ISSN: 2321-7782 (Online)
Volume 3, Issue 1, January 2015
International Journal of Advance Research in Computer Science and Management Studies
Research Article / Survey Paper / Case Study
Available online at: www.ijarcsms.com

A Comparative Study of Clone Detection Tools

Shahid Ahmad Wani[1]
Research Scholar,
Dept. of MMICTBM
Maharishi Markandeshwar University Mullana
Amballa, Haryana, India

Shilpa Dang[2]
Assistant Professor,
Dept. of MMICTBM
Maharishi Markandeshwar University Mullana
Amballa, Haryana, India

Abstract: ***Code reuse is a common activity in software development and is one of the main reasons for code clones. A code clone is a part of the source code that is identical, or highly similar, to another part (clone) in terms of structure and semantics. Various clone detection techniques and tools have been proposed over last few years. Code cloning is found to be a more somber and serious problem in industrial software systems. A large number of clone detection tools are available and in order to make use of the right tool for detection of clones very important. The aim of this study is to analyze various clone detection tools. This study would help to decide which tool is best suitable for detection of code clones. We present the background concepts of cloning, a generic clone detection process and a comparison of four clone detection tools.***

Keywords: Clone detection, Code clone, Code Fragment, Dynamic pattern matching (DPM).

I. INTRODUCTION

Clone detection is an active research area since 1980's and was used to find plagiarism in assignments of students. In 1990's focus shifted towards finding code clone in software system development. Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development industry. Because of this software systems often contain sections of code that are very similar, called code clones. In software programs, we can find different kinds of replication or redundancy. Usually, this kind of replication in the code is called clone, different definitions and taxonomy of code clones have been proposed in the literature [1]. The term "clone" also means "duplicate code", it is an example of bad smell, as defined by Fowler [2]. Clones are often the outcome of copy-paste actions. Such actions are very easy and can considerably reduce programming effort and time as they reuse on hand piece of code rather than rewriting related code from scratch. This practice is normal, especially in device drivers of operating systems where the algorithms are similar [3]. There are numerous other factors such as performance improvement and coding style because of which large software systems may contain a significant proportion of duplicated code.

Code cloning is found to be a more somber and serious problem in industrial software systems [4, 5, and 6]. In presence of clones, the normal operation of the system may not be affected, but if maintenance teams do not take measures to counter the problem, further development may become prohibitively expensive. Clones are supposed to have a negative impact on advancement and evolution [7]. Code clones may harmfully affect the system quality, especially their maintainability and comprehensibility [8, 9]. Moreover, a great deal of cloning increases the system size and often indicate design problems such as missing or omitted inheritance or missing procedural abstraction. Considering the huge amount of duplicated and redundant code and its maintenance cost of large software systems, it is therefore, essential to detect code clones of large software systems for performing the respective maintenance tasks (e.g., refactoring). Fortunately, there are vast research studies to find clones automatically. However, attempts are being undertaken to identify clones [6, 10] and once identified, they can be removed by source code refactoring.

The rest of this paper is organized as follows. After introducing some clone terminology terms in Section 2, we provide a general overview of the clone detection process in Section 3. We present the comparison of clone detection tools in Section 4. In section 5 the conclusion of the paper is presented.

II. CLONE DETECTION TERMINOLOGY

We begin with a basic introduction to clone detection terminology.

A. *Definition 1: Code Fragment.* Any sequence of code lines is a code fragment (CF). It can be of any granularity, e.g., begin-end block, sequence of statements, or function definition. A CF is identified by its file name and begin-end line numbers in the original code base and is denoted as a triple (CF.FileName, CF.BeginLine, CF.EndLine).

B. *Definition 2: Code Clone.* A code fragment CF2 is a clone of another code fragment CF1 if they are similar by some given description of similarity, that is, f (CF1) = f (CF2) where f is the similarity function. Two code fragments that are similar to each other form a clone pair (CF1; CF2), and when many code fragments are parallel or similar, they form a clone class or clone group.

C. *Definition 3: Clone Types.* There are two main kinds of similarity between code fragments. Code fragments can be similar based on their functionality similarity, or they can be similar based of their program text. In the following we provide the types of clones based on both the textual (Types 1 to 3) [11] and functional (Type 4) [12] similarities:

» **Type-1:** Identical code fragments except small variations in white space, layout, and comments.

» **Type-2:** Syntactically identical code fragments except for variations in literals, identifiers, types, layout, comments and whitespaces.

» **Type-3:** These are copied fragments with additional modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, layout, comments and whitespaces.

» **Type-4:** Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

III. CLONE DETECTION PROCESS

A clone detector should try to find pieces of code of high similarity in a software system's source text or code. The main problem here is that it is not known in advance which code fragments may be repeated. Thus the detector really should compare each possible fragment with every other possible code fragment. In this section, an overall summary of the basic steps involved in a clone detection process is provided.

The set of steps that a typical clone detector may follow in general (although not necessarily) are shown in figure 1. A short description of each of the phases is provided in the following subsections.

A. Preprocessing

In this phase of clone detection process the source code is partitioned and comparison domain is determined. The three main purpose of this phase are removing uninteresting parts, determining the source units and determining the comparison units. All the source code that is not of any interest to the comparison phase is filtered out. The remaining source code portioned into a set of disjoint fragments called source units. These units are involved in direct clone relations to each other. Depending on the comparison technique used source units may need to be further partitioned into smaller units.

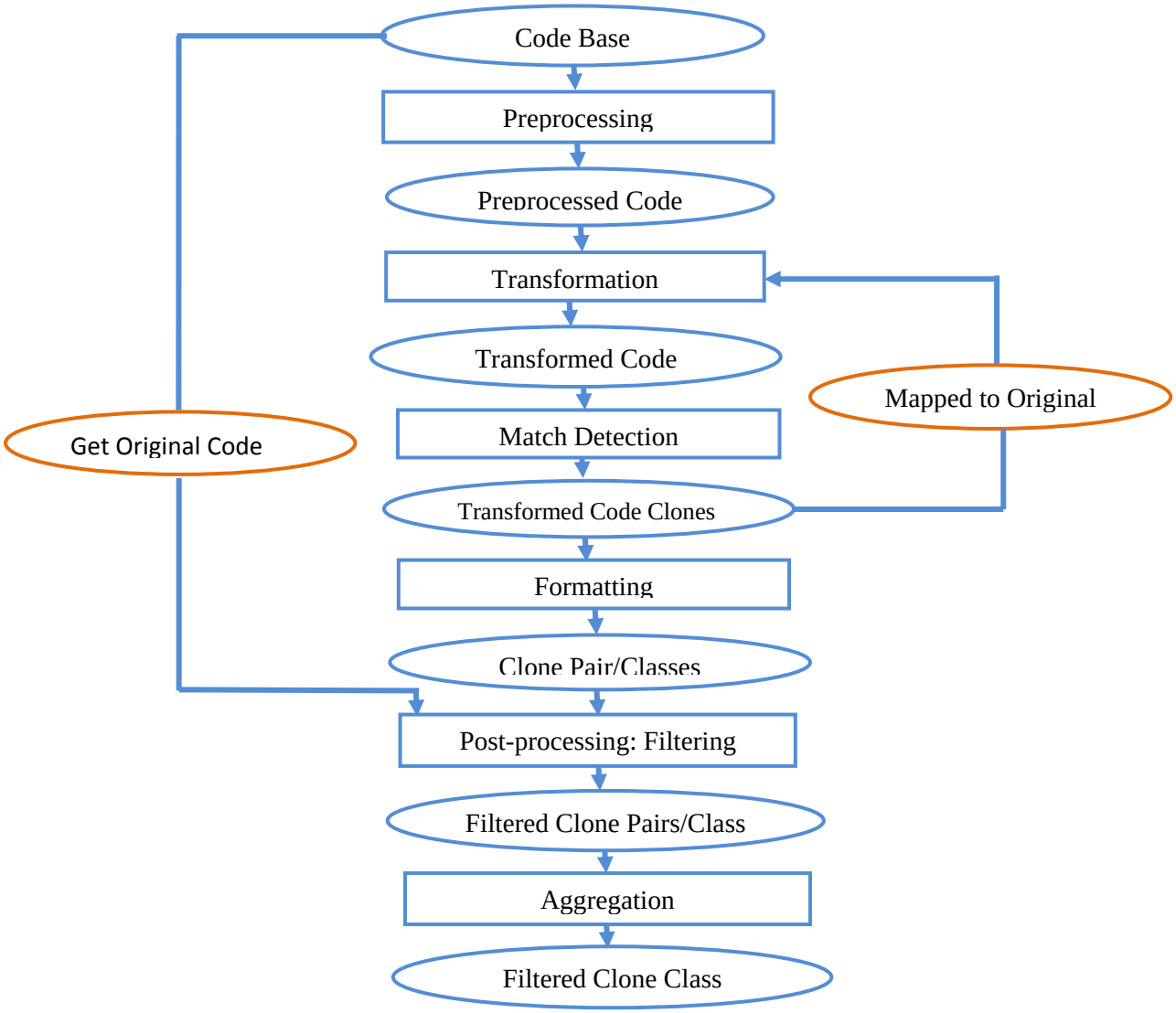

Fig. 1. A generic view of Clone Detection Process

B. Transformation

Once the units of comparison are determined, the source code of the comparison units is transformed to an appropriate intermediate representation for comparison. This transformation is called as extraction by reverse engineering community. Few transformation techniques are extraction, tokenization, parsing, generating PDG, removal of comments, removal of white spaces, normalizing identifiers, and pretty printing of source code.

C. Match Detection

The transformed code is then given input to a comparison unit where it is compared with each other to locate matches. Often larger units are formed by joining adjacent similar comparison units. 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 code clone pair is usually represented as the source coordinates of each of the corresponding or matched fragments in the transformed code. Other popular matching algorithms in addition to simple normalized text comparison used in clone detection include dynamic pattern matching (DPM) [15], suffix-trees [13, 14] and hash value comparison [5, 6].

D. Formatting

In this phase, the conversion of the clone pair list obtained with respect to the transformed code to a clone pair list obtained with respect to the original code base is performed. After finding the location of the clone pair from the previous phase, it is then converted into line numbers on the original source files.

E. Post-processing / Filtering

In this phase, code clones are filtered or ranked using manual analysis or automated heuristics. In manual analysis of clones is done where human experts filter out false positive clones or spurious clones. Suitable format visualization of cloned source can help speed up this manual filtering. While automated heuristics can be defined often based on diversity, frequency, length, or other characteristics of clones in order to rank or filter out clone candidates automatically [13].

F. Aggregation

Some clone detection tools directly identify clone classes and most of tools return only clone pairs as the result. The clone pairs should be aggregated to classes, clone groups, or clusters etc and there is no need to reduce amount of data.

IV. COMPARISON OF CLONE DETECTION TOOLS

There are many clone detection techniques and their corresponding tools, and therefore, a comparison of these tools is worth in order to pick the right tool for a particular purpose of interest. In this we compare four clone detection tools. We compare Bauhaus an abstract syntax tree based tool, two token based tools CP-Miner and CCFinderX, and PMD a string matching based tool. In Table 1 we show the comparison of these tools based on various parameters. The parameters with which the tools can be compared are known as clone detection challenges. Some of the parameters used for comparing the different tools/techniques are listed below:

- **Portability:** The tool should be portable in terms of multiple dialects and languages. A clone detection tool must be portable and easily configurable for different types of dialects and languages to tackle the syntactic variations of those languages.
- **Precision:** The tool should be good enough so that it detect less number of false positives i.e., the tool should find duplicated code with higher precision.
- **Recall:** The tool should be capable of locating and finding most (or even all) of the clones of a system of interest.
- **Scalability:** As duplication is the most problematic in complex systems, the tool should be able to find clones from large code bases with efficient use of memory.
- **Robustness:** A good tool should be robust in terms of the different editing activities that might be applied on the copied fragment so that it can detect different types of clones with higher precision and recall.

TABLE 1

Comparison of Clone Detection Tools

Tools	Bauhaus	CP-Miner	PMD	CCFinderX
Techniques	AST	Token-Based	String Matching	Token-Based
Functionality	Looks for portions of identical code, variations in variable names, identifiers and portions of identical code with added or removed statements.	Detect copy pasted code segments. Finding copy pasted bugs.	Looks for potential problems like duplicate code, possible bugs, and dead code.	Code clone detector Analysis in metrics of code clones

Strengths	Detect bigger clones Detect Type 1 and Type 2 clones	Can detect modification in copy pasted statements Fast detection of clones	Finds occasionally real defects Finds bad practices Finds larger number of clones	Effective use of multi core CPU Fast detection of clones
Weakness	Detects lesser clones from a software system	Incorrectly matching copy pasted code segments	Slow duplicate code detector	Find only fewer clones
Supporting Language	C, C++, C#, Java, Ada	C, C++, Java	Java, C, C++, JSP, PHP, Ruby	C/C++, Java, COBOL, VB, C#
Detect Clone Types	Type 1, Type 2, and Type 3 clones	Type 1 and Type2 clones	Type 1 and Type 2 clones	Type 1 and Type 2 Clones
Portability	Portable with many languages and dialects	Limited portability	Can work with many programming languages	Available for windows and Unix platforms and can support many languages
Scalability	Can handle medium size software systems	Highly Scalable	Finds duplicate code in large code bases	Limited scalability
Robustness	Finds three types of clones therefore this tool is highly robust	Limited Robustness	This tool is not robust in finding all types of clones	Finds only two types of clones, thus is not very much robust

In table 1, all the tools exploit different detection technique and therefore provide different results. Though all the tools have almost same functionality but they differ in various other attributes. From table 1 we find that only the AST-based tool Bauhaus can identify Type 3 clone while as the rest of the tools can only find Type 1 and Type 2 clones. Bauhaus bigger but less number of clones while as PMD can find small but larger number of clones. CP-Miner and CCFinderX are fast in clone detection. All the tools support various programming languages presently used in software industry, thus all are at the same level in terms of portability. Bellon et al. conducted an experiment using six clone detection tools and found that tools behave complementary in terms of precision and recall [11]. A complete comparison of the tools is presented in table 1. From the table it can be concluded that all the tools have their possible advantages and disadvantages and no tool is dominating the other.

V. Conclusion

Clone detection is an active research area and work has been carried out on a larger scale in detection and removal of clones from software. In this paper, we have focused on clone detection tools; a brief but complete survey is presented. Although an independent comparison study is not available for the recent tools, it is commonly agreed that no tool excel others. All approaches have their distinct advantages and drawbacks and further improvement or more hybrid approach is required for overcoming the limitations of the tools while preserving the strengths. We hope that this study may help people associated with clone detection in understanding the range of available tools and selection of tools most appropriate for their needs.

References

1. C. K. Roy and J. R. Cordy, “A survey on software clone detection research”, Tech. Rep. 2007-541, pp.115, School of Computing, Queen’s University, Kingston, Ontario, Canada, 2007.
2. M. Fowler, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley Longman, Boston, Ma, USA, 1999.
3. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, “CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code”, In IEEE Transactions on Software Engineering, Vol. 32(3): pp. 176-192, March 2006.
4. Brenda Baker, “On Finding Duplication and Near-Duplication in Large Software Systems”, In Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95), pp. 86-95, Toronto, Ontario, Canada, July 1995.
5. Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, “Clone Detection Using Abstract Syntax Trees”, In Proceedings of the 14th International Conference on Software Maintenance (ICSM'98), pp. 368-377, Bethesda, Maryland, November 1998.
6. Jean Mayrand, Claude Leblanc, Ettore 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'96), pp. 244-253, Monterey, CA, USA, November 1996.

7. Reto Geiger, Beat Fluri, Harald C. Gall and Martin Pinzger, “Relation of Code Clones and Change Couplings”, In Proceedings of the 9th International Conference of Fundamental Approaches to Software Engineering (FASE'06), pp. 411-425, Vienna, Austria, March 2006

8. Simon Giesecke, “Clone based Reengineering for Java on Eclipse Platform”, Masters Thesis, Carl von Ossietzky UniversitÄat Oldenburg, Germany, September 2003.

9. Simon Giesecke, “Generic modelling of code clones”, In Proceedings of Duplication, Redundancy, and Similarity in Software, ISSN 16824405, Dagstuhl, Germany, July 2006.

10. Jens Krinke, “Identifying Similar Code with Program Dependence Graphs”, In Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), pp. 301-309, Stuttgart, Germany, October 2001

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

12. 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 (2008

13. 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, 28(7):654-670 (2002).

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

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

