(IJACSA) International Journal of Advanced Computer Science and Applications,
Vol. 11, No. 6, 2020

Enhanced Pre-processing and Parameterization Process of Generic Code Clone Detection Model for Clones in Java Applications

Nur Nadzirah Mokhtar[1], Al-Fahim Mubarak-Ali[2], Mohd Azwan Mohamad Hamza[3]
Faculty of Computing, Universiti Malaysia Pahang
26300 Gambang, Pahang, Malaysia

***Abstract*—Code clones are repeated source code in a program. There are four types of code clone which are: Type 1, Type 2, Type 3 and Type 4. Various code clone detection models have been used to detect code clone. Generic Code Clone model is a model that consists of a combination of five processes in detecting code clone from Type-1 until Type-4 in Java Applications. The five processes are Pre-processing, Transformation, Parameterization, Categorization and Match Detection process. This work aims to improve code clone detection by enhancing the Generic Code Clone Detection (GCCD) model. Therefore, the Preprocessing and Parameterization process is enhanced to achieve this aim. The enhancement is to determine the best constant and weightage that can be used to improve the code clone detection result. The code clone detection result from the proposed enhancement shows that private with its weightage is the best constant and weightage for the Generic Code Clone Detection Model.**

***Keywords*—Code clone; code clone detection model; java applications; computational intelligence**

I. Introduction

Duplicated codes or better known as code clone are similar source codes that exist in a program [1-3]. Code clone brings maintenance issues in software. The more source codes are cloned in a program, the more memory and time needed in processing the software. At times, it also happens due to the software developer code writing practices [4]. Apart from that, if a source code contains bugs copied to the other parts of the software, the same bugs will be copied together throughout the program. This compromises the security of the software [3]. Code clone occurrence also depends on the deficiency of a programming language. As an instance, the Java programming language. Java is a worldwide open-sourced programming language used to develop open-source applications. In an experiment conducted to see the occurrence of code clones in Java applications, a total of 6% out of 512 000 lines of codes or 30 720 lines of codes from tested Java applications contains clones. One of the reasons for this occurrence is due to the absence of generic modules in Java [5].

At the initial stages of code clone detection, various approaches have been introduced. The approaches include text-based approach [6] [7], metric-based approach [8-10], tree-based approach [11-14], token-based approach [15-17] and graph-based approach [18-20]. The drawback of existing approaches is the lack of detecting all types of code clones [21]. In order to overcome this issue, code clone detection models were introduced to detect code clones that causes bad effect to the software. Code clone detection models is a model with combinatorial and structured processes that helps to detect and display detection result of code clone. Code clone detection models are recent development in the field of software clone and very little in terms of availability as tool, yet the existing code clone detection models have been a frontal movement in terms of having a combined process that detects code clone nevertheless of the diverse code clone jargons and programming languages.

As mentioned before, a model is an effort of unifying different processes to detect all code clone types. The effort can be seen through the Unified Clone Model [22] although this model is still in the design phase. Generic Pipeline Model [23-24] is a code clone detection model that detects on exact which is Type-1 and near-exact clones which is Type-2 in Java applications. An enhanced was proposed for this model by proposing a concatenation process, but it more focused on improving the time rather than improving the clone detection [25]. The disadvantage of this model is it only detect clones for Type-1 and Type-2. The state of the art model can be referred to as the Generic Code Clone Detection Model [26]. This model detects clones from Type-1 until Type-4 in Java applications. Type-3 refers to the source code that has modified semantically and Type-4 refers to the source code that has been modified further compared to Type-3.

This work focuses on improving code clone detection by enhancing the Generic Code Clone Detection Model through determining the best constant and weightage for Generic Code Clone Detection Model. Section 2 describes the Generic Code Clone Detection Model. Section 3 shows the implementation of the proposed enhancement while Section 4 discusses the findings of this work. This paper is summarized and concluded in Section 5.

II. Generic Code Clone Detection Model

Generic Code Clone Detection is a model that was designed and developed with an objective of detecting code clone from Java programming language [26]. It was designed into five processes which are elaborated in detail in upcoming sections. Fig. 1 illustrates the diagrammatic view of the model together with a brief narrative of the processes involved.

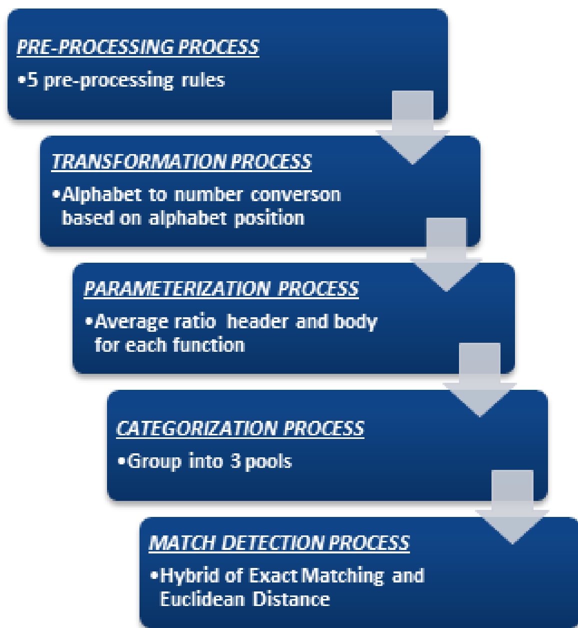

Fig. 1. Illustration of Generic Code Clone Detection Model [26].

A. Pre-Processing Process [26]

This model is initiated through this process. Source code alludes to the codes written in a source document of an application. It fills in as the contribution for the procedure. The source codes need to experience five joined rules used to accomplish the point of this process. Table I shows the rundown of these five rules. The yield of this process is standardized source codes or otherwise called source units. The source unit is still as source code. Each source unit speaks to a component of the source code.

B. Transformation Process [26]

This process is after the pre-processing process. The main objective of this process is changing the output of the previous process which is the source units into a more calculable format. The calculable format which is in the form of numbers are called as transformed source units and serves as the input in the determination of the parameters that will be used in the next process. The numerical form for this process is acquired from a letter to number substitution concept. The substitution is done based on the location of the alphabet. As an example, b is the third alphabet in the vocabulary sequence; therefore, b is changed to 02. This change is done on other alphabets.

The yield for this process is source units that has been transformed in number form. The source units that has been transformed are split to two branch which are the header (h) and body (b). Header refers to the transformed source unit that starts at the very first of the line of code and ends before the start of the body part of a transformed source unit. The body (b) is the body of a transformed source unit. As an instance in explaining the concept of a header (h) and body (b) in a Java function, assume a Java function named Function C with the written composition of:

```
public static void myMet ()
{
System.out.println("I love java");
}
```

After going through the initial pre-processing process, the source unit of Function C appear as:

public static void myMet systemoutprintln i love java

Therefore, the header and body of a function of Function C:

header (h): public static void myMet

body (b) : systemoutprintln i love java

C. Parameterization Process

This process starts after the transformation process. The transformed source units from the previous process serves as the input for this process. The attribute or parameter used for clone detection in this model is the average ratio for both header and body. Before demonstrating the step by step calculation for the average ratio header and body of a function, four important metrics is extracted from transformed source units. Table II shows attained metrics from the transformed source units.

To gain an average ratio of a function, the ratio of the header (h) and body (b) of the respective function must be gained initially. From the previous transformation process, the access modifier of all the function or method that has been changed to the value of public. Therefore, all the functions that has been changed to transformed source unit consist the equal access modifier value after going through the previous process. By using the value of public as the standard value, respective source units are divided with this standard value. It is done so that the header and body ratio value of each code of the transformed source unit is acquired. As an instance in calculating the average ratio for each transformed source unit, let's presume a transformed source unit contains a header, *TSUXa*, with body, *TSUXb*.

TABLE I. FIVE PRE-PROCESSING RULES

Pre-processing Rule	Description
PR-1	Import and package lines are excluded.
PR-2	Comment lines are excluded.
PR-3	Empty statements are excluded.
PR-4	Access modifier of a function is replaced with public.
PR-5	All the written source code lines is changed to lowercase format.

TABLE II. METRIC EXTRACTED FROM TRANSFORMED SOURCE UNITS

Metrics	Description
header code count	Total source code count in the header
body code count	Total source code count in the body
header ratio	header (h) ratio
body ratio	body (b) ratio
average header ratio	header (h) average ratio
average body ratio	body (b) average ratio

Therefore, the ratio of the transformed source unit is:

$$RA = \frac{(A1,A2,A3 \ldots An)}{P1} \quad (1)$$

$$RB = \frac{(B1,B2,B3 \ldots An)}{P1} \quad (2)$$

in which;

P1 is public access modifier value

RA is ratio value of header for each source units that has been transformed

RB is ratio value of body for each source units that has been transformed

A1, A2 A3..An is value of header in source units that has been transformed

B1, B2, B3.. Bn is value of body in source units that has been transformed

Once each function acquired the ratio of header and body, the next step is the calculation of the average ratio header and average ratio body of each transformed source unit. The formula of average ratio header and average ratio body in each transformed source units are:

$$AVRA = \frac{(RA)}{CA} \quad (3)$$

$$AVRB = \frac{(RB)}{CB} \quad (4)$$

in which:

AVRA is the value of average ratio for header in a transformed source unit

AVRB is the value of average ratio for body in a transformed source unit

CA is the total count of source code for header in a transformed source unit

CB is the total count of source code for body in a transformed source unit

The output of this process is the mentioned metrics; in which will be used in the next categorization process.

D. Categorization Process [26]

This process starts after parameterization process. The objective of this process is to pool the source units that has been transformed into a pool of code clones based on the exact ratio value of average ratio header and body for respective functions. This process uses metrics acquired from the parameterization process as input. The categorization is completed by grouping it into three pools using the average ratio value of the header and body of source units that has been transformed.

The first pool is for transformed source units for different functions that has the same value of header. As an instance, if transformed source unit for function E has the same header average ratio value with transformed source unit B, therefore these two transformed source units are categorized into the same group. This process will be continued until all the transformed source units that have the same average value of the header are categorized in the same pool. The second pool is for transformed source units for different functions that has the same value of body.

After the first pooling process, if transformed source unit for function E has the same body average ratio value with transformed source unit B, therefore these two transformed source units are grouped into the same category. This process will be continued until all the transformed source units that have the same average value of the body are categorized in the same pool. The remaining transformed source units is categorized into another category or better known as the third pool.

E. Match Detection Process [26]

This process comes after categorization process and it is the last process in this model. The main objective of this process is detecting code clone. The pool from the previous process is utilized as input for this process. Combination of exact matching and Euclidean Distance is used to detect code clone for this model. Exact matching is used on the first two pools to detect Type-1 and Type-2 clone. Once the detection is done for Type-1 and Type-2 from the first and second pools, the remaining transformed source units from the first and second pools is combined together with the third pool. As for the remaining average ratio header and body value in the third pool, Euclidean distance is used for Type-3 and Type-4 clone detection. As for the Euclidean distance application in this process, presume there are two transformed source units which are M and N. Therefore, the Euclidean distance, ED, between transformed source unit M and transformed source unit N is calculated as:

$$\mathrm{EDMN} = (headerM - headerN)^2 + (bodyM - bodyN)^2 \quad (5)$$

where;

EDMN is Euclidean distance of transformed source unit M and N

headerM is the average ratio header value of M

bodyM is average ratio body value of M

headerN is average ratio header value of N

bodyN is average ratio body value of N

As for the remaining body and header value in the final pool, the mathematical equation which is the Euclidean distance is utilized. Once the equation is utilized upon the remaining average ratio values of the functions, all the functions is gathered to Type-3 and Type-4 depending on the distance value that is gained. Range 0f 0.85 to 1 is categorized as Type-3 while the rest is categorized as Type-4.

III. Proposed Enhancement

The enhancement of the Generic Code Clone Detection Model [26] is focused on two of its process which is Pre-processing and Parameterization Process.

A. Enhancement on Pre-Processing Process

Pre-processing is a process that normalizes source code to produce better source units to be processed for clone detection. The enhancement done in this process is the removal of function regularization rule; which is PR-4: Regularize function access keyword to public. This is to maintain the original function keyword of a function. Therefore, the enhanced pre-processing remains with four pre-processing rules. Fig. 2 shows the enhanced Pre-processing process is elaborated in the form of pseudo code.

B. Enhancement on Parameterization Process

This process aims to create parameters or metrics that will be used for the categorization and match detection process. Therefore, the enhancement done in this process is the change of value access function based on three access functions and their respective weightage. The three access function is public with the weightage of 162102120903, private with the weightage of 16180922012005 and protected with the weightage of 161815200503200504. These values are based on the concept of the alphabet to number that has been explained in the Transformation Process. Fig. 3 shows the enhanced parameterization process is elaborated in the form of pseudo code.

```
Java application, J1
Source file, [S1, S2, S3, ...Sn]
Source code, [SC1, SC2, SC3, ...SCn]
Source unit, [SU1, SU2, SU3, ...SUn]
Pre-processing Rule 1, PR-1 Remove package and import statements.
Pre-processing Rule 2, PR-2 Remove comments.
Pre-processing Rule 3, PR-3 Remove empty lines.
Pre-processing Rule 4, PR-4 Regularize source codes to lowercase.

1. Read source file S1 in J1
2. For each S1,
3. Check SCi
4.    For each existing SCi
5.       Apply PR-1
6.       Apply PR-2
7.       Apply PR-3
8.       Apply PR-4
9.       Repeat on the remaining source codes [SC2, SC3, ...SCn] in
         S1
10. Continue step 2 until 10 on the remaining source files [S2, S3,
    ...Sn] in J1
```

Fig. 2. Enhanced Pre-Processing Process Pseudo Code.

```
Transformed source unit, [TSU1, TSU2, TSU3, ... TSUn]
header of source unit, [h1, h2, h3, ...hn]
body of source unit, [b1, b2, b3, ...bn]
Transformed source units in header, [TSUh1, TSUh2, TSUh3, ... TSUhn]
Transformed source units in body, [TSUb1, TSUb2, TSUb3, ... TSUbn]
Value of access function and weightage that starts with public, P1
Value of access function and weightage that starts with public, P2
Value of access function and weightage that starts with public, P3
Code count for each transformed source unit in header, [Ch1, Ch2, Ch3, ... Chn]
Code count for each transformed source unit in body, [Cb1, Cb2, Cb3, ... Cbn]
Ratio for transformed source unit in header, [Rh1, Rh2, Rh3, ... Rhn]
Ratio for transformed source unit in body, [Rb1, Rb2, Rb3, ... Rbn]
Average ratio for transformed source unit in header, [AVRh1, AVRh2, AVRh3, ... AVRhn]
Average ratio for transformed source unit in body, [AVRb1, AVRb2, AVRb3, ... AVRbn]

1. Read a transformed source unit TSU1
2.   For transformed source unit header, h1
3.     Calculate Rh1 by dividing each value in h1 with P1
4.     Count code for source unit in header, Ch1
5.     Calculate AVRh1 by dividing Rh1 with Ch1
6.   For transformed source unit body, b1
7.     Calculate Rb1 by dividing b1 with P1
8.     Count code for source unit in body, Cb1
9.     Calculate AVRb1 by Rb1 with Cb1
10. Continue with step 1 until 9 on the remaining transformed
    source units [TSU2, TSU3, ... TSUn] in finding remaining
    [AVRh2, AVRh3, ...AVRhn] and remaining [AVRb1, AVRb2,
    AVRb3, ...AVRbn]
11. Repeat step 1 until step 10 with P2
12. Repeat step 1 until step 10 with P3
```

Fig. 3. Enhanced Parameterization Process Pseudo Code.

IV. Result and Discussion

This section is divide into three subsections. The first subsection describes the result of the overall clone pair for Java applications from Bellon's benchmark data [27]. The second subsection describes the result of the overall clone pair based on the clone type for Java applications from Bellon's benchmark data [27]. The third subsection discusses the obtained results.

A. Overall Clone Pair in Java Application

Fig. 4 shows the overall clone pair for Java applications from Bellon's benchmark data. Based on Fig. 4, the highest overall clone pair detected for Eclipse-ant is from protected weightage with 7681 clone pairs. The second highest overall clone pair detected for Eclipse-ant is from private weightage with 4454 clone pairs. It is 42% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for Eclipse-ant. The third overall clone pair detected for Eclipse-ant is from the existing GCCD with 2688 clone pairs. It is 65% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for Eclipse-ant. The lowest overall clone pair detected for Eclipse-ant is from public weightage with 2654 clone pairs. It is 65.4% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for Eclipse-ant.

As for the Eclipse-jdtcore application, the highest overall clone pair detected for Eclipse-jdtcore is from protected weightage with 39974 clone pairs. The second highest overall clone pair detected for Eclipse-jdtcore is from private weightage with 15406 clone pairs. It is 61.5% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for Eclipse-jdtcore. The third overall clone pair detected for Eclipse-jdtcore is from the existing GCCD with 11268 clone pairs. It is 71.8% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for Eclipse-jdtcore. The lowest overall clone pair detected for Eclipse-jdtcore is from public weightage with 10767 clone pairs. It is 73.1% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for Eclipse-jdtcore.

As for the j2sdk1.4.0-javax-swing application, the highest overall clone pair detected for j2sdk1.4.0-javax-swing is from protected weightage with 56312 clone pairs. The second highest overall clone pair detected for j2sdk1.4.0-javax-swing is from private weightage with 8993 clone pairs. It is 84% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for j2sdk1.4.0-javax-swing. The third overall clone pair detected for j2sdk1.4.0-javax-swing is from public weightage with 7393 clone pairs. It is 86.9% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for j2sdk1.4.0-javax-swing. The lowest overall clone pair detected for j2sdk1.4.0-javax-swing is from the existing GCCD with 7281 clone pairs. It is 87.1% lower compared to the overall clone pairs detected from the protected weightage, that is, the highest overall clone pair detected for j2sdk1.4.0-javax-swing.

As for the Netbeans-javadoc application, the highest overall clone pair detected for Netbeans-javadoc is from private weightage with 937 clone pairs. The second highest overall clone pair detected for Netbeans-javadoc is from protected weightage with 654 clone pairs. It is 30.2% lower compared to the overall clone pairs detected from the private weightage; which is the highest overall clone pair detected for Netbeans-javadoc. The lowest overall clone pair detected for Netbeans-javadoc is from the existing GCCD and the public weightage with 595 clone pairs. It is 36.5% lower compared to the overall clone pairs detected from the private weightage; which is the highest overall clone pair detected for Netbeans-javadoc. The next subsection discusses the overall clone pair based clone type for each Java application from the Bellon benchmark data.

B. Overall Clone Pair based on Clone Type

Table III shows the overall clone pair based on the clone type for Java applications from Bellon benchmark data. As for Eclipse-ant application, the highest number of Type-1 clone pairs in Eclipse-ant was produced through the protected weightage which is 424 clone pairs. The second highest number of Type-1 clone pairs in Eclipse-ant was produced through the private weightage which is 246 clone pairs. The existing GCCD produced 185 clone pairs for Type-1; which is the same as the enhancement of the GCCD done using public weightage. This is the lowest amount of clone pair for Type-1 in Eclipse-ant. As for Type-2 clone in Eclipse- ant, the highest Type-2 clone pair in Eclipse-ant was produced through protected weightage with 916 clone pairs. The second highest Type-2 clone pair in Eclipse-ant was produced through private weightage with 750 clone pairs. The third highest Type-2 clone pair in Eclipse-ant was produced through protected weightage with 648 clone pairs. The lowest Type-2 clone pair in Eclipse-ant was produced through the existing GCCD with 552 clone pairs. As for Type-3 clone in Eclipse- ant, the highest Type-3 clone pair in Eclipse-ant was produced through protected weightage with 2296 clone pairs. The second highest Type-3 clone pair in Eclipse-ant was produced through private weightage with 2481 clone pairs. The third highest Type-3 clone pair in Eclipse-ant was produced through the existing GCCD with 581 clone pairs. The lowest Type-3 clone pair in Eclipse-ant was produced through the public weightage with 578 clone pairs. As for Type-4 clone in Eclipse-ant, the highest Type-4 clone pair in Eclipse-ant was produced through protected weightage with 4225 clone pairs. The second highest Type-4 clone pair in Eclipse-ant was produced through the existing GCCD with 1370 clone pairs. The third highest Type-4 clone pair in Eclipse-ant was produced through the public weightage with 1243 clone pairs. The lowest Type-4 clone pair in Eclipse-ant was produced through the private weightage with 977 clone pair.

As for the Eclipse-jdtcore application, the highest Type-1 clone pair in Eclipse-jdtcore was produced through protected weightage with 1008 clone pairs. The second highest Type-1 clone pair in Eclipse-jdtcore was produced through the private weightage with 766 clone pairs. The third highest Type-1 clone pair in Eclipse-jdtcore was produced through the public weightage with 627 clone pairs. The lowest Type-1 clone pair in Eclipse-ant was produced through the existing GCCD with 626 clone pairs. As for Type-2 clone in Eclipse-jdtcore, the highest Type-2 clone pair in Eclipse-ant was produced through protected weightage with 2952 clone pairs. The second highest Type-2 clone pair in Eclipse-jdtcore was produced through the existing GCCD with 2886 clone pairs. The third highest Type-2 clone pair in Eclipse-jdtcore was produced through the public weightage with 2882 clone pairs. The lowest Type-2 clone pair in Eclipse-jdtcore was produced through the private weightage with 2660 clone pairs. As for Type-3 clone in Eclipse-jdtcore, the highest Type-3 clone pair in Eclipse-jdtcore was produced through protected weightage with 22854 clone pairs. The second highest Type-3 clone pair in Eclipse-jdtcore was produced through the private weightage with 9634 clone pairs. The third highest Type-3 clone pair in Eclipse-jdtcore was produced through the existing GCCD with 4265 clone pairs. The lowest Type-3 clone pair in Eclipse-jdtcore was produced through the public weightage with 3866 clone pairs. As for Type-4 clone in Eclipse-jdtcore, the highest Type-4 clone pair in Eclipse-jdtcore was produced through protected weightage with 13169 clone pairs. The second highest Type-4 clone pair in Eclipse-jdtcore was produced through the existing GCCD with 3491 clone pairs. The third highest Type-4 clone pair in Eclipse-jdtcore was produced through the public weightage with 3392 clone pairs. The lowest Type-4 clone pair in Eclipse-jdtcore was produced through the private weightage with 2346 clone pairs.

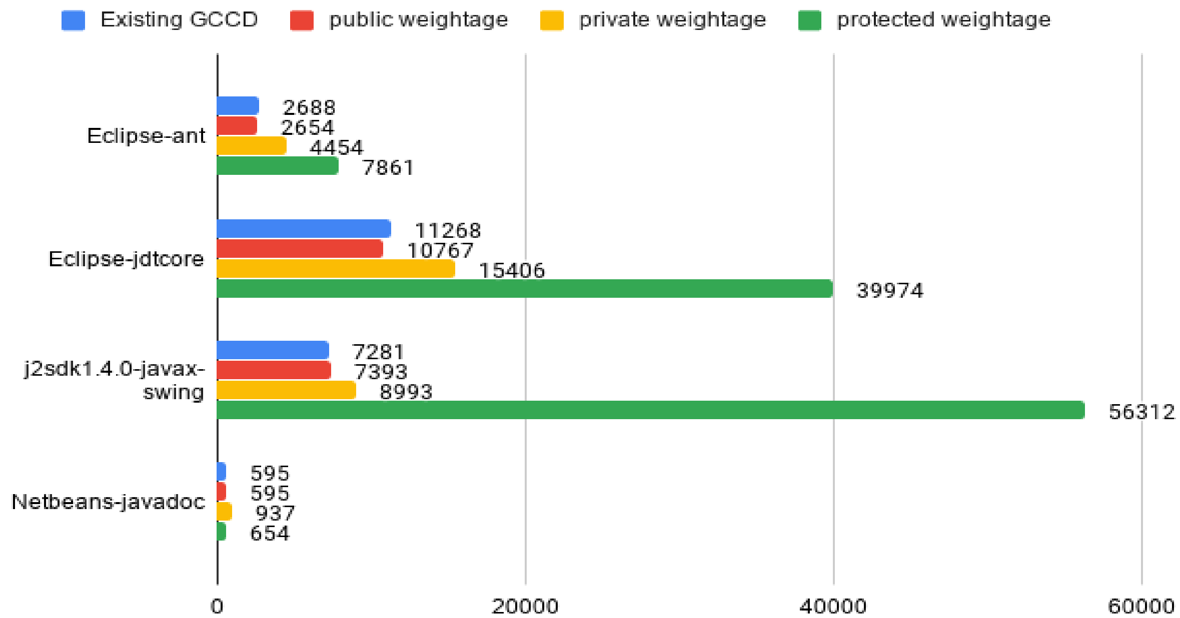

Fig. 4. Overall Clone Pair for Java Applications from Bellon Benchmark Data.

TABLE III. OVERALL CLONE PAIR BASED ON CLONE TYPE FOR JAVA APPLICATIONS FROM BELLON BENCHMARK DATA

Java Application	Clone Type	Existing GCCD	public weightage	private weightage	protected weightage
Eclipse-ant	Type-1	185	185	246	424
	Type-2	552	648	750	916
	Type-3	581	578	2481	2296
	Type-4	1370	1243	977	4225
Eclipse-jdtcore	Type-1	626	627	766	1008
	Type-2	2886	2882	2660	2952
	Type-3	4265	3866	9634	22845
	Type-4	3491	3392	2346	13169
j2sdk1.4.0-javax-swing	Type-1	877	891	1021	1330
	Type-2	3697	3713	3709	4259
	Type-3	1710	1774	1977	27316
	Type-4	997	1015	2286	23407
Netbeans-javadoc	Type-1	99	99	120	182
	Type-2	341	341	393	425
	Type-3	102	102	304	11
	Type-4	53	53	120	36

As for the j2sdk1.4.0-javax-swing application, the highest Type-1 clone pair in j2sdk1.4.0-javax-swing was produced through protected weightage with 1330 clone pairs. The second highest Type-1 clone pair in j2sdk1.4.0-javax-swing was produced through the private weightage with 1021 clone pairs. The third highest Type-1 clone pair in j2sdk1.4.0-javax-swing was produced through the public weightage with 891 clone pairs. The lowest Type-1 clone pair in j2sdk1.4.0-javax-swing was produced through the existing GCCD weightage with 877 clone pairs. As for Type-2 clone in j2sdk1.4.0-javax-swing, the highest Type-2 clone pair in j2sdk1.4.0-javax-swing was produced through protected weightage with 4259 clone pairs. The second highest Type-2 clone pair in j2sdk1.4.0-javax-swing was produced through the public weightage with 3713 clone pairs. The third highest Type-2 clone pair in j2sdk1.4.0-javax-swing was produced through the private weightage with 3709 clone pairs. The lowest Type-2 clone pair in j2sdk1.4.0-javax-swing was produced through the existing GCCD with 3697 clone pairs. As for Type-3 clone in j2sdk1.4.0-javax-swing, the highest Type-3 clone pair in j2sdk1.4.0-javax-swing was produced through protected weightage with 27316 clone

pairs. The second highest Type-3 clone pair in j2sdk1.4.0-javax-swing was produced through the private weightage with 1977 clone pairs. The third highest Type-3 clone pair in j2sdk1.4.0-javax-swing was produced through the public weightage with 1774 clone pairs. The lowest Type-3 clone pair in j2sdk1.4.0-javax-swing was produced through the existing GCCD with 1710 clone pairs. As for Type-4 clone in j2sdk1.4.0-javax-swing, the highest Type-4 clone pair in j2sdk1.4.0-javax-swing was produced through protected weightage with 23407 clone pairs. The second highest Type-4 clone pair in j2sdk1.4.0-javax-swing was produced through the private weightage with 2286 clone pairs. The third highest Type-4 clone pair in j2sdk1.4.0-javax-swing was produced through the public weightage with 1015 clone pairs. The lowest Type-4 clone pair in j2sdk1.4.0-javax-swing was produced through the existing GCCD with 997 clone pairs.

As for the Netbeans-javadoc application, the highest Type-1 clone pair in Netbeans-javadoc was produced through protected weightage with 182 clone pairs. The second highest Type-1 clone pair in Netbeans-javadoc was produced through the private weightage with 120 clone pairs. The lowest Type-1 clone pair in Netbeans-javadoc was produced through the private weightage and the existing GCCD with 99 clone pairs. As for Type-2 clone in Netbeans-javadoc, the highest Type-2 clone pair in Netbeans-javadoc was produced through protected weightage with 425 clone pairs. The second highest Type-2 clone pair in Netbeans-javadoc was produced through the private weightage with 393 clone pairs. The lowest Type-2 clone pair in Netbeans-javadoc was produced through the private weightage and the existing GCCD with 341 clone pairs. As for Type-3 clone in Netbeans-javadoc, the highest Type-3 clone pair in Netbeans-javadoc was produced through private weightage with 304 clone pairs. The second highest Type-3 clone pair in Netbeans-javadoc was produced through the public weighthage and the existing GCCD with 102 clone pairs. The lowest Type-3 clone pair in Netbeans-javadoc was produced through the protected weightage with 11 clone pairs. As for Type-4 clone in Netbeans-javadoc, the highest Type-4 clone pair in Netbeans-javadoc was produced through private weightage with 120 clone pairs. The second highest Type-4 clone pair in Netbeans-javadoc was produced through the public weightage and the existing GCCD with 53 clone pairs. The lowest Type-4 clone pair in Netbeans-javadoc was produced through the protected weightage with 36 clone pairs.

C. Discussion

The main aim of this work is to improve the code clone detection for Java applications by enhancing the Pre-processing and Parameterization process in the Generic Code Clone Detection Model. The pre-processing rule has been reduced from five rules to four rules by removing the regularization of function access modifiers. After that, the Parametrization process was enhanced with three different access functions and weightage. The three access functions are public with the weightage of 162102120903, private with the weightage of 16180922012005 and protected with the weightage of 161815200503200504. These values are based on the concept of the alphabet to number that has been explained in the Transformation Process. The result from these changes has been described in subsection 4.1 and subsection 4.3. Based on the result shown, the protected with the weightage of 161815200503200504 has shown more clone pair detection in three Java applications compared to the other success function. The three Java applications are Eclipse-ant, Eclipse-jdtcore and j2sdk1.4.0-javax-swing. The remaining Java application which is Netbeans-javadoc has more clone pair revealed through private with the weightage of 16180922012005 but has the second most clone pair detected through protected with the weightage of 161815200503200504.

This happens due to the enhancement made to the GCCD model. First is the removal keyword regularization rule from the pre-processing process. As mentioned previously, the pre-processing process of the GCCD at the start does the process of removing source code from uninteresting information. The uninteresting information is removed through the five rules previously that had been adopted in this process. The rules include removing packages and import statements, removing comments, removing empty lines, regularizing function access keyword to public and regularizing source codes to lowercase. These rules were set after taking into consideration in not making many changes to the original source codes. Too many changes in the source codes may cause critical information to be changed or removed; therefore, keeping a minimum set of rules ensures the most of the information of the source code such as the source code line and length is intact. The idea of removing the keyword regularization rule is to minimize the change of a function by sustaining original source code of a function. Furthermore, the different weightage of a constant influence the result. Based on the result, the higher the weightage value, the more clones are detected.

V. Conclusion

The idea of this work is to improve code clone detection in Java applications by enhancing the Generic Code Clone Detection Model. The enhancement involves by enhancing the Pre-processing and Parameterization Process. Based on the result shown, it can be concluded that the best constant and weightage for Generic Code Clone Detection Model is protected with a weightage value of 161815200503200504.

Acknowledgment

The authors would like express gratitude to Ministry of Higher Education Malaysia and Universiti Malaysia Pahang in supporting this work through the Fundamental Research Grant Scheme (FRGS Grant ID: RACER/1/2019/ICT01/UMP//1).

References

[1] J. Yang, Y. Xiong and J. Ma, "A function level Java code clone detection method," 2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Chengdu, China, 2019, pp. 2128-2134.

[2] D. K. Kim, “Enhancing code clone detection using control flow graphs, ” International Journal of Electrical and Computer Engineering (IJECE), Vol.9, No.5, October 2019, pp. 3804~3812.

[3] M. S. Rahman and C. K. Roy, "On the Relationships Between Stability and Bug-Proneness of Code Clones: An Empirical Study," 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), Shanghai, 2017, pp. 131-140.

[4] M. Pyl, B. van Bladel and S. Demeyer, "An Empirical Study on Accidental Cross-Project Code Clones," 2020 IEEE 14th International Workshop on Software Clones (IWSC), London, ON, Canada, 2020, pp. 33-37.

[5] M. Dagenais, J. F. F. Patenaude, E. Merlo, and B. Lagu¨e, “Clones occurrence in Java and Modula-3 software systems,” Advances in Software Engineering, 2002, pp. 95–110.

[6] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika and B. V. Saranya, "Detection of Type-1 and Type-2 Code Clones Using Textual Analysis and Metrics," 2010 International Conference on Recent Trends in Information, Telecommunication and Computing, Kochi, Kerala, 2010, pp. 241-243.

[7] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA, 2001, pp. 107-114.

[8] Vishwachi and S. Gupta, "Detection of near-miss clones using metrics and Abstract Syntax Trees," 2017 International Conference on Inventive Communication and Computational Technologies (ICICCT), Coimbatore, 2017, pp. 230-234.

[9] Z. Li and J. Sun, "An iterative, metric space based software clone detection approach," The 2nd International Conference on Software Engineering and Data Mining, Chengdu, 2010, pp. 111-116.

[10] M. Sudhamani and L. Rangarajan, "Code clone detection based on order and content of control statements," 2016 2nd International Conference on Contemporary Computing and Informatics (IC3I), Noida, 2016, pp. 59-64.

[11] Y. Yang, Z. Ren, X. Chen and H. Jiang, "Structural Function Based Code Clone Detection Using a New Hybrid Technique," 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, 2018, pp. 286-291.

[12] L. Büch and A. Andrzejak, "Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection," 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 2019, pp. 95-104.

[13] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang and X. Liu, "A Novel Neural Source Code Representation Based on Abstract Syntax Tree," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 783-794.

[14] H. Yu, W. Lam, L. Chen, G. Li, T. Xie and Q. Wang, "Neural Detection of Semantic Code Clones Via Tree-Based Convolution," 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, 2019, pp. 70-80.

[15] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic tokenbased code clone detection system for large scale source code,” IEEE Transactions on Software Engineering. 28(7), pp. 654–670.

[16] W. Toomey, "Ctcompare: Code clone detection using hashed token sequences," 2012 6th International Workshop on Software Clones (IWSC), Zurich, 2012, pp. 92-93.

[17] P. Wang, J. Svajlenko, Y. Wu, Y. Xu and C. K. Roy, "CCAligner: A Token Based Large-Gap Clone Detector," 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, 2018, pp. 1066-1077.

[18] C. M. Kamalpriya and P. Singh, "Enhancing program dependency graph based clone detection using approximate subgraph matching," 2017 IEEE 11th International Workshop on Software Clones (IWSC), Klagenfurt, 2017, pp. 1-7.

[19] Y. Higo, U. Yasushi, M. Nishino and S. Kusumoto, "Incremental Code Clone Detection: A PDG-based Approach," 2011 18th Working Conference on Reverse Engineering, Limerick, 2011, pp. 3-12.

[20] P. Gautam and H. Saini, "Type-2 software cone detection using directed acyclic graph," 2017 Fourth International Conference on Image Information Processing (ICIIP), Shimla, 2017, pp. 1-4.

[21] J. Harder, "The limits of clone model standardization," 2013 7th International Workshop on Software Clones (IWSC), San Francisco, CA, 2013, pp. 10-11.

[22] C. J. Kapser, J. Harder and I. Baxter, "A common conceptual model for clone detection results," 2012 6th International Workshop on Software Clones (IWSC), Zurich, 2012, pp. 72-73.

[23] B. Biegel and S. Diehl, “Highly Configurable and Extensible Code Clone Detection,”. 2010 17th Working Conference on Reverse Engineering, 2010, pp. 237–241.

[24] B. Biegel and S. Diehl, “JCCD: a flexible and extensible API for implementing custom code clone detectors,” Proceedings of the 2010b Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10), 2010, pp. 167-168.

[25] A. Mubarak-Ali, S. Sulaiman and S. M. Syed-Mohamad, "An enhanced generic pipeline model for code clone detection," 2011 Malaysian Conference in Software Engineering, Johor Bahru, 2011, pp. 434-438.

[26] A. Mubarak-Ali and S. Sulaiman, “Generic Code Clone Detection Model for Java Applications,” IOP Conference Series: Materials Science and Engineering, Volume 769, The 6th International Conference on Software Engineering & Computer Systems, 25-27 September 2019, Pahang, Malaysia.

[27] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," in IEEE Transactions on Software Engineering, vol. 33, no. 9, pp. 577-591, Sept. 2007.