Kodhai and Kanmani *Journal of Software Engineering Research and Development* 2014, **2**:12
www.jserd.com/content/2/1/12

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

Method-level code clone detection through LWH (Light Weight Hybrid) approach

Egambaram Kodhai[1*] and Selvadurai Kanmani[2]

* Correspondence:
kodhaiej@yahoo.co.in
[1]Research Scholar, Department of CSE, Pondicherry Engineering College, Puducherry, India
Full list of author information is available at the end of the article

Abstract

Background: Many researchers have investigated different techniques to automatically detect duplicate code in programs exceeding thousand lines of code. These techniques have limitations in finding either the structural or functional clones.

Methods: We propose a LWH (Light Weight Hybrid) approach combining textual analysis and metrics for the detection of method-level syntactic and semantic clones in C and Java projects. This approach has been experimenting for the detection of all four types of clones by a specific set of metrics assessment and textual comparison. A tool named CloneManager has been developed in Java to support the experiments carried out and to validate the proposed approach.

Results: A benchmark dataset widely referred in the literature and medium to large size open-source projects developed in C or Java. Java is used for the experiments.

Conclusions: The results show that the proposed approach is able to detect all four types of clones accurately with the precision and recall values ranging from 88% to 100%.

Keyword: Clone detection; Function clones; Source code metrics; String-matching

1 Introduction

Copying code fragments and then reusing them through the paste option with or without minor modification or adaptation is called “Code Cloning” and the pasted code fragment is called a “clone”. Most of the software systems comprise a substantial quantity of code clones; typically 10–15% of the source code in large software systems are part of single or more code clones (Kapser and Godfrey 2006).

In literature, (Bellon et al. 2007) 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 (Marcus and Maletic 2001; Komondoor and Horwitz 2001; Krinke 2001; Gabel et al. 2008; Liu et al. 2006). So far, there is a lack of technique for the detection of all four types of clones in literature.

Clones may be useful from different points of view (Kapser and Godfrey 2008). Clones carry important domain knowledge and thus studying clones may assist in understanding it (Pate et al. 2011). Moreover, the software clone research has promoted academic-industrial collaboration. Software Practitioners used to copy and modify the existing project's clones frequently to meet the needs of the clients and users in their new projects (Petersen 2012).

A number of clone detection techniques have been proposed in literature. Among them, Text-based techniques are lightweight and are able to detect accurate clones with higher recall values, where recall refers to the overall percentage of clones exist in the source code that have been detected by the clone detector. However, it failed to detect suitable syntactic units (Bellon et al. 2007). Token-based techniques are fast with high recall, but failed in precision. Precision refers to the quality of clones returned by the clone detector. Parser-based techniques are worthy in detecting syntactic clones. However, they give low recall values (Bellon et al. 2007). Metric-based techniques are able to detect syntactic as well as semantic clones with high precision values. They are also very fast in detecting both syntactic and semantic clones. However, they fail to detect some of the actual clones (Bellon et al. 2007). PDG (Program Dependency Graph) based techniques are able to find more semantic clones, where PDG is a directed graph which represents the dependencies among program elements in a program. However, sub-graph comparisons are very costly (Koschke et al. 2006). These limitations in existing methods provide a path to investigate hybrid or combinational techniques in order to overcome them.

Although numerous techniques and tools have been proposed for code clone detection (Kamiya et al. 2002), only little has been known about, which detected code clones are appropriate for refactoring and how to extract code clones for refactoring. A technique that helps to process the code clones is called Refactoring. Refactoring is defined as "restructuring an existing body of code, altering its internal structure without changing its external behaviour" (Fowler 1999). By refactoring the clones detected, one can potentially improve understandability, maintainability and extensibility and reduce the complexity of the system (Fowler 1999).

The granularity of clones can be free with no syntactic boundaries or fixed within predefined syntactic boundaries such as method or block (Roy and Cordy 2007). Clone granularity is fixed at different levels, such as files, classes, functions/methods, begin-end blocks, statements or sequences of source lines.

Clone detection techniques have been proposed with free granularity, mostly with more than six lines of code (Kamiya et al. 2002; Koschke et al. 2006). On the analysis of different clone detection techniques, most of the matches tend to be methods/functions of 1-5 lines of code. Most of these methods are setter/getter functions which are valid set of clones. Only limited detectors used function clones as granularity. Function/Method clones are simply clones that are restricted to refer to entire function or method. Function/Method clones appear to be the most promising points of refactoring for all clone types. They are larger and tend to have a significant amount of code in common.

The techniques that return only Function/Method level clones are suitable for architectural refactoring as they represent a meaningful code segment. It is not so in the case of detecting clones with fixed number of lines in a continuous unsegmented file of code. Tools have been proposed in the literature, which analyses these clones further to extract meaningful codes for refactoring support (Kapser and Godfrey 2006; Ueda et al. 2002; Zibran and Roy 2013). Function/Method clones are the meaningful clones which are also useful for software maintenance and evolution phases. Thus, it motivates researchers to fix the granularity as function/method level (Mayland et al. 1996; Roy and Cordy 2008).

In this paper, a LWH (Light Weight Hybrid) approach has been proposed with a combination of textual comparison and metrics computation. As there is no need for external parsing, this approach is of light weight. Moreover, a model has been arrived to detect syntactic and semantic clones which will cover all four types of clones. For experimental validation, a tool has been developed using the proposed LWH approach to detect method/function level clones for both C and Java projects. This tool has been developed in Java and it has been named as CloneManager. Experimental results show that, the proposed tool CloneManager is efficient and accurate in detecting all types of clones.

This paper is presented in five major sections. Section 2 discusses the literature review for clone detection. Section 3 introduces the basic definitions and background details of code clone detection. The detailed implementation of the proposed method as a tool is elaborated in Section 4. Section 5 summarizes the experimental results. Section 6 concludes the paper.

2 Literature review

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 cloning research, we carried out a quantitative review of related publications. Clone detection research has proved that software systems have 9%-17% of duplicated code (Zibran et al. 2011). (Thummalapenta et al. 2009) indicated that in most of the cases, clones are changed consistently and for the remaining inconsistently changed cases, clones undergo independent evolution. Effective code clone detection will support perfective maintenance. Up to the present, several code clone detection methods have been proposed (Petersen 2012; Al-Batran 2011; Leitner et al. 2013). Comparison and evaluation of code clone detection techniques and tools have been carried out by (Bellon and Koschke 2014; Bellon et al. 2007) and (Roy and Cordy 2007; Roy et al. 2009).

A clone detection process is usually done by converting the source code into another form that is handled by an algorithm to detect the clones. A rough classification is then carried out depending on the level of matches found. Token-based techniques (Li et al. 2006; Leitao 2004; Basit et al. 2007) use a similar sequence matching algorithm. However, its accuracy is not that adequate as the normalization, and also token conversion process may bring false positive clones in result set. Many of the clone detection approaches have used Abstract Syntax Tree (AST) and suffix tree representation of a program to find clones (Evans et al. 2009; Evans and Fraser 2005; Greenan 2005; Pate et al. 2011; Koschke 2012). Some of the clone detection techniques use an AST that is generated by a pre-existing parser. (Baker 1997) describes one of the earliest applications of suffix trees for the clone detection process. An algorithm based on feature-vector computation over AST was applied by Lee et al. (2010) to detect similar clones. However, all of them use parsing, which results in heavy-weighted approach.

Lighter weight techniques were proposed in the literature without the use of parsing namely text-based techniques and metrics-based techniques. Text-based techniques (Wettel and Marinescu 2005; Ducasse et al. 1999) are investigated by comparing two code fragments with each other to find longest common subsequences of same text/

strings to detect clones. Though these techniques detect clones they are not low in precision values. Metric-based techniques identify a set of suitable metrics to detect a particular type of clone. By a quantitative assessment of the metric values in the source code, the clone detection is done. (Kapser and Godfrey 2004) chaos Cyclomatic complexity as the corroboration metric. However, they have only proved that their technique works well to locate the clone segments across several versions of a software system using a very small test set.

Hybrid techniques were also proposed in the literature. (Marco Funaro et al. 2010) proposed a hybrid technique using Abstract Syntax Tree to identify clone candidates and textual methods to discard false positives. (Leitao 2004) also proposed a hybrid approach with the combination AST and PDG. Both approaches use parsing which results in heavy-weight. As text-based techniques preserve higher recall, metrics-based techniques preserve higher precision and both of them are light-weight, a hybrid technique with the combination of textual analysis and metrics, is experimenting in this paper for the detection of all four types of clones.

3 Background

Clones may be compared on the basis of the program text that has been copied. A related definition of cloning was described by (Bellon et al. 2007), who defined the types of code clones based on the degree and type of similarities.

Textual similarity

- **Type-1** is an exact copy without modifications (except for whitespace and comments).
- **Type-2** is a syntactically identical copy; except some changes in variable name, data type, identifier name, etc.
- **Type-3** is a copied fragment with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

Functional similarity

- **Type-4** Two or more code fragments that perform the same computation, but implemented through different syntactic variants.

Table 1 illustrates the four types of clones. The clone pair (a, b) is of type-1 which have exactly the same code except the alignment, space and comment. The clone pair (a, c) is of type-2 which have minor differences in function names and parameters. The clone pair (a, d) is of type-3 with additional statements in code, as they need not be

Table 1 Illustration of 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 = t/++s; return t; }
				int func4()
				{ int n = 5;
				return ++n; }

functionally similar. The clone pair (a, e) is of type-4 clones with no similarity in code, but the output of the functions are same.

The results of the code clone detection are presented as clone pairs and clone clusters.

- **Clone Pair (CP) or Code Fragment (CF):** pair of code portions/fragments that are identical or similar to each other.
- **Clone Cluster (CC) or Clone Class or Clone Set (CS):** the union of all clone pairs that have code portions in common.

The quality of clone detection by any tool is assessed by two key parameters precision and recall as defined in Figure 1.

Precision

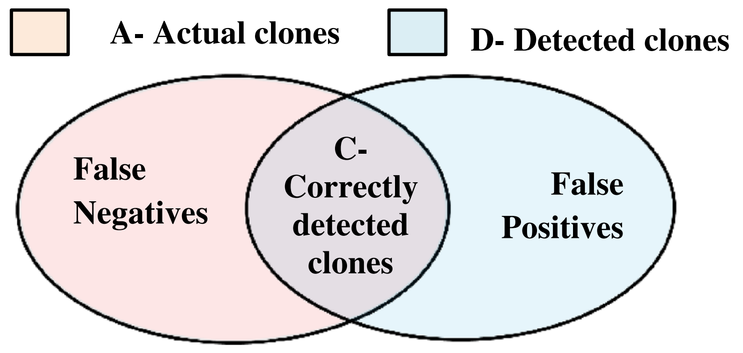
Precision is the ratio of the number of correctly detected clones to the total number of detecting clones by the proposed tool.

Recall

Recall is the ratio of the number of correctly detected clones by the proposed tool to the total number of actual clones in the project by reference values.

4 Methods

This section describes the proposed LWH approach for automatic detection of function clones in C or Java source code. A tool CloneManager has been developed in Java in order to experiment the proposed approach. This tool accepts a C or Java source project as the input and separates the functions/methods present in it. A built-in hand-coded parser (Moonen 2001) is used to process these methods following an island-driven parsing approach (Moonen 2001). Having identified the methods, different source code metrics is computed for each method and stored in a database. With the help of these metric values the near equal methods are extracted and are subjected to textual comparison to detect potential clone pairs.



$$\text{Precision} = \frac{|C|}{|D|} * 100 \qquad \text{Recall} = \frac{|C|}{|A|} * 100$$

Figure 1 Illustration of precision and recall.

The overall process is carried out in three major stages: Pre-processing, detection and post-processing. Figure 2 shows the overall system diagram of the proposed system. The following subsections, explain the steps in each of the stages.

4.1 Pre-processing

This stage includes the process of comment, white space removal and source code conversion or standardization (formatting). All files are scanned for filtering the uninteresting statements such as comments and white spaces. The final step is re-structuring of the code into a standard form which is needed for establishing clone fragments similarity (Ducasse et al. 2006). This helps in the identification of the cloned methods, thus yielding a significant gain in the Recall. Figures 3 and 4 illustrates the removal of comments and white spaces and statement standardization.

4.2 Method detection

Another potentially useful analysis could be to extract the methods alone, as the granularity is method-level. The standard form of source code scans for the detection of methods of adopting an 'island-driven parsing' (Moonen 2001). In order to extract isolated phrases or to detect certain features of a text island parser is used instead of a full-fledged parser.

It is a grammar-based method for extracting parts of a program as required from unwanted parts which need not be precisely parsed. In the island driven parsing system

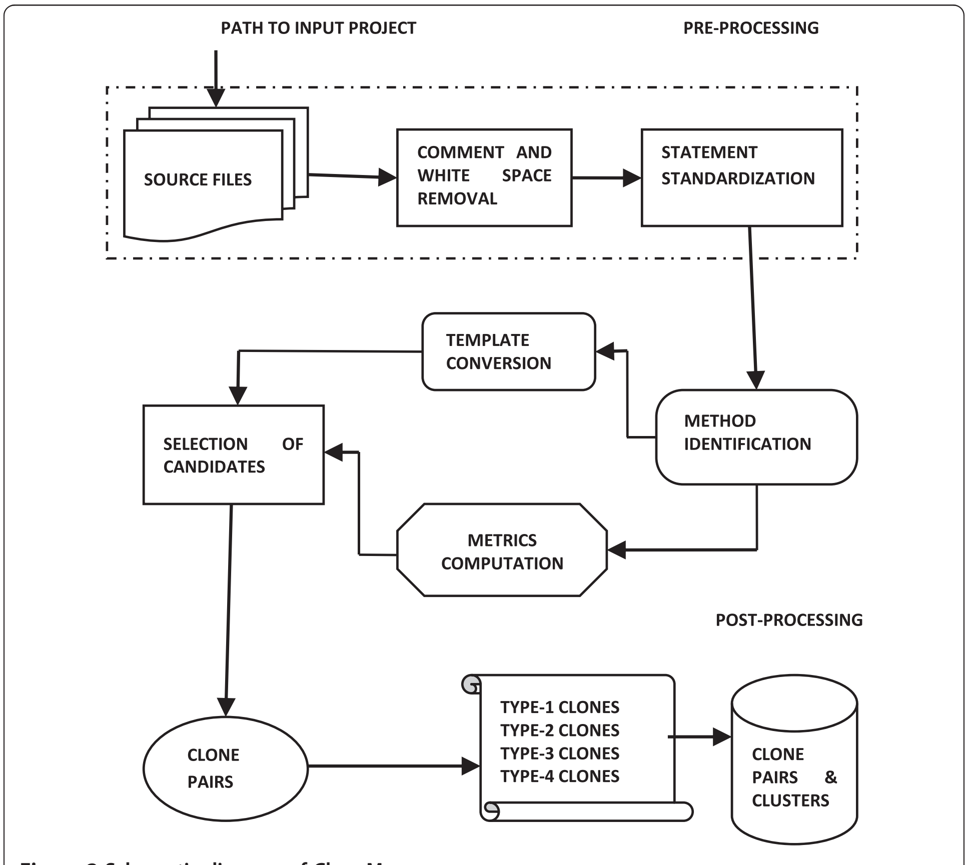
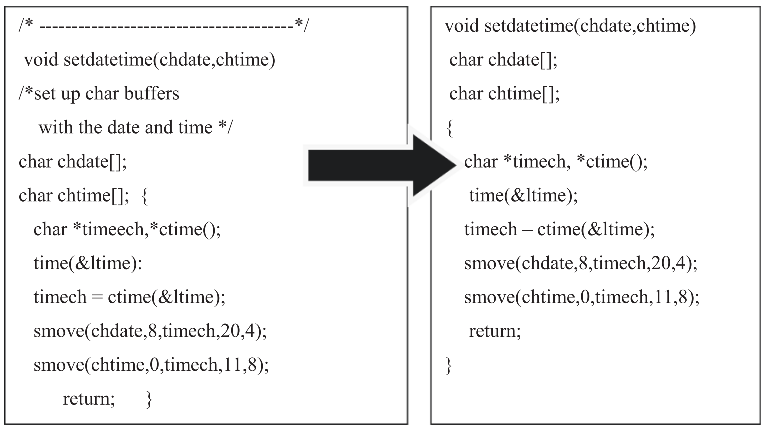


Figure 2 Schematic diagram of CloneManager.



```
/* --------------------------------------*/
 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);
   smove(chdate,8,timech,20,4);
   smove(chtime,0,timech,11,8);
    return;
}
```

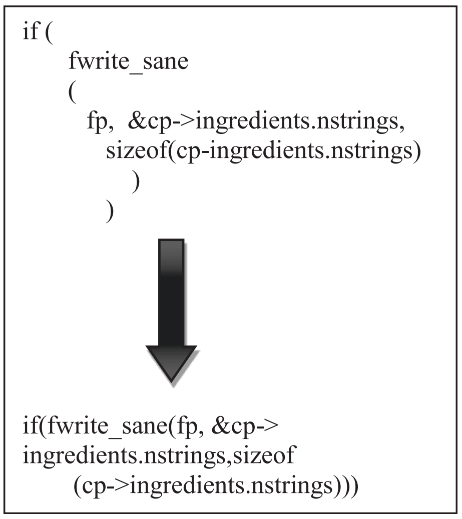
Figure 3 Process of comment and white space removal in pre-processing.

(Moonen 2001), parsing does not start at the beginning of the word network, but rather can start at confident regions within the network, at places known as islands. It provides a mechanism to find out the required elements to be compared.

Using this approach, the method definitions are extracted and collected by means of a hand-coded parser and saved for further reference. An interesting fragment is the piece of code that can be parsed and reduced to a nonterminal, method declaration. This approach takes text files and returns the structured fragments containing methods. For each method, it keeps track of the exact location within the file. An extracted method consists of a list with three elements such as (i) the method name (ii) the file name and the methods start and end positions (iii) the method content.

4.3 Template conversion

In addition to the standardization of source code, template conversion is exploited. This converts the original source code into a new form, having a uniform pattern for



```
if (
      fwrite_sane
      (
        fp,  &cp->ingredients.nstrings,
          sizeof(cp-ingredients.nstrings)
             )
          )
```

```
if(fwrite_sane(fp, &cp->
ingredients.nstrings,sizeof
       (cp->ingredients.nstrings)))
```

Figure 4 Statement standardization.

the permitted equivalent constructs between the clone pairs of the same type. An equivalent constructs contain invariant and variant parts as defined below.

- The invariant is part of the source code construct which is not expected to change between the clone versions.
- The variant is part of the source code constructs which are allowed to have changes among clone versions.

In this tool, variant part has been employed for detection of type-2, type-3 and type-4 clones.

4.3.1 Template conversion for type-1 and type-2

For type-2, as per the definition of literature the function identifiers, variable names, data-types, etc., are the only allowed differences in functions. Hence, to minimize the differences between the code fragments we bring out a uniform intermediate representation of the source code.

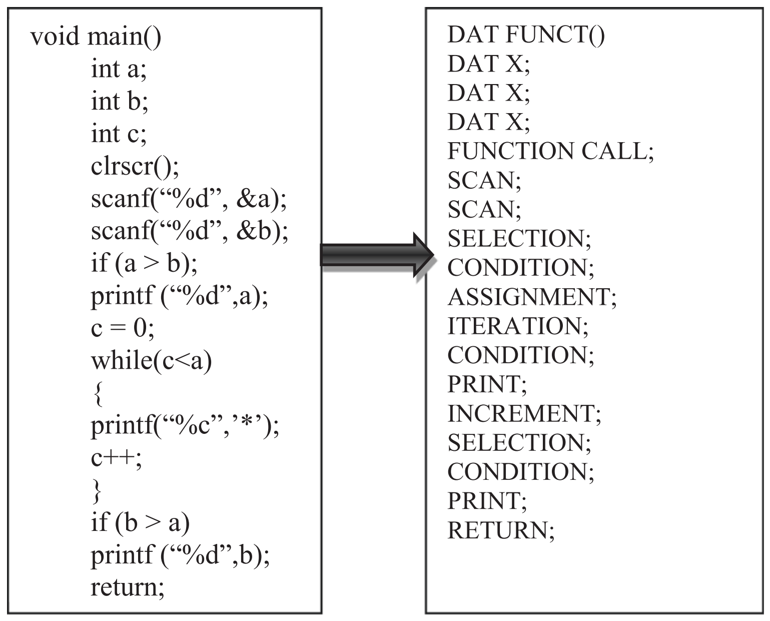
In case of type-2 detection, clone methods may contain a difference in the identifiers, literals, types, white space, layout and comments. To match all these differences, a common template is arrived. For instance, to avoid name differences, the names of the identifier are converted into common name as X and all the data-type declarations are converted into common data-type namely DAT. Figure 5 shows the template conversion for type-2 clones.

4.3.2 Template conversion for type-3 and type-4

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.

4.3.2.1 Iterative equivalence The control looping structures are *for, while* and *dowhile*. The three patterns present in looping are initialization, condition and increment/decrement; these are separated and written, each in a separate line. The common template form *iteration* helps in replacing the above three patterns. Both open braces and close braces are neglected while writing due to the changes in the order of the statement changes in order and nested statements in the source code. Table 2 shows the different types of variants among the source code. Figure 6 shows the template conversion for type-3 & type-4 clones.

4.3.2.2 Conditional equivalence The conditional structures are *if, else* and *elseif*. In these statements, the conditions are separately written in new line following the template form *selection*. The nested operations are split separately and rewritten in each new line. In case of the ternary operator "?:" the condition and other statements are separately printed in order to get the similar pattern.



```
void main()
        int a;
        int b;
        int c;
        clrscr();
        scanf("%d", &a);
        scanf("%d", &b);
        if (a > b);
        printf ("%d",a);
        c = 0;
        while(c<a)
        {
        printf("%c",'*');
        c++;
        }
        if (b > a)
        printf ("%d",b);
        return;
```

```
DAT FUNCT()
DAT X;
DAT X;
DAT X;
FUNCTION CALL;
SCAN;
SCAN;
SELECTION;
CONDITION;
ASSIGNMENT;
ITERATION;
CONDITION;
PRINT;
INCREMENT;
SELECTION;
CONDITION;
PRINT;
RETURN;
```

Figure 5 Template conversion for type-2 clones.

4.3.2.3 Input equivalence The input statements such as *scanf, system.in, input.readline*. In these statements, the variable alone will follow the template form *read*. For the multiple inputs, single input statements are separately written on each line as illustrated in Table 2.

4.3.2.4 Output equivalence The output statements such as *printf, system.out*. In these statements, the output variables alone follow the template form *write*. The print statements which are just printing any comments or statements are neglected. Also the multiple outputs, single print statements are separately written on each line.

4.3.2.5 Declaration equivalence The declaration statements start with keywords such as *char, int, long int, double, float, and string*. In this case, multiple declarations in a single statement are split and written, with each line as a single declaration statement. Table 2 shows the conversion of multiple declarations into single declaration.

4.3.2.6 Braces The braces are used in the programming languages for grouping the statements of looping and nesting. Both the open and close braces are neglected while writing due to the changes made in ordering.

4.4 Metrics computation

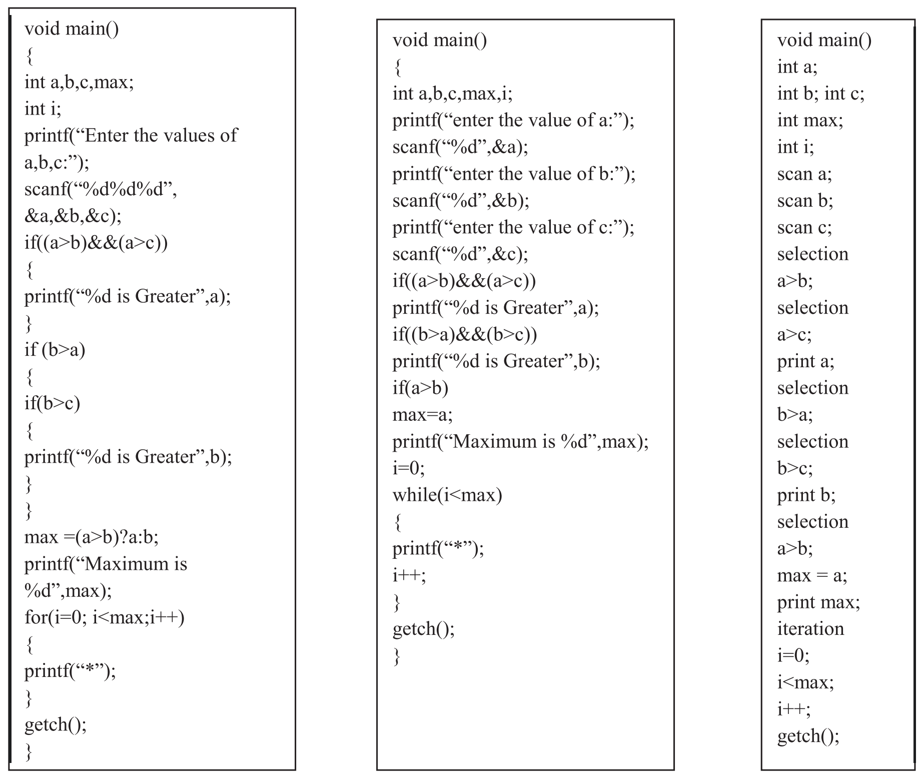
The previous method detection step produces a set of methods. In this step, we calculate the metric values for each of these methods to extract the potential clone pairs. A set of 12 count metrics has been proposed for the detection of these cloned methods.

Table 2 Types of variants among the source code patterns

S.No	Equivalence category	Possible constructs	Proposed pattern
1	Iterative equivalence	for	iteration
		while	<initial>
		do-while	<condition>
			<incre/decre>
2	Conditional equivalence	if	selection
		else	<condition>
		else-if	
		?:	
		switch	
3	Input equivalence	scanf	read <variable>
		system.in	
		input.readline	
4	Output equivalence	printf	write <variable>
		system.out	
5	Declaration equivalence	int	Multiple Declaration
		char	to Single line declaration
		float	
		double	*Example*
		string	int x
			int y
		Example	int z
		int x,y,z	char c
		char c,s	char s
6	Braces	{ }	Braces are removed in the code

Metrics, which are calculated using the simple counting formula are called as count metrics. These count metrics have been proposed for each type of cloned methods based on the necessity. Table 3 gives the list of metrics used for the detection of clones and their descriptions are briefed as follows:

1. **No. of Lines:** This indicates the number of effective lines of code in each method presents between the '{' and '}', indicating the start and end of the function definition.
2. **No. of Arguments:** This indicates the total number of arguments passed to the method irrespective of the data-types and the order of the arguments passed.
3. **No. of Local Variables:** The count value of the number of local variables declared within the function definition is represented by this metric. The number of variables used by the function or the number of global variables or the number of times the variables are used is not considered.
4. **No. of Function Calls:** This value gives a picture of the number of function calls made by the method. It is usually a measure of the flow of control in a source code and it gives an overall view of the functionality of both the defined and the called methods.



```
void main()
{
int a,b,c,max;
int i;
printf("Enter the values of
a,b,c:");
scanf("%d%d%d",
&a,&b,&c);
if((a>b)&&(a>c))
{
printf("%d is Greater",a);
}
if (b>a)
{
if(b>c)
{
printf("%d is Greater",b);
}
}
max =(a>b)?a:b;
printf("Maximum is
%d",max);
for(i=0; i<max;i++)
{
printf("*");
}
getch();
}
```

```
void main()
{
int a,b,c,max,i;
printf("enter the value of a:");
scanf("%d",&a);
printf("enter the value of b:");
scanf("%d",&b);
printf("enter the value of c:");
scanf("%d",&c);
if((a>b)&&(a>c))
printf("%d is Greater",a);
if((b>a)&&(b>c))
printf("%d is Greater",b);
if(a>b)
max=a;
printf("Maximum is %d",max);
i=0;
while(i<max)
{
printf("*");
i++;
}
getch();
}
```

```
void main()
int a;
int b; int c;
int max;
int i;
scan a;
scan b;
scan c;
selection
a>b;
selection
a>c;
print a;
selection
b>a;
selection
b>c;
print b;
selection
a>b;
max = a;
print max;
iteration
i=0;
i<max;
i++;
getch();
```

Figure 6 Template conversion for type 3 & type 4 clones.

5. **No. of Conditional Statements:** This includes the conditional statements in each method like the number of 'if', 'else if' and 'else' statements, etc., defined in the method. It is considered important as it determines the overall semantics of the method.
6. **No. of Iteration Statements:** This gives a count of the iterative control structures used within the method definition. Statements defining "while", "do" and "for" are considered in this metric. These are also important in identifying the pattern of execution of the method.

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 through Function Calls
11	No. of Selection Statements
12	No. of Assignment Statements

7. **No. of Return Statements:** It gives the number of return structures defined within the method. It indicates the number of exits present within the method definition.
8. **No. of Input Statements:** The various types of input statements used in the method to obtain the values of variables, the choice of the user, etc., are identified and counted. These play a vital role in judging the similarities between various methods.
9. **No. of Output Statements:** Similar to the count of input statements, the output statements also make a significant contribution to the analysis of the content of the method. Simple output statements used for the purpose of formatting the output and information texts are neglected while the valid values and results from the method passed to the buffers, console, etc. are taken under consideration.
10. **No. of Assignments through Function Calls:** This metric count the number of variables which gets the value by the assignment of a return value from a function call. These give an exclusive classification for the variables and their values and hence are taken into interest.
11. **No. of Selection Statements:** This metric is used for identifying selection statements in each method which include conditional operators, cases, etc. These statements along with the conditional statements produce branches and are hence analyzed to find out the pattern of execution of the method.
12. **No. of Assignment Statements:** This metric gives the count of the number of assignment statements in each method that modify the values of the various variables used in the method. The statements may be simple assignments, arithmetic expressions, unary operators, etc.

Apart from these 12 count metrics, four more metrics 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 which are additionally used is given below. A detailed description is present in literature (Adamov 1987, Fenton 1991, Moller 1993). Note that these metrics are computed compositionally from statements, two functions (in C) and methods (in Java).

13. S COMPLEXITY(**S**) = *FAN OUT*(***S***)
    where *FAN OUT(**S**)* is the number of individual function calls made within S.
14. D COMPLEXITY(***S***) = *GLOBALS*(***S***)/(*FAN OUT*(***S***) + 1)

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. MCCABE(**S**) = 1 + d, where d is the number of control decision statements in **S**.
16. ALBRECHT(**S**) = $\begin{cases} p1 \ast \mathit{VARS\ USED\ AND\ SET}(\mathbf{S})+ \\ p2 \ast \mathit{GLOBAL\ VARS\ SET}(\mathbf{S}) + \\ p3 \ast \mathit{USER\ INPUT}(\mathbf{S}) + \\ p4 \ast \mathit{FILE\ INPUT}(\mathbf{S}) \end{cases}$

where,
*VARS USED AND SET(**S**)* is the number of data elements set and used in the statement **S**,
*GLOBAL VARS SET(**S**)* is the number of global data elements set in the statement **S**,

*USER INPUT(**S**)* is the number of read operations in statement **S**,

*FILE INPUT(**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. These values are chosen according to the literature (Adamov 1987).

All 16 metrics are calculated for each method and stored for comparison and extraction processes. For type-1, type-2 and type-4, a constraint is posed that a cloned method pair must have an identical set of metric values. Thus, the database records containing identical metric values for method pairs are shortlisted for the type-1, type-2 and type-4 clone detection. The metrics are computed for each of the methods and are compared to be shortlisted by the formulas as indicated in Table 4.

4.5 Type-1 clone detection

With the shortlisted set of methods that are obtained, a textual comparison of the method pairs in the formatted and normalized code is done to identify the exactness of the extracted pairs. As per the definition, exact copy and paste of source code without any modification is called as type-1 clones. Methods having an exact equality score, which means, number of similar lines must be equal to the total number of lines in the method, are declared as type-1 cloned methods. The methods with same computed metric values and same as a textual comparison are declared as clone pairs. The detection criteria used for the identification of types of clones are tabulated in Table 4.

4.6 Type-2 clone detection

Type-2 cloned methods are syntactically identical code fragments except for variations in identifiers, literals, types, white space, layout and comments. Hence the textual comparison is performed on the template code created by the tool. The methods with the same computed metric values and same patterns for template comparison are short listed as clone pairs. The comparison in the template identifies type-1 cloned method along, with type-2 cloned methods. So they need to be removed separately. Further, for this reason textual comparison with original source code is compared to identify the differences in the parameters.

4.7 Type-3 clone detection

Copied code fragments with further modifications like statements can be changed, added or removed are considered as type-3 clones. In this case Range values of the calculated metrics are considered rather than the original values due to the wide variation in the syntactical structure of the methods. Thus to identify the clones, two different Range of metric values is identified which are suitable to detect type-3 clones. These

Table 4 Criteria for clone types 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	Range1 >=90%	-	Range2 >=85%
Type-4	Same	No match	Same

Range values are calculated for the methods in pairs. Range1 is the ratio of the actual metric value to the average metric value in the methods which are suspected to be clones.

$$\text{Range1} = \frac{\text{Actual metric value of method} \ast 100}{\text{Average metric value of methods}}$$

If any method is having more than 90% value for Range1, they are shortlisted under the possibilities for type-3 method clones. Then Range2 is calculated as the ratio of equal number of lines which are similar to the suspected method by the total number of lines in a method in the template code.

$$\text{Range2} = \frac{\text{No. similar lines in a method} \ast 100}{\text{Total no. of lines in a method}}$$

The method pairs having more than 85% values of Range2 in template methods are declared as type-3 clones. In the literature, there is no clear range specified for type-3 clones. The Range1 and Range2 values are equal for type-1, 2 and 4. Hence, for type-3 range has been explored with different values from 85% to 100%, and arrived this threshold value as a range after so many trial rounds.

4.8 Type-4 clone detection

Type-4 clones are the results of semantic similarity between two or more code fragments. In this type of clones, the cloned fragment is not necessarily copied from the original. Two code fragments may be developed by two different programmers to implement the same kind of logic, making the code fragments similar in their functionality. Thus the semantics of the cloned fragments remain the same while the structural and syntactical representation may show changes.

For type-4, first the two considered methods are taken and their computed metric values are considered. If the computed metric values are same for these two methods, then they are compared with the template methods. If they are also same, then the textual comparison of the source code is checked. If they are completely different, then they are categorized under type-4.

4.9 Post-processing

The output from the previous phase is in the form of clone pairs. The results of the tool CloneManager are given as clone pairs and clone clusters. The identified clone methods called as "potential clone pairs", are then clustered separately for each type and the clusters are uniquely numbered. *Clustering* is the process of grouping the clone pairs into classes or clusters so that clone pairs within a cluster are highly similar to one another, but are very dissimilar to clone pairs in other clusters. These clone pairs and clusters of all four types of clones are stored each in a text file separately.

5 Results and discussion

To validate the proposed LWH approach, the performance of the tool CloneManager is assessed for detecting the function clones in a number of open source systems. Based on the literature, Bellon's benchmark dataset (Bellon et al. 2007) has been chosen for code clone data which provides the details of reference set for eight software systems. For the remaining unclassified data, clone details are collected through manual verification process. Moreover, the experiments are carried out and presented as guidance provided by Wohlin et al. (2012).

5.1 Experimental setup

To evaluate the tool, source code of seven C projects and seven Java projects have been used. The experimental analysis has been carried out with a medium sized C project Weltab 11,000 lines to a large sized C project called Linux with 6,265,000 lines. Table 5 gives the size details of the projects, namely # files: number of files in the project, KLOC: number of thousand lines of code in the project and #methods: number of functions/methods in the project.

(Bellon and Koschke 2014; Bellon et al. 2007; Koschke et al. 2006) also measured the precision (refer section 3) and recall (refer section 3) of clone detection tools. Bellon created a benchmark set of clones by random sampling and evaluating a random subset of the union of clones detected by all clone detection tools in the study. This resulted in an oracled set of clones known to be true positives. Each reference clone was classified into one of three types: exact clones (Type-1); parameterized clones (Type-2); and clones with additional changes (Type-3). Six clone detection tools were used in the study: Dup (token-based), CCFinder (token-based), CloneDr (AST sub–tree), Duplix (PDG), CLAN (AST metrics), and Duploc (normalized lines of code).

Bellon's work produced the results for four C projects, namely Cook, Postgresql, Snns, Weltab and four Java projects, namely Eclipse-ant, Java netbeans-javadoc, Eclipse-jdtcore, J2sdk-swing. Finally, the precision and recall values in percentage are measured for each project by all the tools. Moreover, in literature, some researchers have used Bellon's benchmark for evaluation of their technique (Koschke et al. 2006; Selim et al. 2010; Hotta et al. 2014). Hence, in order to evaluate the proposed tool CloneManager, Bellon's benchmark has been adopted. For the remaining six projects, manual validation is carried out for the purpose of evaluation.

Table 5 Overview of the open source projects used by CloneManager

Language	Project name	#files	KLOC	# methods
C	Cook	287	70	1362
	Apache-httpd-2.2.8	496	275	4301
	Postgresql	314	202	4669
	Snns	138	94	2201
	Weltab	39	11	123
	Wget	23	17	219
	Linux-2.6.24.2	9491	6265	154977
Java	Eclipse-ant	161	35	1754
	EIRC	54	11	588
	Java Netbeans-Javadoc	97	14	972
	Eclipse-jdtcore	582	148	7383
	JHotDraw 5.4b1	233	40	2399
	Spule	50	13	420
	J2sdk-swing	414	204	10971

5.2 Results

The results of the experiments are summarized, in this section. It presents the numbers of clone pairs and clone clusters detected for different categories of clone types by our proposed tool CloneManager. In Table 6, the third column is the clone type-1 with the number of detected clone pairs and the clone clusters. Columns 4, 5 and 6 hold the same set of data for type-2, 3 and 4 respectively.

From the data presented in Table 6, the following observations were made.

- Linux with 6265,000 lines has only 39119 clone pairs in total. On the other hand, J2sdk-swing with only 204,000 of lines has 27559 clone pairs in total. This shows that, the number of lines in the projects is not directly proportional to the number of clone pairs.
- The smallest size project in our observation was Weltab with 11,000 lines. However, it had 333 clones in total.
- It is interesting to note, Wget has no type-1 matches, which means that they do not have exact functions in the code. The size of Wget is 17,000 lines. Moreover, they have the least number of clone pairs 17 in total.

On comparing the clone types obtained it has been observed that the no. of clones in type-2 clones is higher than type-1 clones and less type-3 clones; all projects have the least number of type-4 clones. This shows us that, the number of clones increases as the type increases and falls down for the type-4. In other words, the number of clones increases in textual similarity and decreases in functional similarity. These observations lead to an interesting inference: programmers do not write code with different logic for the same external behaviour.

On analysing the experimental results it has been observed that, on average, above 15% of the methods in open source Java code is type-1 clones, whereas only above 2.5% of C

Table 6 CloneManager: number of detected clones pairs and clone clusters

S.No	Project name	Type-1		Type-2		Type-3		Type-4	
		Clone pairs	Clone clusters	Clone pairs	Clone clusters	Clone pairs	Clone clusters	Clone pairs	Clone clusters
1	Cook	18	5	157	30	280	98	7	3
2	Apache-httpd-2.2.8	183	107	242	143	711	276	10	4
3	Postgresql	28	4	240	42	530	203	7	3
4	Snns	109	63	160	86	495	191	9	4
5	Weltab	46	8	115	11	160	20	12	5
6	Wget	0	0	4	2	11	2	2	1
7	Linux-2.6.24.2	5953	1505	7386	2265	25767	7918	13	5
8	Eclipse-ant	363	92	372	96	426	119	10	4
9	EIRC	117	35	119	35	149	47	6	3
10	Java Netbeans-Javadoc	193	80	199	83	304	110	8	3
11	Eclipse-jdtcore	1427	323	5573	587	4378	660	15	7
12	JHotDraw 5.4b1	291	137	299	142	598	208	10	4
13	Spule	60	11	69	14	113	19	4	2
14	J2sdk-swing	8115	516	8205	558	11209	843	30	14

functions are type-1 clones. Thus it shows that function clones appear more in open source Java code than C. After analysing the detected clones, it is observed that this is due to the large number of 'small getter and setter methods' in Java programs which are not present in C. From overall analysis, it has been observed that the level of cloning is found to be less in C than Java projects. Also, it is found that C projects have very less type-1 clones, less than 10% in some and to a large extent independently of the system size.

As a result of all these analysis, it could be inferred that,

- Most of the Java systems have significantly fewer clone clusters than clone pairs, indicating the fact that there are many pairs of functions in the systems that are similar to each other.
- Average number of clone pairs per clone cluster is more or less consistent for C and Java systems for different clone types.
- C systems show a faster growing ratio for type-3 clones than the Java systems, indicating the fact that there might be more type-3 clones in the C than the Java systems.

5.3 Procedure to determine reference data

The Bellon's benchmark (Bellon and Koschke 2014) results are used for the tool evaluation. Bellon's benchmark has evaluated 8 projects for different tools (Cook, Postgresql, Snns, Weltab, Eclipse-ant, Java netbeans-javadoc, eclipse-jdtcore and J2sdk-swing). He has evaluated experimental result with the manually evaluated values as reference values, which was only 2%. However, he produced his complete experimental results for all projects. Thus the results are taken from his benchmark, assuming that they are accurate. The complete results of Bellon's tool experiment are available at http://www.bauhaus-stuttgart.de/clones/. For the remaining six projects (Apache-httd-2.2.8, Wget, Linux-2.6.24.2, EIRC, Jhotdraw 5.4b1 and Spule), which are not available in Bellon's benchmark, manual evaluation was carried out with the help of semi-automated tools.

Using the standardization tool named fscodeformat64, both C and Java codes are standardized. Comments above the methods are examined carefully, which informs the method description. This helps to analyse the methods, with similar semantic methods, may be type-4 clones. These methods alone are extracted separately and by checking external behaviour, type-4 clones are detected. All the methods are extracted by removing the other codes by simple program developed in Java. The methods with similar codes are detected using another simple program. They are counted as type-1 clones and extracted separately in a file. Then the manual process is carried out to detect the type-2, 3 clones.

Two students in a batch are allocated for the manual detection of clones for 2 open source projects. They took 15 days training from the faculty, before starting their work. They took six months to complete this task. Two batches are allocated in parallel and thus 14 students helped to evaluate this work manually and took 21 months to complete this task. Moreover, one batch students' results are also verified by the other batch, mutually. Finally, to cross check the accuracy of these manual processes, some samples clones have been picked from the reference set of data and monitored whether these clones have been detected by the students. To carry out this evaluation process, misclassification is calculated as follows

Table 7 Misclassification report for sample clones

Project	Actual clones [A]	Detected clones [D]	Correctly detected clones [C]	False negatives [N]	False negatives in %	False positives [P]	False positives in %
Cook	14	14	14	0	0	0	0
Apache-httpd-2.2.8	20	20	19	1	5	0	0
Postgresql	11	11	11	0	0	0	0
Snns	14	15	14	0	0	1	6
Weltab	10	10	10	0	0	0	0
Wget	8	8	8	0	0	0	0
Linux-2.6.24.2	20	20	20	0	0	0	0
Eclipse-ant	15	15	14	1	6	0	0
EIRC	13	13	13	0	0	0	0
Java Netbeans-Javadoc	10	10	10	0	0	0	0
Eclipse-jdtcore	16	16	16	0	0	0	0
JHotDraw 5.4b1	24	24	24	0	0	0	0
Spule	15	16	15	0	0	1	6
J2sdk-swing	21	21	21	0	0	0	0

1. $False\ \ negative\ \ in\ \ \% \ = \frac{[N]}{[A]} * 100$
2. $False\ \ positive\ \ in\ \ \% \ = \frac{[P]}{[D]} * 100$

Where False Negative [N] = Actual clones [A] – correctly detected clones[C] which reports the number of clones failed to be detected.

False Positive [P] = Detected Clones [D] – correctly detected clones[C] which reports the number of clones wrongly detected as clones.

Actual clones [A] are the reference clones.

The Table 7 shows the misclassification report for the sample clones considered. From the Table 7, it is clear that the manual detection of clones is merely correct.

Table 8 CloneManager: precision and recall of type-1 clones

Project name	Actual clones (A)	Detected clones (D)	Correctly detected clones (C)	Precision %	Recall %
Cook	18	18	18	100	100
Apache-httpd-2.2.8	203	192	183	95	90
Postgresql	28	29	28	96	100
Snns	118	110	109	97	92
Weltab	46	46	46	100	100
Wget	0	0	0	-	-
Linux-2.6.24.2	6764	6470	5953	92	88
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
J2sdk-swing	8820	8196	8115	99	92

Table 9 CloneManager: precision and recall of type-2 clones

Project name	Actual clones (A)	Detected clones (D)	Correctly detected clones (C)	Precision %	Recall %
Cook	160	160	157	98	98
Apache-httpd-2.2.8	252	249	242	97	96
Postgresql	250	252	240	95	96
Snns	161	170	160	94	99
Weltab	115	115	115	100	100
Wget	4	4	4	100	100
Linux-2.6.24.2	7774	8116	7386	91	95
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
J2sdk-swing	8728	8918	8205	92	94

5.4 Evaluation of the tool CloneManager

From the standard benchmark results, a reference set is obtained for the evaluation of the parameters precision and recall. These values have been evaluated for all four types of clones and are given in Tables 8, 9, 10 and 11 respectively.

Table 8 shows the precision and recall values of type-1 clones for all the projects. Column 2 holds the number of actual clones (A) from the reference set for all the projects. Column 3 holds (D) the number of detected clones by our tool CloneManager. Column 5 holds (C) the number of correctly detected clones by our tool. Then, values for the two parameters precision and recall are computed using the formula given in Figure 1.

Table 10 CloneManager: precision and recall of type-3 clones

Project name	Actual clones (A)	Detected clones (D)	Correctly detected clones (C)	Precision %	Recall %
Cook	291	291	280	96	96
Apache-httpd-2.2.8	807	756	711	94	88
Postgresql	576	552	530	96	92
Snns	526	505	495	98	94
Weltab	160	160	160	100	100
Wget	11	11	11	100	100
Linux-2.6.24.2	28007	27411	25767	94	92
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 11 CloneManager: precision and recall of type-4 clones

Project name	Actual clones (A)	Detected clones (D)	Correctly detected clones (C)	Precision %	Recall %
Cook	8	7	7	100	87
Apache-httpd-2.2.8	11	11	10	90	90
Postgresql	7	7	7	100	100
Snns	10	10	9	90	90
Weltab	13	13	12	92	92
Wget	2	2	2	100	100
Linux-2.6.24.2	14	14	13	92	92
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

From the data presented that has been given in Tables 8, 9, 10 and 11, it could be seen that, CloneManager has resulted in higher values for precision and recall for all the clone types. As precision and recall are the best parameters for the evaluation of clone detection tools, it could be concluded that the proposed CloneManager is found to be an effective tool for detecting all types of clones. Figures 7 and 8 shows the precision and recall values in graph for all the projects. Finally the result of the Linux project shows that the tool CloneManager is able to detect clones even for larger systems in size. This proves that the tool CloneManager is also scalable.

5.5 Comparison with existing tools

In literature, there are two approaches with method-level granularity: CLAN (Mayland et al. 1996) and NICAD (Roy and Cordy 2008) which is closely comparable to our own. In this section, the proposed tool has been compared with CLAN and NICAD. The first tool considered for analysis is the CLAN clone detection with metrics based clone detection technique and method-level granularity. CLAN gathered different metrics for code fragments and compared these metric vectors instead of comparing the code

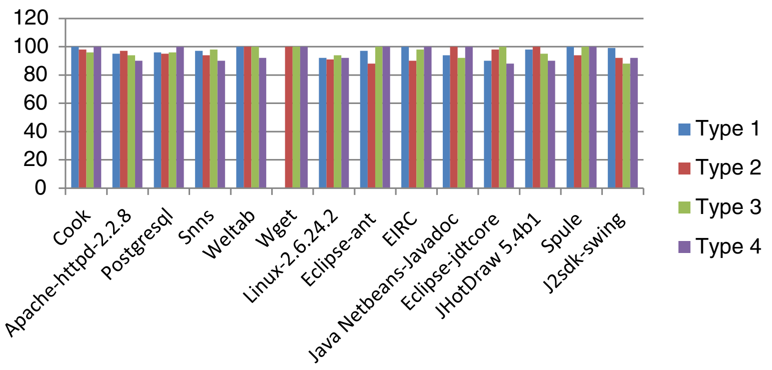


Figure 7 Precision in % of all the projects.

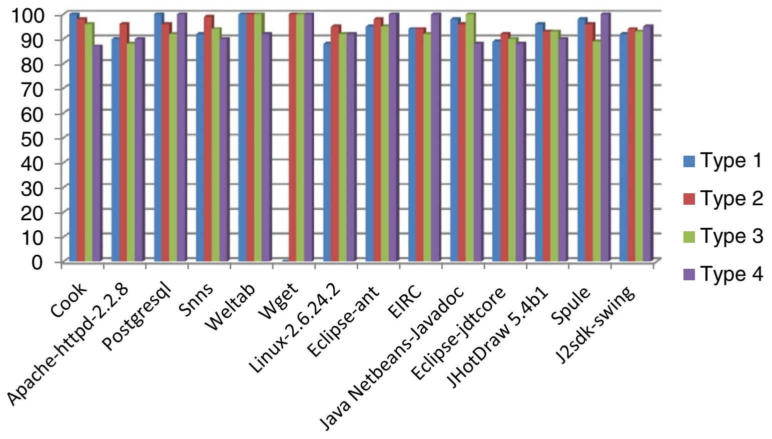


Figure 8 **Recall in % for all the projects.**

directly. An allowable distance (for instance, Euclidean distance) for these metric vectors can be used as a hint for similar code.

The second is NICAD (Roy and Cordy 2008) a parser-based, language specific, lightweight approach using simple text-line comparison which finds function clones with the aid of TXL. TXL (Cordy et al. 2002) is a programming language specifically designed for manipulating and experimenting with programming language notations and features using source to source transformation.

Because of limited space, only one system presented here. We have chosen Weltab, because some of the earlier experimental tools have used it to evaluate their work. The tool CloneManager ran successfully for all the projects in Table 5. The comparison of the results of all the projects with these two existing tools is done in the same way as Weltab. 321 clone pairs for type-1, 2, 3 were obtained altogether using the proposed LWH approach, while CLAN has obtained only 101 match clone pairs. Moreover, the CloneManager tool further classified clones pairs as clone clusters. In addition, type-4 clones are also detected by the tool CloneManager. The results obtained by these two existing tools are presented in Table 12 along with the computed values for the proposed tool CloneManager.

NICAD reported 8 exact-match and 20 near-miss clone clusters which are nothing but the type-1 and type-3 clone clusters found in Weltab. The implemented proposed method, have obtained similar results. NICAD having claimed to have obtained 100% when compared with Bellon's benchmark results, which concludes that the proposed method has also accomplished the same output.

Though NICAD has proved to effectively detect the function clones, the initial phases employ an external parser. Whereas, the proposed method uses a hand-coded parser,

Table 12 Comparison of clone pairs and clone clusters for Weltab

TYPE	CLAN	NICAD		CloneManger	
	Clone pairs	Clone pairs	Clone clusters	Clone pairs	Clone clusters
Type-1	46	46	8	46	8
Type-2	27	-	-	115	11
Type-3	28	160	20	160	20
Type-4	-	-	-	12	5
Total	101	206	28	333	44

Table 13 Comparison of run-time with NICAD and proposed tool CloneManager

Projects	NICAD in minutes	CloneManager in minutes
Cook	5.13	5.01
Apache-httpd-2.2.8	18.21	16.12
Postgresql	9.59	8.48
Snns	5.23	5.09
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

external lexers or parsers have not been deployed. Moreover, 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 Roy called it as exact clones (type-1). Then Roy matches with threshold value 0.10, 0.20, 0.30 and called it as 10%, 20%, 30% of dissimilarity in the clones respectively. It is able to detect near-missed clones (type-3) but fails to detect type-2 and type-4 clones.

From the Table 13, shows the comparison of the run-time of the proposed tool CloneManager with the NICAD tool. It is easier to notice from the Table that the time taken by the proposed tool is lesser than NICAD. Thus the proposed tool proves to have time complexity better than NICAD.

Table 14 shows the comparison of the Precision and Recall parameters of the tool CLAN with the proposed tool CloneManager. In Table 14 T1, T2, T3 stands for type-1, type-2, type-3 respectively. The projects which have Precision and Recall data are taken from the standard Bellon's benchmark. Moreover, the data were only available for type-1, type-2 and type-3. From the Table it is observed that the proposed tool CloneManager is very high in Precision and Recall.

5.6 Threats to validity

In this section, the various factors that threaten the validity of our results are summarized. The common guidelines (Yin 2002) are followed for empirical studies.

Table 14 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
Cook	33	10	3	4	16	12	100	98	96	100	98	96
Postgresql	9	2	0	0	19	11	95	95	96	100	96	92
Snns	8	11	4	11	6	2	96	94	98	92	99	94
Weltab	15	35	0	33	6	0	100	100	96	100	100	98
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

5.6.1 Internal validity

Threat of internal validity corresponds to the ability of our experiments to link the independent and dependent variables. The threat may be revealed through experimental or human errors. Bellon's benchmark was used as a reference set for the comparison of detecting clone results. The Bellon reference corpus was manually built by Bellon using only 2% of the clones suggested by the six clone detectors. For unbiased comparison, it is necessary to rebuild the clone references by considering the results of all clone detectors, which is beyond the scope of this paper.

We carried out manual analysis to verify the correctness of the clone detection using semi-automated tools/manual. The manual assessment can be subject to human errors. However, all the participants of this work are graduate students carrying out projects in the area of software clones. Thus we trust that each one has agent expertise to keep the plausible human errors to the minimum.

5.6.2 External validity

Threats to external validity are about how to generalize our results. We had done our comparison with 14 open source projects of various size and application domains that are written in two popular programming languages C and Java. However, this does not declare that the findings can be held true for other programming languages. Moreover, we planned to explore more systems written in various programming languages.

5.6.3 Construct validity

Construct validity threats are related to the relation between theory and observation. It corresponds to the suitableness of our evaluation parameters. We mainly focused on the precision, recall and run-time for the evaluation of our tool. These evaluation parameters measured high in precision & recall values and low in run-time values. However, the usage of the memory is slightly higher, as our approach uses the intermediate results such as generating templates in two different methods. Moreover, it will not affect so much as we can see the vast development of physical storage capacity and speed of access growing rapidly day-by-day.

6 Conclusion

In this paper, we have proposed a LWH approach 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 reduce the total comparison cost of avoiding the exponential rate of comparison by using the metrics. Since the string matching/textual comparison is performed over the shortlisted 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 C and Java.

As a future work, first we have planned to enhance the technique for Web Static pages. Second, we have also planned to enhance the tool for clone removal by using the refactoring technique. Third, if there are some simple modifications in the source code, then the clone has to be detected in the whole software from the scratch. It surely takes the same or more time to do the same process. This time can be reduced to a considerable extent, by making it to retain the previous clone detection results with the intermediate values and thus produce the results in a fraction of time for the next revisions. Next we have planned to enhance our tool with this incremental process.

Appendix A

The details of the open source projects chosen for the experimentation and evaluation of the clone detection tool CloneManager, is as follows

1. **Cook** is a tool for constructing files. It is given a set of files to create, and recipes of how to create them.
2. **Apache HTTP** Server project develops and maintains an open source HTTP (Hypertext Transfer Protocol) server for modern operating systems, including UNIX and Windows NT (New Technology).
3. **PostgreSQL** (Database) runs on many different operating systems.
4. **SNNS** (Stuttgart Neural Network Simulator) is a neural network simulator originally developed at the University of Stuttgart.
5. **Weltab** which is a Vote tabulation system.
6. **Wget** a free software package for retrieving files using HTTP, HTTPS and FTP, the most widely-used Internet protocols.
7. **Linux** is the open source operating system.
8. **Eclipse Ant** is the premier build tool for Java developers, and Integrating Ant with Eclipse provides a good solution for web development.
9. **Eight IRC (EIRC)** will be an Internet Relay Chat(IRC) client in windows that will also be translated to Swedish hopefully.
10. **JavaNetbeans- javadoc** tool provides an easy way to write API documentation for source code and software projects using the Java programming language.
11. **Eclipse-jdtcore** - The Java model is the set of classes that model the objects associated with creating, editing, and building a Java program.
12. **JHotDraw** is a Java GUI framework for technical and structured Graphics.
13. **Spule** stands for "secure practical universal lecture evaluator". Spule is a program to automatize the evaluation of lecture polls.
14. **J2sdk-swing** provides many enhancements to the existing graphics package.

Appendix B

The comparative results of the tool CloneManager with CLAN and NICAD tools for all chosen open source projects, are presented in the following tables (Tables 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27 and 28).

Table 15 Clone pairs and clone clusters for cook

TYPE	CLAN	NICAD		CloneManager	
	CP	CP	CC	CP	CC
Type-1	-	7	5	18	5
Type-2	200	-	-	157	30
Type-3	249	280	98	280	98
Type-4	-	-	-	7	3
Total	449	287	103	462	136

Table 16 Clone pairs and clone clusters for SNNS

TYPE	CLAN	NICAD		CloneManager	
	CP	CP	CC	CP	CC
Type-1	100	109	63	109	63
Type-2	108	-	-	160	86
Type-3	110	495	191	495	191
Type-4	-	-	-	9	4
Total	318	604	254	773	344

Table 17 Clone pairs and clone clusters for Postgresql

TYPE	CLAN	NICAD		CloneManager	
	CP	CP	CC	CP	CC
Type-1	200	7	7	28	4
Type-2	200	-	-	240	42
Type-3	530	530	203	530	203
Type-4	-	-	-	7	3
Total	830	537	210	805	252

Table 18 Clone pairs and clone clusters for Weltab

TYPE	CLAN	NICAD		CloneManager	
	CP	CP	CC	CP	CC
Type-1	46	46	8	46	8
Type-2	27	-	-	115	11
Type-3	28	160	20	160	20
Type-4	-	-	-	12	5
Total	101	206	28	333	44

Table 19 Clone pairs and clone clusters for Eclipse-ant

TYPE	CLAN	NICAD		CloneManager	
	CP	CP	CC	CP	CC
Type-1	10	363	92	363	92
Type-2	54	-	-	372	96
Type-3	24	426	119	426	119
Type-4	-	-	-	10	4
Total	88	789	211	1171	311

Table 20 Clone pairs and clone clusters for Eclipse-jdtcore

TYPE	CLAN	NICAD		CloneManager	
	CP	CP	CC	CP	CC
Type-1	1030	1427	323	1427	323
Type-2	6050	-	-	5573	587
Type-3	3031	4378	660	4378	660
Type-4	-	-	-	15	7
Total	10111	5805	983	11393	1577

Table 21 Clone pairs and clone clusters for Netbeans-javadoc

TYPE	CLAN	NICAD		CloneManager	
	CP	CP	CC	CP	CC
Type-1	28	193	80	193	80
Type-2	28	-	-	199	83
Type-3	29	304	110	304	110
Type-4	-	-	-	8	3
Total	85	497	190	704	276

Table 22 Clone pairs and clone clusters for J2sdk-swing

TYPE	CLAN	NICAD		CloneManager	
	CP	CP	CC	CP	CC
Type-1	936	8115	516	8115	516
Type-2	936	-	-	8205	558
Type-3	937	11209	843	11209	843
Type-4	-	-	-	30	14
Total	2809	19324	1359	27559	1931

The following projects are compared to NICAD tool alone, as the data was not available for CLAN tool.

Table 23 Clone pairs and clone clusters for Apache-httpd 2.2.8

TYPE	NICAD		CloneManager	
	CP	CC	CP	CC
Type-1	183	107	183	107
Type-2	-	-	242	143
Type-3	711	276	711	276
Type-4	-	-	10	4
Total	894	383	1146	530

Table 24 Clone pairs and clone clusters for wget

TYPE	NICAD		CloneManager	
	CP	CC	CP	CC
Type-1	0	0	0	0
Type-2	-	-	4	2
Type-3	11	2	11	2
Type-4	-	-	2	1
Total	11	2	17	5

Table 25 Clone pairs and clone clusters for Linux

TYPE	NICAD		CloneManager	
	CP	CC	CP	CC
Type-1	5953	1505	5953	1505
Type-2	-	-	7386	2265
Type-3	25767	7918	25767	7918
Type-4	-	-	13	5
Total	31720	9423	39119	11693

Table 26 Clone pairs and clone clusters for EIRC

TYPE	NICAD		CloneManager	
	CP	CC	CP	CC
Type-1	117	35	117	35
Type-2	-	-	119	35
Type-3	149	47	149	47
Type-4	-	-	6	3
Total	266	82	391	120

Table 27 Clone pairs and clone clusters for JHotDraw

TYPE	NICAD		CloneManager	
	CP	CC	CP	CC
Type-1	291	137	291	137
Type-2	-	-	299	142
Type-3	598	208	598	208
Type-4	-	-	10	4
Total	889	345	1198	491

Table 28 Clone pairs and clone clusters for Spule

TYPE	NICAD		CloneManager	
	CP	CC	CP	CC
Type-1	60	11	60	11
Type-2	-	-	69	14
Type-3	113	19	113	19
Type-4	-	-	4	2
Total	173	30	246	46

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
KE carried out the systematic reviews, identified the issues in the existing work. KE and KS designed architecture and implementation of the proposed algorithms. The dataset collection, experiments and result analysis are conducted by both KE and KS. The format of the manuscript was decided by KE and KS. The manuscript was prepared by KE, corrections and reviews are made by KS. Both authors read and approved the final manuscript.

Authors' information
Mrs. Kodhai. E is currently working as Associate Professor in the Department of Information Technology 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 14 years of experience in teaching in various engineering colleges. She is currently pursuing her Ph.D in Software Clones. Her Research interests include Software Maintenance and Evolution. She has published more than 30 papers in international conference and journals.
Dr. Kanmani. S received her B.E (CSE) and M.E (CSE) from Bharathiar University, Coimbatore, India and Ph.D from Anna University, Chennai, India. She is working as Professor in the Department of Information Technology at Pondicherry Engineering College. She has published nearly 63 research papers. She is currently a supervisor guiding 8 Ph.D scholars. She is an expert in Software Testing. Her areas of interests include Software Engineering, Genetic algorithms and Data Mining.

Acknowledgements
We appreciate the insightful comments from the three anonymous reviewers. Their comments were very helpful for us to improve the paper. We also express our thanks to Pondicherry Engineering College for their support in performing this research.

Author details
[1]Research Scholar, Department of CSE, Pondicherry Engineering College, Puducherry, India. [2]Department of IT, Pondicherry Engineering College, Puducherry, India.

Received: 14 November 2013 Accepted: 23 September 2014
Published online: 22 October 2014

References
Adamov R (1987) Literature review on software metrics. Institute of computer science, University of Zurich, Zurich
Al-Batran B, Sch¨atz B, Hummel B (2011) Semantic clone detection for model-based development of embedded systems. Model Driven Eng. Languages and Syst. 6981:258–272
Baker BS (1997) Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. SIAM J on Computing 26(5):1343–1362
Bellon S, Koschke R (2014) Detection of Software Clones: Tool Comparison Experiment. URL: http://www.bauhaus-stuttgart.de/clones/. Accessed 29 Jan 2014
Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and Evaluation of Clone Detection Tools. IEEE Transactions on Software Engineering 33(9):577–591
Basit H, Pugliesi S, Smyth W, Turpin A, Jarzabek S (2007) Efficient Token Based Clone Detection with Flexible Tokenization. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'07). ACM, Croatia, pp 513–515
Cordy JR, Dean TR, Malton AJ, Schneider KA (2002) Source Transformation in Software Engineering using the TXL Transformation System. J Information and Software Technology 44(13):827–837
Ducasse S, Nierstrasz O, Rieger M (2006) On the effectiveness of clone detection by string matching. J on Software Maintenance and Evolution 18(1). doi:10.1002/smr.317, http://scg.unibe.ch/archive/papers/Duca06iDuplocJSMEPaper.pdf
Ducasse S, Rieger M, Demeyer S (1999) A Language Independent Approach for Detecting Duplicated Code. In: 15th International Conference on Software Maintenance (ICSM'99). IEEE, Oxford, England, pp 109–118
Evans W, Fraser C (2005) Clone Detection via Structural Abstraction. Technical Report MSR-TR-2005-104. Microsoft Research, Redmond, WA
Evans WS, Fraser CW, Ma F (2009) Clone Detection via Structural Abstraction. Software Quality Journal 17:309–330
Fenton E (1991) Software metrics: a rigorous approach. Chapman and Hall
Fowler M (1999) Refactoring: improving the design of existing code. Wesley, Addison
Funaro M, Braga D, Campi A, Ghezzi C (2010) A hybrid approach (syntactic and textual) to clone detection. In: 4th International Workshop on Software Clones. ACM 2010 ISBN 978-1-60558-980-0, Cape Town, South Africa, pp 79–80
Gabel M, Jiang L, Su Z (2008) Scalable Detection of Semantic Clones. In: 30th International Conference on Software Engineering. ICSE 2008, Leipzig, Germany, pp 321–330
Greenan K (2005) Method-Level Code Clone Detection on Transformed Abstract Syntax Trees using Sequence Matching Algorithms. Student Report. University of California, Santa Cruz, Winter. available at http://users.soe.ucsc.edu/~ejw/courses/290gw05/greenan-report.pdf
Hotta K, Yang J, Higo Y, Kusumoto S (2014) How Accurate Is Coarse-grained Clone Detection? Comparision with Fine-grained Detectors. In: Eight International workshop on software clones. Electronic Communications of the EASST, Antwerp, Belgium
Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE Computer Society Transactions on Software Engineering 28(7):654–670
Kapser C, Godfrey M (2004) Aiding comprehension of cloning through categorization. In: International Workshop on Principles of Software Evolution. IEEE Computer Society, Kyoto, Japan, pp 85–94

Kapser CJ, Godfrey MW (2006) Supporting the analysis of clones in software systems: Research articles. J of Software Maintenance: Research and Practice 18(2):61–82
Kapser C, Godfrey MW (2008) Cloning considered harmful: Patterns of cloning in software. Empirical Software Engineering 13(6):645–692
Komondoor R, Horwitz S (2001) Using Slicing to Identify Duplication in Source Code. In: 8th International Symposium on Static Analysis. SAS 2001, Paris, France, pp 40–56
Koschke R (2012) Large-Scale Inter-System Clone Detection Using Suffix Trees. In: European Conference on Software Maintenance and Reengineering. IEEE Computer Society Press. University of Szeged Congress Centre (SZTE TIK), Szeged, Hungary, pp 309–318
Koschke R, Falke R (2006) Frenzel P (2006) Clone detection using abstract syntax suffix trees. Working Conference on Reverse Engineering, IEEE Computer Society Press, In
Krinke J (2001) Identifying Similar Code with Program Dependence Graphs. In: 8th Working Conference on Reverse Engineering. WCRE 2001, Stuttgart, pp 301–309
Lee M, Roh J, Hwang S, Kim S (2010) Instant code clone search. In: Fundamental of Software Engineering, pp 167–176
Leitao A (2004) Detection of Redundant Code Using R2D2. Software Quality Journal 12(4):361–382
Leitner A, Ebner W, Kreiner C (2013) Mechanisms to Handle Structural Variability in MATLAB/Simulink Models. In: Favaro J, Morisio M (ed), vol 7925. ICSR 2013, LNCS, Pisa, Italy, pp 17–31
Li Z, Lu S, Myagmar S, Zhou Y (2006) CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering 32(3):176–192
Liu C, Chen C, Han J, Yu P (2006) GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In: 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp 872–881
Marcus A, Maletic J (2001) Identification of High-level Concept Clones in Source Code. In: 16th IEEE International Conference on Automated Software Engineering. ASE 2001, Coronado Island, San Diego, CA, USA, pp 107–114
Mayland J, Leblanc C, Merlo E (1996) Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In: International Conference on Software Engineering 96. IEEE and ACM, Berlin, Germany
Moller K (1993) Software metrics: a practitioner's guide to improved product development. Hall, Chapman and
Moonen L (2001) Generating Robust Parsers using Island Grammars. In: 8th Working Conference on Reverse Engineering (WCRE'01). IEEE Computer Society, Washington, DC, USA, p 13
Pate J, Tairas R, Kraft N (2011) Clone Evolution: a Systematic Review. J of Software Maintenance, Research and Practice
Petersen H (2012) Clone detection in Matlab Simulink models. Master's thesis. Tech. Univ. Denmark
Roy CK, Cordy JR (2007) A survey on software clone detection research. Tech. Rep. 541. Queen's University, Kingston, Canada
Roy CK, Cordy JR (2008) NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In: 16th IEEE International Conference on Program Comprehension. IEEE Computer Society 2008, Amsterdam, The Netherlands, pp 172–181
Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming 74(7):470–495
Selim GMK, Foo KC, Zou Y (2010) (2010) Enhancing Source-Based Clone Detection Using Intermediate Representation. Working Conference on Reverse Engineering, In
Thummalapenta S, Cerulo L, Aversano L, Penta MD (2009) An empirical study on the maintenance of source code clones. Empirical Software Engineering 15(1):1–34
Ueda Y, Kamiya T, Kusumoto S, Inoue K (2002) Gemini: Maintenance Support Environment Based On Code Clone Analysis. In: 8th IEEE Symposium on Software Metrics. IEEE Computer Society 2002 ISBN 0-7695-1339-5, Ottawa, Canada
Wettel R, Marinescu R (2005) Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments. In: 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05). 115f, Timisoara, Romania
Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in Software Engineering. Springer Berlin, Heidelberg
Yin RK (2002) Design and methods. ICSM'00, 3rd edition. IEEE Computer Society 2002 ISBN 0-7695-1819-2, Montreal, Quebec, Canada
Zibran M, Roy CK (2013) Conflict-aware Optimal Scheduling of Code Clone Refactoring. IET Software 7(3):167–186
Zibran M, Saha R, Asaduzzaman M, Roy C (2011) Analyzing and forecasting near-miss clones in evolving software: An empirical study. In: International Conference on Engineering of Complex Computer Systems. IEEE Xplore Digital Library, Las Vegas, USA, pp 295–304

doi:10.1186/s40411-014-0012-8
Cite this article as: Kodhai and Kanmani: **Method-level code clone detection through LWH (Light Weight Hybrid) approach.** *Journal of Software Engineering Research and Development* 2014 **2**:12.