



An Emerging Approach towards Code Clone Detection: Metric Based Approach on Byte Code

Kanika Raheja

CSED, Thapar university, India.

Rajkumar Tekchandani

CSED, Thapar university, India.

Abstract— In software programs different kind of redundancy can be found. This kind of redundancy in the code is called as code clone. The impact of cloning on software maintenance is problematic and is of great concern. So the need for identification of clones is necessary and this process is called Clone detection. Various clone detection techniques are proposed, that are quite efficient in providing clones. The approaches mainly followed for clone detection are text based comparison and token based approach. These techniques take a huge amount of time and the process of comparison is very tedious and expensive. Moreover tree based techniques are very complex. The present study addresses a metric based technique which can be applied as a pre-filtration step to reduce the complexity encountered with the previous techniques. MCD Finder is proposed for calculating metrics of a Java program. It uses byte code to calculate the metrics of Java programs, instead of using any transformed representation. The reason of using byte code is that it is platform independent and represents the unified structure of the code. So the proposed approach also detected semantic clones up to some extent. This approach can be used independently and it can be combined with other approaches to detect clones.

Keywords— Clone detection, metrics computation, hybrid approach, complexity, byte code

I. INTRODUCTION

The software life cycle comprises of three steps: first we have to clearly define the requirements, implement these requirements; and then we have to maintain the software and evolve it according to user's requirements. But from the development point of view maintenance is the most crucial activity in terms of cost and effort. Code clones are considered one of the bad smells of software system [2] and indicators of poor maintainability. Various studies show that the software system with code clones is difficult to maintain as compared to non-cloned code software system [1].

Code clones are the result of copy paste activities which are syntactically or semantically similar. The reason behind cloning can be intentional or unintentional. To exacerbate the situation cloning is performed rapidly and without a care about the context. This means that error free code becomes buggy after cloning [11]. Furthermore programmer often copies the other's code without fully understanding it. Considering all these factors, and from a maintenance point of view it is beneficial to detect clones and remove them by constantly monitoring software. Code clone detection could be useful for maintenance, re-engineering and plagiarism detection. For clone detection various techniques and tools had been proposed on the basis of detecting various types of clones. Clones are classified on the basis of textual and functional similarity [3] [4]. Identical code segment comes in type-1 clones; identical code segment with slight renaming comes in type-2 clones; and renamed cloned segment with extra statements added or deleted comes in the category of type-3 clones and all these types of clones comes in the category of textual similar clones [5]. Type-4 clones come in the category of functional clones and these types of clones are semantic clones. The detection of all these types of clone varies according to the clone detection technique. For example text based technique can detect type1 clone only [5]. Token based approach detects type1 and type2 clones. These techniques cannot detect type3 clones. Generally for detecting type3 clones Abstract syntax Tree based approach [7] and for detecting type-4 clones Program Dependence graph approach [8] is used. Another technique for clone detection is a metric based approach that does not work directly on the source code for calculating clones.

This process of clone detection can be done manually by comparing lines one by one but for the large program this process will become very tedious. Hence a tool should be designed for detecting the clones because it is very important from a maintenance point of view. Various tools had been proposed for detecting clones that are based on different algorithms [1] [10]. For example CCFinder [6] is a standalone tool that is based on a token based approach that is efficient in detecting type 1 and type 2 clones.

A clone detection system should give precise and useful information about clones since a large number of clones are expected to be found in large software system. In the presented work metric based approach for selecting potential clones is used as all the other technique requires a large amount of comparison. Potential clones are selected with the help of this approach. And by combining this approach with other approaches a hybrid approach can be designed. Section II describes the related work. Section III illustrates how this technique of metric based approach for Java program is implemented and section IV illustrates the results. Section V describes the conclusion and future scope.

II. RELATED WORK

Clone Detection has been an active research area since 1990's [1] [12]. Code clone detection is directly related to maintenance of software, code refactoring and reengineering. The survey shows the various techniques and algorithms to detect clones [5], [13]. Yue Jia et al. [14] proposed that the algorithm to detect clones should be precise and efficient. He proposed a new algorithm, KClone which is a novel combination of local dependence analysis and lexical analysis with a high value of precision. Chanchal K. Roy et al. [5] in his survey paper broadly classify the various techniques for clone detection. Clone detection approaches are broadly classified in five techniques [5] which are described below:

A. Textual comparison

The Textual approach uses no transformation of the source code before applying the comparison, and source code is used directly in the clone detecting process. The text based approach is the efficient technique but it can detect type1 clone only [13]. The integrity of this approach cannot be assured because it cannot detect the structural type of clones having different coding but same logic. Johnson approach [17] uses "fingerprints" on substrings of the source code to detect clones.

B. Token based comparison

Token Based Approach needs a parser or lexer to normalize the code in the form of tokens. First of all source code is changed in the form of tokens by using parser or lexer, then the comparison is applied on the changed code (intermediate representation). This technique is more robust as compared to the text based approach when blank spaces and comments are present in the source code. But its accuracy is not that much good because while conversion (source code in the token sequence) various false positive may introduce in the code. Various tools are proposed for clone detection that is based on token based approach. Toshihiro Kamiah et al. [6] proposed a tool named CCFinder that is based on token based approach but this technique requires a parser to parse the code into tokens. So the proposed approach presented in this paper can be combined with this approach to make a hybrid approach and the advantage is that we don't need to apply the parsing on whole software, only the part of the software in which potential clones are detected, code normalization is applied.

C. Abstract Syntax Tree Based Comparison

In contrast to token based approach abstract syntax tree based approach converts source code into an abstract syntax tree and traverse the tree for finding a similar sub tree. If similarity is found the code for this sub tree is termed as clone [15]. The result obtained through this comparison is quite efficient but it is very complex to create an abstract syntax tree and the scalability is not so good.

D. Program Dependency Graph Comparison

Program Dependence Graph basically shows control flow and data dependencies. Once the PDG is obtained from the source code, an isomorphic graph comparison [8] is applied to find the similarity, and the original code slices represented by a sub graph are returned as a clone. This approach is more robust and efficient as they analyze both semantic and syntactic clones. But the problem with this approach is, for large software it is very difficult to obtain the program dependence graph and cost is also very high.

E. Metric Based Comparison

This approach calculates metrics from source code and uses these metrics to measure clones in software. Rather than working on source code directly this approach use metrics to detect clones [9]. Various tools are available for calculating metrics of source code. Columbus is the tool which provides metrics that are useful in detecting clones, but this tool does not work for Java programs. And the tool available for calculating Java code metrics is Source Monitor [18] but the metrics provided by this tool are not efficient in providing the result of clone detection. Other tools that are available for calculating Java code metrics are highly complex like Datrix which are designed for extending the quality of Java code [16]. In the presented work MCD Finder a tool for metric based clone detection is proposed. The metrics calculated by this tool are useful for predicting clones in the Java software [16] and it is easy to use. Table-1 compares the various clone detection approaches.

TABLE I
CLONE DETECTION APPROACHES

Type of Comparison	Portability	Type of clones	Efficiency	Integrity
Text Based	Good	Syntax (Type 1 Clone)	High	Depends on algorithm
Token Based	Average	Syntax (Type1 and Type 2)	Low	Good
Abstract Syntax Tree Based	Poor	Syntactic(Type1, Type 2 and Type3)	High	Depends on algorithm
Program Dependence Graph Based	Poor	Syntactic and Semantic	High	Medium

III. PROPOSED WORK

The proposed method is based on a metric based approach for clone detection. Firstly the metrics are calculated from this tool and then the comparison algorithm is applied on these metrics to detect clones. The method consists of three phases: Adaptation Phase, Computation Phase and Measurement Phase. Section illustrates Adaptation phase, section B describes Computation phase, section C describes Measurement phase and section D explains the flow diagram of the tool.

A. Adaptation phase

This is the first phase of the proposed method. In this phase the code is adapted according to requirement of tool means tool requires as input the .class file for the program. So the Java source code should be compiled to make it adaptable according to tool's requirement. The tool takes byte code as an input program rather than the Java source code because the byte code is platform independent. The perception is derived from the fact that while converting the code into byte code the compiler would map 'for' and 'while' loops into the same unified representation. As compiler will generate a unified representation of code by reordering, and reordering reduces dissimilarity in code segments, so it yields better results.

The Byte code is similar to other low level language. Java byte code uses machine instruction to stimulate basic functionalities. Java byte code is generated by using javac compiler provided by JDK environment. In the proposed approach byte code is used to calculate various metrics for the source code as it is platform independent and the other reason to use byte code is that a unified representation of source code is generated and various syntactic dissimilarities of various loops and conditions are also changed in a unified format shown in Fig-3. So up to some extent it can detect some semantic clones and can also eliminate some challenges of semantic clones.

B. Computation Phase

After the adaptation phase the next step is to perform computation. Computation is in the form of various metrics that helps in identifying the potential clone. Bruno Lague et al. [16] provided various metrics to perform Java software evolution and clone detection and these metrics are calculated with the help of this tool. Besides these metrics some object oriented metrics are also calculated from the proposed tool that helps in identifying potential clones. Fig-4 depicts that we have to open a file and the metrics that were listed by Bruno Lague et al. [16] is calculated. And the metrics that are calculated with the help of proposed tool are:

1. Number of calls from method
2. Number of statements
3. Number of parameters passed into the function
4. Number of conditional statements
5. Number of Non-local variable
6. Number of Total variable
7. Number of public variables
8. Number of private variables
9. Number of protected variables

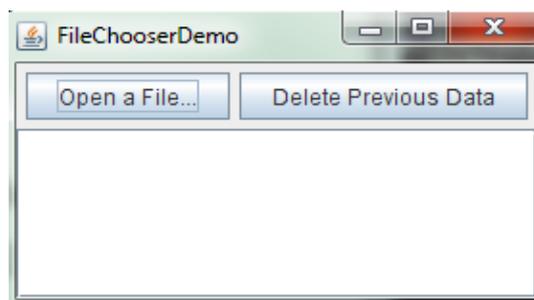


Fig-1 Choosing a File for calculating metrics

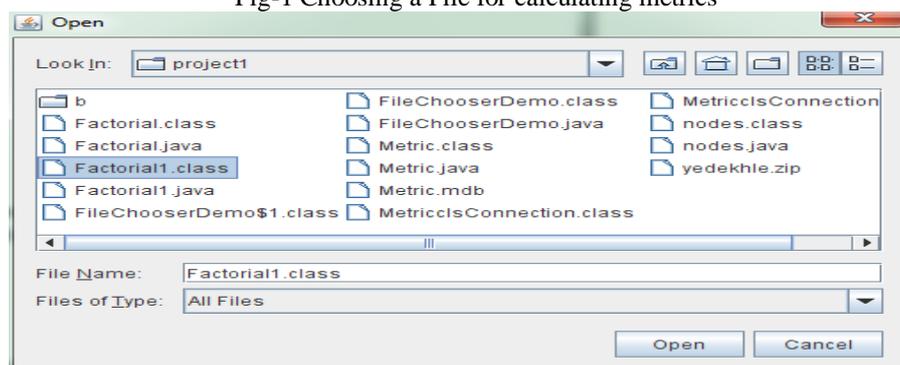


Fig-2 Choosing .class file

When we open a file the metrics calculated will be stored in a database. Now these calculated metrics are mapped into the excel sheets. This mapping to excel sheet is made to store the data for future use and comparison is applied to these excel sheets. And the calculated metric is shown in Fig-3 and Fig-4. Metrics are calculated both at the function level and class level.

ID	ClassName	totalfunc	totalif	LOC	totalV	pubV	priV	friV	proV
21	Factorial1.java	1	0	20	0	0	0	0	0
*	(New)								

Fig-3 Metrics at class level

classname	functionname	localvariable	LOC
Factorial1.java	main	3	17
*			

Fig-4 Metrics at function Level

C. Measurement Phase

After the computation phase is completed, and these metrics are mapped into excel sheets and comparison is performed. The comparison is performed on the basis of similarity of the metric value. This tool represents as output which metric's value is same and on the basis of this result potential clones are identified.

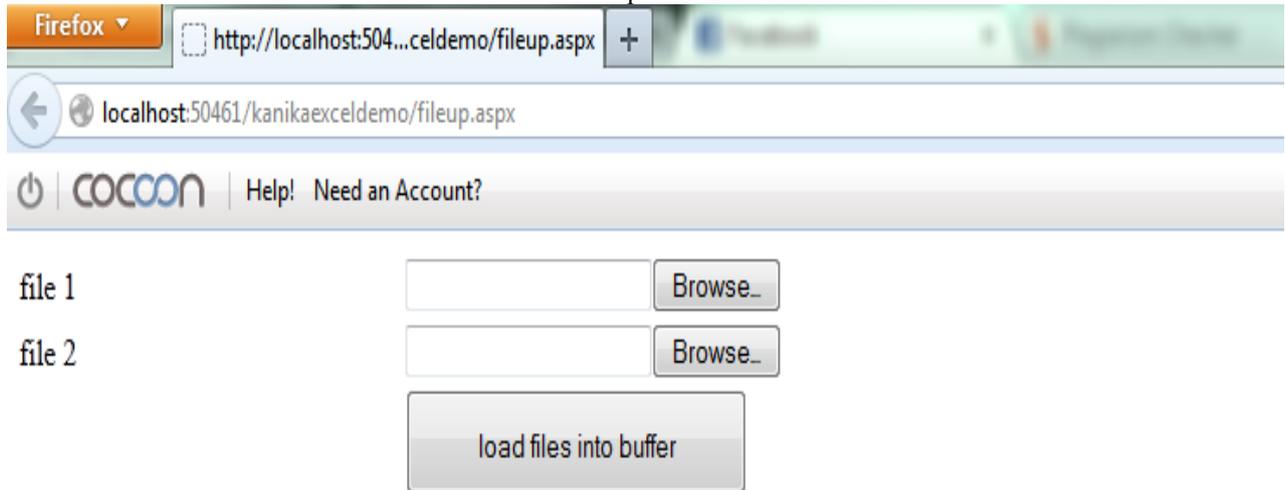
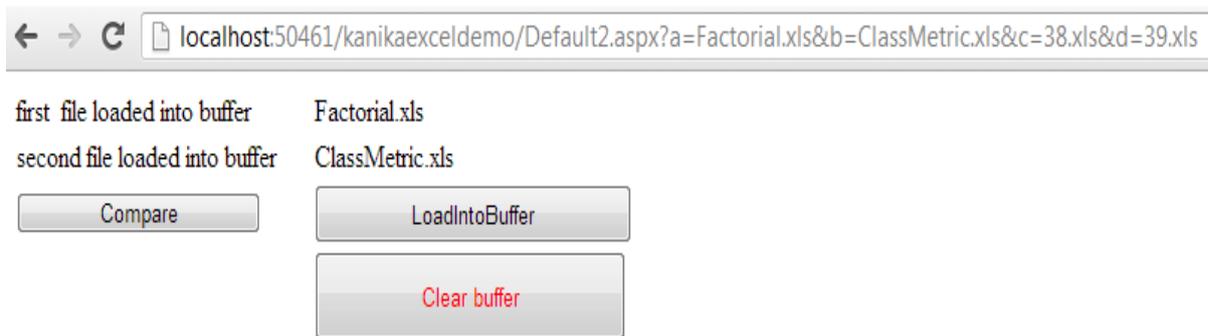


Fig-5 Choosing excels files for comparison



Total_Function	Totalif	LOC	totalv	pubV	priV	proV	FrndV
			0	0	0		

Fig-6 Comparison of Class Level Metric

Fig-6 shows the comparison of various class level metric on the basis of similarity. Similarly the function level metric calculated are also compared to find the similarity. And after analyzing the results potential clones are detected. And this approach is very efficient in providing potential clones.

Section D describes the flow graph of the tool. The flow graph depicts step by step working of the tool. First of all compilation of Java code is made to convert it into byte code and then the metric computation is performed. These calculated metrics are stored in the database as shown in fig-3. And the process of comparison is applied to detect potential clones.

D. Flow diagram

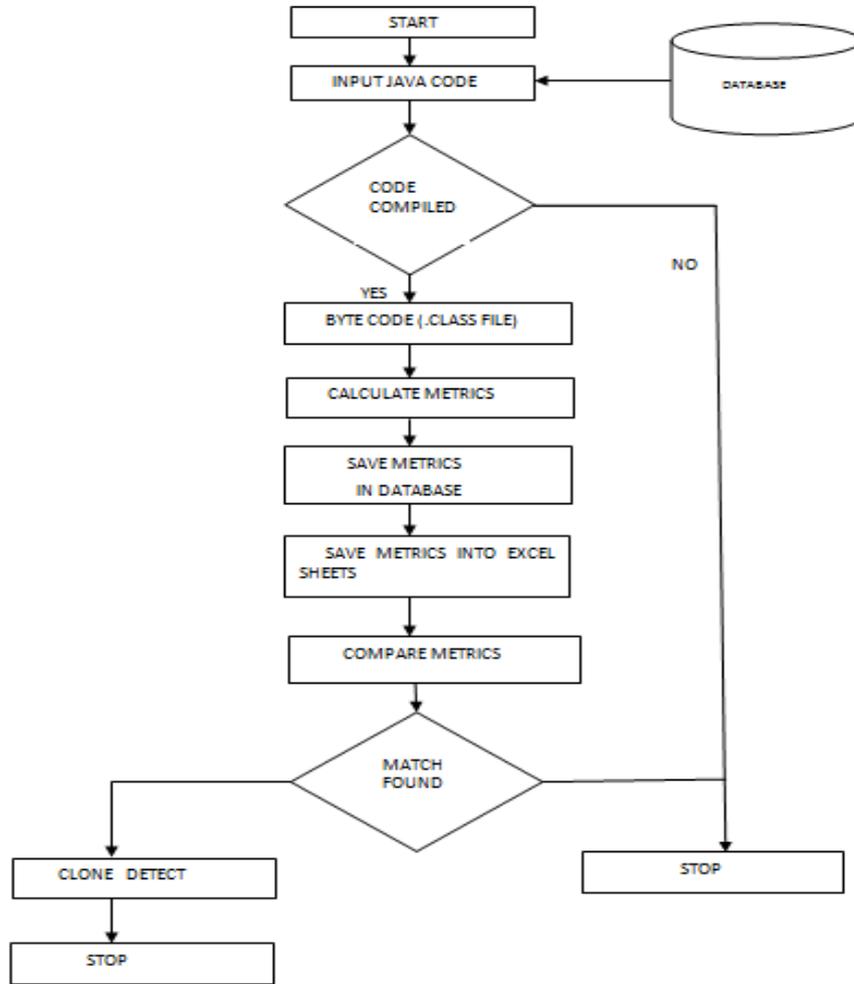


Fig-7 Flow Graph of Tool

IV .RESULTS AND DISCUSSION

The above calculated results are analyzed and studied; the results calculated are very efficient in providing the potential clones. As we have calculated various metrics and then compared the metrics on the basis of similarity, the results obtained are very efficient as they are providing information about semantic clones as well. We have compared the same two programs with different syntactical structure but same meaning and this tool is able to detect those potential clones and semantic clones are detected as well.

We have studied source monitor [18] tool and used its metrics to compare the results for the same program. The tool is efficient in calculating metrics but those metrics are not useful for detecting any type of clone.

The screenshot shows a web browser window with the URL: localhost:50461/kanikaexceldemo/Default2.aspx?a=kanu1.xls&b=kanu2.xls&c=24.xls&d=25.xls

Below the browser, there is a control panel with the following text and buttons:

- first file loaded into buffer: kanu1.xls
- second file loaded into buffer: kanu2.xls
- Buttons: Compare, LoadIntoBuffer, Clear buffer

At the bottom, there is a table with the following data:

lines	statements	PercentBranchStatement	MethodCallStatements	PercentLineWithComment	ClassesAndInterface	MethodPerClass	AverageStatementPer.Method
58*	39	0	0	0	1	18	1

Fig-8 Loading Excel Files

Now consider the same example with CCFinder [6] as it is token based approach it does not find those clones that are same in logic as shown in Fig-12.

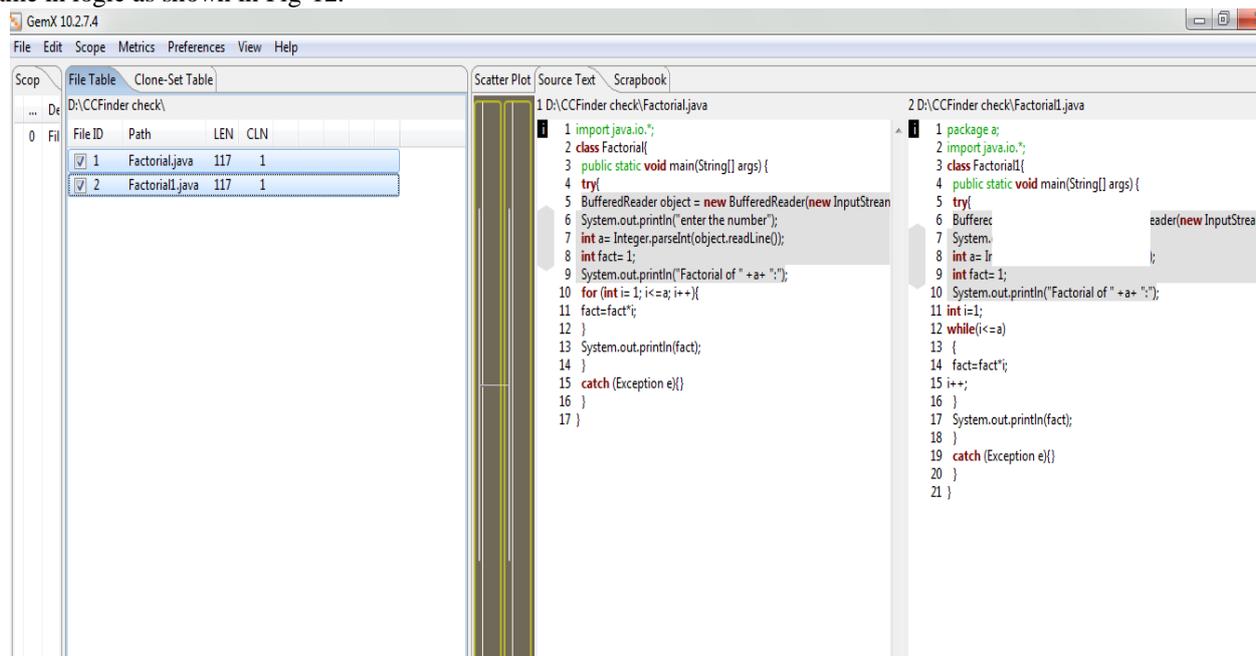


Fig-9 Detecting Clones using CCFinder

As it is clear from the fig-9 semantic clones can't be detected. Thus the approach used in this paper is efficient in providing semantic clone as a potential clone but this tool is not able to detect all the clones.

V. Conclusion and future Scope

The tool developed works only for the Java language and is easy to use. In the presented work metric based approach is used to identify the potential clones which do not work directly on source code. Since the byte code has been taken as input to calculate metric's value, so up to some extent it is able to identify the semantic clones. Moreover byte code is platform independent which makes this tool more efficient than already existing tools. As an application of abstract syntax tree based approach and program dependence graph approach on all code is tedious work so the proposed tool have reduced the work by identifying potential clones.

In future this approach can be integrated with other approaches like abstract syntax tree based approach and the program dependence graph approach to make this a hybrid approach to efficiently detect semantic clones. With the help of proposed tool potential clones has been calculated and on these potential clones these techniques could be applied to make it more efficient and cost effective.

Acknowledgment

We would like to express our gratitude to all those who gave us the possibility to complete this paper. We express our gratitude to Thapar University for providing us labs to complete our work and all the faculty members who supported us.

REFERENCES

- [1] Brenda Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", In *Proceedings of the Second Working Conference on Reverse Engineering*, pp 86-95.
- [2] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
- [3] Stefan Bellon, "Vergleich von Techniken zur Erkennung duplizierten quellcodes", Diploma Thesis, No. 1998, University of Stuttgart (Germany), Institute for Software Technology, September 2002.
- [4] Gilad Mishne and Maarten de Rijke, "Source Code Retrieval Using Conceptual Similarity", In *Proceeding of the 2004 Conference on Computer Assisted Information Retrieval (RIA0'04)*, pp. 539-554, April 2004.
- [5] Chanchal Kumar Roy and James R. Cordy, "A Survey on Software Clone Detection Research", Technical Report No. 2007-541, School of Computing Queen's University at Kingston Ontario, Canada, September 26, 2007.
- [6] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, "CCFinder: A Multilingual Token-Based Code Clone Detection System for Large Scale Source Code", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, VOL. 28, NO. 7, JULY 2002.
- [7] W. S. Evans, C. W. Fraser, F. Ma, "Clone Detection via Structural Abstraction", *Software Quality Journal*, Vol. 17, pp. 309-330, 2007.
- [8] C. Liu, C. Chen, J. Han, P. S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis" Conf. On Knowledge Discovery and Data Mining, pp. 872-881, 2006.

- [9] 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 and Technology (IJCET) Volume 3, Issue 1, pp. 273-288, January-June (2012).
- [10] Lablanc. C, J. Mayland, and E. M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software Maintenance (ICSM) '96, Moterey, California, pp. 244-253,1996.
- [11] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, pp. 55–64,2007.
- [12] Baker, B.S., "Parameterized Pattern Matching: Algorithms and Applications," *Journal Computer System Science*, Vol. 52 (1): 2842, February 1996.
- [13] Koschke, R., "Survey of Research on Software Clones", In *Dagstuhl Seminar 06301*, pp. 24, 2006.
- [14] Yue Jia, David Binkley, Mark Harman, Jens Krinke and Makoto Matsushita, "KClone: A Proposed Approach to Fast Precise Code Clone Detection," In *Proc. Of The Third International Workshop on Detection of Software Clones (IWSC 2009)*, pp. 12-16, 2009.
- [15] Yu, D. Peng, X. Zhao, W., "An Automatic Refactoring Method of Cloned Code Using Abstract Syntax Tree and Static Analysis," *Journal of Chinese Computer Systems*, Vol.28,2008.
- [16] Jean-Francois Patenaude, Bruno Laguë, "Extending Software Quality Assessment Techniques to Java Systems", *Seventh International Workshop on Digital Object Identifier*, pp. 45- 56, 1999.
- [17] J. Johnson, "Visualizing Textual Redundancy in Legacy Source", *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative research, CASCON 2004*, pp. 171-183, 1994.
- [18] Nitin Bhide, Code Analysis and Visualization Tool. [Online] Available:<http://thinkingcraftsman.in/articles/codeanalysisistools.htm>,2009.