An Empirical Study of a Hybrid Code Clone Detection Approach on Java Byte Code

Aritra Ghosh, Young Lee
Department of Electrical Engineering and Computer Science
Texas A&M University - Kingsville
Kingsville, TX 78363. USA
aritra.tech@gmail.com, young.lee@tamuk.edu

***Abstract*- Code clones increase the complexity of the system; therefore the software maintenance costs. Code clone detection techniques have been proposed and evaluated based on metric value and runtime evaluations. But in the existing methods, many false positive clones are detected. In this paper, we suggest a hybrid approach combining Program Dependence Graph-based technique with Metric-based technique to improve the precision of clone detection. We conduct a case study on two open source code Java projects such as Eclipse-ant and Eclipse-JDT core to show the effectiveness of our tool. The application of this hybrid technique is then compared with the existing clone detection technique, CloneDR. The result shows that our tool increases the performance in precision, recall, false positive and false negative compared to CloneDR.**

***Keywords*- Code Clone, Byte code, Metrics, False positive, False negative, Precision, Recall.**

1. INTRODUCTION

In today's community, the software has become the important entity and is an integral part of our life. Due to the development of technology, writing source code for a software system is no longer the most difficult part of software development in respect to cost and effort. Relatively, software maintenance and evolution have become the most challenging parts [1]. The term software maintenance refers to the modification of a software product after delivery to correct faults, to improve the performance or other attributes [2]. On the contrary, evolution refers to the process of developing software initially, then repeatedly updating it for various reasons [3]. Although for small systems, maintenance and evolution may not be an issue; for large software systems, their effects cannot be ignored. It has been found that almost 40-80% (average 60%) of the costs of developing a typical software system is consumed on the maintenance phase [4], which indicates there is a need for state-of-the-art techniques, methods, and tools to support maintenance and evolution.

Programmers often use code fragments by simple copy and paste them with or without adaptation. These identical code fragments are called as software clones [5]. Due to the copy-paste habits of programmers, clones are inevitable in software development. Previous studies have reported that the total quantity of cloning in software systems varies from 5-15% and can be even 50% of the main code [6]. Although some positive impacts of clones have been identified, their negative impacts cannot be ignored (e.g. increased program size, update anomalies) [7]. A code fragment having a bug causes the same problem to all other fragments copied from it. Fixing the bug requires the developer to check and update all copied locations as necessary. Enhancing a code fragment also requires the developer to look for its duplicated code fragments to ensure that changes are propagated to all desired locations, which also multiplies the work need to be done [8]. So, clones are treated as a "bad smell" [9] in code and are a major contributor to project maintenance difficulties.

Table 1.1: Description of different Clone Detection Techniques

Techniques	Description	Example
Text-based	1. Compare every line of code as a string. 2. Oldest and simplest technique. 3. It can detect code clones quickly compared with other detection techniques. 4. This technique requires no pre-processing on the source code.	SDD, NICAD, DuDe
Token-based	1. The source code is compiled and transformed into a sequence of tokens.	CCFinder, iClones, CP-Miner

DOI: 10.5176/2251-3043_5.2.367

© The Author(s) 2017. This article is published with open access by the GSTF

	2. Detects similarities of tokens as code clones. 3. Detection speed is lesser as compared with text-based techniques.	
Metric-based	1. Collect various metrics vectors and compare them. 2. The source code is transformed to its equivalent AST or PDG representation.	Davey
AST-based	1. A program is parsed to an abstract syntax tree and then divided into sub trees. 2. Common sub trees are regarded as code clones.	Asta, Tairas, Deckard
PDG-based	1. Code clones are detected by comparing PDGs created from source code. 2. Isomorphic sub graphs are regarded as code clones.	Duplix, Gabel

There are various clone detection techniques which can be classified into following categories [10]; According to Murakami et al., some text or metric-based techniques cannot identify code clones; whereas AST or PDG-based techniques need much time to find code clones. Text-based techniques can able to locate type I clone only; whereas token-based techniques can identify Type II clone also. AST-based techniques can find type III clone but this approach requires complex algorithm and parser and metric based tools are suitable for a large software system; but it cannot be applied to source code directly [11].

Program Dependence Graph-based technique is the only way which can detect code clones syntactically and semantically both. C. K. Roy et al. and Bellon et al. proved that technique indicates small recall and precision value. There are certain hybrid tools which are a combination of the abstract syntax tree, text-based or metric based approach; but can detect only syntactic clones [11].

The novel aspect of the work is using metric and program dependence graph-based technique in the detection process. To achieve this aim, the following objectives must be fulfilled:

I. To detect code clones in more efficient way, a novel approach is still needed.
II. Both syntactic and semantic clones should be detected.
III. The tool should be light weighted.
IV. Many false positive clones are detected which should be removed to get high precision value.

To fix those deficiencies, we suggest a hybrid approach for detecting code clones. It combines program dependence graph-based technique with metric-based technique.

The rest of the paper is organized as follows: Section 2 elaborates literature review. We provide an overall summary of the proposed methodology in Section 3. Section 4 gives the implementation and experimental result. Section 5 refers to the conclusion and future work.

2. LITERATURE REVIEW

Clone detection is widely open research area from last many years [12]. There are several techniques and tools are mentioned in literature and broadly categorized into following types:

2.1 Text-Based Code Clone Detection Techniques

In this method, line by line comparison has been made on two code fragments by textual similarity exists between them. These techniques do not require any filtration or normalization process [13] and apply directly on the source code. Johnson et al. [14] [15] enhances this process for better maintenance and reengineering of legacy systems. He found fingerprints for substring present in the source program and used them for comparison purpose. NICAD [16] is mainly a hybrid approach which uses tree concept along with text-based technique to detect clones. This tool works in two stages. Firstly flexible, pretty printing and normalization process is used to identify potential clones and then line by line textual comparison is made on these potential clones to find actual clones. SDD [17] is another text-based tool which is efficient to identify near-miss clones in the large software system. SDD algorithm uses the concept of n- neighbor approach for finding a number of repetitions in the system. Ducasse et al. [18] use string-based matching along with scattering plot diagrams to visualized clones present in the system.

2.2 Token-Based Code Clone Detection Techniques

Token-based clone detection technique uses the concept of parsing or lexical analysis to detect code clones. In this technique, the normalization process is used to convert source code into the intermediate stage which is the chain of tokens. These tokens are generated with the help of any parser and comparison algorithms are applied on them to detect clones. Dup [19] is a combination of text-based and token-based technique which divides a program in parameterized and non-parameterized tokens to find the Type I and the Type II clones. It uses hashing function to find the Type I clone and position index for the type II clones. CCFinder [20] is one of the efficient token based tools which can detect code clones from Java, C, C++, COBOL and many other source program files. This tool convert source file into series of tokens and then a comparison of these tokens are made with the help of suffix tree algorithm. CP-Miner [21] uses least common

© The Author(s) 2017. This article is published with open access by the GSTF

subsequence approach to detect clone activities in large software systems.

2.3 Tree Based Code Clone Detection Techniques

Type III clones or near-miss clones are tree-based in which code is modified [22]. In this source code is represented by abstract syntax trees in contrast to tokens and then pattern matching is applied to them to find similar sub trees which are considered as code clones [23]. Ira D. Baxter et al. [24] presents a tool CloneDr which generates abstract syntax tree by using parser; then three comparison algorithms are used to detect code clones. Jiang et al. [25] present a tool named DECKARD introduced a novel approach of a characteristic vector in Euclidian space to find similar sub trees.

2.4 Graph Based Code Clone Detection Techniques

In this technique, program dependence graph is obtained from source code as an intermediate state. To detect code clones, isomorphic sub-graphs are identified [23]. Komondor et al. [26] proposed an Approach called PDG-DUP that uses program dependence graphs (PDGs) and program slicing to search non-contiguous and intertwined clones that involve variable renaming and statement reordering. Krinke [27] presented an approach for classifying similar code fragments in programs based on searching identical sub-graphs in attributed directed graphs called 'Duplix'. Liu et al. [28] proposed a tool called GPlag which uses the PDG-based algorithm to analyze the graph and to detect the clones. It uses an algorithm called sequential pattern mining to discover copy/paste. It found the clone with high precision and added the new feature like text clone and text clone file ratio. Gabel et al. [29] projected a scalable detection algorithm for finding semantic clones. This algorithm is depended on selecting PDG sub graph based on its related structured syntax.

2.5 Metrics Based Code Clone Detection Techniques

In metric based code clone detection technique, different metrics of code are calculated, and code clones should possess similar values of these metrics. Jean Mayrand et al. [30] used this technique in the tool named 'Datrix' in which 21 metrics are calculated by four categories viz. name, layout, expression and control flow of program [31]. Kontogiannis et al. [32] use metrics technique in two different ways to detect code clones. In a first way, metrics are calculated for whole program or function. It compares data by data and control flow among methods. In a second way, it uses to do statement by statement analysis of the whole block by applying dynamic programming techniques.

2.6 Hybrid Code Clone Detection Techniques

There are certain hybrid tools which use the combination of above discussed syntactic and semantic techniques to detect clones. By utilizing benefits of various methods, it can identify all types of clones with more efficiency and accuracy. Koschke et al. [33] present a process which overcomes the limitation of token-based techniques by using Abstract Syntax tree with a combination of suffix tree algorithms. Leitao [34] applies a combination of both structural changes detection techniques and semantic techniques on programs to detect clones.

3. PROPOSED METHODOLOGY

Code clones are considered as a huge threat to maintenance and software efficiency. It is not feasible to track code clones manually. Hence, various clone detection techniques and tools are proposed, but there are certain limitations. So clone detection is an open research area.

3.1 Proposed Approach

The proposed work presents an automated clone detection tool for Java programs. This tool combines PDG based and metrics based technique to detect code clone efficiently. Our proposed tool focuses on the semantic information carried in PDG and applies comparison operation on this to find probable clones. After detection of a potential clone, it is necessary to verify whether they are real clones or false positives. To solve this purpose, various metrics are calculated, and comparison of yield values has been made. Hence, this tool goes through different phases during its clone detection life cycle. Figure 3.1 shows the proposed overall structure of our approach.

© The Author(s) 2017. This article is published with open access by the GSTF

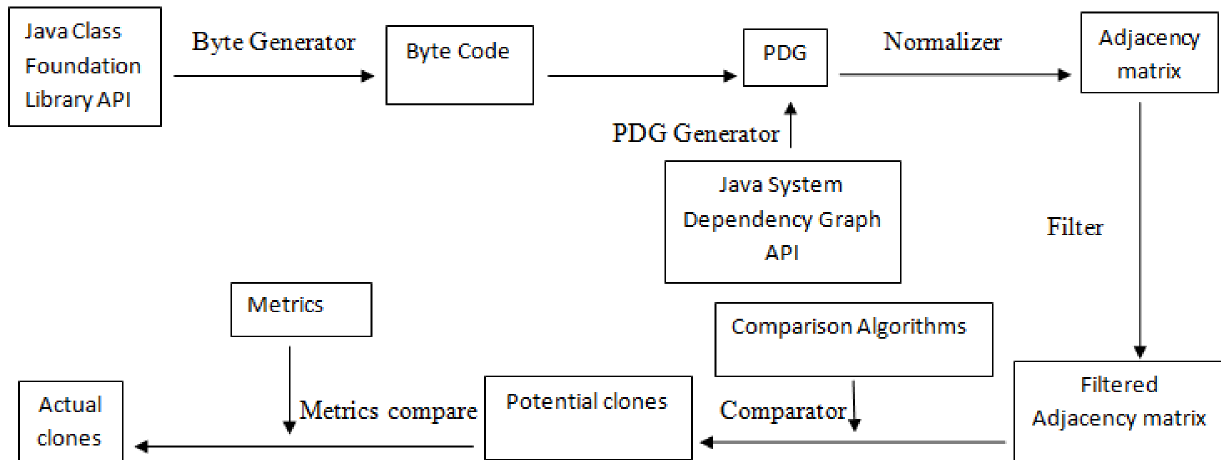

Figure 3.1: Overall architecture of our approach

3.2 Design

A. *Preprocessing Phase*

Jar files are kinds of Zipping files. Thus before extracting features from class files, we need to decompress the Jar file using the Java Class Foundation Library API, java.util.Jar to finish this job. We use the JarResource to obtain the byte codes of all classes in a Jar file. This is the first phase which determines whether inputted files are in .class format or not. As proposed system is only for Java programs and works on Java byte code, therefore, input files should be Java file with .class extension. Java .class file contains compiled byte code of particular file, and it transforms the code into a unified format by removing all syntactic dissimilarity that exists in the program. Java byte code is a complied code (low-level language code) of a program written by a programmer in a high-level language. It is considered as an intermediate representation of source code which makes Java programs independent of any platform. The key purpose of using java byte code is that API which we utilized for the creation of PDG and calculation of metrics works only on Java byte code. Moreover, it helps to find semantic clones by removing syntactic dissimilarities.

B. *Conversion Phase*

In clone detection process, the comparison algorithms are mostly applied for the intermediate stage except for text based detection process. The extraction of the intermediate stage from the source code is done in this phase. This intermediate stage can be tokens, trees, and graphs based on the clone detection technique applied. As program dependence graph-based technique is used here to detect code clone, so PDGs are obtained from source code during this phase of the proposed system. To get PDG, Java System Dependence Graph API is used which is the only Java API available to perform this function. After extraction of PDG, analysis on control dependence and data dependence nodes are made and stored in the form of adjacency matrices. If one node is data dependent on other then this relation is represented by 1, control dependence is represented by 2, and independent nodes are represented by 0.

C. *Normalization Phase*

Normalization phase is used to remove irrelevant differences exist in code fragments like whitespaces, comments, and layout. But program dependence graphs are independent of these changes, so it is not required to remove these differences from code fragments. However, PDG is sensitive to addition, deletion or modification in any statement and if such conditions do not contribute to any data and control flow of program, then they should be removed to reduce the number of comparisons and to detect clones more accurately. Due to these reasons, there is a need to further modify these extracted matrices and obtained filtered matrices. Hence, we remove those nodes which are control and data independent so that information about only semantically similar nodes should retain in the matrix which helps us to detect clones even when any data or control independent statement is added or deleted.

D. *Evaluation Phase*

After normalization, filtered matrices are obtained and fed into ‘Evaluation’ phase where it determines whether any similarity holds between both code fragments or not. For this purpose, a comparison algorithm compares the values of both matrices to find the corresponding node to node dependence between programs i.e. if one node is either

© The Author(s) 2017. This article is published with open access by the GSTF

control, data or independent of other nodes in the first matrix then corresponding node should contain same values on other nodes in the second matrix. If this similarity exists then, they are potential clones otherwise not. Hence, comparison algorithm is used to check data and control flow rather than any textual similarity. Therefore, if two program exhibits similar control and data flow, then they are considered to be potential clones.

E. Metrics Computation Phase

Now after getting code fragments as a potential clone, it is necessary to verify whether they are real clones or not which can be done by calculating metrics and then making the comparison on yield values. For this purpose, proposed tool calculates various control flow metrics and object-oriented metrics at class and function level for Java programs. To calculate object-oriented metrics, Java reflection API is used whereas 'control flow' metrics are considered with the help of program dependence graph (PDG). In Table 3.1, we are showing the different metrics used in our system.

Table 3.1: Metrics used in the system

Metric Type	Acronym	Description
Control Flow Metrics	Complexity	McCabe cyclomatic complexity
	C nodes	Number of control nodes
	D nodes	Number of data nodes
	Ed counts	Number of edges
Class Metrics	Fanout	Number of methods called
	T count	Total variables
	pubV	Number of public variables
	Protected	Number of protected variables
	Private	Number of private variables
Function Metrics	F name	Function name
	P type	Type of parameter passed
	R type	Return type
	n Parameter	Total number of parameters

3.3 Implementation

The tool proposed here can detect all types of clone efficiently. This tool can identify both Type I and Type II clones. Moreover, they are again verified by comparing obtained metrics. For Type III clones, insertion and deletion made on the same statement that does not affect control and data dependency exist between them can be simply detectable. As the comparison is prepared by control and data relation, that remain same in a modified version of the same program. However, when any data or control independent statement is added, then it is already removed by filter function, so the system can detect Type III clones efficiently.

Type III clone detection can be illustrated with the help of the following example: Two Java .jar files InputJar1.jar and InputJar2.jar are entered into the proposed system. Here both files differ in the position of independent control declaration. Addition and reordering of this statement do not affect the flow of the program and consider as type III clone.

Now PDG of both files should be obtained with the help of function used to generate them. After obtaining PDG, an adjacency matrix is obtained which represents data dependency with 1, control dependence with 2 and independent nodes with 0.

Jar files are kinds of Zipping files. Thus before extracting features from class files, we need to decompress the Jar file using the Java Class Foundation Library API, java.util.Jar to finish this job. We use the JarResource class to obtain the byte codes of all classes in a Jar file.

The first function which is used after creation of program dependence graph is filter function which removes all the data and control independent statements which do not affect the flow of the program. In this task, if the value of both row and column corresponding to a particular node is zero then this node is considering as an independent node and should be removed from the matrix as it does not affect the control and data flow of the program. Compare function

© The Author(s) 2017. This article is published with open access by the GSTF

is used to detect potential clones. In this node to node comparison is made as for clones programs similar dependency should exist between programs. This function takes filtered adjacency matrix of both files and finds whether clone relation may exist between them or not.

Clone ratio can be defined as a percentage of nodes matched. If all the nodes of code fragments are not matched then it tells how much percent of total code is considered as code clones. After finding potential clones, next step is to prove them real clones. Hence, various metrics for these potential clones are calculated with the help of Count_Metric algorithm. After collecting metrics of both files, their values are compared with the help of in built comparison function using Java Array class. If all the metrics values are equal then, they are actual clones otherwise not.

4. IMPLEMENTATION AND EXPERIMENTAL RESULT

To estimate the performance and efficiency of our approach, we have performed an experiment. The purpose of this experimentation was to compare the usefulness, understandability, and performance of this way.

4.1 Experimental Method

The working of proposed tool starts with Adaption Phase i.e. by giving two Java jar files as input with the help of the user. For this purpose startup page of the tool is created by using Java frames. To choose files with the help of the user, Java FileChooser function is added which allow selecting only .jar files. The input file is selected with the help of File buttons. File1 and File2 Button handle the fileChooser event and display the absolute path of java .jar file on java text box. When File 1 or File 2 button is clicked by the user, then open dialogue box is appeared to choose Java .jar files to find clones in the system.

After uploading two .jar files, we get the results of desired clones. Various object-oriented metrics and control metrics are calculated to prove them as actual clones. Hence in this way, all the phases of proposed tool are followed step by step. We prepared two sets of classes - two open source code Java projects such as Eclipse-ant and Eclipse-JDT core.

1) Apply our clone detection program and other clone detection programs to these two sets.
2) Show the results.
3) Randomly choose a couple of classes from Eclipse-ant and change names and syntax a little.
4) Copied these updated classes on the files in Eclipse-JDT core.
5) Apply our clone detection program and other clone detection programs to these two sets.
6) Show the results.

Table 3.2 shows the details of the scenarios.

Table 3.2: Description of test cases

Test Cases	Result
Original and target JAR files are the same	1st part of Figure 3.2 shows that the Certainty percent = 100 Number of similar function =1120 Number of functions in 1st JAR = 1120 Number of functions in 2nd JAR = 1120
Two different JAR files	2nd part of Figure 3.2 shows that the Certainty percent = 23.07 Number of similar function=3 Number of functions in 1st JAR = 1120 Number of functions in 2nd JAR = 13
Two identical JAR files, each one containing only one .class. In the second file, we modify the following in one method only • Method name • Return data type • Input parameter data type	3rd part of Figure 3.2 shows that Certainty percent = 33.33 Number of similar function=1 Number of functions in 1st JAR = 3 Number of functions in 2nd JAR = 3

Figure 3.2 shows the result of three test cases we have examined.

© The Author(s) 2017. This article is published with open access by the GSTF

```
-----Final comments-----
Total number of functions in .class files of first JAR file is : 1120
Total number of functions in .class files of second JAR file is : 1120
Number of matched functions in second JAR file is : 1120
Certainty Percentage is : 100.0
```

```
-----Final comments-----
Total number of functions in .class files of first JAR file is : 1120
Total number of functions in .class files of second JAR file is : 13
Number of matched functions in second JAR file is : 3
Certainty Percentage is : 23.076923
```

```
-----Final comments-----
Total number of functions in .class files of first JAR file is : 3
Total number of functions in .class files of second JAR file is : 3
Number of matched functions in second JAR file is : 1
Certainty Percentage is : 33.333336
```

Figure 3.2: Three test case scenarios

4.2 Experimental Result

It is pretty difficult to make an accurate clone set because of the ambiguity of clones. Though there are several benchmarks on clone detectors. Bellon and his colleagues found exact set checking manually [35]. They experimented on eight open source projects with six clone detectors to prove them whether there are actual clones or not. For a huge number of collected clones, they arbitrarily choose few of them.

Roy et al. compared clone detectors with four distinct scenarios [35]. The main thought of their study is

a. False Negatives and False Positive

False negative in % = $|N|/|A| * 100$

False positive in % = $|P|/|D| * 100$

Where,

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

using mutation by which technique mutants are generated and injected and evaluates detectors with them.

Ducasse et al. used string matching on some different languages including COBOL, Java, or C++, etc. to find high precision and recall values [35].

We followed the Bellons' benchmark. We took four open source Java projects to evaluate the result for our tool. It presents the numbers of 'actual', 'detected' and 'correctly detected' clones for different categories of clone types by our proposed tool.

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

Actual clones [A] are the reference clones.

Table 4.2 shows the false negative and false positive determined by the clones for the projects.

Table 4.2: Calculated false negative and false positive

Projects	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	False Negatives [N]	False Negatives in %	False Positives [P]	False Positives in %
Apache-httpd-2.2.8	20	20	19	1	5	0	0
Eclipse-ant	15	15	14	1	6	0	0
Eclipse-jdtcore	16	16	16	0	0	0	0
J2sdk-swing	21	21	21	0	0	0	0

© The Author(s) 2017. This article is published with open access by the GSTF

b. Precision and Recall

The quality of the system can be estimated through the quality metrics. The quality metrics considered in the proposed methodology are:

- Precision
- Recall

1. Precision: Precision measures the proportion of actual clones which are correctly identified [24].

Precision = Number of clones correctly found / Total number of clones

2. Recall: Recall measures the proportion of non-clones which are correctly identified [24].

Recall = Number of clones found correct / Total number of clones in the source code

High precision shows that there are mostly appropriately recognized code clones and low precision indicates that all the code clones are not true. On the other hand, high recall demonstrates that most of the code clones in the source code have been identified; low recall indicates that most of the code clones in the source code have not been located. While comparing code clone detection techniques, precision and recall values are judged for accuracy.

From the data presented that has been given in Tables 4.3 and 4.4, it could be seen that our proposed tool 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 tool is found to be efficient for identifying all kinds of clones.

Table 4.3: Calculated Precision and Recall value for type I type II clone

Projects	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision in %	Recall in %	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision in %	Recall in %
Apache-httpd-2.2.8	203	192	183	95	90	252	249	242	97	96
Eclipse-ant	382	374	363	97	95	379	422	372	88	98
Eclipse-jdtcore	1603	1585	1427	90	89	6057	5686	5573	98	92
J2sdk-swing	8820	8196	8115	99	92	8728	8918	8205	92	94

This tool can identify both the Type I and the Type II clones efficiently. For Type III clones, insertion and deletion made on the same statement that does not affect control and data dependency exist between statements. As a comparison is finished by control and data relation, that remain same in a modified version of the same program. However, when any data or control independent statement is added, then it is already removed by filter function, so the system can detect Type III clones efficiently.

Table 4.4: Calculated Precision and Recall value for type III and type IV clone

Projects	Type III					Type IV				
	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision in %	Recall in %	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision in %	Recall in %
Apache-httpd-2.2.8	807	756	711	94	88	11	11	10	90	90

© The Author(s) 2017. This article is published with open access by the GSTF

Eclipse-ant	448	426	426	100	95	10	10	10	100	100
Eclipse-jdtcore	4864	4378	4378	100	90	17	17	15	88	88
J2sdk-swing	12052	12737	11209	88	93	31	32	30	92	95

From the data presented that has been given in Tables 4.3 and 4.4, it could be seen that our tool 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 tool is found to be an efficient tool for identifying all kinds of clones.

4.3 Comparison with Existing Tools

In this section, we compared our tool with CloneDR using the same example sets. CloneDR is an existing Java clone detection tool which identifies both exact and near-miss clones in software systems. It can find clones with the different format, variable names, and code snipers.

Table 4.5: Calculated false negative and false positive (CloneDR)

Projects	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	False Negatives [N]	False Negatives in %	False Positives [P]	False Positives in %
Apache-httpd-2.2.8	20	20	18	2	6	0	0
Eclipse-ant	15	15	12	3	7	0	0
Eclipse-jdtcore	16	16	16	0	0	0	0
J2sdk-swing	21	21	21	0	0	0	0

Table 4.6: Calculated Precision and Recall value for type I and type II clone (CloneDR)

Projects	Type I					Type II				
	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision in %	Recall in %	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision in %	Recall in %
Apache-httpd-2.2.8	203	191	181	90	88	252	249	242	97	96
Eclipse-ant	382	374	363	97	95	379	422	372	88	98
Eclipse-jdtcore	1603	1585	1427	90	89	6057	5680	5571	90	90
J2sdk-swing	8820	8195	8110	92	90	8728	8915	8200	89	88

Table 4.7: Calculated Precision and Recall value for type III and type IV clone (CloneDR)

© The Author(s) 2017. This article is published with open access by the GSTF

Projects	Type III					Type IV				
	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision in %	Recall in %	Actual Clones [A]	Detected Clones [D]	Correctly Detected Clones [C]	Precision in %	Recall in %
Apache-httpd-2.2.8	807	756	711	94	88	11	11	10	90	90
Eclipse-ant	448	426	426	100	95	10	10	10	100	100
Eclipse-jdtcore	4864	4377	4377	100	90	17	17	14	87	87
J2sdk-swing	12052	12737	11208	87	92	31	33	30	92	94

4.4 Analysis of the Result

From the above tables, we can compare our model with the existing CloneDR tool in respect to False Negative, False Positive, Precision, and Recall. Here we are observing that False Positive value of our model is low compared to CloneDR. False Positive reports the number of clones wrongly detected as clones. So we have a better result. Again, the Recall value of our model is high compared to CloneDR. High recall admits that most of the source code clones have been found. It means the performance of our model is more accurate compared to CloneDR.

4.5 Threads of Validity

A. Internal Validity: Threat of internal validity is about the capacity of our experiments to relate the dependent and independent variables. The threat may be exposed through investigational or individual errors. We did a manual analysis to validate the accurateness of the clone detection. The manual evaluation can have human errors. Again, there is a lot of metric parameters from which we used few of them. More metric value comparison may change the result.

B. External Validity: Threats to external validity correspond to the way of generalizing our results. We had done our comparison with other existing tools in respect to precision and recall. However, this does not declare that the same result would be found for other programming languages.

C. 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 precision & recall values and low in run-time values.

5. CONCLUSION AND FUTURE WORK

The proposed tool is a hybrid approach tool which combines program dependence graph-based clone detection technique with metrics-based technique. Program dependence graph technique is used to find potential clones in the system while the metrics-based technique is used to verify them as actual clones. As PDG carries semantic information of system, hence proposed tool can detect both syntactic as well as semantic similar code clones. The proposed tool finds code clones only for programs written in Java language. This tool goes through five phases during its clone detection life cycle. Java byte code is given as input to the system as it removes all structural dissimilarities that exist in the system and converts code fragments into unified code format. PDG is obtained with the help of Java System Dependence Graph API which is displayed in java frame with the help of Java Swings. The adjacency matrix is achieved with the help of Java System Dependence Graph API where data dependency among nodes represented by 1, control dependency by 2 and independent nodes by 0. These adjacency matrices are filtered to remove independent nodes. Node by node comparison is made to prove them potential clones. Various object-oriented metrics at the class level and function level are computed using reflection API. Various control metrics are calculated with the help of obtained PDG. The proposed tool compares these metrics values to find whether potential clones are actual clones or not.

This approach is implemented only for Java programs. In future it can be adapted for other languages like C++, C#, etc. so that it becomes language independent. More metrics can be calculated with it to get more understandable results. The efficiency of the tool can be improved for type IV clone where reordering of control and data dependent statement is associated. Calculated metrics can also be used to rank code clones for efficient clone management. This tool can be further enhanced by using clone removal techniques after detecting actual clones.

© The Author(s) 2017. This article is published with open access by the GSTF

REFERENCES

[1] K. H. Bennett, and V.T Rajlich, “Software Maintenance and Evolution: a Roadmap,” in ICSE '00 Proceedings of the Conference on The Future of Software Engineering Pages 73-87.

[2] Shahid Hussain, Muhammad Zubair Asghar, Bashir Ahmad and Shakeel Ahmad, “A Step towards Software Corrective Maintenance: Using RCM model,” (IJCSIS) International Journal of Computer Science and Information Security, Vol. 4, No. 1 & 2, 2009.

[3] Mrs. E.Kodhai, V.Vijayakumar, G. Balabaskaran, T.Stalin, and B.Kanagaraj, “Method Level Detection and Removal of Code Clones in C and Java Programs using Refactoring,” International Journal of Computer Communication and Information System (IJCCIS) – Vol2. No1. ISSN: 0976–1349 July – Dec 2010.

[4] Robert L. Glass, “Frequently Forgotten Fundamental Facts about Software Engineering,” an article in IEEE Software May/June 2001.

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

[6] C.K. Roy, and J.R. Cordy, “A Survey on Software Clone Detection Research,” Queen’s School of Computing Tech. Report 2007-541, Kingston, 2007, 115 pp.

[7] C.J. Kapser, and M.W. Godfrey, ““Cloning Considered Harmful” Considered Harmful: Patterns of Cloning in Software,” Emp. Soft. Eng., 13(6), 2008, pp. 645-692.

[8] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code,” in OSDI, San Francisco, 2004, pp. 289–302.

[9] T. Bakota, R. Ferenc, and T. Gyimothy, “Clone Smells in Software Evolution,” in ICSM, Paris, 2007, pp. 24–33.

[10] D. Gayathri Devi, and Dr.M.Punithavalli, “Developing a Novel and Effective Clone Detection Using Data Mining Technique,” International Journal of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 8, August 2012.

[11] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto, “Gapped Code Clone Detection with Lightweight Source Code Analysis,” ICPC 2013, San Francisco, CA, USA, 978-1-4673-3091-6/13/$31.00 c 2013 IEEE.

[12] C. K. Roy, M. F. Zibran, R. Koschke, “The Vision of Software Clone Management: Past, Present, and Future (Keynote Paper),” in IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR- WCRE), Software Evolution Week, 2014, pp.18-33.

[13] C. K. Roy, James R. Cordy, and Rainer Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” in Science of Computer Programming, May 2009, pp. 470-495.

[14] J. Johnson, “Identifying redundancy in source code using fingerprints,” in: Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), 1993, pp. 171-183.

[15] J. Johnson, “Visualizing textual redundancy in legacy source,” in: Proceedings of Conference of the Centre for Advanced Studies on Collaborative research, (CASCON), 1994, pp. 171-183.

[16] C. Roy and J. Cordy, “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” In 16th IEEE International Conference on Program Comprehension, 2008, pp. 172–181.

[17] Seunghak Lee and Jeong Iryoung, “SDD: high-performance code clone detection system for large scale source code,” In Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, 2005, pp. 140-141.

[18] S. Ducasse, M. Rieger, and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” in Proceedings of the 15th International Conference on Software Maintenance *(ICSM’99)*, September 1999, pp. 109–118.

[19] Baker, Brenda S, “On finding duplication and near-duplication in large software systems,” In Proceedings of 2nd Working Conference on Reverse Engineering, IEEE, 1995, pp. 86-95.

[20] T. Kamiya, S. Kusumoto, K. Inoue, “CCFinder: A multi-linguistic token- based code clone detection system for large scale source code,” in IEEE Transactions on Software Engineering, 2002, pp. 654-670.

[21] Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou, “CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code,” Software Engineering, IEEE Transactions, vol. 32, March 2006, pp. 176-192.

[22] D. Rattan, Rajesh Bhatia, and Maninder Singh, “Software clone detection: A systematic review,” Information and Software Technology, vol. 55, no. 7, 2013, pp 1165-1199.

[23] Jiang, Zhen Ming, and Ahmed E. Hassan, “A framework for studying clones in large software systems,” In Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2007, pp. 203-212.

[24] I. D. Baxter, A. Yahin, L. Moura, M. SantAnna, L. Bier, “ Clone Detection using abstract syntax trees,” in Proceedings of the 14th International Conference on Software Maintenance (ICSM ‟98), Bethesda, Maryland, USA, 1998, pp. 368-378.

[25] Jiang, Lingxiao, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. “Deckard: Scalable and accurate tree-based detection of code clones,” In Proceedings of the 29th International conference on Software Engineering, Minneapolis, MN, USA, 2007, pp. 96-105.

[26] R. Komondoor, S. Horwitz, “Using slicing to identify duplication in the source code,” in Proceedings of the 8th International Symposium on Static Analysis (SAS' 01), Vol. LNCS 2126, Paris, France, 2001, pp. 40-56.

[27] J. Krinke,“Identifying Similar code with program dependence graphs,” In Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, 2001, pp. 301-309.

[28] C. Liu, C. Chen, J. Han, P. S. Yu, “GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis,” In Conference on Knowledge Discovery and Data Mining, 2006, pp. 872-881.

[29] Gabel, Mark, Lingxiao Jiang, and Zhendong Su, “Scalable detection of semantic clones,” In 30th International Conference on Software Engineering, (ICSE'08), ACM/IEEE, 2008, pp. 321-330.

[30] Johnson, J. Howard, “Identifying redundancy in source code using fingerprints,” In Proceedings of the conference of the Centre for Advanced Studies on Collaborative research: software engineering- IBM Press, vol. 1, 1993, pp. 171-183.

[31] J. Mayrand, Claude Leblanc, and Ettore M. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” In Proceedings of International Conference on Software Maintenance, IEEE, 1996, pp. 244-253.

[32] A. Kostas Kontogiannis, Renator DeMori, Ettore Merlo, M. Galler, and Morris Bernstein, “Pattern matching for clone and concept detection,” In Reverse engineering, Springer US, 1996, pp. 77-108.

[33] R. Koschke, Raimar Falke, and Pierre Frenzel, “Clone Detection using abstract syntax suffix trees,” In 13th Working Conference on Reverse Engineering (WCRE'06), IEEE, 2006, pp. 253-262.

© The Author(s) 2017. This article is published with open access by the GSTF

[34] Leitao Antonio Menezes, "Detection of redundant code using R 2 D 2," Software quality journal, vol. 12, no. 4, 2004, pp. 361-382.

[35] G. Anil Kumar, Dr. C.R.K.Reddy, Dr. A. Govardhan, "AN EFFICIENT METHOD-LEVEL CODE CLONE DETECTION SCHEME THROUGH TEXTUAL ANALYSIS USING METRICS," INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING & TECHNOLOGY (IJCET) ISSN 0976 – 6367(Print) ISSN 0976 – 6375(Online) Volume 3, Issue 1, January- June (2012), pp. 273-288.

Authors' Profile

Aritra Ghosh received the M.S. degree in Computer Science from Texas A&M University-Kingsville in 2015. He is currently pursuing PhD in Computer Science, Florida Atlantic University. His research interests include Source Code Visualization, Tracking Software Evolution, Data Mining and Machine Learning, Language for Mobile Sensor Application, Human in loop in Self-Adaptive System.

Young Lee received the PhD degree in Computer Science and Software Engineering from Auburn University in 2007. He is an Associate Professor in the Department of Electrical Engineering and Computer Science, Texas A&M University-Kingsville. His research interests include Software Visualization, Reverse Engineering, Computer Science Education, and STEM Education.

© The Author(s) 2017. This article is published with open access by the GSTF