Method-level Code Clone Detection for Java through Hybrid Approach

Egambaram Kodhai[1] and Selvadurai Kanmani[2]
[1]Department of Computer Science and Engineering, Pondicherry Engineering College, India
[2]Department of Information Technology, Pondicherry Engineering College, India

Abstract*:* *A Software clone is an active research area where several researchers have investigated techniques to automatically detect duplicated code in programs. However their researches have limitations either in finding the structural or functional clones. Moreover, all these techniques detected only the first three types of clones. In this paper, we propose a hybrid approach combining metric-based approach with textual analysis of the source code for the detection of both syntactical and functional clones in a given Java source code. This proposal is also used to detect all four types of clones. The detection process makes use of a set of metrics calculated for each type of clones. A tool named CloneManager is developed based on this method in Java for high portability and platform-independency. The various types of clones detected by the tool are classified and clustered as clone clusters. The tool is also tested with seven existing open source projects developed in Java and compared with the existing approaches.*

Keywords: *Clone Detection, Functional Clones, Source code metrics, String-matching.*

Received October 21, 2013; accepted June 24, 2014

1. Introduction

Copying code fragments and then re-use by pasting with or without minor modifications or adaptations is called *Code Cloning* and the pasted code fragment is called a "clone". Clone detection is a research problem where there is no precise definition. Code clones are the source of heated debates among software maintenance researchers [3].

Clones are compared on the basis of the program text that has been copied. A related definition of cloning was described by Bellon *et al*., who defined the types of code clones based on the degree and type of similarities [1].

- *Type 1.* is an exact copy without modification (except for whitespace and comments)
- *Type 2.* is a syntactically identical copy; only variable, type, or function identifiers have been changed
- *Type 3.* is copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments
- *Type 4.* Two or more code fragments that perform the same computation but implemented through different syntactic variants

The granularity of clones can be free with no syntactic boundaries or fixed within predefined syntactic boundaries such as method or blocks. Clone granularity is fixed at different levels such as files, classes, functions/methods, begin-end blocks, statements or sequences of source lines.

In the literature there are number of clone detection techniques has been proposed with free granularity. Only limited detectors used function clones as granularity. The techniques that return only function clones are useful for architectural refactoring [9]. Moreover, function clones are the meaningful clones which are more useful for software maintenance and evolution phases [19].

In this paper, we propose a code clone detection method through hybrid approach. It is the combination of textual analysis using metrics to detect all the four types of clones. We also implemented a tool in Java using this approach. Our tool, detects function clones found in either the given Java source code projects at method level efficiently and accurately.

This paper is divided into four major sections. Section 2 presents the literature review for clone detection. Section 3 describes the implementation of the proposed method. Section 4 discusses the results obtained using our proposed method. Finally, section 5 describes the conclusion of the paper.

2. Motivation for Clone Detection

There has been more than a decade of research in the field of software clones. To understand the growth and trends in different dimensions of clone research, the research has been carried out with a quantitative review of related publications. In literature, Bellon *et al*. [1] has classified and defined four types of clones. A number of techniques have been proposed for the detection of type-1, type-2, and type-3 clones as per the definition of clone literature. However, for type-4 clones called semantic clones, very few attempts were made with limitations to detect them [10, 17]. So far, there is a lack of technique for the detection of all four types of clones in literature.

Table 1. An example for the four types of clones.

Source code(a)	**Type 1 clone(b)**	**Type 2 clone(c)**	**Type 3 clone(d)**	**Type 4 clone(e)**

int main()
{
int x = 1;
int y = x + 5;
return y;
}
	int main()
	{
	int x = 1;
	int y = x + 5;
	return y; // output
	}
		int func2()
		{
		int p = 1;
		int q = p + 5;
		return q;
		}
			int main()
			{
			int s = 1;
			int t = s + 5;
			t/++s;
			return t;
			}
				int func4()
				{
				int n= 5;
				return ++n;
				}

In this paper, we propose a code clone detection method through hybrid approach. It is the combination of textual analysis using metrics to detect all the four types of clones. We also implemented a tool in Java using this approach. Our tool, detects function clones found in either the given Java source code projects at method level efficiently and accurately.

This paper is divided into further into four major sections. Section 3 presents the literature review for clone detection. Section 4 describes the implementation of the proposed method. Section 5 discusses the results obtained using our proposed method. Finally, section 6 describes the conclusion of the paper.

3. Literature Review

Code cloning or the act of copying code fragments and making minor alterations is a well-known problem leading to duplicated code fragments or clones [10, 15]. Of course, the normal functioning of the system is not affected, but without counter measures, further development may become prohibitively expensive [4, 5].

Effective code clone detection will support for the perfective maintenance. Hariharan S in his paper identified some key parameters that would help to identify plagiarism [13]. Up to the present, several code clone detection methods have been proposed [2, 14, 18, 23, 25]. Several clone detection methods have used the Abstract Syntax Tree (AST) representation of a program to find clones [7, 8,12]. Generally, a clone detection tool uses an AST that is generated by a pre-existing parser.

Baker [11] describes one of the earliest applications of suffix trees to the clone detection process. In this work, instead of AST nodes, a token-like structure produced after the lexical analysis is used to find duplicates. The use of biological sequence matching algorithms is evident in [12]. It uses string alignment algorithm that inspired by dynamic programming methods. These methods are useful in the detection of near exact clones.

Godfrey and Zou [11] chose cyclomatic complexity as the corroboration metric. On a very small test set they have shown this approach can work for locating the clone segments across several versions of a software system. Thummalapenta *et al*. [25] indicated that in most of the cases clones are changed consistently and for the remaining inconsistently changed cases, clones mainly undergo independent evolution.

Ducasse *et al*. describe a clone detection algorithm with two steps [6]. The first step is to transform the code. Further normalization was considered by Ducasse *et al*. They found that these forms of normalization dropped precision from 94% to 70% in one case study and from 42% to 11.5% in another. This normalization improved recall by as much as 20%.

Merlo *et al.* [18] proposed a technique to detect function clones. He identified type-1 and type-2 clones. He maintains a high precision and a low recall. His tool did not detect type-3 and type-4 clones.

Chanchal K. Roy *et al.* [22] proposed a technique to detect function clones. However, he did not classify the clone types 1,2 or 3 as specified in the literature. Instead of that, the tool fixed some threshold value. If the threshold value is 0.0 then exact match (type-1) and it starts matches with threshold value 0.10, 0.20, 0.30. It means 10%, 20%, 30% of dissimilarity in the clones. It is able to detect near-missed clones (type-3) but fails to detect type2 clones.

The limitations in existing methods show a way to investigate hybrid or combinational techniques to overcome them. Our proposal is the detection of function clones using textual analysis and metrics approach. It also detects all four types of clones as specified in the literature.

4. Implementation of Clone Detection

A method is proposed to detect function code clones in Java source codes through textual analysis and metrics. It is implemented in Java. The tool accepts a Java source project as the input and identifies various functions/methods present in it. Then a built-in hand-coded parser [24] is used to analyze the various methods following an island-driven parsing approach [24]. Having identified the methods, different source code metrics are computed for each method and stored in the database. With the help of these metric values the possible potential clone pairs are extracted and are further put forth for the textual comparison.

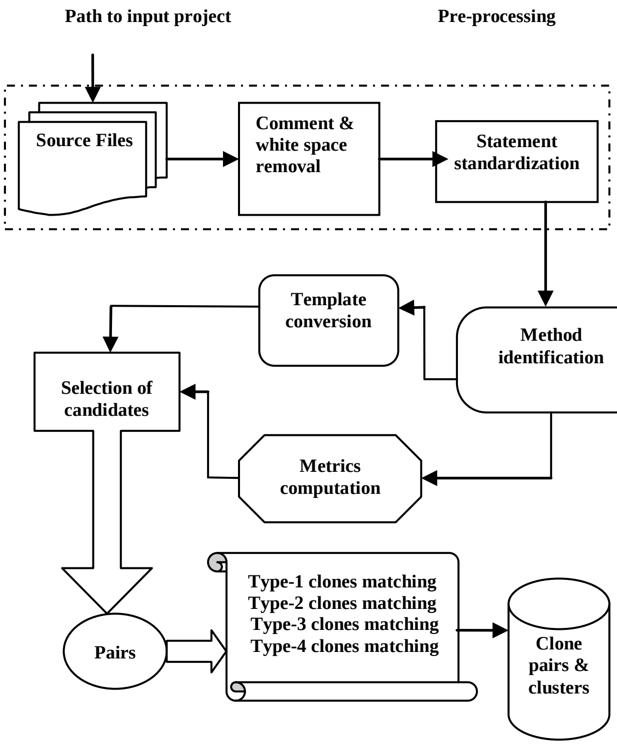
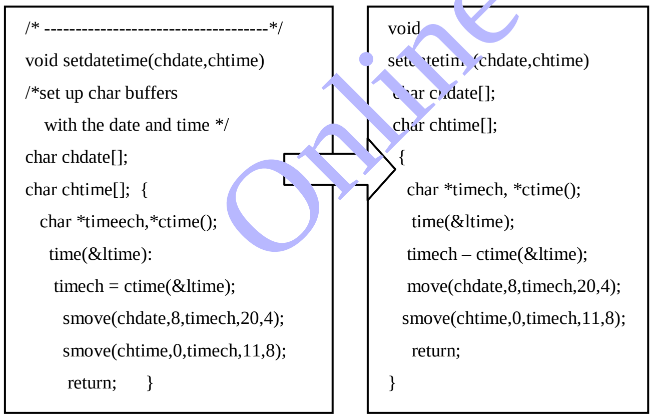


Figure 1. Overall Block Diagram

In the following subsections, we explain the design of the tool using the proposed method for the detection of four types of clones. The detection tool is thus lightweight i.e., it doesn't employ any external parsers and requires a less overhead compared to other methods.
The detection process is carried out in three major steps: A pre-processing, detection and post-processing. Figure 1 is the overall block diagram of the proposed system.



```
/* ------------------------------------*/
void setdatetime(chdate,chtime)
/*set up char buffers
   with the date and time */
char chdate[];
char chtime[];  {
   char *timeech,*ctime();
     time(&ltime):
      timech = ctime(&ltime);
        smove(chdate,8,timech,20,4);
        smove(chtime,0,timech,11,8);
          return;      }
```

```
void
setdatetime(chdate,chtime)
 char chdate[];
 char chtime[];
 {
    char *timech, *ctime();
     time(&ltime);
    timech – ctime(&ltime);
    move(chdate,8,timech,20,4);
   smove(chtime,0,timech,11,8);
     return;
 }
```

Figure 2. Comment and White Space Removal and standardized source code for pre-processing

4.1. Pre-Processing

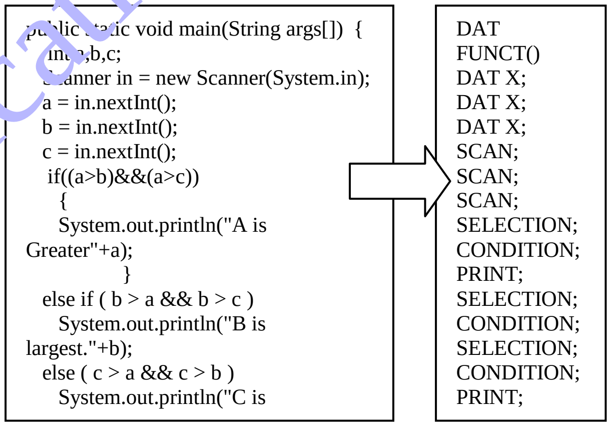
This phase includes the processes of comment and white space removal and source code standardization. In this step all the files are scanned for the removal of comments, whitespaces. The final step is re-structuring of the code into a standard form which is important for establishing clone fragments similarity [6]. These steps help in identification of the cloned methods thus yielding a significant gain in the recall. Figure 2 illustrates the comment and white space removal and statement standardization for pre-processing phase.

4.2. Template Conversion

This Template conversion converts the original source code into a new form having a uniform notation for the permitted equivalent constructs between the clone pairs of same type. In this tool we have employed variant part for the purpose of detection of type-2, type-3 and type-4 clones.

4.2.1. Template conversion for Type-1 and Type-2

For type 2 as per the definition of literature the function identifiers, variable names, types etc., are the only allowed difference in functions. Hence to minimize the differences between the code fragments due to the editing activities of the programmer we bring out a uniform intermediate representation of the source code. Figure 3 shows a sample template conversion for type1 and type2.



```
public static void main(String args[])  {
  int a,b,c;
  Scanner in = new Scanner(System.in);
  a = in.nextInt();
  b = in.nextInt();
  c = in.nextInt();
   if((a>b)&&(a>c))
     {
      System.out.println("A is
Greater"+a);
                }
    else if ( b > a && b > c )
      System.out.println("B is
largest."+b);
    else ( c > a && c > b )
      System.out.println("C is
```

```
DAT
FUNCT()
DAT X;
DAT X;
DAT X;
SCAN;
SCAN;
SCAN;
SELECTION;
CONDITION;
PRINT;
SELECTION;
CONDITION;
SELECTION;
CONDITION;
PRINT;
```

Figure 3. Template Conversion for type 1 and type 2

4.2.2. Template conversion for Type-1 and Type-2

In type-3 and type-4 clone detection, various constructs like iterations and branches may also change between clone methods. A slightly different form of representation is needed to be generated. Thus the following representations help in generalizing the various deviations and constructs and in identifying the various types of cloned methods.

- *Iterative equivalence*: The control looping structures are *for*, *while* and *do while*. In looping statements, the three patterns present in looping are initialization, condition and increment/decrement are separated and they are printed each in separate line. The common template form *iteration* is used. Both open braces and close braces are neglected in printing due to the change in order and nested statements in the source code.

Table 2. Type of variants among the source code patterns

S.No	Name of the pattern	Possible variations in the source code presentation	Proposed template form

1	Iterative equivalence	for
		while
		do-while
			iteration
			<initial>
			<condition>
			<incre/decre>

2	Conditional equivalence	if
		else
		else-if
			selection
			<condition>

3	Input equivalence	system.in input.readline	read <variable>
4	Output equivalence	system.out	write <variable>
5	Declaration equivalence	int char float double string <u>Example</u> int x,y,z char c,s	Multiple declaration To Single line declaration <u>Example</u> int x int y int z char c char s
6	Braces	{ }	Braces are removed in the code

- *Conditional equivalence*: The conditional structures are *if, else* and *else if*. In these nested statements, the conditions are separately printed in new line following the template form *selection*. The operations are split separately and rewritten in each new line.
- *Input equivalence*: The input statements are *system.in, input.readline*, etc. In these statements, the variable alone will follow the template form *read*. For the multiple inputs which are given in a single input statement are separately printed in each line.
- *Output equivalence*: The output statements are system.out, etc. In these statements, the output variables alone are printed following the template form *write*. The print statements which are just printing any comments or statements are neglected. Also the multiple outputs which are printed in a single print statement are separately printed in each line.
- *Declaration equivalence*: The declarations statements starts with keywords such as *char, int, long int, double, float, string,* etc. In this case, multiple declarations in a single statement are split and reprinted in each line as a single declaration statement. The table 2 shows the conversion of multiple declarations into single declaration.

4.3. Method Detection

The standard form of source code is scanned for detecting various methods by adopting an 'island-driven parsing'[24] approach and the method definitions are extracted and collected by means of a hand-coded parser and saved for further reference. The end positions of the method and the total no. of lines in each method are also noted.

4.4. Metrics Computation

A set of 12 count metrics are proposed for the detection of these cloned methods. Metric sets are proposed for each type of cloned methods based on the necessity. They are as shown in the Table 3.

Table 3. Metrics applied to methods

S.No	Metrics
1	No. of Lines
2	No. of Arguments
3	No. of Local Variables
4	No. of function Calls
5	No. of conditional statements
6	No. of iteration statements
7	No. of Return Statements
8	No. of Input Statements
9	No. of Output Statements
10	No. of Assignments from Function Calls
11	No. of Selection Statements
12	No. of Assignment Statements

Apart from the above 12 count metrics 4 more metrics as shown in equations 1 to 4 are also used. The features examined for these metric computations are, Global and local variables defined or used, Functions called, Files accessed, I/O operations and defined/used parameters passed by reference and by value.
Let S be a code fragment. The description of the four metrics used is given below. A detailed description is given in [38-40]. Note that these metrics are computed compositionally from statements, to functions and methods.

$$13.\ \text{S COMPLEXITY}(s) = \text{FAN OUT}(s) \qquad (1)$$

Where FAN OUT(s) is the number of individual function calls made within s.

$$14.\ \text{D COMPLEXITY}(s)=\text{GLOBALS}(s)/(\text{FAN OUT}(s)+1) \qquad (2)$$

Where GLOBALS(s) is the number of individual declarations of global variables used or updated within s. A global variable is a variable which is not declared in the code fragment *s*.

$$15.\ \text{MCCABE}(s) = 1 + d, \qquad (3)$$

where d is the number of control decision statements in *s*.

$$16.\ \text{ALBRECHT}(s)=\left\{\begin{array}{l}\text{p1*VARSUSEDANDSET}(s)+\\ \text{p2*GLOBALVARSSET}(s)+\\ \quad\text{p3* USER INPUT}(s)\end{array}\right. \qquad (4)$$

p4* FILE INPUT(s)

Where VARSUSEDANDSET(s) is the number of data elements set and used in the statement s,
GLOBALVARSSET(s) is the number of global data elements set in the statement s,
USERINPUT(s) is the number of read operations in statement s,
FILEINPUT(s) is the number of files accessed for reading in s.
The factors p1, .., p4, are weight factors. The values chosen are p1 = 5, p2 = 4, p3 = 4 and p4 = 7. The values are chosen as given in the literature [1].
The computed metric's values for each method are stored for comparison and extraction processes. For type-1, type-2 and type-4 we pose a constraint that a cloned method pair must have an identical set of metric's values. Thus the database records containing identical metric's values are short-listed for the type-1 and type-2 clone detection. The metric's are computed for each of the methods and are compared to be short-listed by the formulas. Table 1 gives the list of metrics used for the detection of clones.

4.5. Clone Detection

With the short-listed set of methods, a textual comparison of the method pairs in the formatted and normalized code is done to identify the exactness of the extracted pairs. The detection method used for the identification of the clone types are tabulated in Table 4. The comparison in the template identifies type-1 cloned method along with type-2 cloned methods. So they need to be listed separately. For this reason textual comparison with original source code is made to identify the differences in the parameters.

Table 4. Criteria for Clones Type detection.

Clone Type	Standardized Source Code		Template Code
	Metrics Comparison	Textual Comparison	Template Comparison
Type 1	Same	Same	-
Type 2	Same	Difference in Parameters	Same
Type 3	Range 1>= 90%	-	Range 2>= 85%
Type 4	Same	No match	Same

Fortype -3 clone detection, Range values are calculated. Range1 is the ratio of the actual metric value to the Average metric values in the methods.
i.e.,

$$range1=\left[\frac{Actualmetricvalueofamethod * 100}{Averagemetricvaluesofthemethods}\right] \quad (5)$$

If any method having more than 90% value for range1, they are short-listed under the possibilities for type-3 method clones. Then range2 is calculated as the ratio of equal no of lines in a method to the total no of lines in a method.

i.e.,

$$range2=\left[\frac{No.ofsimilarline\sin amethod * 100}{Tota\ln o.ofline\sin amethod}\right] \quad (6)$$

The methods having more than 85% values of range2 in template methods are declared as type-3 clones.

For type-4, first the two considered methods are taken and their metric values are calculated. If the two methods are having all its metrics values equal then they are compared with the template methods. If they are also the same then the textual comparison of the source code is checked. If they are completely different then they are categorized under Type-4.

4.6. Post-processing

The results of the code clone detection are given as clone pairs and clone clusters. The identified clone methods called "potential clone pairs", are then clustered separately for each type and the clustered separately for each type and the clusters are uniquely numbered. The association of similar pairs into a single group called a cluster or a class. Each clone cluster may be defined as a unique set of methods that are similar within themselves. These clone pairs and clusters are stored each in a text file separately.

5. Experiment and Results

In this experiment we have applied CloneManager to find function clones in a number of open source systems. We have then used a set of metrics to analyze the results. We manually verify all detected clones and provide a complete catalogue of different clones in a variety of formats. This section introduces the systems we have studied and the metrics used, including a brief overview of our definition and methodology for manual verification of the detected clones.

5.1. Experimental Setup and Datasets

The proposed method is implemented and experimented with seven Java Projects. Table 3 lists a statistical overview of open source projects which are taken for the performance analysis of our CloneManager tool. We have only considered *.Java* files in the calculations. All clones detected in this study were validated by hand.
In Table 5, the second column is the list of open source project names as input project. The third column is the no. of files. The fourth column is the no. of lines in the source code in thousands. The last column is the no. of methods in each project.

Table 5. Projects chosen as dataset for CloneManager

S.No	Input projects	#files	LOC in K	# methods
1	Eclipse-ant	161	35	1754
2	EIRC	54	11	588
3	Java Netbeans-Javadoc	97	14	972
4	Eclipse-jdtcore	582	148	7383

5	JHotDraw 5.4b1	233	40	2399
6	Spule	50	13	420
7	J2sdk-swing	414	204	10971

The effectiveness of clone detection by any tool is basically measured by two key parameters namely,

- *Recall*: Fraction of actual clones identified as candidates
- *Precision*: Fraction of candidates that are actually clones

5.2. Results

In Table 6, the third column is the clone type-1, which has the no. of clones detected and the no. of clone clusters. Column 4, 5 and 6 has the same set of data for type 2, 3 and 4 respectively.

From the Table 6 results we observed that J2sdk-swing with only 204,000 of lines have 27559 clones in total. This shows that the no. lines are not directly propositional to the no. of clones in the code.

We can notice that there is significantly more function cloning in our open source Java. On average, about 15% of the methods in open source Java programs are type-1 clones—those with no changes at all (except changes in formatting, whitespace and comments). After detecting clones we noticed this in large part due to the large number of small accessor and iterator methods in Java programs.

When we plot the percentage of type-1 clones, we can see that Java show similar percentages of clones for similar clone sizes. While it is difficult to provide the exact statistics for the types of smaller methods for all the systems, we manually examined the small clones of the systems and found that there are in fact many accessor methods in Java systems.

It is interesting to notice that most systems have significantly fewer clone classes than clone pairs, indicating the fact that there are many pairs of functions in the systems that are similar to each other with higher numbers for Java systems. It is also interesting to see that while average number of clone pairs per clone class is more or less consistent for Java systems for different clone types.

5.3. Evaluation of CloneManager tool with Parameters

In comparison with a reference set obtained from the standard set of results gathered from the other detection tools the precision (PREC) and recall (REC) of the tool for all 4 type of clones has been estimated as in Tables 7,8, 9 and 10.

The table 7 shows the precision and recall of type1 clones for all the projects. The column 2 holds [A] the number of actual clones detected for all the datasets. The Column 3 holds [D] the number of detected clones by our tool CloneManager. The column 5 holds [C] the number of correctly detected clones by our tool. These values are used to calculate the two parameters precision and recall for evaluation. The formula to calculate Precision=[C]/[D]*100 and Recall=[C]/[A]*100.

From the above calculated values for precision and recall as shown in Figures 4 and 5, we come to know that our system shows high values in precision and recall. Thus our tool proves to provide high in precision and recall, which are the best parameters for the evaluation of clone detection tools. Finally, we are able to get results for the J2sdk-swing system also which is larger in size. This proves that our system is also scalable.

Table 6. No. of detected clones and clone clusters for all the datasets.

S. No	Projects	Type-1		Type-2		Type-3		Type-4	
		Clones	Clone Clusters	Clones	Clone Clusters	Clones	Clone Clusters	Clones	Clone Clusters
1	eclipse-ant	363	[illegible]	372	96	426	119	10	4
2	EIRC	117	35	119	35	149	47	6	3
3	Java Netbeans-Javadoc	193	80	199	83	304	110	8	3
4	eclipse-jdtcore	1427	[illegible]	5573	587	4378	660	15	7
5	JHotDraw 5.4b1	291	[illegible]	299	142	598	208	10	4
6	spule	60	11	69	14	113	19	4	2
7	j2sdk-swing	[illegible]	516	8205	558	11209	843	30	14

Table 7. Precision and recall of type-1 clones for all the projects.

Project	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision %	Recall %
Eclipse-ant	382	374	363	97	95
EIRC	124	117	117	100	94
Java Netbeans-Javadoc	196	205	193	94	98
Eclipse-jdtcore	1603	1585	1427	90	89
JHotDraw 5.4b1	303	296	291	98	96
Spule	61	60	60	100	98

Table 8. Precision and recall of type-2 clones for all the projects.

Project	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision %	Recall %
Eclipse-ant	448	426	426	100	95
EIRC	161	152	149	98	92
Java Netbeans-Javadoc	304	330	304	92	100
Eclipse-jdtcore	4864	4378	4378	100	90
JHotDraw 5.4b1	643	629	598	95	93
Spule	126	113	113	100	89

J2sdk-swing	12052	12737	11209	88	93

Table 9. Precision and recall of type-3 clones for all the projects

Project	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision %	Recall %
Eclipse-ant	10	10	10	100	100
EIRC	6	6	6	100	100
Java Netbeans-Javadoc	9	8	8	100	88
Eclipse-jdtcore	17	17	15	88	88
JHotDraw 5.4b1	11	11	10	90	90
Spule	4	4	4	100	100
J2sdk-swing	31	32	30	92	95

Table 10. Precision and recall of type-4 clones for all the projects

Project	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision %	Recall %
Eclipse-ant	379	422	372	88	98
EIRC	126	132	119	90	94
Java Netbeans-Javadoc	207	199	199	100	96
Eclipse-jdtcore	6057	5686	5573	98	92
JHotDraw 5.4b1	321	299	299	100	93
Spule	71	73	69	94	96

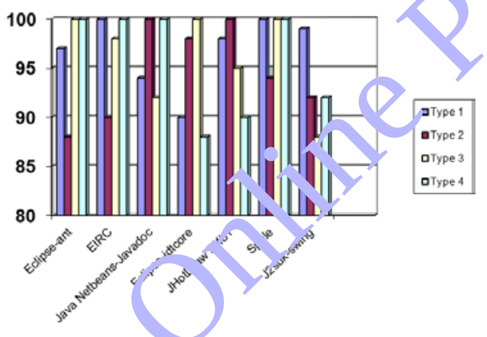


Figure 4. Precision in % for all the projects

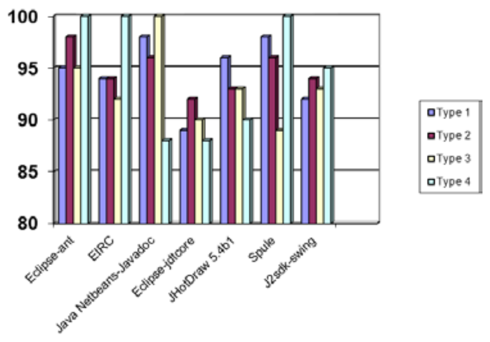


Figure 5. Recall in % for all the projects

5.4. Comparison with existing tools

The first tool considered for analysis is the CLAN clone detection with metrics based clone detection technique developed by Merlo [18] with the method-level granularity. The second is NICAD [22], a parser-based, language specific, lightweight approach using simple text -line comparison which finds functional clones with the aid of TXL. Even though there are number of tools developed for clone detection, we chose only these two existing tools because they detect the functional clones as our CloneManager tool does.
In case of Eclipse-ant we have obtained 1171 clone pairs for type 1,2, 3 altogether using our standardization and normalization techniques while Merlo has obtained only 88 match clone fragments. Moreover we have also classified the clones as clone clusters and detected the type 4 clones. The results obtained by these tools are computed as in the Table 11. NICAD having obtained 1154 of clone fragments.

Table 11. Clone Fragments and Clone Clusters for Eclipse-ant

TYPE	CLAN	Nicad		CloneManger	
	CF	CF	CC	CF	CC
Type1	10	363	92	363	92
Type2	54	365	94	372	96
Type3	24	426	119	426	119
Type4	-	-	-	10	4
Total	88	1154	305	1171	311

Nicad tool did not classify the clones types 1,2 or 3 as specified in the literature. Instead of that, the tool fixed some threshold value. If the threshold value is 0.0 then exact matches (type-1) and it starts matches with threshold value 0.10, 0.20, 0.30. It means 10%, 20%, 30% of dissimilarity in the clones. It is able to detect near-missed clones (type-3) but fails to detect type-2 clones. We have compared the results of all the projects with these two existing tools like weltab.

Table 12. Comparison of run-time with NICAD

Projects	NICAD in minutes	CloneManager in minutes
Eclipse-ant	1.57	1.35
Java Netbeans-Javadoc	0.42	0.38
Eclipse-jdtcore	17.43	16.02
JHotDraw 5.4b1	2.48	2.05
J2sdk-swing	35.24	30.37

From the Table 12 we compared the run-time of our with the NICAD tool. Second and third column shows the results for time taken by NICAD in minutes and by our tool CloneManager respectively. It is easier to notice from the table that the time taken by our tool is lesser than NICAD. Thus our tool proves to have good time complexity.

Table 13 shows the comparison of the precision and recall parameters of the tool CLAN with our tool CloneManager. We have taken only the projects which have precision and recall data from the standard bench bellon *et al*. Moreover, the data was available for type-1, 2 and 3 alone. From the table we observed that our tool is very high in precision and recall.

Table 13. Comparison of the tool CLAN with the tool CloneManager

Projects	**CLAN**						**CloneManager**					
	Precision %			**Recall %**			**Precision %**			**Recall %**		
	T1	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**
Eclipse-ant	11	9	0	5	20	0	97	88	100	95	98	95
Java Netbeans-Javadoc	7	6	6	33	9	13	94	100	92	98	96	100
Eclipse-jdtcore	4	4	0.8	4	53	12	90	98	100	89	92	90
J2sdk-swing	7	7	0.2	69	25	1	99	92	88	92	94	93

6. Conclusions

In this paper we have proposed a light-weight technique to detect method-level clones for both textual similarity and functional similarity types with the computation of metrics combined with simple textual analysis technique. We could improve the precision and reducing the total comparison cost by avoiding the exponential rate of comparison by using the metrics. Since the string matching/textual comparison is performed over the short-listed candidates, a higher amount of recall could be obtained. The early experiments prove that this method can do atleast as well as the existing systems in finding and classifying the function clones in Java.

As a future work we have planned to enhance the technique for web static pages. Secondly, we also planned to enhance the tool for clone modification by using the refactoring technique. Finally, we have planned to detect the clones in incremental process for next revision of projects.

References

[1] Bellon.S., “Detection of software clones — tool comparison experiment”, *http://www.bauhaus-stuttgart.de/clones, 2009*.

[2] Bellon.S, Koschke.R, Antoniol.G, Krinke.J, and Merlo.E, “Comparison and Evaluation of Clone Detection Tools” *IEEE Transactions on Software Engineering*, Vol. 33, no.9, pp. 577-591, September 2007.

[3] Bettenburg.N, Shang.W., Ibrahim.M., Adams. B., Zou.Y., Hassan.E., “An empirical study on inconsistent changes to code clones at the release level. Science of Computer Programming, vol.77, pp. 760-776, 2012.

[4] Cataldo.M, Mockus.A, Roberts.A, Herbsleb.D, “Software Dependencies, Work Dependencies and Their Impact on Failure,” *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864-878, November/December 2009.

[5] Demeyer.S., Ducasse.S. and Nierstrasz.O., “Object-Oriented Reengineering Patterns”, *Morgan Kaufmann and DPunkt*, 2002.

[6] Ducasse.S., Nierstrasz.O. and Rieger.M., “On the effectiveness of clone detection by string matching”, *Journal on Software Maintenance and Evolution*, vol. 18, no.1, January 2006.

[7] Evans.S. , Fraser.W. , Ma.F., “Clone Detection via Structural Abstraction”, *Software Quality Journal*,Vol. 17, pp. 309-330, 2009.

[8] Evans. W. and Fraser. C., “Clone Detection via Structural Abstraction” *Technical Report MSR-TR 2005-104*, Microsoft Research, Redmond, WA, 2005.

[9] Fowler. M., *Refactoring: improving the design of existing code*, Addison Wesley, 1999.

[10] Gabel, M., Jiang, L. and Su, Z., “Scalable Detection of Semantic Clones”, *30th International Conference on Software Engineering, ICSE 2008*, pp. 321-330, 2008.

[11] Godfrey.W. and Zou.L., “Using origin analysis to detect merging and splitting of source code entities” *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[12] Greenan. K., “Method-Level Code Clone Detection on Transformed Abstract Syntax Trees using Sequence Matching Algorithms”, *Student Report*, University of California - Santa Cruz,2005.

[13] Hariharan. S., “Automatic Plagiarism Detection Using Similarity Analysis”, *The International Arab Journal of Information Technology*, Vol. 9, No. 4, July 2012. Pp. no. 322-326.

[14] Kamiya.T., Kusumoto.S. and Inoue.K., “CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code” *IEEE Computer Society Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[15] Kapser.C and Godfrey.W. ”Cloning considered harmful: Patterns of cloning in software”. *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[16] Kapser.J and Godfrey.W , “Supporting the analysis of clones in software systems: Research articles,” *Journal of Software Maintenance: Research and Practice*, vol. 18, no. 2, pp. 61–82, 2006.

[17] Liu. C., Chen.C., Han.J. and Yu.P., “GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis”, *12th ACM*

SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2006, pp. 872-881, 2006.

[18] Mayland.J., Leblanc.C. and Merlo.E., “Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics,” *International Conference of Software Engineering* 96, pp. 244- 253,1996.

[19] Nguyen.H, Nguyen.T, Pham. N.H., Al-Kofahi.J, Nguyen. T.N, “Clone Management for Evolving Software” *IEEE Transactions on Software Engineering*, 2011.

[20] Pate.J., Tairas. R. and Kraft.N., “Clone Evolution: a Systematic Review”, *Journal of Software Maintenance: Research and Practice*, Vol. 25, no.3, pp.261-283, 2013.

[21] Petersen, H., “Clone detection in Matlab Simulink models”, Master’s *thesis*, *Tech. Univ. Denmark*, 2012.

[22] Roy.C. and Cordy.J., "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," *The 16th IEEE International Conference on Program Comprehension*, pp.172-181, 2008.

[23] Roy.C., Cordy.J , Koschke.R , “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach” *Science of Computer Programming*, Vol.74, no.7, pp. 470–495, 2009.

[24] Satta.G. and Stock.O., “Bidirectional context-free grammar parsing for natural language process”, *Artificial Intelligence*, vol.69, pp. 123–164, 1994.

[25] Thummalapenta. S, Cerulo.L, Aversano.L, and Di Penta.M, “An empirical study on the maintenance of source code clones,” *Empirical Software Engineering*, vol.15, no.1, pp.1–34, 2009.

[26] Zibran. M. and Roy. C., “Conflict-aware Optimal Scheduling of Code Clone Refactoring”, *IET Software*, Vol. 7, no.3, pp. 167 – 186, 2013.

Egambaram Kodhai is currently working as Associate Professor in the Department of Computer Science and Engineering at Sri Manakula Vinayagar Engineering College affiliated to Pondicherry University, Puducherry, India. She has completed her M.C.A from Cauvery College for women, Trichy affiliated to Bharathidasan University, Trichy and M.E. in Computer Science and Engineering from Vinayaka Mission’s Kirupananda Variyar Engineering College, Salem. She has more than 15 years of experience in teaching in various engineering colleges. Her Research interests include Software Clones. She has published more than 40 papers in international conferences and journals.

Selvadurai Kanmani received her BE and ME degree in Computer Science and Engineering from Bharathiar University and PhD from Anna University, Chennai. She has been the faculty of the Department of Computer Science and Engineering, Pondicherry Engineering College since 1992. She has published about 150 papers in international conferences and journals. Her research interests are software engineering and data mining techniques. She is a member of Computer Society of India, ISTE and Institute of engineers, India.