ISSN 2320-6802

INTERNATIONAL JOURNAL FOR ADVANCE RESEARCH IN ENGINEERING AND TECHNOLOGY

WINGS TO YOUR THOUGHTS.....

Comparative Analysis of Tree-Based and Text-Based Technique for Code Clone Detection

Manisha Gahlot
A.P. (CSE)
PDM College of Engg.
Bahadurgarh

Deepak Sethi
A.P. (CSE)
FET, Mody University of Science and Technology
Laxmangarh

Abstract: *Many studies have been conducted on the evolution of code clones. However, after a decade of active research, there has been a lack of progress in understanding the evolution of near-miss software clones, where code statements have been added, deleted, or modified in the copied fragments in a program. Given that there are a significant amount of near-miss clones in the software systems, we believe that without studying the evolution of near-miss clones, one cannot have a complete picture of the clone evolution. In this paper, we have advance the art in the evolution of clone research in the context of both exact and near-miss software clones. Software clones are considered harmful in software maintenance and evolution. In this paper, we advance the art in clone detection and analysis in several ways. First, we use clone detection methods, called Solid SDD and CloneDR that can detect clones. Solid SDD is a text-based approach and CloneDR is a tree based approach. We compare these two clone detection tools on the basis of duplicates found, execution time and clone sets etc.*

1. INTRODUCTION

A *clone* is a program fragment that identical to another fragment. A *near miss clone* is a fragment, which is nearly identical to another. Clones usually occurs when an idiom is copied and optionally edited, producing exact or near-miss clones. Copying and pasting source code results in multiple, identical code fragments (code clones) throughout a system, which may then be individually modified to fit their specific tasks. Though the contents of these clones may vary, some amount of similarity must be maintained between them; otherwise the programmer would not have made an exact duplicate as a base to work from. The correspondence between a pair of clones can be a useful piece of information when programmers modify or debug source code [1]. This can be helpful especially when programmers are working with someone else source code or if it has been a while since they have worked on their own. Maintaining the cloning relationship is thus a very important consequence of copying and pasting, and a responsibility left to the software maintainer. Without tracking clones over time, identifying and consistently changing clones can be problematic [2, 3]. Due to the increase in source code maintenance as a result of cloning, there has been a variety of research with the aim to manage clones. One large area of clone management research focuses on clone detection and removal [4, 5]. In cases where it may not be easy or possible to remove clones, other research proposes ways make consistent changes between clones [6,7]. A variety of research looks into the topic of clone-related bugs and inconsistent changes to clones [2, 8, 9, 10, 11, and 12]. Still other research recognizes that a common type of clone is a parameterized one [13, 14], where the programmer intends to make only small changes to the pasted code, like to the identifier names and constants. Copying a code fragment and reusing it by pasting with or without minor modifications is a common practice in software development, and as a result software systems often have sections of code that are similar, called software clones or code clones.

Drawbacks of code duplication:-

i) Increased probability of bug propagation
ii) Increased probability of introducing a new bug
iii) Increased probability of bad design
iv) Increased difficulty in system improvement
v) Increased maintenance cost
vi) Increase resource requirement

Importance of clone Detection:-

i) Helps in program understanding
ii) Helps aspect mining research
iii) Find usage patterns
iv) Detects malicious software
v) Detects plagiarism and copyright infringement
vi) Helps software evolution research
vii) Helps in code compaction

2. Why Do Clones Occur?

Software clones appear for a variety of reasons:

i) Code reuse by copying pre-existing idioms
ii) Coding styles
iii) Instantiations of definitional computations
iv) Failure to identify/use abstract data types
v) Performance enhancement
vi) Accidents

Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development. As a result software systems often contain sections of code that are very similar, called code clones. Previous research shows that a significant fraction (between 7% and 23%) of the code in

INTERNATIONAL JOURNAL FOR ADVANCE RESEARCH IN ENGINEERING AND TECHNOLOGY
WINGS TO YOUR THOUGHTS.....

a typical software system has been cloned [1, 2]. While such cloning is often intentional [8] and can be useful in many ways [3, 4], it can be also be harmful in software maintenance and evolution [5,10]. For example, if a bug is detected in a code fragment, all fragments similar to it should be checked for the same bug [6]. Duplicated fragments can also significantly increase the work to be done when enhancing or adapting code [7,9]. Many other software engineering tasks, such as program understanding (clones may carry domain knowledge), code quality analysis (fewer clones may mean better quality code), aspect mining(clones may indicate the presence of an aspect), plagiarism detection, copyright infringement investigation, software evolution analysis, code compaction (for example, in mobile devices), virus detection, and bug detection may require the extraction of syntactically or semantically similar code fragments, making clone detection an important and valuable part of software analysis [11].

3. CLONE TERMINOLOGY

Clone detection tools normally report clones in the form of Clone Pairs (CP) or Clone Classes (CC) or both. Both these terms are based on a similarity relation between two or more cloned fragments. The similarity relation between the cloned fragments is an equivalence relation (i.e., a reflexive, transitive, and symmetric relation) [11]. A clone relation holds between two code fragments if (and only if) they are the same sequences, where sequences may refer to the original character text strings, strings without whitespace, sequences of tokens, transformed or normalized sequences of tokens, and so on. In the following, clone pair and clone class are defined in terms of the clone relation.

Definition 1: Code Fragment. A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, beginend block, or sequence of statements. 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).

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

Definition 3: Clone Types. There are two 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 we provide the types of clones based on both the textual (Types 1 to 3) [17] and functional (Type 4) [15] similarities:

Type 1: Identical code fragments except for variations in whitespace, layout and comments.

Type 2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

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.

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

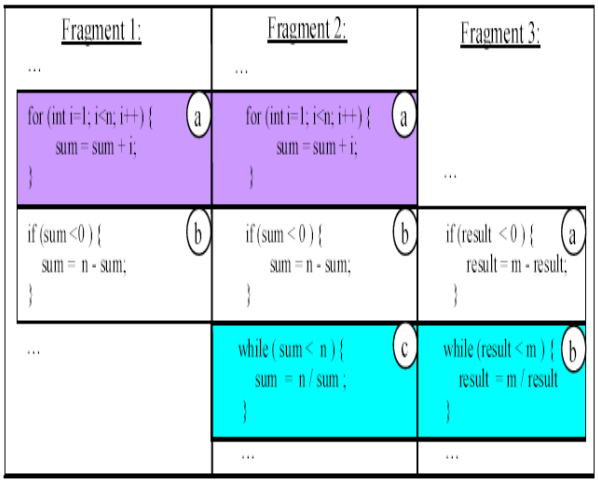

Figure 3.1: Clone pair and clone class

Definition 4: Clone Pair: A pair of code fragments is called a clone pair if there exists a clone relation between them, i.e., a clone pair is a pair of code fragments which are identical or similar to each other. For instance, the first code segment (a) in fragment 1 of Figure 3.1 together with segment (a) in fragment 2 form a clone pair. Clones may subsume each other and we are typically interested only in the maximally long clones. For instance, segment (a) could be joined with the subsequent segment (b) in fragment 1 of Figure 1 to form a larger clone together with the joined segments (a) and (b) in fragment 2. A maximally long clone pair is one whose two fragments can be extended neither to the left nor to the right to form a larger clone. For the three code fragments, Fragment 1 (F1), Fragment 2 (F2) and Fragment 3 (F3) of Figure 3.1, we have the following maximally long clone pairs: <F1(a + b), F2(a + b)>, <F2(b + c), F3(a + b)> and <F1(b), F3(a)>. We note that the clone relation defined by maximally long clone pairs

is not transitive. For instance, if we have the following string modelling code fragments adbdadb, we observe that d occurs three times in this string. The second and third d form a maximally long clone pair, but the first and last d do not form a maximally long clone pair because they can be extended neither to the left nor to the right to form the subsuming clone pair of the two abc. Only if we do not insist on maximal length, clone pairs may be summarized to equivalence classes, also known as clone classes.

Definition 5: Clone Class: A clone class [16] is the equivalence class formed by the (non-maximal) clone relation, that is, it is the maximal set of code fragments in which any two of the code fragments is a clone pair. For the three code fragments of Figure 3.1, we get the clone class <F1(b), F2(b), F3(a)> where the three code fragments F1(b), F2(b) and F3(a) each form clone pairs with the others, that is, there are three clone pairs, <F1(b), F2(b)>, <F2(b), F3(a)> and <F1(b), F3(a)>. A clone class is therefore simply the union of all clone pairs that have code fragments in common [7]. Clone classes may be further summarized into clone class families if they occur in the same context.

Definition 6: Clone Class Family: The group of all clone classes that have the same domain is called a clone class family [11] or super clone. The domain of a clone class is the set of source entities from which its source fragments stem. The particular source entities to be considered as domains depend on the particular programming language or scope of interest, but common examples are files, functions, classes, or packages. Previous research has shown that a significant fractions of the code in software systems is cloned, depending on the domain and origin of the software system. Baker [7] found that in large systems between 13% - 20% of source code can be cloned code, Lague et al. [16] have reported that between 6.4% and 7.5% of functions were cloned in ¨ the systems they studied, and Baxter et al. [11] found that 12.7% of code in a large software system was cloned. Mayrand et al. [15] estimated that industrial source code contains 5% – 20% duplicated code, and Kapser and Godfrey [16] have reported that as much as 10%–15% of source code of a large system was cloned. In one object-oriented COBOL system, the rate of duplicated code was found to be even higher, at 50%.

4. OVERVIEW OF CLONE DETECTION TECHNIQUES AND TOOLS

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 four main categories: textual, lexical, syntactic, and semantic. In this section we summarize the state of the art 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.

i. Textual Approaches

Textual approaches (or text-based techniques) use little or no transformation / normalization on the source code before the actual comparison, and in most cases raw source code is used directly in the clone detection process. Johnson pioneered text-based clone detection. His approach [17] uses “fingerprints” on substrings of the source code. First, code fragments of a fixed number of lines (the window) 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 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 [17].

ii. 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 [17]. 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 they are lexically not farther than a user-defined threshold away from each other. 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. Because syntax is not taken into account, clones found by token-based techniques may overlap different syntactic units [17].

iii. Syntactic Approaches

Syntactic approaches use a parser to convert source programs into parse trees or abstract syntax trees (ASTs)

which can then be processed using either tree-matching or structural metrics to find clones. Tree-matching Approaches: Tree-matching approaches (or tree-based techniques) find clones by finding similar subtrees. 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 [17]. 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. To avoid the complexity of full subtree comparison, recent approaches use alternative tree representations A novel approach for detecting similar trees has been presented by Jiang et al. [17] in theirtool 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.

Metrics-based Approaches: Metrics-based techniques gather a number of metrics for code fragments and then compare metrics vectors rather than code or ASTs directly. One popular technique involves fingerprinting functions, metrics calculated for syntactic units such as a class, function, method and statement that yield values that can be compared to find clones of these units [17].

iv. Semantic Approaches

Semantics-aware approaches have also been proposed, using static program analysis to provide more precise information than simply syntactic similarity. In some approaches, the program is represented as a program dependency graph (PDG). The nodes of this graph represent expressions and statements, while the edges represent control and data dependencies. This representation abstracts from the lexical order in which expressions and statements occur to the extent that they are semantically independent.

v. Hybrids

In addition to the above, there are also clone detection techniques that use a combination of syntactic and semantic characteristics. Leitao [17] provides a hybrid approach that combines syntactic techniques based on AST metrics and semantic techniques (using call graphs) in combination with specialized comparison functions.

5. ALGORITHM

Step 1: Build projects in High Level languages like C#, Java or C++ etc.
Step 2: Load this projects in clone detector tools (Text based or Tree based).
Step 3: Set the parameters like Project Name, Source Folder, Output Folder, language, clone size, gap decay etc.
Step 4: Convert the project into Datasets.
Step 5: Find clones.
Step 6: Generate report on cloning files and generate clone overview diagrammatically.
Step 7: Compare and analyze these results.

6. IMPLEMENTATION AND RESULTS

Table 6.1 to 6.5 shows the comparative analysis of Solid SDD and CloneDR tools. It shows File Count, Total Source Lines of Code (SLOC), Estimated SLOC before preprocessing, Estimated SLOC before preprocessing, Total CloneSets, Total CloneSets, Near-miss CloneSets, Near-miss CloneSets, SLOC in clones % etc.

Table 6.1 CloneDR Statistics

Clone Detection Statistics (CloneDR)

Statistic	Value
File Count	42
Total Source Lines of Code (SLOC)	2509
Estimated SLOC before preprocessing	2548
Expanded SLOC after preprocessing	2548
Total CloneSets	18
Exact-match CloneSets	6
Near-miss CloneSets	12
Number of cloned SLOC	973
SLOC in clones %	38.2%
Estimated removable SLOC	707
Possible SLOC reduction %	27.7%
Possible SLOC reduction in expanded file %	27.7%

INTERNATIONAL JOURNAL FOR ADVANCE RESEARCH IN ENGINEERING AND TECHNOLOGY
WINGS TO YOUR THOUGHTS…..

Table 6.2 Result of banking portal

Potential gain of code duplication removal:	
423	statements
9.89	% of total code
Potential gain of removing the first 5% duplication instances:	
156	statements
36.88	% of potential gain
3.65	% of total code

Table 6.3 Result of friend's world project

Potential gain of code duplication removal:	
418	statements
21.18	% of total code
Potential gain of removing the first 5% duplication instances:	
72	statements
17.22	% of potential gain
3.65	% of total code

Table 6.4 Result of jdk7 project

Potential gain of code duplication removal:	
4195	statements
6.6	% of total code
Potential gain of removing the first 5% duplication instances:	
1397	statements
33.3	% of potential gain
2.2	% of total code

Table 6.5 Result of learning project

Level A
Average cloned % 72.57
Average fan-out 5
Total #clones 73
Average #clones 1.92
Total #statements 1614
Average #statements 42.47

Files: 42
Found duplicates: 23

7. CONCLUSION AND FUTURE WORK

In this paper, we have focused on detection techniques, we have compared two techniques one is Text-Based (Solid SDD) and other is Token Based (Clone DR). Both techniques work very efficiently. Here, we first develop a project and then apply both techniques on it. The simulated results are shown. Then we focus on three projects, one is open-source project (jdk-7) and the other two are real-life based project (banking and social networking website). In all these projects, Solid SDD shows good results as shown in tables.

We hope that the results of this study may assist new potential users of clone detection techniques in understanding the range of available methods and selecting those most appropriate for their needs. We hope it may also assist in identifying remaining open research questions, avenues for future research, and interesting combinations of techniques. The evaluation results of this thesis are based on estimating the performance of techniques using the most lenient values of all tunable parameters, and thus our findings differ from the results of empirical studies. In future, we can compare more clone detection techniques to get good results.

References

[1] Rahman, F. Bird, C. Devanbu, P. 2010. Clones: What is that smell? IEEE Working Conference on Mining Software Repositories (MSR), 2-3 May 2010, 72-81.

[2] Harder, J. Gode, N. 2010. Quo vadis, clone management? In Proceeding of IWSC '10 - 4th International Workshop on Software Clones. 2010, 85-86.

[3] Mutharaju, R. and Maier, F. A MapReduce Algorithm for EL+.23rd International Workshop on Description Logics (DL2010), 2010, 464-474.

[4] Stefan Bellon. Detection of software clones tool comparison experiment, http://www.bauhaus-stuttgart.de/clones, last visited July 2010.

[5]Amazon Elastic Computing, http://aws.amazon.com/ec2/, last visited January 2011.

[6] B. Biegel and S. Diehl, "JCCD: A Flexible and Extensible API for Implementing Custom Code Clone Detectors," 25th IEEE/ACM International Conference on Automated Software Engineering, pp. 167- 168, 2010.

[7] C.K. Roy and J.R. Cordy, A Mutation Injection-based Automatic Framework for Evaluating Clone Detection Tools, in: Proceedings of the 4th International Workshop on Mutation Analysis, Mutation 2009, 10pp. (2009) (submitted).

[8] M. Kim, G.Murphy, An Empirical Study of Code Clone Genealogies, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/SIGSOFT FSE 2005, pp. 187-196 (2005).

INTERNATIONAL JOURNAL FOR ADVANCE RESEARCH IN ENGINEERING AND TECHNOLOGY

WINGS TO YOUR THOUGHTS.....

[9] 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 (2006).

[10] 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 (1996).

[11] C.K. Roy and J.R. Cordy, A Survey on Software Clone Detection Research, Queen's Technical Report:541, 115 pp. (2007).

[12] Kevin Jalbert, Jeremy S. Bradbury, "Using Clone Detection to Identify Bugs in Concurrent Software", 26th IEEE International Conference on Software Maintenance(2010).

[13] Anna Corazza, Sergio Di Martino, Valerio Maggio, Giuseppe Scanniello," A Tree Kernel Based Approach for Clone Detection" 26th IEEE International Conference on Software Maintenance(2010).

[14] Philipp Schugerl, Juergen Rilling, Philippe Charland," Reasoning about Global Clones Scalable Semantic Clone Detection", IEEE (2011).

[15] James R Cordy, Chanchal K. Roy, "The NiCad Clone Detector" 19th IEEE International Conference on Program Comprehension (2011).

[16] James R Cordy, Chanchal K. Roy, "DebCheck: Efficient Checking for Open Source Code Clones in Software Systems" 19th IEEE International Conference on Program Comprehension (2011).

[17] Chanchal K. Roy_,a, James R. Cordya, Rainer Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach" Preprint submitted to Science of Computer Programming February 24, 2009.