# Clone Type Based Comparison among Code Clone Detection Tools and Techniques

**Tarek Salah Uddin Mahmud**

**Bachelor of Science in Software Engineering**

**Institute of Information Technology, University of Dhaka**

**Class Roll: BSSE 0508**

**Registration Number: 2012-912-067**

**Session: 2012-13**

A submitted in partial fulfillment for the degree of

Bachelor of Science in Software Engineering

in the

**Institute of Information Technology**

**University of Dhaka**

**DHAKA, BANGLADESH**

November 2016

# ABSTRACT

Over the last decade many techniques for code clone detection have been proposed. To understand the characteristics of different tools and techniques and where to use those perfectly, comparison between those are needed. There have been a number of comparison and evaluation studies to relate those which provided significant contributions to the clone detection research. These also exposed how challenging it is to compare different tools for some reasons. They are the diverse nature of the detection techniques, the lack of standard similarity definitions and the absence of benchmarks. There is no comparison that helps to understand which tool or technique works better in different types of code clones. In this research project, a comprehensive analysis is provided on the performances of currently available clone detection tools and techniques.

# ACKNOWLEDGEMENT

I would like to express my heartiest gratitude to Dr. Kazi Muheymin-Us-Sakib for his support and timely guidance during the thesis compilation. He has been meticulous and fastidious to bring the best out of me throughout the thesis.

# TABLE OF CONTENTS

Abstract ..... iii
Acknowledgement ..... iv
List of tables ..... vii
List of figures: ..... viii
1 Introduction ..... 1
1.1 Overview ..... 1
1.2 Comparison among code clone detection tools and techniques ..... 2
1.3 Research questions ..... 2
1.4 Contribution of the research project ..... 3
1.5 Organization of the research project ..... 4
2 Background studies ..... 5
2.1 Code Clone ..... 5
2.1.1 Code fragment ..... 5
2.1.2 How duplicate codes are created ..... 6
2.1.3 Problems associated with duplicated codes ..... 6
2.1.4 Clone types ..... 7
2.2 Code Clone Detection ..... 8
2.2.1 Code analysis ..... 8
2.2.2 Application of code clone detection ..... 9
2.2.3 Clone detection process ..... 9
3 Related work ..... 14
3.1 Textual Approaches ..... 16
3.2 Lexical Approaches ..... 18
3.3 Tree-matching Approaches ..... 19
3.4 Comparison Studies Among Code Clone Detection Tools and Techniques ..... 20
3.5 summary ..... 22
4 Methodology ..... 23
4.1 Overview of the implemented tool ..... 23
4.2 Description of the implemented tool ..... 25
4.2.1 Johnson ..... 25
4.2.2 CCFinder ..... 26

4.2.3 CloneDigger ..... 27
Sequence Detection Algorithm ..... 28
4.3 Summary ..... 29
5 Implementation and result analysis ..... 30
5.1 Environmental Setup ..... 30
5.2 Comparative analysis ..... 32
5.2.1 Type 1 (Change in formatting) ..... 33
5.2.2 Type 2 (change in identifiers) ..... 35
5.2.3 Type 3 (add or delete lines) ..... 37
5.2.4 Type 4 (reorder data dependent and control statements) ..... 39
5.3 Summary ..... 42
6 Conclusion ..... 43
6.1 Discussion ..... 43
6.2 Future Work ..... 44
7 References ..... 45

# LIST OF TABLES

Table 3.1: Taxonomy of Clone Detection Techniques and Tools……15
Table 5.1: Evaluation of the tools on clone type-1……33
Table 5.2: Evaluation of the tools on clone type-2……35
Table 5.3: Evaluation of the tools on clone type-3……37
Table 5.4: Evaluation of the tools on clone type-4……39
Table 5.5: Coverage on the scenarios……41

# LIST OF FIGURES:

Figure 2.1: Two code fragments which are clones……5
Figure 2.2: Clone types……7
Figure 2.3: A Generic Clone Detection Process……12
Figure 4.1: Overview of the implemented tool……24
Figure 5.1: Clone type-1 scenarios……34
Figure 5.2: Clone type-2 scenarios……36
Figure 5.3: Clone type-3 scenarios……38
Figure 5.4: Clone type-4 scenarios……40
Figure 5.5: Coverage of the techniques……42

# 1 INTRODUCTION

Code clone detection refers the automated process of finding similarities among code sections. Cloning unnecessarily increases program size. Since many maintenance efforts correlate with program size, this increases the maintenance effort. It requires the developers to find and update several fragments while changing any module. Clone detection techniques first analyze the source code, represent the code in their proposed ways such as text, tokens, Abstract syntax tree etc. and then perform matching algorithms to detect the clones. Various techniques and tools have been proposed for detecting clones. Each of these techniques have their own merits, but are not useful in all the scenarios where a code can be cloned.

## 1.1 OVERVIEW

Several techniques for detecting code clones have been proposed in literature. To understand the characteristics of different tools and techniques and where to use those perfectly, comparison between those are needed. There have been a number of comparison and evaluation studies to relate those which provided significant contributions to the clone detection research. These also exposed how challenging it is to compare different tools for some reasons. They are the diverse nature of the detection techniques, the lack of standard similarity definitions and the absence of benchmarks.

Each study has chosen a number of tools and compared them using precision, recall, computational complexity and memory use. But it is very important to know which tool or technique should be used in different types of clones. It will give the user an idea about which one s/he should use in particular scenario. Unfortunately, there is no comparison that helps to understand which tool or technique works better in different types of code clones. There is also no evaluation of the most recent tools, such as iClones and clone detection feature of visual studio.

## 1.2 COMPARISON AMONG CODE CLONE DETECTION TOOLS AND TECHNIQUES

To detect duplicated code, numerous techniques have been successfully applied on industrial systems. These techniques can be roughly classified into three categories. (i) text based, the program is divided into a number of strings (typically lines) and these strings are compared against each other to find sequences of duplicated strings; (ii) token based, a laxer tool divides the program into a stream of tokens and then searches for series of similar tokens; (iii) tree based, after building a complete parse tree one performs pattern matching on the tree to search for similar sub–trees. Several tools have been proposed based on these techniques. Some of those are SDD (Data structure of an inverted index and an index with n neighbor distance concept) [1] (text based), CPD (Karp Rabin string matching algorithm with frequency table of tokens) [2], iClones (adapted to detect clones over multiple versions at a time) [3] (token based), CloneDigger (XML representation of ASTs and anti-unification/code abstraction) [5], CloneDr (Hashing of syntax trees and tree comparison) [4] (tree based) and clone detection feature in visual studio enterprise. One of the first comparative experiments was conducted by Bailey and Burd [6], who compared three state of the art clone detection and two plagiarism detection tools. Although they were able to verify all the clone candidates, the limitations of the case study in terms of a single subject system, modest system size and validation subjectivity may make their findings less than definitive. Considering the limitations of Burd and Bailey's study, Bellon et al. set out to conduct a larger tool comparison experiment [7] on the same three clone detection tools used in Burd and Bailey's study and three additional clone detection tools. They also used a more diverse set of larger software systems, consisting of four Java and four C systems. While their study is the most extensive to date, only a small proportion of the clone detection tools were evaluated. Bellon's framework has been reused in experiments by Koschke et al. [8] and Ducasse et al. [9]. To date, no one works with the different types of code clones.

## 1.3 RESEARCH QUESTIONS

As found from the previous discussion, to use a tool or technique, it is important to know the nature or those. The general lack of evaluation is worsen by the fact that there are no agreed upon evaluation criteria or representative benchmarks. Finding such universal criteria is difficult,

since techniques are often designed for different purposes and each has its own tunable parameters. The following research question requires our attention here:

**RQ:** What tool or technique should one use when the scenario of the clone is known?

To address this question the following sub questions must be answered.

- What kind of matches are found?

  Depending on the maintenance task at hand, someone may be looking for specific kinds of duplication. For instance, during a problem assessment phase, maintainers want to obtain an overall report of the amount of duplication existing in all program files. On the other hand, during a restructuring phase, maintainers are interested in a duplication tool that detects only the programming constructs that one can restructure using a particular tool. Therefore a refactoring tool, moving methods in the class hierarchy, is interested only in duplicated method bodies.

- How does it perform?

  When using a detection technique one wishes it will detect all types of code clones. Therefore you need to establish the performance of each technique for different types of clones.

## 1.4 CONTRIBUTION OF THE RESEARCH PROJECT

Duplicate code is a major problem in software development. It requires the developers to find and update several fragments while changing any module. There are many tools and techniques for detecting code clones. But there is no comparative study that helps a user to know which tool works better in different types of clones. The results of this study may assist new potential users of clone detection techniques in understanding the range of available techniques and tools and selecting those most appropriate for their needs. It may also assist in identifying remaining open research questions, avenues for future research, and interesting combinations of techniques.

## 1.5 ORGANIZATION OF THE RESEARCH PROJECT

This section gives an overview of the remaining chapters of this research project. The chapters are organized as follows

- Chapter 2: Background Study, In this chapter, background knowledges needed to study on code clone is presented.
- Chapter 3: Literature Review, In this chapter, a brief about researches targeting code clone detection, and its impact, evolution and comparison among techniques is presented.
- Chapter 4: Methodology, Brief description about the implementation of three techniques in a tool is described in this chapter.
- Chapter 5: Result Analysis, This chapter discusses the results of the experiment and explains the performances of the tools and techniques.
- Chapter 6: Conclusion In this concluding chapter, a brief of the whole work is summarized and future directions are described.

# 2 BACKGROUND STUDIES

## 2.1 CODE CLONE

Code clone is a computer programming term for a sequence of source code that occurs more than once. It can be located either within a program or across different programs owned or maintained by the same entity. Cloning unnecessarily increases program size. It requires the developers to find and update several fragments while changing any module. This section contains the related topics which should be known before introducing code clone.

### 2.1.1 Code fragment

A Code Fragment (CF) is any sequence of code lines (with or without comments). Clone is detected using comparison between the fragments in a source code. It can be of any types of code, for example function definition, begin-end block, or sequence of statements. A CF is identified by its file name and begin-end line numbers in the original code base.

Let CF1 and CF2 are two code fragments. CF2 is a clone of CF1 if they are similar by some given definition of similarity, that is, f(CF1) = f(CF2) where f is the similarity function. A similarity function can be defined in various ways such as, exact match between fragments, match fragments after removing the comments or normalizing identifiers. Two fragments that are similar to each other form a clone pair (CF1, CF2), and when many fragments are similar, those form a clone class or clone group. Fig 2.1 shows two code fragments which are clones by nature.

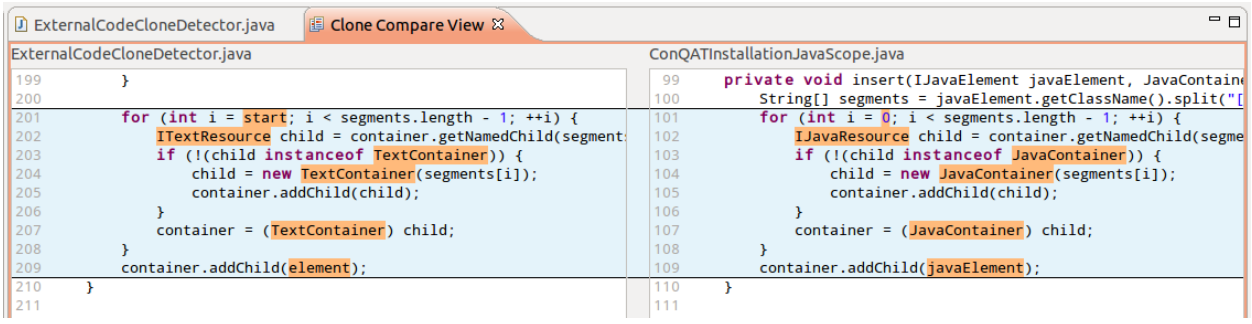



Figure 2.1: Two code fragments which are clones.

### 2.1.2 How duplicate codes are created

There are a number of reasons why duplicate codes may be created, including:

- When multiple programmers are working on different parts of the same program at the same time. Since they are working on different tasks, they may be unaware their colleague has already written similar code that could be repurposed for their own needs.
- Copy paste programming, in which a section of code is copied because it is workable. In most cases this operation involves slight modifications in the cloned code such as renaming variables or inserting/deleting codes. A copy is created due to the programmer does not truly knowing the language or not having the time to do it properly. For example, a programmer has to write a sort function where he has no idea how to write it in java. He will search this in the code base or through internet. And put it in the program. Thus a clone has been made.
- Functionality that is very similar to that in another part of a program is required and a developer independently writes code that is very similar to what exists elsewhere. That such independently rewritten code is typically not exactly similar.
- Generated code, where having duplicate codes may be desired to increase speed or ease of development. For example, to design a desktop application in java everyone uses the auto generated code generated by the IDE.

### 2.1.3 Problems associated with duplicated codes

Duplicated clones introduces many problems in software development and management. Inappropriate code duplication generally makes editing more difficult due to unnecessary increases in complexity and length. This may lead to,

- Increased maintenance costs,
- More human errors,
- Forgotten or overlooked pieces of code,
- Greater file size,
- Indicative of a sloppy design.

### 2.1.4 Clone types

Code can be cloned in several ways. There are four main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting into another location. In the following the types of clones are provided based on both the textual (Types 1 to 3) [1] and functional (Type 4) [2] similarities:

- Type-1: Identical code fragments except for variations in whitespace, layout and comments. Fig 2.2(b) shows this type of code clone.
- Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments. Fig 2.2(c) shows this type of code clone.
- Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments. Fig 2.2(d) shows this type of code clone.
- Type-4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants. Fig 2.2(e) shows this type of code clone.

```
void  sumProd(int n) {
float  sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
    {sum=sum + i;
     prod = prod * i;
     foo(sum, prod); }}
```

(a)Original Code

```
void  sumProd(int n) {
float  sum=0.0; //C1
float prod =1.0;
 for (int i=1; i<=n; i++)
    {sum=sum + i;
     prod = prod * i;
     foo(sum, prod); }}
```

```
void  sumProd(int n){
float  s=0.0; //C1
float p =1.0;
for (int j=1; j<=n; j++)
    {s=s + j;
     p = p * j;
     foo(s, p); }}
```

(b)

```
void  sumProd(int n) {
float  sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
    {sum=sum + i;
     prod = prod * i;
     foo(prod); }}
```

(c)

```
void  sumProd(int n) {
float  sum=0.0; //C1
float prod =1.0;
int i=0;
 while (i<=n)
   { sum=sum + i;
     prod = prod * i;
     foo(sum, prod);
      i++ ;  }}
```

(d) (e)

Figure 2.2: Clone types.

## 2.2 CODE CLONE DETECTION

Code clone detection refers the automated process of finding similarities among code sections. Cloning unnecessarily increases program size. Since many software maintenance efforts correlate with program size, this increases the maintenance effort. It requires the developers to find and update several fragments while changing any module.

### 2.2.1 Code analysis

Source code analysis is the automated testing of source code for the purpose of debugging a computer program or application before it is distributed or sold. The source code is the most permanent form of a program, even though the program may later be modified, improved or upgraded. Source code analysis can be either static or dynamic.

- In static analysis, debugging is done by examining the code without actually executing the program. This can reveal errors at an early stage in program development, often eliminating the need for multiple revisions later.
- After static analysis has been done, dynamic analysis is performed in an effort to uncover more subtle defects or vulnerabilities. Dynamic analysis consists of real-time program testing.

Clone is detected through static analysis of a source code.

### 2.2.2 Application of code clone detection

Finding clones is typically useful in the following cases:

- When updating existing code**.** At the time of fixing a bug, or responding to changes in requirements, one usually start by finding the location in the code that s/he need to change. Before making the change, search for clones of that code segment. If clones are discovered:
    1. Consider whether it is needed to make the same change to each clone.
    2. Consider also whether this is a good opportunity to refactor the cloned code into a shared method or class.
- When merging multiple code bases. Suppose a retail banking software system maintained by Tata Consultancy Services (TCS), that is in active use by a number of banks (with different codebases). The company decided to form a common codebase for all these banks. It will be very difficult and a huge business lose if they need duplicated efforts in-
    1. Delivering common features,
    2. Maintain these common features separately.

    Code clone detection is needed here to identify the duplicate features and making a generalized feature that can support all these banks.

### 2.2.3 Clone detection process

A clone detector must try to find pieces of code of high similarity in a system's source text. The main problem is that it is not known which code fragments may be repeated. Thus the detector really should compare every possible fragment with every other possible fragments. Such a comparison is prohibitively expensive from a computational point of view and thus, several measures are used to reduce the domain of comparison before performing the actual comparisons. Even after identifying potentially cloned fragments, further analysis and tool support may be required to identify the actual clones.

In this section, an overall summary of the basic steps in a clone detection process will be provided. This generic overall picture allows us to compare and evaluate clone detection tools with respect to their underlying mechanisms for the individual steps. It also allows to evaluate their level of support for these steps.

- **Preprocessing**: At the beginning of any clone detection approach, the source code is partitioned and the domain of the comparison is determined. There are three main objectives in this phase:
    - **Remove uninteresting parts**: Whole source code uninteresting to the comparison phase. Those are filtered out in this phase. For example, partitioning is applied to embedded code to separate different languages (for example, SQL embedded in Java code, or Assembler in C code).
    - **Determine source units**: After removing the uninteresting code, the remaining source code is partitioned into a set of disjoint fragments called source units. These units are the largest source fragments that may be involved in direct clone relations with each other. Source units can be at any level of granularity, for example, files, classes, functions/methods, begin-end blocks, statements, or sequences of source lines.
    - **Determine comparison units**: Source units may need to be further partitioned into smaller units depending on the comparison technique used by the tool. For example, source units may be subdivided into lines or even tokens for comparison. Comparison units can also be derived from the syntactic structure of the source unit.
- **Transformation**: Once the units of comparison are determined, if the comparison technique is other than textual, the source code of the comparison units is transformed to an appropriate intermediate representation for comparison.
    - **Extraction**: Extraction transforms source code to the form suitable as input to the actual comparison algorithm. Depending on the tool, it typically involves one or more of the following steps.

- **Tokenization**: In case of token-based approaches, each line of the source is divided into tokens according to the lexical rules of the programming language of interest. The tokens of lines or files then form the token sequences to be compared. All whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequences. CCFinder [4] and Dup [3] are the leading tools that use this kind of tokenization on the source code.
- **Parsing**: In case of syntactic approaches, the entire source code base is parsed to build a parse tree or (possibly annotated) Abstract Syntax Tree (AST). The source units to be compared are then represented as subtrees of the parse tree or the AST, and comparison algorithms look for similar subtrees to mark as clones [5, 6, 7]. Metrics-based approaches may also use a parse tree representation to find clones based on metrics for subtrees [8, 9].
- **Control and Data Flow Analysis**: Semantics-aware approaches generate Program Dependence Graphs (PDGs) from the source code. The nodes of a PDG represent the statements and conditions of a program, while edges represent control and data dependencies. Source units to be compared are represented as subgraphs of these PDGs. The techniques then look for isomorphic subgraphs to find clones [10, 11]. Some metrics-based approaches use PDG subgraphs to calculate data and control flow metrics [8, 9].

- **Normalization**: Normalization is an optional step intended to eliminate superficial differences such as differences in whitespace, commenting, formatting or identifier names.
  - **Removal of whitespace**: Almost all approaches disregard whitespace, although line-based approaches retain line breaks.
  - **Removal of comments**: Most approaches remove and ignore comments in the actual comparison.

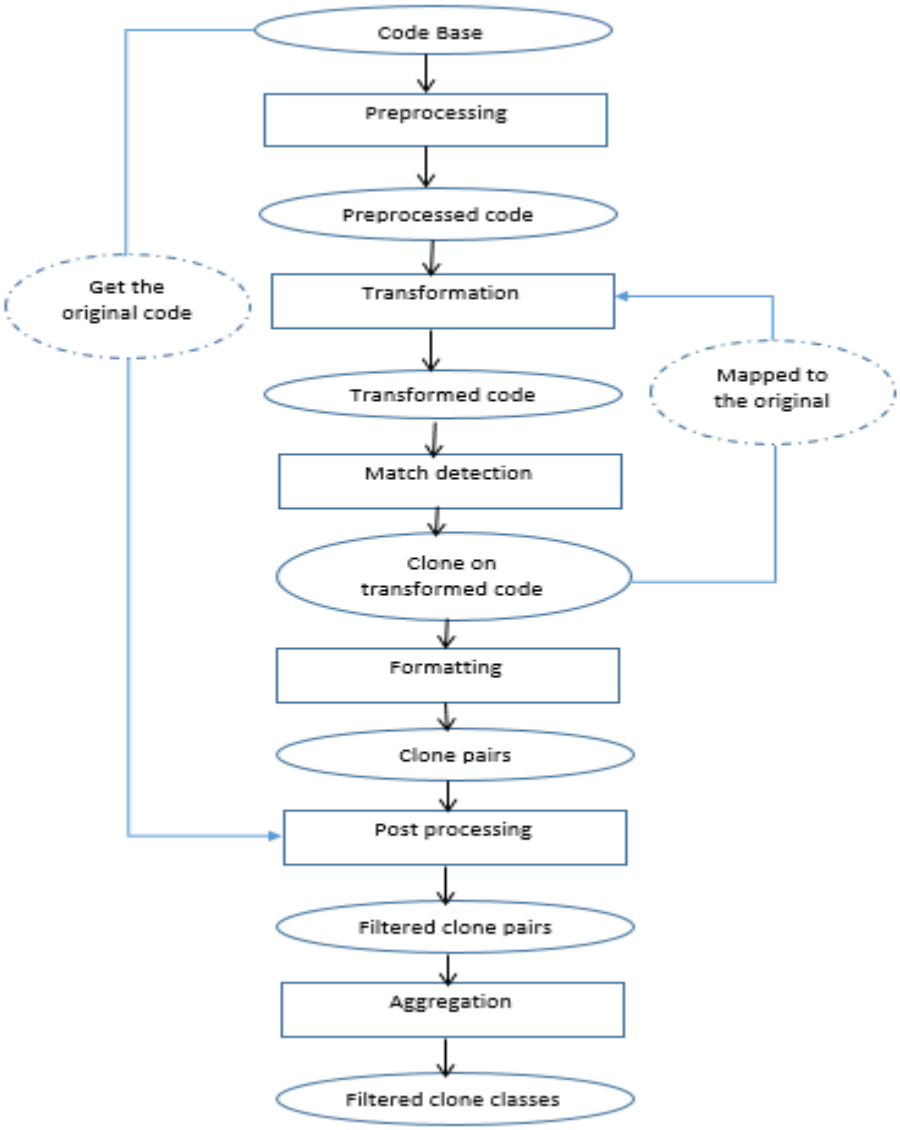



Figure 2.3: A Generic Clone Detection Process

- **Normalizing identifiers**: Most approaches apply an identifier normalization before comparison in order to identify parametric Type-2 clones. In general, all identifiers in the source code are replaced by the same single identifier in such normalizations.
- **Pretty-printing of source code**: Pretty printing is a simple way of reorganizing the source code to a standard form that removes differences in layout and spacing. Pretty printing is normally used in text-based clone detection approaches to find clones that differ only in spacing and layout.

- **Match Detection**: The transformed code is then fed into a comparison algorithm where transformed comparison units are compared to each other to find matches. The output of match detection is a list of matches in the transformed code which is represented or aggregated to form a set of candidate clone pairs. Each clone pair is normally represented as the source coordinates of each of the matched fragments in the transformed code.
- **Formatting**: In this phase, the clone pair list for the transformed code obtained by the comparison algorithm is converted to a corresponding clone pair list for the original code base. Source coordinates of each clone pair obtained in the comparison phase are mapped to their positions in the original source files.
- **Post-processing**: In this phase, clones are ranked or filtered using manual analysis or automated heuristics.
- **Aggregation**: While some tools directly identify clone classes, most return only clone pairs as the result. In order to reduce the amount of data, perform subsequent analyses or gather overview statistics, clones may be aggregated into clone classes.

# 3 RELATED WORK

Many clone detection approaches have been proposed in the literature. Based on the level of analysis applied to the source code, the techniques can roughly be classified into three main categories: textual, lexical and syntactic.

1. Textual approach: Textual approaches (or text-based techniques) use little or no transformation on the source code before the actual comparison, and in most cases raw source code is used directly in the clone detection process. For example, SDD, NICAD, Simian1 etc.
2. Lexical approach: Lexical approaches (or token-based techniques) begin by transforming the source code into a sequence of lexical "tokens" using compiler-style lexical analysis. The sequence is then scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones. Lexical approaches are generally more robust over minor code changes such as formatting, spacing, and renaming than textual techniques. For example, Dup, CCFinder, CP-Miner etc.
3. Syntactic approaches: Syntactic approaches (or tree-based approaches) use a parser to convert source programs into parse trees or abstract syntax trees which can then be processed using either tree matching or structural metrics to find clones. For example, CloneDr, Deckard, CloneDigger etc.

In this section, the state of the art is summarized in automated clone detection by introducing and clustering available clone-detection tools and techniques by category. The techniques can be distinguished primarily by the type of information their analysis is based on and the kinds of analysis techniques that they use. Table 3.1 provides a high-level overview of the techniques and tools in the form of a taxonomy where the first column shows the underlying approach of the tools/techniques, the second column either shows the name of the corresponding tool or the last name of the first author has been used as the tool name (if no tool name is found) and the third column shows their one sentence description.

Table 3.1: Taxonomy of Clone Detection Techniques and Tools

| Approach | Tool/1stAuthor | One Sentence Description |
|---|---|---|
| Text-based Approach | Johnson | Hashing of strings per line, then textual comparison |
| | Duploc | Hashing of strings per line, then visual comparison using dot plots |
| | DuDe | Composes smaller isolated fragments of duplication with scatter-plot |
| | SDD | Data structure of an inverted index and an index with n-neighbor distance concept |
| | NICAD | Syntactic pretty-printing, then textual comparison with thresholds |
| | Simian | Textual comparison with flexible options (such as, ignore all identifiers) |
| Token-based Approach | Dup | Suffix trees for tokens per line |
| | CCFinder | Token normalizations, then suffix-tree based search |
| | RTF | Flexible tokenization and suffix-array comparison |
| | CP-Miner | Data mining for frequent token sequences |
| | CPD | Karp-Rabin string matching algorithm with frequency table of tokens |
| | CloneDetective | Normalized token comparison integrated with Visual Studio |
| | clones | Normalized token comparison with suffix-tree |

| | | |
|---|---|---|
| | iClones | *clones* is adapted to detect clones over multiple versions at a time |
| Tree-based Approach | CloneDr | Hashing of syntax trees and tree comparison |
| | cpdetector | Serialization of syntax trees and suffix-tree detection |
| | Deckard | Metrics for syntax trees and metric vector comparison with hashing |
| | CloneDetection | XML representation of ASTs with frequent item sets techniques of data mining |
| | CloneDigger | XML representation of ASTs and anti-unification/code abstraction |

## 3.1 TEXTUAL APPROACHES

Textual approaches use little or no transformation/normalization on the source code before the actual comparison. In most cases raw source code is used directly in the clone detection process. And uses textual comparison or string matching between code fragments for clone detection.

Johnson [12, 13] pioneered text-based clone detection using “fingerprints” on substrings of the source code. First, code fragments of a fixed number of lines are hashed. A sliding window technique in combination with an incremental hash function is used to identify sequences of lines having the same hash value as clones. To find clones of different lengths, the sliding window technique is applied repeatedly with various lengths. This approach can only identify the exact matches in the code fragments. It cannot handle the whitespaces and comments.

One of the newer text-based clone detection approaches is that of Ducasse et al. [14]. The technique is based on dot plots. A dot plot – also known as a scatter plot – is a two-dimensional chart where both axes list source entities. In the case of the approach by Ducasse et al., comparison entities are the lines of a program. There is a dot at coordinate (x, y) if x and y are

equal. Two lines must have the same hash value to be considered equal. Dot plots can be used to visualize clone information, where clones can be identified as diagonals in dot plots. The detection of clones in dot plots can be automated, and Ducasse et al. use string-based dynamic pattern matching on dot plots to compare whole lines that have been normalized to ignore whitespace and comments. Diagonals with gaps indicate possible Type-3 clones, and a pattern matcher is run over the matrix to find diagonals with holes up to a certain size. This approach can identify the type-1 clone perfectly, but have a poor performance in type-3 clones. It missed some nearby clones, which are called near miss clones.

An extension of the Ducasse et al. approach is used by Wettel & Marinescu [15] to find near miss clones using dot plots. Starting with removal of whitespace and comments, use string based dynamic pattern matching in the lines having the same hash value. Then the algorithm chains together neighboring lines to identify certain kinds of Type-3 clones which are missed in the approach followed in Ducasse et al. It improves the performance of Ducasse et al. in detecting type-3 clones, but cannot find all the type-3 clones.

SDD [16] is another similar approach that uses the method proposed in Ducasse et al. First, normalize the source code by removing whitespace and comments. Then using the string based dynamic pattern matching it gets the hash value of the strings and put them in the dot plots. Then this approach applies an n-neighbor approach in finding near-miss clones. It can detect the near miss clones missed by the approach proposed in Ducasse et al.

NICAD [17] is also text-based approach. However it exploits the benefits of tree-based structural analysis, which is based on lightweight parsing to implement flexible pretty-printing, code normalization, source transformation and code filtering. Although NICAD is essentially a hybrid technique, it is considered as a text-based approach because it uses textual comparisons in the matching part of the process. It can identify some type-2 clones as well as type-1 and type-3 clones.

Marcus and Maletic [18] apply Latent Semantic Indexing (LSI) to source text in order to find high level concept clones, such as, Abstract Datatypes (ADTs), in the source code. This information retrieval approach limits its comparison to comments and identifiers, returning two code

fragments as potential clones or a cluster of potential clones when there is a high level of similarity between their sets of identifiers and comments.

Using the row code, without normalizing the identifier, these techniques obtains better performance in type-1 and type-3 clones. But still there is no analysis on the performance of the text-based approaches in detecting type-2 and type-4 clones.

## 3.2 LEXICAL APPROACHES

Lexical approaches (or token-based techniques) begin by transforming the source code into a sequence of lexical "tokens" using compiler-style lexical analysis. The sequence is then scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones. Lexical approaches are generally more robust over minor code changes such as formatting, spacing, and renaming than textual techniques.

Efficient token-based clone detection was pioneered by Brenda Baker. In Baker's tool Dup [3], lines of source files are first divided into tokens by a lexical analyzer. Tokens are split into parameter tokens (identifiers and literals) and non-parameter tokens, with the non-parameter tokens of a line summarized using a hashing function, and the parameter tokens are encoded using a position index for their occurrence in the line. This encoding abstracts away from concrete names and values of parameters, but not from their order, allowing for consistently parameter-substituted Type-2 clones to be found. All prefixes of the resulting sequence of symbols are then represented by a suffix tree, a tree where suffixes share the same set of edges if they have a common prefix. If two suffixes have a common prefix, obviously the prefix occurs more than once and can be considered a clone.

The technique allows one to detect Type-1 and Type-2 clones, and Type-3 clones can be found by concatenating Type-1 or Type-2 clones if those are lexically not farther than a user-defined threshold away from each other. These can be summarized using a dynamic-programming technique [19]. Kamiya et al. later extended this technique in CCFinder [4], using additional source normalizations to remove superficial differences such as changes in statement bracketing such as, **if(a) b=2; vs. if(a) {b=2;}**. CCFinder is itself used as the basis of other techniques, such as

Gemini [21], which visualizes near-miss clones using scatter plots, and RTF [20], which uses a more memory-efficient suffix-array in place of suffix trees and allows the user to tailor tokenization for better clone detection.

CP-Miner [22] is another state-of-the-art token-based technique, which uses frequent subsequence data mining to find similar sequences of tokenized statements. A token- and line-based technique has also been used by Cordy et al. [23] to detect near-miss clones in HTML web pages. An island grammar is used to identify and extract all structural fragments of cloning interest, using pretty-printing to eliminate formatting and isolate differences between clones to as few lines as possible. Extracted fragments are then compared to each other line-by-line using the Unix diff algorithm to assess similarity.

As syntax is not taken into account, clones found by token-based techniques may overlap different syntactic units. However, using either preprocessing or post-processing, clones corresponding to syntactic blocks can be found if block delimiters are known or lightweight syntactic analysis such as island parsing is added.

## 3.3 TREE-MATCHING APPROACHES

Tree-matching approaches (or tree-based techniques) find clones by searching similar sub trees. Variable names, literal values and other leaves (tokens) in the source may be abstracted in the tree representation, allowing for more sophisticated detection of clones.

One of the pioneering tree-matching clone detection techniques is Baxter et al.'s CloneDr [7]. A compiler generator is used to generate a constructor for annotated parse trees. Subtrees are then hashed into buckets. Only within the same bucket, subtrees are compared to each other by a tolerant tree matching. The hashing is optional but reduces the number of necessary tree comparisons drastically.

This approach has been adapted by the AST-based clone detectors of Bauhaus [24] as ccdiml. The main differences from CloneDr are ccdiml's explicit modeling of sequences, which eases the search for groups of subtrees that together form clones, and its exact matching of trees. Yang et

al. [6] has proposed a dynamic programming approach for handling syntactic differences in comparing similar subtrees.

Wahler et al. [5] find exact and parameterized clones at a more abstract level by converting the AST to XML and using a data mining technique to find clones. Structural abstraction, which allows for variation in arbitrary subtrees rather than just leaves (tokens), has been proposed by Evans et al. [2] for handling exact and near-miss clones with gaps.

To avoid the complexity of full subtree comparison, recent approaches use alternative tree representations. In the approach of Koschke et al. [25], AST subtrees are serialized as AST node sequences for which a suffix tree is then constructed. This idea allows to find syntactic clones at the speed of token-based techniques. A function-level clone detection approach based on suffix trees has been proposed by Tairas and Gray based on Microsoft's new Phoenix framework [26].

A novel approach for detecting similar trees has been presented by Jiang et al. [27] in their tool Deckard. In their approach, certain characteristic vectors are computed to approximate the structure of ASTs in a Euclidean space. Locality Sensitive Hashing (LSH) is then used to cluster similar vectors using the Euclidean distance metric (and thus can also be classified as a metrics based techniques) and thus finds corresponding clones.

## 3.4 Comparison Studies Among Code Clone Detection Tools and Techniques

Although there is no work in the literature that provides a scenario-based evaluation of the techniques and tools similar to this study, several tool comparison experiments have been conducted to estimate the abilities of the tools in terms of precision, recall, and time and space requirements.

One of the first experiments was conducted by Bailey and Burd [28], who compared three state-of-the-art clone detection and two plagiarism detection tools. They began by validating all the clone candidates of the subject application obtained with all the techniques of their experiment

to form a human oracle, which was then used to compare the different techniques in terms of several metrics to measure various aspects of the reported clones.

Although they were able to verify all the clone candidates, the limitations of the case study in terms of a single subject system, modest system size and validation subjectivity may make their findings less than definitive. Moreover, the intention of their analysis was to assist in preventative maintenance tasks, which may have influenced their clone validation process.

Considering the limitations of Burd and Bailey's study, Bellon et al. set out to conduct a larger tool comparison experiment [1] on the same three clone detection tools used in Burd and Bailey's study and three additional clone detection tools. They also used a more diverse set of larger software systems, consisting of four Java and four C systems totaling almost 850 KLOC. As in the study of Burd and Bailey, a human oracle validated a random sample of about 2% of the candidate clones from all the tools evenly and blindly. While their study is the most extensive to date, only a small proportion of the clone candidates were oracled and several other factors may have influenced the results [29]. Bellon's framework has been reused in experiments by Koschke et al. [25] and Ducasse et al. [14] (partially), but without any improvements to the framework.

Rysselberghe and Demeyer [30, 31] have evaluated prototypes of three representative clone detection techniques, providing comparative results in terms of portability, kinds of duplication reported, scalability, number of false matches, and number of useless matches. However, they did not make a reference set, used relatively small subject systems and did not provide the reliability of the judge(s) that validated the found clones. Moreover, rather than quantitative evaluation of the detection techniques, their intention was to determine the suitability of the clone detection techniques for a particular maintenance task (refactoring) which might have influenced their clone validation.

Another interesting study has been conducted by Bruntink et al. [32], in which several clone detection techniques are evaluated in terms of finding known cross-cutting concerns in C programs with homogeneous implementations.

## 3.5 SUMMARY

Although there are different comparative analysis is available, each of them has chosen a number of tools and compared them using precision, recall, computational complexity and memory use. But it is very important to know which tool or technique should be used in different types of clones. It will give the user an idea about which one they should use in particular scenario. Unfortunately, there is no comparison that helps to understand which tool or technique works better in different types of code clones. There is also no evaluation of the most recent tools, such as iClones and clone detection feature of visual studio.

# 4 METHODOLOGY

A tool to identify code clones in source code is developed in this chapter. As discussed in the earlier chapters a comparison study among will be proposed based on clone types. For this, eight techniques will be considered. Five of them are free tools found through the internet. The others are implemented as a tool where user input a java project and select one of the approaches and the tool perform the selected approach on the given source code. Then it will show the matched lines found in the source code.

## 4.1 OVERVIEW OF THE IMPLEMENTED TOOL

A comparative study among code clone detection tools and techniques are proposed in this research project. The matrices of the comparison are the clone types. For this study, the following tools and techniques are considered

1. Johnson [12, 13]
2. SDD [16]
3. CCFinder [4]
4. CPD [34]
5. Clones [28]
6. CP-Miner [22]
7. CloneDigger [11] and
8. CloneDr [7]

CP-Miner and CPD are free tool, SDD and cloneDr are found as eclipse plugin and clones is a visual studio clone detection feature. The other three techniques will be implemented as a tool. The overview of the implemented tool is shown in Figure 4.1.

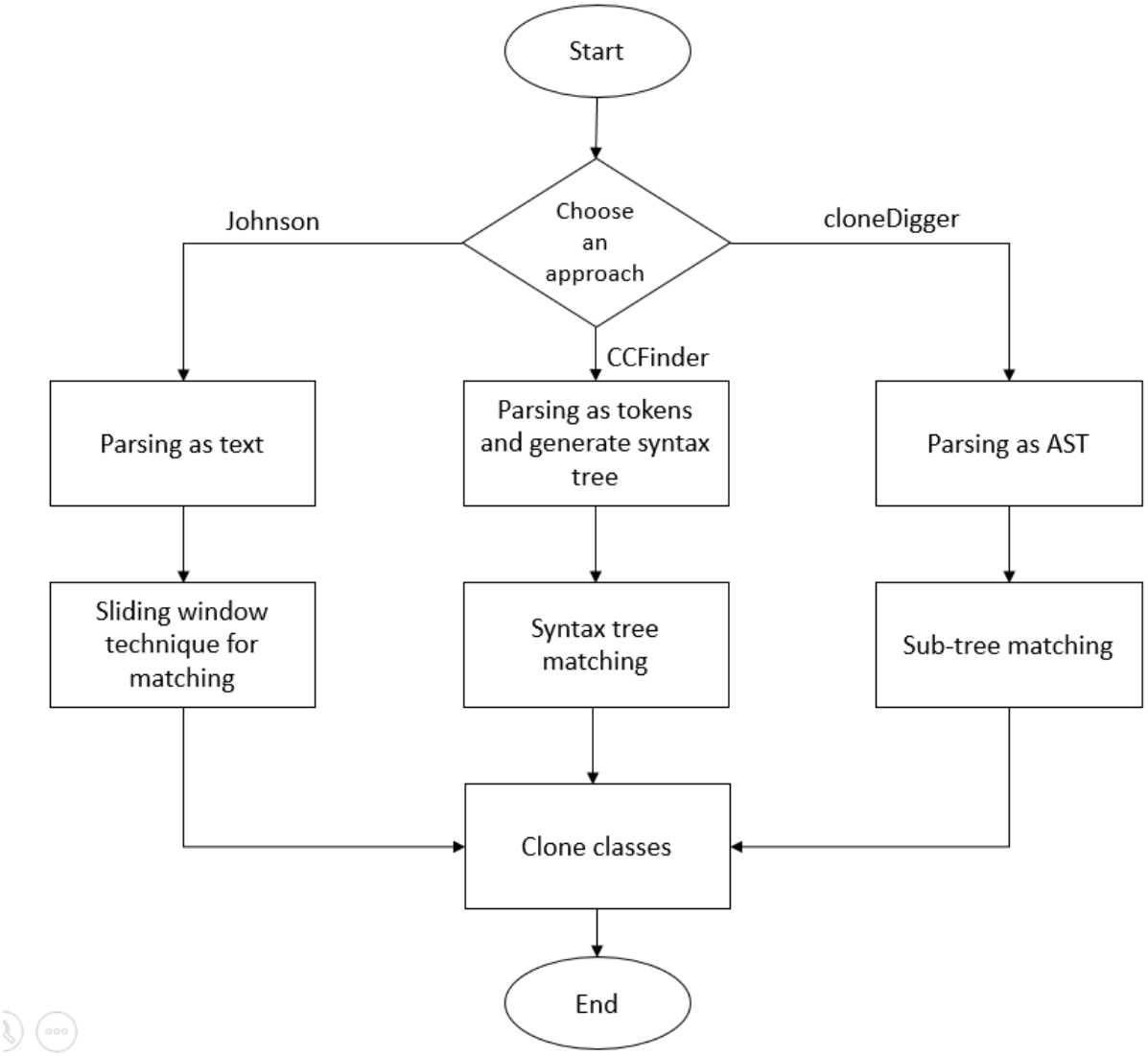



Figure 4.1: Overview of the implemented tool.

User will give a java project as input a java project and give a choice which approach s/he is willing to use. Then the tool generate clone classes using the chosen approach. Johnson is a text based approach of detecting clones. It parses the whole source code as text and matches the code fragments using sliding window technique. CCFinder is a token-based approach which first normalizes the identifiers and parse the normalized source as tokens. Then generate suffix tree using the token sequences and perform a tree matching algorithm to detect the clones. On the other hand CloneDigger is a tree-based clone detection approach. At first it generates Abstract Syntax Tree (AST) by parsing the source code. Then matches the sub-trees to detect the clones.

## 4.2 DESCRIPTION OF THE IMPLEMENTED TOOL

. The tool contains three approaches. Those are

1. Johnson [12, 13],
2. CCFinder [4] and
3. CloneDigger [11].

A brief description of the approaches is given in the following sub-sections.

### 4.2.1 Johnson

It is the earliest approach of clone detection. It is a text based clone detection technique where the source is considered as text and analyzed by the way documents are analyzed. It does not use any modification in the source code when matching. The sliding window technique used in the matching phase of this approach. The approach can be summarized as,

1) For each file being considered, apply a text to text transformation to discard characters not to be considered for matching. For this study, this is an identity transformation (output equals input). However, various types of approximate matching can be accommodated by discarding different parts of the input.
2) Generate a set of substrings that cover the source (i.e., every character of text appears in at least one substring).
3) Identify which of the substrings match (i.e., have the same sequence of characters).
4) Transform this database of raw matches into a form that more concisely expresses the same information.
5) Perform task-specific data reduction.
6) Summarize high-level matches.

Steps (2) and (3) are information collecting phases, (4) is an information-preserving transformation, (5) an aggregation and simplification phase, and (6) the presentation of results in a useful form. Phase (1) provides greater sensitivity for particular types of input.

### 4.2.2 CCFinder

CCFinder is token-based code clone detection tool. It is proposed by Kamiya et al. [4]. CCFinder uses a suffix-tree algorithm with both time and space complexities O(mn), where m is the maximum length of involved clones and n is the total length of the source file. If it would be assumed that m does not depend on n and it is bounded by some fixed length, the time and space complexities will practically be O(n).

The optimizations employed inCCFinder to handle large source files are as follows:

- **Alignment of Token Sequence**: Source code has its inherent granularity such as character, token, statement, or block. Code portions of a code clone should begin at their boundary. For example, a code portion, which begins at the middle of a statement X and ends the middle of a statement Y, is less useful than a code portion which begins at the beginning of Y. As a simple filtering for this purpose, it allow only specific tokens at the beginning of clones as leading tokens. Keywords that initiate statements are leading tokens. In C and C++ source files, those keywords are '#', '{', keywords for selection statements (else, if, switch, etc.), iteration statements (do, for, and while), jump or structured exception handling statements (break, catch, return, etc), and declarations (class, enum, typedef, etc). Also, tokens following keywords that terminate statements (';', ')'} or labels (':') are also leading tokens. The number of nodes in the suffix tree was reduced to one third by this filtering. This technique might slightly reduce the sensitivity of clone detection, but practically it is very important to make the technique scalable.
- **Repeated Code Removal:** Repetition of a short code portion tends to generate many clone pairs. For example, consider the following code:

```
switch (c) {
        case '0' : value = 0; break;
        case '1' : value = 1; break;
        case '2' : value = 2; break;
        case '3' : value = 3; break;
        case '4' : value = 4; break;
```

```
}
```

Now, consider that the following code section is also included in the target source files:

```
case 'a':
        flag = 2;
        break;
```

In this case, five code portions make a clone class, { a2-a2,a3-a3, ..., a6-a6, b1-b3 }, where each pair of the code portions makes a clone pair, and the number of maximal clone pairs are ${}_6C_2$ = 15, in total. To avoid this explosion of clone pairs, a heuristic approach is introduced. Upon building a suffix-tree, if a repetition of a2 is identified at a3, the succeeding repetition section (a3-a6) is not intentionally inserted into the tree, so that a part of the clone pairs is not being reported. However, the clone pair (a2-a2, b1-b3) is still extracted, which offers sufficient information. The repeated code removal process also prevents detection of self-clones, e.g., (a2-a5, a3- a6), or repetition of "constant" declarations.

- **Concatenation of Tokens:** Just before computing the match in the token sequence, abutting tokens, except for punctuator keywords, are concatenated. This process reduces the length of a token sequence in exchange for an increase in variation of the tokens.
- **Division of Large Archive of Source Files:** If the total size of source files exceeds the memory space for a single suffix-tree, the tool automatically employs a 'divide and conquer' approach. The input source files are divided into several parts. For each combination of the parts, a sub-suffix tree is built to extract clone pairs. The total collection of clone pairs will finally be the output. Let m be the number of subsets of source files, and then the number of pairs of the chunks (i.e., the number of constructed subsuffix trees) is ${}_mC_2$. Therefore, the time complexity becomes O($m^2$).

### 4.2.3 CloneDigger

CloneDigger is a tree-based code clone detection technique. In order to find code clones using AST it is needed to compare each subtree to each other subtree in AST. Computing the similarities of all subtree pairs are not efficient, which complexity of computation is O($N^3$),where N is number of nodes in AST. To increase the scalability of the approach a hash function is used that

partitions the AST into similar subtrees. If there are two subtrees whose similarity exceeds the threshold then these subtrees are called clones.

Hashing function is used to hash subtrees into some buckets if the mass of the subtree exceeds the mass threshold(implemented by basic algorithm given below).The single subtree clone were detected by using hashing function but the subtree sequence clone cannot be detected. To overcome from this problem a list structure is built where each list is associated with a sequence in the program and stores the hash codes of each subtree element of associated sequence. The algorithm for implementing this is given as below-

**Basic Algorithm**

```
1   Clones=ϕ
2   For each subtree i
3       If mass(i)>=Threshold
4       Then hash i to bucket
5   For each subtree i and j in the same bucket
6       If Compare tree(i,h) > SimilarityThreshold
7       Then For each subtree s of i
8           If IsMember(clone s,s)
9           Then RemoveClonePair(clone s,s)
10      For each subtree s of j
11          If IsMember(clone s,s)
12          Then RemoveClonePair(clone s,s)
13      AddClonePair(clones,i,j)
```

**SEQUENCE DETECTION ALGORITHM**

```
1   Build the list structure s describing sequences
2   For k=MinimumSequenceLengthThreshold to MaximumSequenceLength
3       Place all subsequences of length k into buckets according to subsequence hash
```

```
4       For each subsequence i and j in the same bucket
5           If CompareSequences(i,j,k)>SimilarityThreshold
6           Then
7               RemoveSequenceSubclonesOf(clones i,j,k)
8               AddSequenceClonePair(clones i,j,k).
```

Clone detection by using abstract syntax tree and comparing each subtree or subtree sequence results in finding out exact and near miss clone.

## 4.3 SUMMARY

A tool was developed that implements three approaches, that are chosen for evaluation but implementation is not available, to perform an investigation on comparison among code clone detection techniques. This tool takes java source code as input, can able to apply three approaches on it and output the clone classes available on the source code. Using this tool the three approaches will be evaluated in the next chapter.

# 5 IMPLEMENTATION AND RESULT ANALYSIS

This chapter aims to experimentally evaluate the performance of the used approaches (for example Johnson, CloneDigger etc.) by applying these on some source codes with different types of clones. Three approaches are implemented in Java programming language as a tool. The environment setup for the tool and the other 5 tools are discussed at first. Then a brief description of the scenarios of different clone types and the performances of the tools on these scenarios will be provided.

## 5.1 ENVIRONMENTAL SETUP

This section discusses the equipment that were used to implement the tool as well as to run the experimental procedures for evaluating the approaches. Eight approaches are evaluated in this experiment. They are –

1. Johnson [12, 13] (implemented)
2. SDD [16]
3. CCFinder [4] (implemented)
4. CPD [34]
5. Clones [28]
6. CP-Miner [22]
7. CloneDigger [11] (implemented) and
8. CloneDr [7]

The implemented approaches are implemented as a tool. This tool is implemented in Java programming language. In order to implement the tool, following tools and libraries are used

- Eclipse Mars (4.5.0) []
- JavaParser (2.3.1) []

The experiments on the tool are performed on following PC configuration

- 2.5GHz Intel Core i5
- 4GB RAM
- Windows 10 64bit

- Java SE 1.8

The environment setup for the other tools is given below:

1. **SDD:** This is an eclipse plugin. It is platform independent. Eclipse Mars is used for running this approach. The experiments on this tool are performed on following PC configuration
    - 2.5GHz Intel Core i5
    - 4GB RAM
    - Windows 10 64bit
    - Java SE 1.8
2. **CPD:** This is a linux based tool. The experiments on this tool are performed on following PC configuration
    - 2.5GHz Intel Core i5
    - 4GB RAM
    - Ubuntu 14.04 64bit
3. **Clones:** This is a clone detection feature incorporated with the "Visual Studio". We use this on "Visual Studio 2013". The experiments on this tool are performed in the similar PC configuration used in SDD.
4. **CP-Miner:** This is a software that is developed for windows platform. The experiments on this tool are performed in the similar PC configuration used in SDD.
5. **CloneDr:** This is also an eclipse plugin and also platform independent. The experiments on this tool are performed in the similar PC configuration used in SDD.

## 5.2 COMPARATIVE ANALYSIS

Clone detection techniques are often inadequately evaluated, and only a few studies have looked at some of the techniques and tools []. Of these, the Bellon et al. [2] study is the most extensive to date, with a quantitative comparison of six state-of-the-art techniques, essentially all of those with tools targeted at the C and Java languages. However, even in that careful study, only a small proportion of the clones were oracled, and a number of other factors have been identified as potentially influencing the results []. The general lack of evaluation is exacerbated by the fact that there are no agreed upon evaluation criteria or representative benchmarks. Finding such universal criteria is difficult, since techniques are often designed for different purposes and each has its own tunable parameters.

In an attempt to compare all available clone detection techniques more uniformly, a clone-type based approach is proposed on five free and three implemented code clone detection tools. A small set of hypothetical program editing scenarios representative of typical changes to copy/pasted code is designed in Roy et al. [1] (Figure 5.1, 5.2, 5.3, 5.4). Based on these hypothetical scenarios, it is checked how well the various clone detection techniques perform.

From a program comprehension point of view, finding such true clones is useful since understanding a representative copy from a clone group assists in understanding all copies in that group []. Moreover, replacing all the detected similar copies of a clone group by a function call to the representative copy (such as refactoring) can potentially improve understandability, maintainability and extensibility, and reduce the complexity of the system.

Table 5.1, 5.2, 5.3, 5.4 provides an overall summary of the results of evaluations. The result will be shown using the following matrices:

- Perform well: Perform well in detecting a scenario in code when the matching threshold is over 90%. 1 point will be given to the technique for this scenario.
- Depends on the threshold: the technique detects this scenario when the threshold is in between 70% to 90%. 1/2 point will be given to the technique for this scenario.
- Cannot detect: the technique cannot detect this scenario if the threshold is less than 70%. This technique will get 0 point for this scenario.

In the following paragraphs, every scenario is considered in estimating the ability of the techniques to accurately detect those.

### 5.2.1 Type 1 (Change in formatting)

Roy et al. [1] copied a function that calculates the sum and product of a sequence of numbers (*1...n*) three times, making changes in whitespace in the first fragment (*S1(a)*), changes in commenting in the second (*S1(b)*), and changes in formatting in the third (*S1(c)*) (*Figure 5.1*).

An ideal clone detection technique should recognize all three copy-pasted/modified fragments as clone pairs with the original or form a clone class for them. Text-based techniques are sensitive to format alternations and thus, may not detect scenario *S1(c)*. The technique proposed in Johnson et al. [12,13] cannot detect *S1(c)*, because it is a line-based comparison approach. On the other hand, SDD [16] may detect *S1(c)* depending on the threshold in matching the lines.

Those token-based techniques, which ignores formatting and comments can easily detect these exact matches. CPD [34] ignores formatting but does not ignore comments, so it fails in detecting *S1(b)*. CCFinder [4], clones [28], CP-Miner [11] ignore formatting and comments. So, these Perform well in detecting all the scenarios described in Figure 5.1. The performance of the approaches is shown in Table 5.1.

Table 5.1: Evaluation of the tools on clone type-1

| **Citation** | **S1(a)** | **S1(b)** | **S1(c)** |
|---|---|---|---|
| Johnson | Perform well | Perform well | Cannot detect |
| SDD | Perform well | Depends on the threshold | Depends on the threshold |
| CCFinder | Perform well | Perform well | Perform well |
| CPD | Perform well | Cannot detect | Perform well |
| clones | Perform well | Perform well | Perform well |
| CP-Miner | Perform well | Perform well | Perform well |
| CloneDigger | Perform well | Perform well | Perform well |
| CloneDr | Perform well | Perform well | Perform well |
| **Best Rater** | **SDD** | **SDD** | **CloneDr** |

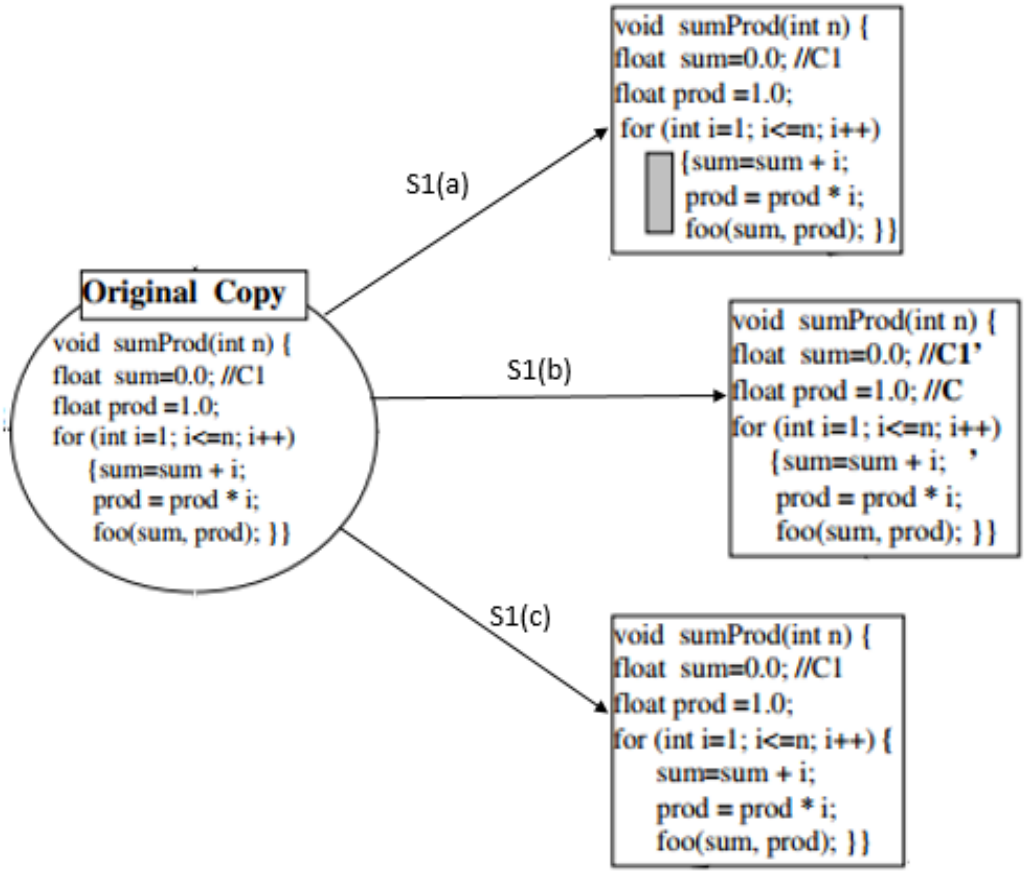



Figure 5.1: Clone type-1 scenarios.

Tree-based techniques ignore formatting differences and comments and should detect these scenarios very well if they look for exact subtrees without ignoring tree-leaves (in most cases they ignore leaves). However, some tree-based techniques use alternative representations of the parse-tree/AST (for example Deckard works on characteristic vectors of the parse-tree) and may not detect them accurately. Moreover, a recent study [29] shows that an AST- based exact matching function clone detection technique can even miss some exact function clones detected by a text-line based technique. Although CloneDigger [11] and CloneDr [7] can detect all the scenarios of type-1 code clones.

SDD [16] is the best rater for *S1(a)*, *S1(b)* although all the tools except CPD [34] perform well in these scenarios. However we rate SDD because it is a text based approach, and text based approaches take a little time in parsing the source code and in textual matching phase. Cloner, CloneDigger has best performance in detecting *S1(c),* but CloneDr is rated as best because of its robust tree matching algorithm.

### 5.2.2 Type 2 (change in identifiers)

Roy et al. [1] made four more copies of the function, using a systematic renaming of identifiers and literals in the first fragment (*S2(a)*), renaming the identifiers (but not necessarily systematically) in the second fragment (*S2(b)*), renaming data types and literal values (but not necessarily systematically) in the third fragment (*S2(c)*), and replacing some parameters with expressions in the fourth fragment (*S2(d)*) (*Figure 5.2*).

Again, an ideal clone detection technique should detect all four modified fragments as clone pairs with the original function or should form a clone class for those. Dup is the robust in detecting scenario *S2(a)* because of its novel use of parameterized suffix-trees.

Table 5.2: Evaluation of the tools on clone type-2

| **Citation** | **S2(a)** | **S2(b)** | **S2(c)** | **S2(d)** |
|---|---|---|---|---|
| Johnson | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| SDD | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| CCFinder | Perform well | Perform well | Perform well | Cannot detect |
| CPD | Perform well | Perform well | Perform well | Cannot detect |
| clones | Perform well | Perform well | Perform well | Cannot detect |
| CP-Miner | Perform well | Perform well | Perform well | Depends on the threshold |
| CloneDigger | Perform well | Perform well | Perform well | Perform well |
| CloneDr | Perform well | Perform well | Perform well | Depends on the threshold |
| **Best Rater** | **CloneDr** | **CloneDr** | **CloneDr** | **CloneDigger** |

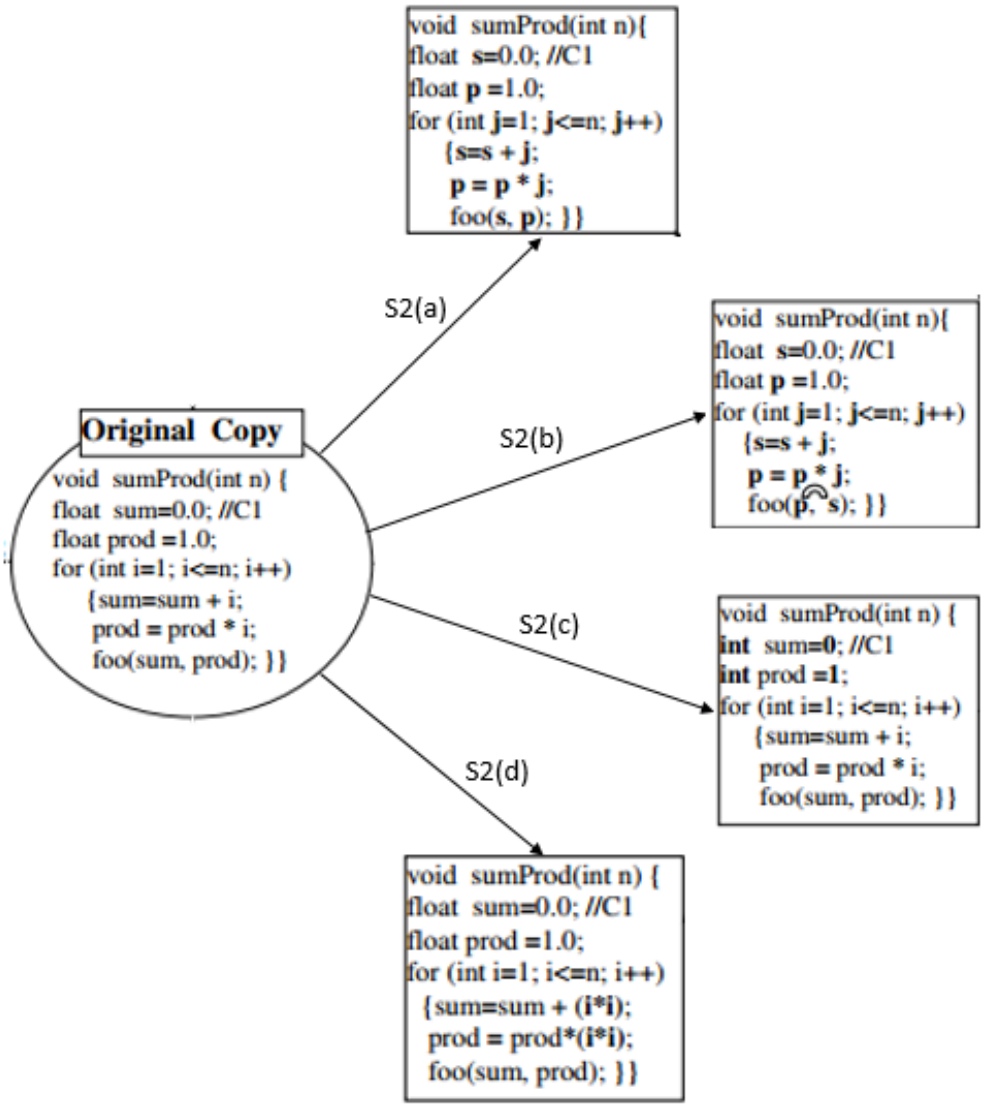



Figure 5.2: Clone type-2 scenarios.

None of the text-based techniques is likely to do well with these scenarios since those normally compare program text without normalization and are therefore fragile to identifier renaming. Token-based techniques can detect scenarios *S2(a)*, *S2(b)* and *S2(c)* well, but are likely to also have many false positives due to their identifier normalizations and transformations. Those generally failed to detect *S2(d),* as it is not really a change in identifiers. However CP-Miner can detect this depending on the threshold in token matching.

Tree-based techniques may also detect scenarios *S2(a)*, *S2(b)* and *S2(c)* well as these techniques normally ignore identifiers and literals in comparison. However CloneDr is the best rater for these

scenarios for its robust sub-tree matching algorithm. On the other hand, for scenario *S2(d)*, tree-based CloneDigger seems to be well suited, as it can apply structural abstraction on arbitrary subtrees. CloneDr also can detect *S2(d)*, but it depends on the threshold. So CloneDigger is the best rater for this scenario.

### 5.2.3 Type 3 (add or delete lines)

Roy et al. [1] made five more copies of the function and this time making small insertions within a line in the first fragment (*S3(a)*), small deletions within a line in the second fragment (*S3(b)*), insertion of some new lines in the third fragment (*S3(c)*), deletion of some lines from the fourth fragment (*S3(d)*), and making changes to some whole lines in the fifth fragment (*S2(e)*) (*Figure 5.3*).

Text-based tools cannot detect this type because those cannot use any normalized or transformed text in the matching phase, and cannot cope with the additional lines. Token-based CP-Miner is likely to work well with these scenarios. CP-Miner uses a frequent subsequence data mining algorithm which allows it to tolerate gaps in cloned segments. The token-based SDD can also identify such scenarios using scatter plot visualization but when threshold is little bit lower. Table 3 shows the performances of the tools on the scenarios of type-3.

Table 5.3: Evaluation of the tools on clone type-3

| **Citation** | **S3(a)** | **S3(b)** | **S3(c)** | **S3(d)** | **S3(e)** |
|---|---|---|---|---|---|
| Johnson | Cannot detect | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| SDD | Perform well | Perform well | Depends on the threshold | Depends on the threshold | Depends on the threshold |
| CCFinder | Depends on the threshold | Depends on the threshold | Cannot detect | Cannot detect | Cannot detect |
| CPD | Cannot detect | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| Clones | Cannot detect | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| CP-Miner | Perform well | Perform well | Perform well | Perform well | Perform well |
| CloneDigger | Perform well | Perform well | Cannot detect | Cannot detect | Cannot detect |

| | | | | | |
|---|---|---|---|---|---|
| CloneDr | Perform well | Perform well | Depends on the threshold | Perform well | Perform well |
| **Best Rater** | **SDD** | **SDD** | **CP-miner** | **CloneDr** | **CloneDr** |

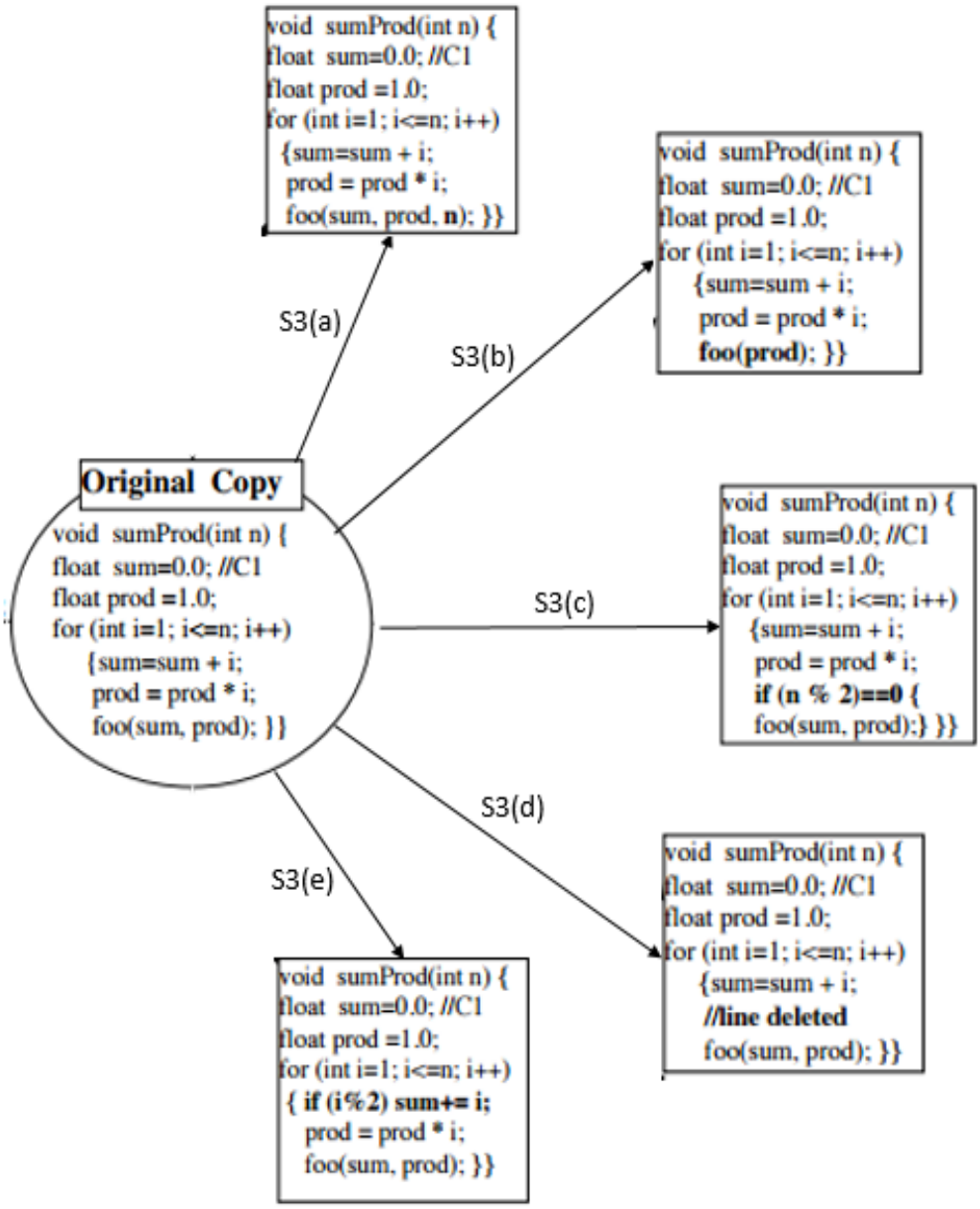



Figure 5.3: Clone type-3 scenarios.

In scenarios *S3(a)* and *S3(b),* SDD, CP-Miner, CloneDigger and CloneDr perform well and we rate SDD for its little execution time. CP-Miner rated best for *S3(c)* as no other tools performs well in this scenario. CP-Miner and CloneDr have well performance in *S3(d)* and *S3(e).* However CloneDr is selected as best rater as tree based techniques have better execution time than token based techniques.

### 5.2.4 Type 4 (reorder data dependent and control statements)

Roy et al. [1] made four more copies of the function and this time reordered the declaration statement in the first fragment (*S4(a)*), reordered data independent statements in the second (*S4(b)*), reorders data dependent statements in the third (*S4(c)*), and replaced a control statement with different one in the fourth (*S4(d)*) (*Figure 5.4*).

Again, it is expected that an ideal clone detection technique should be robust enough to detect such modified code fragments as clone pairs with the original or form a clone class for those. None of the used tools works well in this scenarios. CP-Miner performs well in *S4(a)*, *S4(b)* and *s4(c)*, but only when there is a very low threshold in matching tokens. It appears that only PDG-based techniques are likely to work well with scenarios *S4(a)* and *S4(b)*. PDG-based techniques use data and control flow information, which remains unchanged across reordering of declarations and data independent statements. Reordering of data dependent statements may change data and control flow however, so those may not Perform well with scenario *S4(c)*. To detect scenario *S4(d)*, exhaustive source transformation may be necessary.

Table 5.4: Evaluation of the tools on clone type-4

| **Citation** | **S4(a)** | **S4(b)** | **S4(c)** | **S4(d)** |
|---|---|---|---|---|
| Johnson | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| SDD | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| CCFinder | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| CPD | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| clones | Cannot detect | Cannot detect | Cannot detect | Cannot detect |

| | | | | |
|---|---|---|---|---|
| CP-Miner | Depends on the threshold | Depends on the threshold | Depends on the threshold | Cannot detect |
| CloneDigger | Cannot detect | Cannot detect | Cannot detect | Cannot detect |
| CloneDr | Depends on the threshold | Depends on the threshold | Cannot detect | Cannot detect |
| **Best Rater** | **CloneDr** | **CloneDr** | **CP-Miner** | **None** |

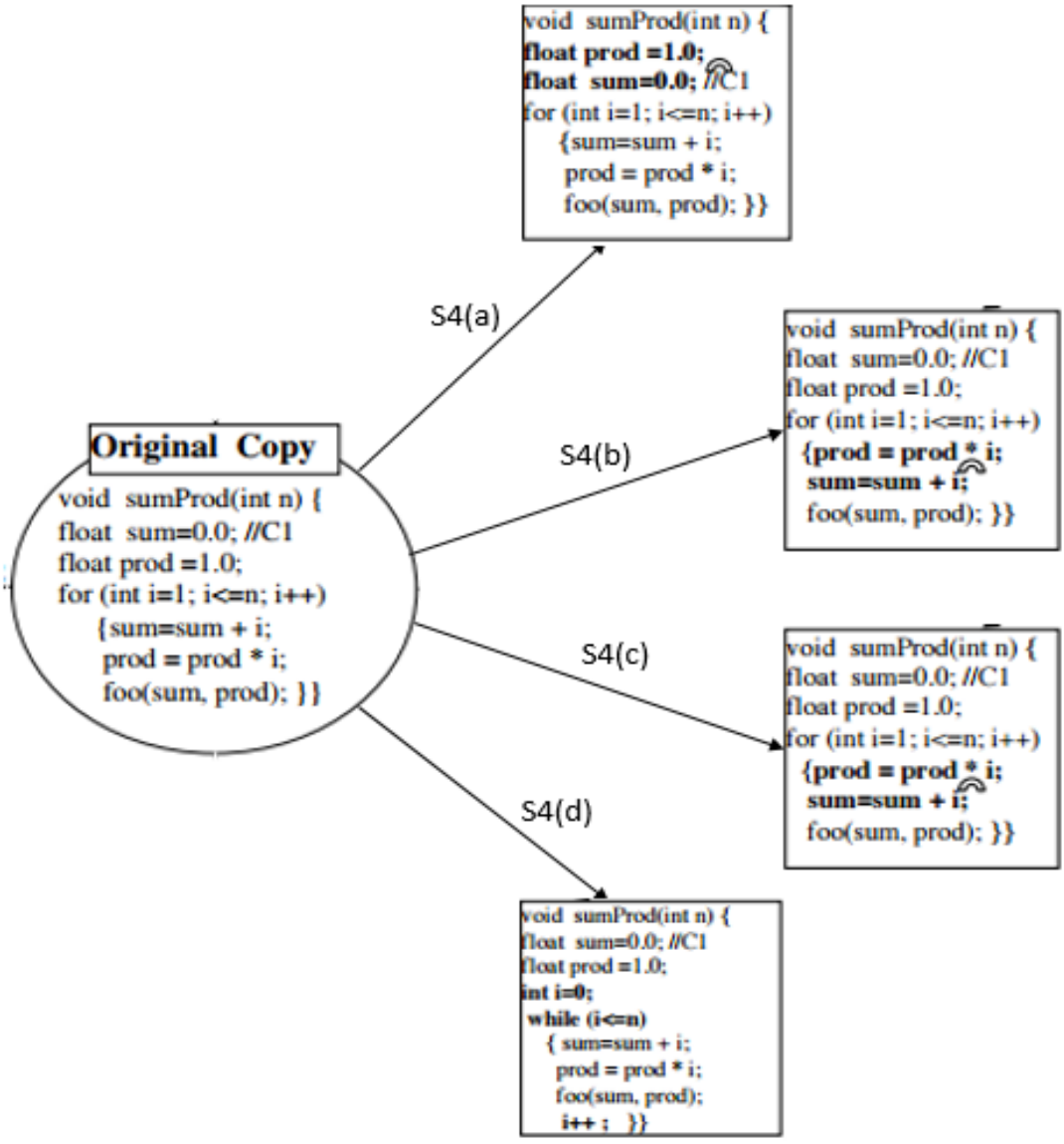



Figure 5.4: Clone type-4 scenarios.

CloneDr and CP-Miner can detect *S4(a), S4(b)* depending on the threshold. Others have very poor performance on detecting this scenarios. However CloneDr is selected as best rater as it is a tree based technique. On the other hand, only CP-Miner can detect the scenario *S4(c).* So it is the best rater for this scenario. No tools can detect the S4(d) with or without threshold.

Table 5.5 estimates the scenario coverage of the technique in points out of 16 and a percentage of scenarios potentially detected, counting (low) and above as potential detection using the matrices described in the start of the section result analysis.

Table 5.5: Coverage on the scenarios.

| **Citation** | Points (out of 16) | Coverage (in percentage) |
|---|---|---|
| Johnson | 2 | 13.50% |
| SDD | 5.5 | 34.38% |
| CCFinder | 7 | 43.75% |
| CPD | 5 | 31.25% |
| clones | 6 | 37.50% |
| CP-Miner | 13 | 81.25% |
| CloneDigger | 9 | 56.25% |
| CloneDr | 12 | 75.00% |

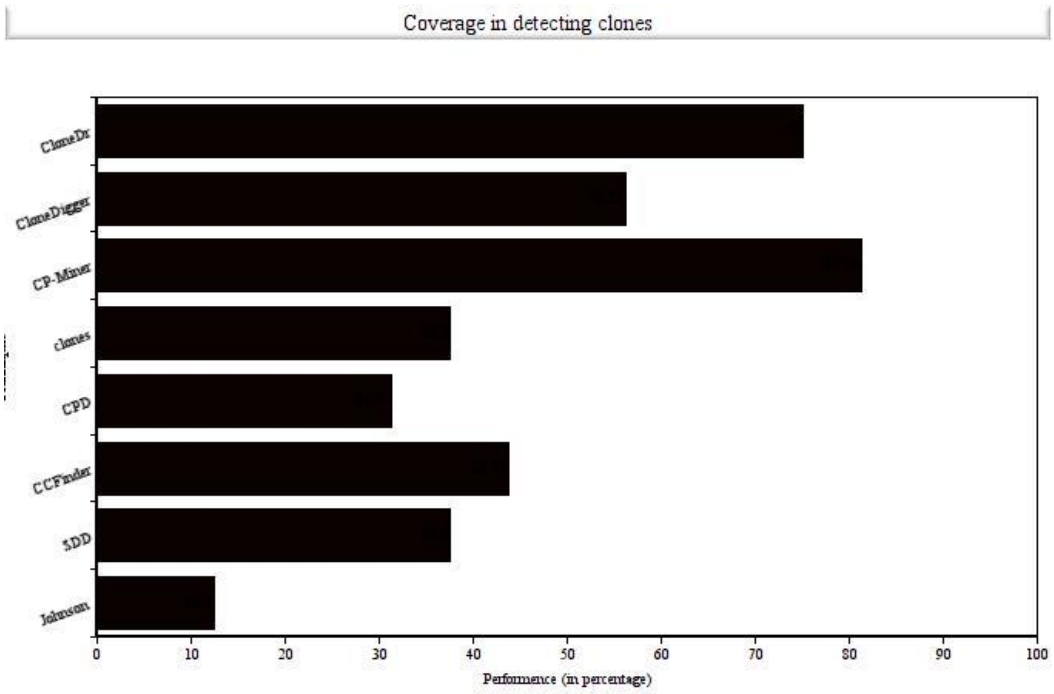



Figure 5.5: Coverage of the techniques.

## 5.3 SUMMARY

This chapter intends to demonstrate the implementation environment and result analysis of the used techniques for this evaluation. The implemented approaches are implemented in Java programming language. The implementation results are analyzed and evaluated with the other five free tools of clone detection. The analysis shows the performance of different techniques in different scenarios of various types of code clones described in [1]. The evaluation will help someone in using the tools in different aspects.

# 6 CONCLUSION

This evaluations are not only intended for experts in clone detection, but also intended for potential new users and builders of clone detection-based tools and applications. It is hoped that it may also assist in identifying remaining open research questions, avenues for future research, and interesting combinations of techniques. The evaluation results of this study are based on estimating the performance of techniques using the most lenient values of all tunable scenarios, and thus the findings differ from the results of empirical studies such as Bellon et al. [2].

## 6.1 DISCUSSION

As a demonstration of how this evaluation can be helpful, example combination of different techniques/tools is provided to handle all of the scenarios used in this paper. Of course, many other combinations can be derived based on user requirements, both in terms of different scenarios and the techniques used. Such a combination might help one to understand how to design a hybrid method to be robust across all types of clones or how to employ a set of different tools to achieve a better result. The last row of Table 1, 2, 3, 4 list the best rated techniques for each of the scenarios. Tempering with the properties of the techniques anyone can select a best choice for each scenario.

For scenarios S1(a), S1(b), S3(a), S3(b) the text based SDD [16] seems best, being very good for S3(a), S3(b) and good for S1(a) and S1(b) while ensuring linear time and space complexity. For scenarios S1(c), S2(b) and S2(c), cloneDr [7] is chosen because it finds syntactic clones in linear time and space. For scenario S2(d), CloneDigger [11] is chosen because it gets a good rating for these scenario and, like cloneDr, it is AST-based, making it a promising choice for a hybrid. For scenarios S3(c), S3(d) and S3(e), CP-Miner seems a good choice. For scenarios S4(a), S4(b) and S4(d), CP-Miner [] might be a good choice, as it can detect such scenarios based on the threshold and it seems to be faster than PDG-based techniques. Thus, the obtained combination is {SDD, CP-Miner, CloneDr, CloneDigger). Several other combinations can easily be obtained based on the results provided in this paper.

## 6.2 FUTURE WORK

Only eight state of art techniques are evaluated in this comparison study and a limited amount of scenarios are used. If more techniques are evaluated with more scenarios then the result will be more interesting. The results of this study may assist new potential users of clone detection techniques in understanding the range of available techniques and tools and selecting those most appropriate for their needs. It may also assist in identifying remaining open research questions, avenues for future research, and interesting combinations of techniques.

# 7 REFERENCES

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, Comparison and Evaluation of Clone Detection Tools, Transactions on Software Engineering, 2007, pp. 33(9):577-591.

[2] M. Gabel, L. Jiang and Z. Su, Scalable Detection of Semantic Clones, in: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 321-330.

[3] B. Baker, A Program for Identifying Duplicated Code, in: Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, 1992, Vol. 24:4957, 24:49-57.

[4] T. Kamiya, S. Kusumoto and K. Inoue, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering, 2002, 28(7):654-670.

[5] V. Wahler, D. Seipel, J. Gudenberg and G. Fischer, Clone Detection in Source Code by Frequent Itemset Techniques, in: Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation, SCAM 2004, pp. 128-135.

[6] W. Yang, Identifying Syntactic Differences between Two Programs, Software Practice and Experience, 1991, 21(7):739-755.

[7] I. Baxter, A. Yahin, L. Moura and M. Anna, Clone Detection Using Abstract Syntax Trees, in: Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, pp. 368-377.

[8] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, Pattern Matching for Clone and Concept Detection, Journal of Automated Software Engineering, 1996, 3(1-2):77-108.

[9] J. Mayrand, C. Leblanc and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, in: Proceedings of the 12th International Conference on Software Maintenance, ICSM 1996, pp. 244-253.

[10] J. Krinke, Identifying Similar Code with Program Dependence Graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, WCRE 2001, pp. 301-309.

[11] R. Komondoor and S. Horwitz, Using Slicing to Identify Duplication in Source Code, in: Proceedings of the 8th International Symposium on Static Analysis, SAS 2001, pp. 40-56.

[12] J. Johnson, Identifying Redundancy in Source Code Using Fingerprints, in: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1993, pp. 171–183.

[13] J. Johnson, Visualizing Textual Redundancy in Legacy Source, in: Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative research, CASCON 2004, pp. 171-183.

[14] S. Ducasse, M. Rieger and S. Demeyer, A Language Independent Approach for Detecting Duplicated Code, in: Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, pp. 109-118.

[15] R. Wettel and R. Marinescu, Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments, in: Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2005, 8pp.

[16] S. Lee and I. Jeong, SDD: High performance Code Clone Detection System for Large Scale Source Code, in: Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA Companion 2005, pp. 140–141.

[17] C.K. Roy and J.R. Cordy, NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible PrettyPrinting and Code Normalization, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, pp. 172-181.

[18] A. Marcus and J. Maletic, Identification of High-level Concept Clones in Source Code, in: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE 2001, pp. 107-114.

[19] B. Baker and R. Giancarlo, Sparse Dynamic Programming for Longest Common Subsequence from Fragments, Journal Algorithms, 2002, Vol. 42 (2):231-254.

[20] H. Basit, S. Pugliesi, W. Smyth, A. Turpin and S.Jarzabek, Efficient Token Based Clone Detection with Flexible Tokenization, in: Proceedings of the 6th European Software Engineering Conference and Foundations of Software Engineering, ESEC/FSE 2007, pp. 513-515.

[21] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, On Detection of Gapped Code Clones Using Gap Locations, in: Proceedings 9th Asia-Pacific Software Engineering Conference, APSEC 2002, pp. 327–336.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, IEEE Transactions on Software Engineering, 32(3):176-192.

[23] J.R. Cordy, T.R. Dean and N. Synytskyy, Practical Language-Independent Detection of Near-Miss Clones, in: Proceedings of the 14th IBM Centre for Advanced Studies Conference, CASCON 2004, pp. 29-40.

[24] Project Bauhaus. URL http://www.bauhaus-stuttgart.de Last accessed 31 August 2016.

[25] R. Koschke, R. Falke and P. Frenzel, Clone Detection Using Abstract Syntax Suffix Trees, in: Proceedings of the 13th Working Conference on Reverse Engineering, WCRE 2006, pp. 253-262.

[26] R. Tairas and J. Gray, Phoenix-Based Clone Detection Using Suffix Trees, in: Proceedings of the 44th Annual Southeast Regional Conference, ACM-SE 2006, pp. 679-684.

[27] L. Jiang, G. Misherghi, Z. Su and S. Glondu, DECKARD: Scalable and Accurate Tree-based Detection of Code Clones, in: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 96-105.

[28] E. Burd, J. Bailey, Evaluating Clone Detection Tools for Use during Preventative Maintenance, in: Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2002, pp. 36-43.

[29] B. Baker, Finding Clones with Dup: Analysis of an Experiment, IEEE Transactions on Software Engineering, 2007, 33(9):608-621.

[30] F. Rysselberghe and S. Demeyer, Evaluating Clone Detection Techniques, in: Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications, ELISA 2003, 12pp.

[31] F. Rysselberghe and S. Demeyer, Evaluating Clone Detection Techniques from a Refactoring Perspective, in: Proceedings of the 9th IEEE International Conference Automated Software Engineering, ASE 2004, pp. 336- 339.

[32] M. Bruntink, A. Deursen, R. Engelen and T. Tourwe, On the Use of Clone Detection for Identifying Crosscutting Concern Code, Transactions on Software Engineering, 31(10):804-818.

[33] Tool Clone Detective (part of ConQAT). URL http://conqat.in.tum.de/index.php/Main Page Last accessed 30 October 2016.

[34] PMD's CPD. URL http://pmd.sourceforge.net/cpd.html Last accessed October 2016.

[35] P. Bulychev and M. Minea, Duplicate Code Detection Using Anti-Unification, in: Spring Young Researchers Colloquium on Software Engineering, SYRCoSE 2008, 4 pp.