IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Code Clone Analysis: Code Clone Types and Detection

[1]Mrs Vani Dave,[2]Mr Sanjeev Kumar Shukla,

[1]M.tech Research Scholar,[2]Assistant Professor and Head of Department

[1]Computer Science and Engineering,

[1]Kanpur Institute of Technology,Kanpur,India

Abstract: A code clone is a Duplicate code exist in a whole source code. The main reason behind the code cloning is copying existing code fragments and using them by pasting with or without minor modifications. Though it has some advantages like it increase the reusability of the code segments but a survey shows that it is harmful more . One of the major problem in such duplicated codes is that if an errors detected in a code fragment, all the other similar codes has to be checked for fixing the same bug.

Another disadvantage is that it increases maintenance cost. So it is necessary to detect the code clone .

In this paper we explain various types of code cloning and the methods of Detecting the code clones.

Index Terms - Code clone ,Types of code clone,Clone Detection Process, Clone Detection Techniques,Clone Detection Tools

I. INTRODUCTION

Code cloning is the process of duplicating existing source **code** for use elsewhere within a software system. Within the research community, **code cloning** is generally a bad practice, so that **code clones** should be removed or refactored where possible.

"The automated process of finding duplicate codes in source code is called clone detection" This paper is divided into Following two categories:

1) Regarding the type of code clones.
2) Describe various detecting methods of Code cloning.

In the first part we describe all types of code clones.

Type I: Identical code fragments except for variations in whitespace and comments.

Type II: Type 2 category includes the code segments which are syntactically same but the changes are in identifiers, literals, types, layout and comments.

Type III: These are the Copied fragments having some modifications. Like statements could be changed, added or deleted.

Type IV: Two or more code segments that perform the same work but implemented through different syntactic models.

In the second part of the paper we describe various approaches to Detecting code clones that are:

1) Text-based Techniques
2) Token-based Techniques
3) Tree-based Techniques.
4) PDG-based Techniques
5) Metrics-based Techniques
6) Hybrid Approaches

2.Code Clone Types

2.1 Type I Clones

Type I clones are identical copy of original. However, there might be some changes like whitespace (blanks, new line(s), tabs etc.), comments and/or layouts. *Type I* is also known as *Exact clones*. Let us consider the following code fragment,

```
if (a     >= b)
{
        c= d +b; // Comment1
           d =d+1;
}
```

```
else
        c =d-a; //Comment2
```

A duplicate copy of this original code could be as follows:

```
if (a>=b) {
//Comment1'
c=d+b;
d=d+1;}
        else// Comment2'
 c=d-a;
```

2.2 Type II Clones

A *Type II* clones are an extension to Type 1 except some possible changes. like name of variables, constants, class, methods and so on, types, layout and comments. The keywords words and the sentence structures are essentially the same as the original one. Let us consider the following code sequence:

```
if (a >= b) {
        c = d = d + 1;}

else

        c = d - a; //Comment2
 d + b; // Comment1
```

A *Type II* clone for the above code can be as follows:

```
if (m >= n)
{ // Comment1'
y = x + n;
  x = x + 5; //Comment3
  }
else
  y = x - m; //Comment2'
```

We can easily compare that the two code segments change a lot in their structure, variable names and value assignments. However, the syntactic structure is still similar in both codes.

2.3 Type III Clones

In *Type III* clones, the duplicate segment is further modified. May be statement(s) are changed, added and/or deleted. Consider the original code segment,

```
if (a >= b) {

        c = d + b; // Comment1

else

        c = d - a; //Comment2
```

we add a statement *e* = 1 then we can get,

```
if (a >= b) {

        c = d + b; // Comment1

        e = 1; // This statement is added

        d = d + 1; }

else

        c = d - a; //Comment2
        d = d + 1;}
```

above is an example of Type 3 code clone as we add 1 statement.

2.4 Type IV Clones

Type IV clones have the semantic similarity between two or more code fragments. Two code fragments may be developed by two different programmers to implement the same logic making the code fragments similar in their functionality. Let us consider the following code fragment 1, where the final value of 'j' is the factorial value of the variable VALUE.

Fragment 1:

```
int i, j=1;

    for (i=1; i<=VALUE; i++)

            j=j*i;
```

Now consider the following code fragment 2, which is actually a recursive function that calculates the factorial of its argument *n*.

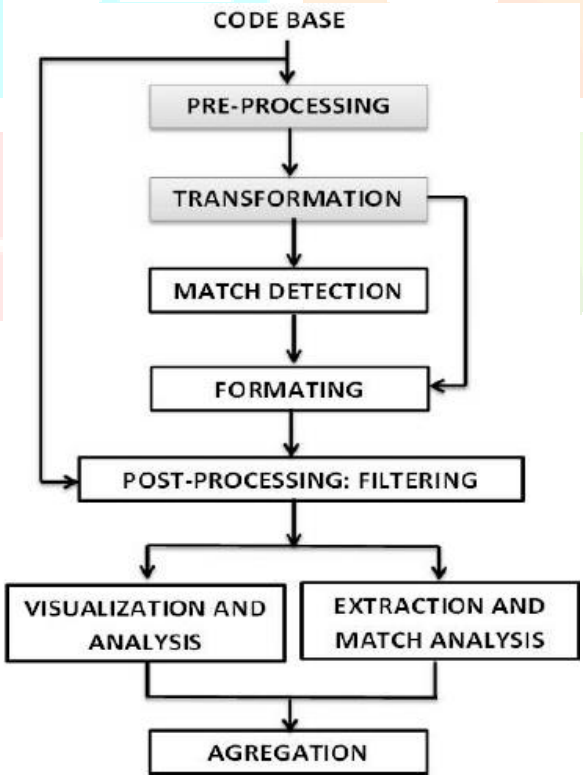
Fragment 2:

```
int factorial(int n) {

  if (n == 0) return 1 ;

  else              return n * factorial(n-1) ;

}
```

From the semantics point of view both the code fragments are similar in their functionality and termed as *Type IV* .

3.Clone Detection Process

A clone detector mainly deals with to find the code similar to the system's source code. The key issue is that we don't know in advance that which code fragments can be found multiple times. Thus the detector has to compare every possible fragment with every other possible fragment essentially But this type of comparison is very expensive from a computational point of view so several techniques has to be apply to reduce the domain of comparison before applying the actual comparison. In this section, we attempt to provide an overall summary of the clone detection process. Figure shows the phases that a clone detector may follow in its detection process. Below figure shows the phases in clone detection process. we provides the brief description of each phase:



Clone Detection Process

3.1 Preprocessing: This is the first phase of any clone detection process. In this we determine the domain of the comparison and then partitioned the target source code. There are mainly three objectives of this phase:

3.1.1 Remove uninteresting parts: All the source code uninteresting to the comparison phase is filtered in this phase. For example, partitioning is applied to embedded code (e.g., SQL embedded in Java code, or Assembler in C code) for separating

3.1.2 Determine Source Units: After removing the uninteresting code, the remaining source code is partitioned into a set of disjoint fragments called source units. Source units are the Domains for the code clones and involve in direct cloning . Granularity can be maintained at different levels of the source code. such as Statements, blocks, procedures ,classes and objects and files or data bases.

3.1.3 Determine comparison unit/granularity: Source units may need to be further divided into smaller units depending on the comparison function of a method. For example, source units can be subdivided into lines or even tokens for comparison.

3.2) Transformation To make the comparison more easy the comparison units of the source code are transformed to another intermediate internal representation. e.g., just removing the whitespace and comments [3] to very complex e.g., generating PDG representation [10, 13] and/or extensive source code transformations [9]. Metrics-based methods usually compute an attribute vector for each comparison unit from such intermediate representations. In the following section we briefly explain transformation approaches. Comparison algorithm uses One or more of the following transformations

3.2.1) **Pretty printing of source code:** Pretty printing is a simple way of reorganizing the source code to a standard form. source code of different layouts can be transformed to a common standard form by using this Technique. The text-based clone detection process uses pretty printing to avoid the false positives that occur due to the different layouts of the similar code segments. Cordy et al. [5] use an *extractor* which generate separate pretty-printed text file for each of the potential clones obtained using an island grammar [7, 22].

3.2.2) Removal of comments: Most of the approaches (except Marcus & Maletic [14] and Mayrand et al. [15]) ignore/remove comments from the source code before performing the actual comparison. Marcus & Maletic search for similarities of concepts extracted from comments and source code elements. Mayrand et al., on the other hand, use metrics to measure the amount of comments and use that metric as a measuring metrics to find clones.

3.2.3) Removal of whitespace: Almost all the approaches (except line-based approaches) disregard whitespace. All whitespace except line breaks can be removed by Line based approaches. Davey et al. [6] use the indentation pattern of pretty printed source text as one of the features for their attribute vector. Mayrand et al. [15] use layout metrics like *number of non-blank lines.*

3.2.4) Tokenization: Each line of the source code is divided into tokens by applying a lexical rule of the programming language. Tokens of all lines are then used to form token sequence(s). All the whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequence. *CCFinder* [9] and *Dup* [3] are the leading tools that use tokenization on the source code.

3.2.5) Parsing: In case of parse tree-based approaches, the entire source code base is parsed to build parse tree or (annotated) abstract syntax tree (AST). In such representation, the source unit and comparison units are represented as sub trees of the parse tree or AST. Comparison algorithm then uses these sub trees to find clones [4,18,19]. Metrics-based approaches may also use such representation of code to calculate of the sub trees and find clones based on the metrics values [11, 15].

3.2.6) Generating PDG: Semantics-aware approaches generate program dependence graphs (PDGs) from the source code. Source units or comparison units are the sub graphs of these PDGs. Detection algorithm then looks for isomorphic sub graphs to find clones [10, 13]. Some metrics-based approaches also use these sub graphs to form data and control flow metrics and also then be used for finding clones [11,15].

3.2.7) Normalizing identifiers: Most of the approaches apply identifier normalizations before going to the comparison phase. All identifiers of the source are replaced by a single token in such normalizations. However, Baker [3] applies systematic normalizations of the identifiers to find parameterized clones.

3.2.8) Transformation of program elements: In addition to identifier normalizations, several other transformation rules may be applied to the source code elements. In this way, different variants of the same syntactic element may treat as similar to find clones [9, 17].

3.2.9) Calculate metrics values: Metrics-based approaches calculate several metrics from the raw and/or transformed (AST, PDG, etc.) source code and use these metrics values for finding clones [15, 11].

3.2.10) Generating PDG: Semantics-aware approaches generate program dependence graphs (PDGs) from the source code. Source units or comparison units are the sub graphs of these PDGs. Detection algorithm then looks for isomorphic sub graphs to find clones [10, 13]. Some metrics-based approaches also use these sub graphs to form data and control flow metrics and also then be used for finding clones [11,15].

3.2.11) Normalizing identifiers: Most of the approaches apply identifier normalizations before going to the comparison phase. All identifiers of the source are replaced by a single token in such normalizations. However, Baker [3] applies systematic normalizations of the identifiers to find parameterized clones.

3.2.12) Transformation of program elements: In addition to identifier normalizations, several other transformation rules may be applied to the source code elements. In this way, different variants of the same syntactic element may treat as similar to find clones [9, 17].

3.2.13) Calculate metrics values: Metrics-based approaches calculate several metrics from the raw and/or transformed (AST, PDG, etc.) source code and use these metrics values for finding clones [15, 11].

3.2.14) The above transformations just provide an overview of the current transformation techniques used for clone detection. Several other types of transformations with different levels can be applied on the source code before going to the *match detection* phase

3.3) Match Detection The next input to a suitable comparison algorithm is transformed code where these units are compared to each other to find a match. Adjacent similar units are summed up to form larger units by using the order of comparison units. For flexed granularity clones, all the comparison units that belong to a source unit are aggregated. On the other hand, for free granularity clones, aggregation is continued till the aggregated sum is above a given threshold for the number of aggregated comparison units. Aggregation is continued till the largest possible group of comparison units are found.

At the end list of matches are found . These matches may be the clone pair candidates or have to aggregate to form clone pair candidates. Each clone pair is normally represented with the location information of the matched fragments in the transformed code. For example, for a token-based approach, a clone pair is represented as a quadruplet (LeftBegin , LeftEnd, RightBegin, RightEnd), where LeftBegin and LeftEnd are the beginning and ending positions (indices in the token sequence) of leading clone, and RightBegin and RightEnd refer to the other cloned fragment that forms clone pair with the first one. Some popular matching algorithms are the su–x-tree [12,16] algorithm [3,9], dynamic pat-tern matching (DPM) [8,11] and hash-value comparison [4, 15]. Several other algorithms are used in the literature.

3.4) Formatting In this phase, line numbers on the original source files are found from each location of the clone pair obtained from the previous phase. The general format of representing a clone pair can be a nested tupple , *f*(FileNameLeft, StartLineLeft, EndLineLeft), (File-NameRight, StartLineRight, EndLineRight)*g* where FileNameLeft represents the location and name of the file containing the leading fragment with StartLineLeft and EndLineLeft showing the boundary of the cloned fragment in that file with respect to the line numbers. In a similar way FileNameRight, StartLineRight and EndlineRight represent the other cloned fragment that forms clone pair with the first.

3.5) Post-processing In this phase, false positive clones are filtered out with manual analysis and/or a visualization tool.

3.5.1) **Manual Analysis** After extracting the original source code, raw code of the clones of the clone pairs are subject to the manual analysis. This phase is used to filtered out the false positive clones.

3.5.2) **Visualization** The obtained clone pair list can be used to visualize the clones with a visualization tool. For removing false positives a visualization tool can speed up the process of manual analysis or other associated analysis.

3.6) Aggregation The clone pairs are aggregated to clusters, classes, cliques of clones, or clone groups in order to reduce the amount of data

The clone detection phases described above are very general.

4.CLONE DETECTION TECHNIQUES

In this section we defines the techniques for code clone detection [1] [2]:

4.1) Textual Approach (Text Based technique)

This approach states that there is no source code transformation before the comparison has done on both sides. In variety of cases, the original source code is used as it is presented in the process of clone detection. For example, NICAD, SDD, Simian 1 etc.

4.2) Lexical Approach (Token Based technique)

To perform the compiler style lexical analysis. initially source code is converted in the lexical sequence, known as tokens. The sequence later scans the identical token sequence of the original code that is resulted as clones. These types of approaches are normally more resilient for small variations in the code. It is defined as spacing, formatting and renaming which is different as compare to textual techniques. For example CCFinder, Dup, CPMiner and so on.

4.3) Syntactic Approach

This approach utilizes a parser for converting a source program in abstract syntax trees or parse trees that can be processed by using structural metrics or tree matching for finding the clones. For example: Deckard, Clone Dr and Clone Digger and so on.

4.4) Semantic Approach

Static program is used in this approach. In comparison to the syntactic similarity it gives the in-depth data. Semantic approach is given in the form of PDG (Program dependency graph) or in the form of

Statements or expressions but the edges shows the dataor Duplex and so on control dependencies. For example, GPLAG, Duplex and so on.

5) Code clone Classification and Technique

	Text based	Token based	AST based	PDG based
Category	Textual approach	Lexical approach	Syntactic approach	Semantic approach
Clone Type	Type-1	Type-1,2	Type-1,2,3	Type-1,2,3
Complexity	O(n)	O(n)	O(n)	$O(n^3)$
Meaning of n	Lines of code	No. Of token	Nodes of AST	Node of PDG

6.Clone Detection Tools

In this section, we list the different clone detection tools available in the literature in a tabular form (however, there are several others). Table 12 shows the tool details where the first column represents the tool name, 2nd column refers the citations for that tool, the 3rd column indicates the languages currently supported by the tool, the 4th column shows whether the tool is a clone detection tools or plagiarism detection tool or designed for other reengineering task, the 5th column represents the approach used in developing the tool, the 6th column indicates whether the tool is for commercial or academic use, the 7th column shows the maximum input size used in validating the tool and the last and 8th column tells us whether the tool was empirically validated or not.

Tool	Citations	Sup. Lang.	Domain	Approach	B.Ground	L.Input	Validated?
Dup	Baker [14, 18]	c, c++, Java	CD/Unix	line-based/text-based	academic	1.1M LOC	With two systems
JPlag	Prechelt et al. [192]	Java, c, c++, Scheme, NL text	PD/Online	Token/Greedy String	Academic	236 LOC	Student assign-ments/artificial data
CloneDr	Baxter et al. [31, 30]	c, c++, Java, COBOL, Others with DMS domain	CD Win-downNT	AST/Tree Matching	Commercial	400K LOC, C Code	Process Control System
DupLoc	Ducasse et al. [74, 72]	Language Indepen-dent/ 45 Mins to adapt	CD/Visual Works2.5	Line/Exact string match-ing	Academic	46K LOC	With 4 sys-tems of dif-ferent lan-guages
CCFinder	Kamiya et al. [122]	C, C++, Java, COBOL and other with lexical analyzer and trans-formation rules	CD/ Win-dows/ NT	Transformation /Token comp. with suffix tree	Academic	10M LOC	With 4 sys-tems
CP-Miner	Li et al. [168, 169]	C, C++	CD & Copy-pasted bugs de-tection /Win-dows/ Linux	Sequence Database/Frequent subsequent mining	Academic	4365K LOC	Several sys-tems
Sim	Gitchell et al. [90]	C	PD/Linux	Parse tree to string / String alignment	Academic	3.5K bytes	With 65 student as-signments
Covet/CLAN	Mayrand [178]	C, Others supported by Datrix	CD	Metrics from Datrix, 4 Points of comp., Ordi-nal scale of 8 cloning level	Academic	507K LOC	With two telecom-munication systems
DiLucca Pro.	Di Lucca et al. [67, 66]	HTML client & ASP server pages	Duplicated web-pages/PD	Sequence of tags/ Leven-shtein distance	Academic	331 files	With 3 web applica-tions
eMetrics	Fabio et al. [46, 161]	HTML & scripting languages	Visual inspec-tion of potential function clones	Gets potential function clones from eMetrics	Academic	403 files	Validated with 4 ap-plications

Continued on Next Page. . .

7.Conclusion:

We justify that code clone is a harmful in software development process. Code clone detection is a current issue in software development industry. The tools of code clone detection have to be integrated with standard IDEs. This paper mails focuses on describing actually what is code clone, Varity of code clones .we also describe the detection process and give the brief of Detection tools and Techniques .I conclude that this paper may serve as a Roadmap to potential users of code detection techniques .It may help them in selecting the right tool or technique.

8. Acknowledgement:

I would like to thank with deep sense of gratitude and respect to my Project Guide **Mr. Sanjeev Kumar Shukla , Kanpur Institute of Technology, Kanpur** for his Valuable suggestions ,guidance and constant encouragement during the paper writing.

I am very much thankful to the College Management and the **Director Prof(Dr) Brajesh Varshney** of the Institute for the help they provided me during the writing the content of this paper .

I would also like to give special thanks to **Mr. Ayush Mishra** for his help and support for writing the paper.

I am also thankful to my Family and friends for their true blessings and encouragement for the completion of paper.

References

[1] A. Aiken. A system for detecting software plagiarism (moss homepage). URL http: //www.cs.berkeley.edu/aiken/moss.html. 2002.

[2] Raihan Al-Ekram, Cory Kapser, Michael Godfrey. Cloning by Accident: An Empir-ical Study of Source Code Cloning Across Software Systems.*International Symposium on Empirical Software Engineering (ISESE'05)*, pp. 376-385, Noosa Heads, Australia, November 2005.

[3] Brenda S. Baker. A Program for Identifying Duplicated Code. In *Proceedings of Computing Science and Statistics: 24th Symposium on the Interface*, Vol. 24:4957, March 1992.

[4] 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.

[5] James Cordy, Thomas Dean, Nikita Synytskyy. Practical Language-Independent Detection of Near-Miss. In *Proceedings of the 14th IBM Centre for Advanced Studies Conference (CASCON'04)*, pp. 1 - 12, Toronto, Ontario, Canada, October 2004.

[6] Neil Davey, Paul Barson, Simon Field, Ray J Frank. The Development of a Software Clone Detector. *International Journal of Applied Software Technology*, Vol. 1(3/4):219-236, 1995

[7] A.van Deursen, T. Kuipers. Building Documentation Generators. In *Proceedings of International Conference on Software Maintenance (ICSM'99)*, Oxford, England, UK, August 1999.

[8] Stephane Ducasse, Matthias Rieger, Serge Demeyer. A Language Independent Ap-proach for Detecting Duplicated Code. In *Proceedings of the 15th International Confer-ence on Software Maintenance (ICSM'99)*, pp. 109-118, Oxford, England, September 1999.

[9] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *Transactions on Software Engineering*, Vol. 28(7): 654- 670, July 2002.

[10] Raghavan Komondoor and Susan Horwitz. Tool demonstration: Finding duplicated code using program dependences. In *Proceedings of the European Symposium on Pro-gramming (ESOP'01)*, Vol. LNCS 2028, pp. 383386, Genova, Italy, April 2001.

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

[12] S. Rao Kosaraju. Faster algorithms for the construction of parameterized su–x trees. In *In Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS95)*, pp. 631638, October 1995.

[13] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceed-ings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pp. 301-309, Stuttgart, Germany, October 2001.

[14] Andrian Marcus and Jonathan I. Maletic. Identiflcation of high-level concept clones in source code.In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pp. 107-114, San Diego, CA, USA, November 2001.

[15] Jean Mayrand, Claude Leblanc, Ettore Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceeding the 12th International Conference on Software Maintenance (ICSM'96)*, pp. 244-253, Monterey, CA, USA, November 1996.

[16] E. McCreight. A space-economical su–x tree construction algorithm. In *Journal of the ACM*, 32(2):262272, April 1976.

[17] S.M. Nasehi, G.R. Sotudeh, and M. Gomrokchi. Source Code Enhancement using Reduction of Duplicated Code. In *Proceedings of the 25th IASTED International Multi-Conference*, pp. 192-197, Innsbruck, Austria, February 2007.

[18] V. Wahler, D. Seipel, Jurgen Wolfi von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings of the 4th IEEE Inter-national Workshop Source Code Analysis and Manipulation (SCAM'04)*, pp. 128135, Chicago, IL, USA, September 2004.

[19] Wuu Yang. Identifying syntactic difierences between two programs. In *SoftwarePrac-tice and Experience*, 21(7):739755, July 1991.

[20] Definition of code clone used in intro:Cory J. Kapser:University of Waterloo, Ontario, Canada, 2009

[21] Chanchal Kumar Roy and James R. Cordy, A Survey on Software Clone DetectionResearch:In proceedings of Technical Report No. 2007-541 School of Computing Queen's University at Kingston Ontario, Canada, September 26, 2007

[22] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pp. 1322, Stuttgart, Germany, October 2001.