International Journal of Computer Applications (0975 – 8887)
Volume 126 – No.7, September 2015

A Novel Composite Approach for Software Clone Detection

Gurvinder Singh
Research Scholar
Punjab Technical University

Jahid Ali, PhD
Director
SSICMIT, Badhani

ABSTRACT

In recent decades, the branch of Clone Detection has undergone a great advancement. This progress is due to the development of various methods, which involves the implementation of complex algorithms and tool chains to offer clone detection. Various clone detection methods that are already available include textual comparison, token comparison, and comparison of Abstract Syntax trees, Suffix trees and Program Dependency Graphs. Moreover, these Clone Detection techniques are limited to a particular programming language environment only. The aim of the paper is to present a survey of the various existing techniques and to develop a tool which is user friendly, easy to maintain and is not limited to small and big software. This method of clone detection can also be implemented to more complex applications such as web based applications. i.e a website code related to PHP or JSP or it can be an application which is linked with internet not a standalone application. In addition to this, the proposed approach is applicable to all the languages and platforms. Hence the proposed system is a platform independent system..

General Terms

Software Cloning

Keywords

Clone detection, Textual comparison, Hybrid approach, code cloning.

1. INTRODUCTION

Clone detection is an area of dynamic research where several tools already exist to encourage code clone detection. Most research has explicitly or implicitly expected that code cloning is destructive and has concentrated on systems for refactoring or expelling code clones from the source code without considering the first choices prompting the code clone. Regardless of an extensive research, there is still need of potential work in the investigation and analysis of near-miss software clones specifically minor to broad alterations have been made to the replicated sections. Software maintenance is the principle driver of aggregate expenses in the lifecycle of long-living software systems. The maintenance phase consists of those changes that are made to a software system after it has been deployed to the client upon client acceptance. The studies show that the major fraction of the annual software expenditure is being spent for maintaining existing software systems. The replication of code fragments across the system often diminishes maintainability as it expands the code size and hinders manual code change, review, and investigation. With the increasing levels of sophistication and complexity of software systems, the standards for software quality and productivity are also getting up to that. Developers continually look for various procedures, tools, and practices to accelerate software development without resulting into additional software defects.

The copied code is called a software **clone** and the process is called software **cloning**. A bug **detected** in one section of code therefore requires correction in all the replicated fragments of code.A code clone is a section of code in source files, identical or similar to another code section. It is a very common practice by developers to copy existing code and pasting it in somewhere else with major or minor edits to increase productivity. This reuse mechanism results in duplicate or very similar code fragments in the code base which are commonly known as code clones. If the presence of clones in program artifacts causes the artifacts to be more frequently changed and the cloned code shows unstable behavior, then clones are considered harmful. Due to huge measure of data involved, it becomes highly difficult to detect code duplication in large code bases or across project boundaries. Clones are connections between different projects. Duplicated fragments often results in significantly increase the work to be done during code optimization. Recent related studies also show that inconsistent changes to cloned code are frequent and lead to severe unexpected behavior. Subsequently, it gets more difficult to maintain the software systems with code clones and can lead to include subtle errors. Hence, while cloning is frequently deliberate and can be helpful in many perspectives still it can also be destructive in software maintenance and evolution.

2. REASONS OF CODE CLONING

Code clones do not occur in software systems by themselves. There may be numerous reasons for cloning the source code. The most usual reason behind it is that it is quick and cheap to just copy the code and place it wherever the similar functionality is needed rather than writing the code from scratch. Most of the times, this scenario happens and a little modification is done to make it distinguishable.

Various factors that may enforce the introduction of clones in a code are as:

- Clones can be introduced in software systems due to many different reuse and programming approaches. The simplest form of reuse mechanism in the development process is copying and pasting existing code with least possible alterations and it is a primary cause of code cloning.
- In case, a new system is produced by merging two software systems identical in functionality, the merged system may show the presence of clones because of the implementations of similar functionality in both systems.
- Code clones can be introduced with good intentions too such as for improved code understand ability which lead to enhance readability, conceptual cohesion/coupling, and traceability and in some situations it can be used to keep software architectures clean and understandable.
- In cases of technology limitations, the use of code cloning is well understood by the developers with the motive to

prevent errors by re-using trusted solutions in new connections.

- In addition to this, few external business forces may necessitate the use of code cloning too. According to J. Cordy [1], most often occurrences of clones have been reported in financial software due to frequent updates and enhancements of an existing system to support similar kinds of new functionality. Since the new applications are not that much different from those of the existing ones, the developer prefers to reuse the existing code by copying and adapting to the new product requirements because of the high risk of software errors when creating new code.
- Sometimes programming languages do lack abstraction mechanisms which ultimately developers repeatedly implement, hence leads to clones.
- The thought that writing reusable code is error-prone especially for a critical piece of code. It is therefore preferred to copy and reuse existing code rather than make new reusable code as introduction of new bugs can be avoided in critical system functionality by keeping the critical piece of code untouched.
- One of the major causes of code cloning in the system is the time limitation on developers. A developer is assigned a specific time deadline to finish a certain project due to which, developers usually look for short way of solving the problems and consequently look for similar existing solutions.
- Sometimes the developer is not familiar to the problem domain at hand and hence looks for existing solutions of similar problems to get better understanding of the view. Once such a solution is found, the developer just adapts the existing solution to his/her needs.
- Programmers may unintentionally repeat a common solution for similar problems using a solution pattern from his/her memory of similar problems. Clones may then unintentionally be created.

3. IMPACTS OF CODE CLONING

Clones have become controversial in the software engineering research domain because of their dual, and contradictory impacts during software maintenance. Along with the numerous positive impacts of clones in terms of faster development and reduction of maintenance cost and effort, there exist potential negative impacts of clones on software maintenance in terms of hidden bug propagation and unintentional inconsistent changes. If code clones are not carefully managed, they can introduce bugs in the system and can also cause propagation of bugs across different portions of the source code. In the following we list some of the consequences of having cloned code in a system:

3.1 Impact on System Modification

It may become challenging to add new functionality in the system or to enhance the existing ones due to the extra time and effort needed to comprehend and adjust the current cloned usage. If a bug is found in a cloned code segment, all of its similar parts ought to be investigated for adjusting the bug in question since there is no surety of this bug already being eliminated from other similar parts during reusing or during maintenance activity. In addition, in keeping up or upgrading a bit of code, duplication increases the measure of work. Code cloning can lead to unused code in the system when the desired solution does not require all of the functionality provided by the clone. If such section is left unchecked, this unused code can bring about issues with code understand ability, readability, and maintainability for the software system lifetime.

3.2 Effect on Faults

If a bug is found in a code fragment and that code fragment is already copied and pasted to several other places without the awareness of this bug, resulting in increased modifications to the source code after the discovery of the bug in any one of the clone fragments. This leads in increasing the probability of bug propagation in the system.

3.3 Effect on Cognitive Effort

Duplication also increases the cognitive effort required by the maintenance engineers to understand a large software system. There are multiple occurrences of a cloned fragment in different places of the system and the maintenance engineers are required to examine all the different instances in order to understand the difference between them.

3.4 Effect on Design

Cloning may additionally introduce bad design, absence of efficiently good inheritance structure. Hence, it gets to be hard to reuse part of the usage in future tasks. It additionally effects the viability of the software.

3.5 Effect on Resource Requirements

Code duplication adds to higher growth rate of the system size. While some domains hardly bother for system size others may require costly hardware upgrade with a software upgrade.

4. CLONE DETECTION PROCESS

Clone detection is the most important and integral part of clone management. Clones from the source code must be identified first before they can be dealt with.

4.1 Preprocessing

The code cloning detection process starts with partitioning of the source code and the domain of the comparison is determined. This phase is mainly responsible for:

- Evacuate uninteresting parts: All the source code irrelevant to the comparison phase is filtered in this stage.
- Determine source units: The source code obtained after removing the uninteresting code is partitioned into a set of disjoint pieces called source units which are the largest source sections suspected ti be involved in direct clone relations with each other.
- Determine comparison units / granularity: Going with the comparison technique, source units may further be partitioned into smaller units. For instance, source units may be partitioned into lines or even tokens for a comparison purpose. Comparison units can likewise be derived from the syntactic structure of the source unit.

4.2 Transformation

Once the comparison units are decided, the source code of the comparison units is transformed to a proper intermediate format for comparison. This change of the source code into an middle representation is regularly called extraction in the reverse engineering community. A few tools support additional normalizing transformations and extraction with a specific aim to distinguish externally distinctive clones.

- Extraction: Extraction transforms source code to the form suitable as input to the actual comparison algorithm. It further involves tokenization, parsing, control and data flow analysis.

- Normalization: Normalization is an optional step intended to eliminate superficial differences such as differences in whitespace, commenting, formatting or identifier names.

4.3 Match Detection
The comparison units take transformed code as input and compares the transformed comparison units to each other to discover matches. The neighboring similar comparison units are collaborated to form larger units. The match detection results into 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.

4.4 Formatting
Here, the resulted clone pair list for the transformed code 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.

4.5 Filtering
In this section, clones are manually analyzed, ranked and filtered or they are fed under automated heuristics.

- Manual Analysis: After the original source code retrieval, clones are manually investigated and the human expert filters the false positive clones or spurious clones. This manual filtering step can be speeded up by visualization of the cloned source code in a suitable format.

- Automated Heuristics: Heuristics can usually be characterized in view of length, diversity, frequency, or other attributes of clones to rank or filter out clone candidates automatically.

4.6 Aggregation
Where a few devices straightforwardly recognize clone classes, most return just clone pairs as the outcome. With a specific end goal to diminish the measure of data, perform subsequent analyses or gather overview statistics, clones may be aggregated into clone classes

5. CLONE DETECTION TECHNIQUES
Many clone detection approaches have been proposed till date. According to the height of analysis applied to the source code, the techniques can generally be classified into four main categories: textual, lexical, syntactic, and semantic.

5.1 Textual Approach
As explained by Chanchal K. Roy et.al [1], Textual approaches use minimal transformation / normalization on the source code before the genuine comparison, and mostly raw source code is used directly in the clone detection process. In this approach, the target source program is considered as sequence of lines/strings. Two code parts are contrasted with each other to discover sequences of same text/strings. Once two or more code fragments are found to be similar in their maximum possible extent (e.g., w.r.t maximum no. of lines) are returned as clone pair or clone class by the detection technique. Because of the purely text-based and/or lexical approach, detected clones do not correspond to structural elements of the language.

As explained by Prajila Prem et.al [6], String based techniques use basic string transformation and comparison algorithms which make them independent of programming languages. Techniques in this category differ in the string comparison algorithm. Comparing calculated signatures per line is one possibility to identify for matching substrings. Line matching, which comes in two variants, is an alternative which is selected as representative for this category because it uses general string manipulations.

Simple line matching is the first variant of line matching in which both detection phases are straightforward. Only minor transformations using string manipulation operations, which can operate using no or very limited knowledge about possible language constructs, are applied. Typical transformations are the removal of empty lines and white spaces. During comparison all lines are compared with each other using a string matching algorithm. This result in a large search space which is usually reduced using hashing buckets. Before comparing all the lines, they are hashed into one of n possible buckets. Afterwards all pairs in the same bucket are compared.

Parameterized line matching: is another variant of line matching which detects both identical as well as similar code fragments. The idea is that since identifier–names and literals are likely to change when cloning a code fragment, they can be considered as changeable parameters. Therefore, similar fragments which differ only in the naming of these parameters are allowed. To enable such parameterization, the set of transformations is extended with an additional transformation that replaces all identifiers and literals with one, common identifier symbol like"$P". Due to this additional substitution, the comparison becomes independent of the parameters. Therefore no additional changes are necessary to the comparison algorithm itself.

5.2 Lexical Approach
Chanchal K. Roy et.al explained in [5], Lexical approach which is also known as token-based technique begin with creating a sequence of lexical tokens out of the source code in a similar manner of compiler lexical analysis. The obtained sequence is further filtered to find out replicated sub-sequences of tokens and the comparing original code is returned as clones. Lexical approaches are by and large more robust over minor code changes such as formatting, spacing, and renaming than textual techniques.

Various tools are proposed for clone detection that is based on token based approach [12].

5.3 Syntactic Approach
In Syntactic approaches, a parser is used to build parse trees or abstract syntax trees from source programs which can further be processed using either tree-matching or structural metrics to find clones.

Chanchal K. Roy et.al explained in [5], the parse tree or AST contains the complete information about the source code. Although the variable names and literal values of the source are discarded in the tree representation, more sophisticated methods for the detection of clones still can be applied.

Tree-based Approaches: Prajila Prem explains that [6] Tree-based method first convert the program to a parse tree or abstract syntax tree (AST) using a parser for the target language. Tree-matching techniques are then used to find similar sub trees, and the corresponding code segments are returned as clone pairs. Variable names, literal values and other tokens in the source may be abstracted in the tree representation, allowing for more sophisticated detection of clones.

Metrics-based Approaches: In Metrics-based techniques, a number of metrics are assembled for code segments and afterwards metrics vectors are compared inspite of code or ASTs directly. One popular technique involves fingerprinting functions, metrics calculated for syntactic like a class, function, method or statement that provides values that can be compared to find clones of these syntactic units. In most cases, the source code is first parsed to an AST or CFG (control flow graph) representation to calculate the metrics. Metrics are calculated from names, layout, expressions and control flow of functions.

5.4 Semantic Approach

Semantics-aware methods have also been proposed, using static program analysis to provide more precise information than simply syntactic similarity.

PDG-based Techniques: Chanchal K. Roy et.al explained in [5], Program Dependency Graph (PDG)-based approaches go a step further in source code abstraction by considering semantic information encoded in a dependency graph that captures control and data flow information. Given the PDG of a subject program, a sub graph isomorphism algorithm is used to find similar sub graphs which are then returned as clones.

Gurunadha Rao Goda et.al [16] proposed a hybrid approach that depends on template conversion and metrics comparison. There are four phases involved in the proposed scheme, namely, input and pre-processing, template conversion, metrics computation and clone type detection. A new technique is introduced, which is the hybrid combination of metric-based approach and textual comparison of the source code for the detection of Clones. Several metrics have been developed to make use of their values during the detection process.

T. Kamiya et.al [17] explains CCFinder detects code clones from source programs, and outputs the locations of the code clones on the source programs. In the detection processing, CCFinder replaces user-defined identifiers such as variable names with special tokens, so that it can regard two similar code fragments as code clones even if they include different user-defined identifiers. The minimum size of code clones to be detected is set by a user in advance. CCFinder can complete code clone detection from systems of millions line scale in a practical timeframe.

Yoshiki Higo et.al [4] developed a software tool, Scorpio. The tool implements multi-threads processing to effectively use the resource of multi-cores CPU. The tool has many options to specify what kinds of duplicate code are detected as code clones. The parameterization has three level: in level 0, the tokens are used as they are; in level 1, the tokens are replaced with their type names; in level 2, all the tokens are replaced with the same special token.

Rajkumar Tekchandani et.al [11],presented an algorithm that can detect semantically equivalent code fragments using formal grammars with the insight that the grammar recovery can be used to find semantically equivalent code fragments. The work also proposed an algorithm to recover the grammar from parse trees.

Iman Keivanloo et.al [7], introduced a novel hybrid clone detection approach named SeClone clone search tool, which is based on multi-layer indexing. This approach considers information retrieval clustering and Semantic Web reasoning methods for clone pair clustering. A clone ontology (CLON) is developed to model the code clone detection vocabulary to support the use of reasoning services and to provide a formal result sharing and integration approach.

Chanchal K. Roy [1], describes the first empirical investigation of function clones in open source software utilizing NICAD. NICAD is a new hybrid clone detection tool which gathers the qualities and overcomes the limitations of both content based and AST-based clone detection techniques to yield exceptionally precise identification of cloned code in software systems. The paper give an inside and out exact investigation of function clones in near about 15 or more open source C and Java frameworks including Apache httpd and the whole Linux Kernel, and confirms every recognized clone and give a complete list of diverse clones in an online archive in a variety of configurations. These outcomes can possibly be utilized as a benchmark for assessing other clone discovery tools.

6. CONCLUSION

This paper mainly focused on detection techniques and clone analysis methods which help for understanding code clones and the different techniques used. The intent of the paper is to present a review of the detection techniques and propose an approach to deal with code clones in any environment. In future the extended work can be enhanced with advanced algorithms with enhancement in research scenario

7. ACKNOWLEDGEMENT

I am Thankful To Punjab Technical University Jalandhar Which Gives Me Opportunity To Work As Research Scholar and Also thanks to my Respected Guide Dr Jahid Ali(Director,SSICMIT) for their kind guidance.

8. REFERENCES

[1] Chanchal K. Roy and James R. Cordy, “An Empirical Study of Function Clones in Open Source Software”, 1095-1350/08 $25.00 © 2008 IEEE

[2] Mark Gabel Lingxiao Jiang Zhendong Su, “Scalable Detection of Semantic Clones”, ICSE’08, May 10–18, 2008, Leipzig, Germany. Copyright 2008 ACM

[3] Chanchal K. Roy, “Detection and Analysis of Near-Miss Software Clones”, 978-1-4244-4828-9/09/$25.00 2009 IEEE

[4] Yoshiki Higo, and Shinji Kusumoto, “Enhancing Quality of Code Clone Detection with Program Dependency Graph”, 2009 IEEE

[5] Chanchal K. Roy, James R. Cordya, Rainer Koschkeb, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach”, Preprint submitted to Science of Computer Programming February 24, 2009

[6] Prajila Prem, “A Review on Code Clone Analysis and Code Clone Detection”, ISSN: 2277-3754 ISO 9001:2008 Certified International Journal of Engineering and Innovative Technology (IJEIT) Volume 2, Issue 12, June 2013

[7] Iman Keivanloo, Juergen Rilling, Philippe Charland, “SeClone - A Hybrid Approach to Internet-scale Real-time Code Clone Search”, 1063-6897/11 $26.00 © 2011 IEEE

[8] Hitesh Sajnani, Joel Ossher, Cristina Lopes, “Parallel Code Clone Detection Using MapReduce”, 2012 IEEE

[9] Norihiro Yoshida, Yoshiki Higo, Shinji Kusumoto, Katsuro Inoue, “An Experience Report on Analyzing Industrial Software Systems Using Code Clone Detection Techniques”, 2012 IEEE

[10] Kanika Raheja1, Raj Kumar Tekchandani2, “An Efficient Code Clone Detection Model on Java Byte Code Using Hybrid Approach”

[11] Rajkumar Tekchandani1, Rajesh Kumar Bhatia2, Maninder Singh, “Semantic Code Clone Detection Using Parse Trees and Grammar Recovery”

[12] Kanika Raheja Rajkumar Tekchandani, “An Emerging Approach towards Code Clone Detection: Metric Based Approach on Byte Code”, Volume 3, Issue 5, May 2013 ISSN: 2277 128X International Journal of Advanced Research in Computer Science and Software Engineering

[13] Al-Fahim Mubarak Ali, Shahida Sulaiman, “A Hybrid Technique in Pre-processing and Transformation Process for Code Clone Detection”, 2014 IEEE

[14] Robin Sharma, “Hybrid Approach for Efficient Software Clone Detection”, IRACST – Engineering Science and Technology: An International Journal (ESTIJ), ISSN: 2250-3498 Vol.3, No.2, April 2013

[15] Deepak Sethi Manisha Sehrawat Bharat Bhushan Naib, “Detection of code clones using Datasets”, Volume 2, Issue 7, July 2012 ISSN: 2277 128X International Journal of Advanced Research in Computer Science and Software Engineering

[16] Gurunadha Rao Goda, Avula Damodaram, “An Efficient Software Clone Detection System based on the Textual Comparison of Dynamic Methods and Metrics Computation”, International Journal of Computer Applications (0975 – 8887) Volume 86 – No 6, January 2014

[17] 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, July 2002.