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
© IAEME: www.iaeme.com/ijcet.html
Journal Impact Factor (2011): 1.0425 (Calculated by GISI)
www.jifactor.com

AN EFFICIENT METHOD-LEVEL CODE CLONE DETECTION SCHEME THROUGH TEXTUAL ANALYSIS USING METRICS

G. Anil kumar[1] **Dr. C.R.K.Reddy**[2] **Dr. A. Govardhan**[3]

[1]MGIT, Dept. of Computer science Hyderabad, India
Email: anilgkumar@mgit.ac.in
[2]CBIT, Dept. of Computer science,Hyderabad, India
Email: crkreddy@cbit.ac.in
[3]JNTUH Dept. of Computer science, Hyderabad, India
Email: govardhan_cse@jntuh.ac.in

ABSTRACT

Code cloning or the act of copying code fragments and making minor, non–functional alterations, is a well known problem for evolving software systems which leads to duplicated code fragments known as code clones. A Clone Detection approach is to find out the reused fragment of code in any application to maintain different types of clones that are being identified by the clone detection techniques. Ever since clone detection evolved, it has been providing better results by reducing the complexity. A different clone detection tool makes the detection process easier and produces efficient results. In many existing systems, main focus is on line by line detection or token based detection to find out the clones in the system. So, it makes the system to take long time to process the entire source code. If the fragment of code is not an exact copy but the functionalities make it similar to each other, then existing system doesn't figure out that type of clones in it. This paper proposes combination of textual and metric analysis of a source code for the detection of all types of clones in a given set of fragment of java source code. Various semantics have been formulated and their values are used during the detection process. This metrics with textual analysis provides less complexity in finding the clones and giving accurate results.

Keywords: *Clone detection, Textual Analysis, Metrics computation, Abstract syntax Trees, Precision and Recall.*

1. INTRODUCTION

Software systems provide vital support for the smooth running of an organization's business. It is the responsibility of maintainers to keep the system up-to date and functioning correctly [6]. The success of free software is evident from the large and growing number of hardware devices that include free software components. Devices such as routers, televisions, set-top boxes and media players are commonly based on software such as the Linux kernel, the Samba file/print server and the BusyBox toolset [13]. Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development. As a result software systems often contain sections of code that are very similar, called code clones [8]. A code clone is a code portion in source files that is identical or similar to another. Clones are introduced because of various reasons such as reusing code by 'copy-and-paste', mental macro (definitional computations frequently coded by a programmer in a regular style, such as payroll tax, queue insertion, data structure access, etc), or intentionally repeating a code portion for performance enhancement, etc [2]. Identifying software clones and understanding how software changes between releases are two important issues for maintainers where a text-based approach is likely to be useful. Maintenance of large software systems under pressure often leads to a phenomenon referred to as software cloning [4].

A clone is a copy of a code fragment. Usually, clones consisting out of more than 5 statements are considered interesting. Since the clone relation is symmetric we better say that the origin and the copy form a clone pair [10]. Clones are frequently introduced by code scavenging, that is, by copying existing code and modifying it. Finding clones in software systems is important in many maintenance, reengineering, and program understanding contexts [9]. Detection and removal of such clones promises decreased software maintenance costs of possibly the same magnitude [1]. One major problem in detecting a clone is that it is impossible to be absolutely certain that one section of code has been copied and pasted from another [6]. Unfortunately, a precise definition of what differentiates a clone from a non-clone is lacking. This can present problems for evaluating clone detectors [9]. A clone detector must try to find pieces of code of high similarity in a system's source text. The main problem is that it is not known beforehand which code fragments may be repeated. Thus the detector really should compare every possible fragment with every other possible fragment. Such a comparison is prohibitively expensive from a computational point of view and thus, several measures are used to reduce the domain of comparison before performing the actual comparisons [8].

A clone detection system should have ability to select clones or to report only helpful information for user to examine clones, since large number of clones is expected to be found in large software systems [2]. Although some researchers argue not to remove clones because of the associated risks, there is a consensus that clones need to be detected at least. Detection is necessary to find the place where a change must be replicated and also useful to monitor development in order to stop the increase of redundancy before it is too late [15]. An important application of clone detection is the improvement of source code quality by refactoring duplicated code fragments [7]. From the analysis of software application it appears that the inclusion of these clones results from the addition of some

extra functionality which is similar but not identical to some existing logic within a system. It seems that when presented with the challenge of adding new functionality the natural instinct of a programmer is to copy, paste and modify the existing code to meet the new requirements and thus creating a software clone [6]. Clone detection techniques attempt at finding duplicated code, which may have undergone minor changes afterward. The typical motivation for clone detection is to factor out copy-paste-adapt code, and replace it by a single procedure [5]. At the beginning of any clone detection approach, the source code is partitioned and the domain of the comparison is determined. There are three main objectives in this phase: remove uninteresting parts, determine source units and determine comparison units / granularity [8]. Code clones can be discovered manually by scavenging through the program source and identifying duplicates one by one. Depending on the size of the program, this manual process can become tedious and labor intensive. An automatic clone detection tool can be beneficial by reducing the time and effort needed to find clones [11]. A good clone detector should scale to large programs, while considering sufficient semantic-level information to detect all three types of clone. This requires that the management of necessary semantic information should be inexpensive in terms of time and memory [14].

Various approaches have been applied in practice with good results. The main technical difficulty is that duplication is often masked by slight differences: eformatting, code modifications, changed variable names and inserted or deleted lines of code all make it harder to recognize software clones [16]. Five established detection tools will be used in the evaluation process; JPlag, MOSS, Covet, CCFinder and CloneDr. JPlag and MOSS are web-based academic tools for detecting plagiarism in student's source code. CloneDr and CCFinder are stand alone tools looking at code duplication in general [6]. Problem Mining is a process change that aims at coping with the existing base of software clones in a system already in service, for which new development and maintenance is still being done [3]. The handling of duplicated code can be very problematic in many respects. An error in one component is reproduced in every copy. Since it is not documented in which places duplicates can be found, it is extremely hard to hand and remove such errors [10]. Duplicated fragments can also significantly increase the work to be done when enhancing or adapting code [12].

SHINOBI is a tool for automatic code clone detection. The main features of SHINOBI are, first, it is highly integrated with Microsoft Visual Studio. For instance, it is implemented as an add-on of Visual Studio. A programmer can easily check and edit detected code clones. Second, SHINOBI automatically detects code clones with source code being edited. The detection process is automatic, implicit, and quick. A programmer can get a list of code clones without noticeable time penalty whenever he develops with the IDE and finally It also highlights code clones to help recognize clones during software maintenance tasks. In the clone detection tool comparison experiment at the First International Workshop on Detection of Software Clones, clones were separated into three categories: Exact copies, with no differences between them, Parameterized copies, where variable and function calls can have different names and/or types have changed and Modified copies, where some modification is done, such as adding or deleting lines of code [11]. Efficient token-based clone detection is based on suffix trees,

originally used for efficient string search [15]. Various approaches have been applied in practice with good results. The main technical difficulty is that duplication is often masked by slight differences: eformatting, code modifications, changed variable names and inserted or deleted lines of code all make it harder to recognize software clones [16].

Hence, we propose an efficient clone detection scheme to detect all types of clones available in the source files. Here we use a hybrid technique based on textual and metric analysis to detect the duplicate codes. The rest of the paper is described as follows. Section 2 briefs about the literature survey. The concept of textual and metric analysis is described in Section 3 and the proposed methodology is explained with necessary equations and diagrams in Section 4. The Results obtained in the proposed method is discussed in Section 5 and Section 6 concludes the work.

2. RELATED WORK

A handful of researches have been presented in the literature for the detection of Clones. Recently, utilizing artificial intelligence techniques like Abstract Syntax Trees, KClone, Substring Matching, Frequent Itemset Techniques have received a great deal of attention among researchers. A brief review of some recent researches is presented here.

Chanchal K. Roy *et al*. [8] proposed that, over the last decade many techniques and tools for software clone detection have been proposed. In that paper, they provide a qualitative comparison and evaluation of the current state-of-the-art in clone detection techniques and tools, and organize the large amount of information into a coherent conceptual framework. We begin with background concepts, a generic clone detection process and an overall taxonomy of current techniques and tools. Then classify, compare and evaluate the techniques and tools in two different dimensions. First, we classify and compare approaches based on a number of facets, each of which has a set of (possibly overlapping) attributes. Second, we qualitatively evaluate the classified techniques and tools with respect to taxonomy of editing scenarios designed to model the creation of Type-1, Type-2, Type-3 and Type-4 clones. Finally, they have provided examples of how one might use the results of this study to choose the most appropriate clone detection tool or technique in the context of a particular set of goals and constraints. The primary contributions of this paper are: a schema for classifying clone detection techniques and tools and a classification of current clone detectors based on this schema, and taxonomy of editing scenarios that produced different clone types and a qualitative evaluation of current clone detectors based on this taxonomy.

Armijn Hemel *et al*. [13] proposed that, Software released in binary form frequently used third-party packages without respecting their licensing terms. For instance, many consumer devices have firmware containing the Linux kernel, without the suppliers following the requirements of the GNU General Public License. Such license violations are often accidental, e.g., when vendors receive binary code from their suppliers with no indication of its provenance. To help find such violations, they have developed the Binary Analysis Tool (BAT), a system for code clone detection in binaries. Given a binary, such as a firm ware image, it attempts to detect cloning of code from repositories

of packages in source and binary form. They have evaluated and compared the effectiveness of three of BAT's clone detection techniques: scanning for string literals, detecting similarity through data compression, and detecting similarity by computing binary deltas.

Yue Jia *et al*. [14] proposed that, in all applications of clone detection it is important to have precise and efficient clone identification algorithms. That work outlines a new algorithm, KClone for clone detection that incorporates a novel combination of lexical and local dependence analysis to achieve precision, while retaining speed. It also reports on the initial results of a case study using an implementation of KClone with which we have been experimenting. The results indicate the ability of KClone to find types-1, 2, and 3 clones compared to token-based and PDG-based techniques, and also reports results of an initial empirical study of the performance of KClone compared to CCFinderX.

Rainer Koschke *et al*. [15] proposed that, reusing software through copying and pasting was a continuous plague in software development despite the fact that it creates serious maintenance problems. Various techniques have been proposed to find duplicated redundant code (also known as software clones). This study has compared those techniques and shown that token-based clone detection based on suffix trees is extremely fast but yields clone candidates that are often no syntactic units. Current techniques based on abstract syntax trees on the other hand find syntactic clones but are considerably less efficient. It describes how they can made use of suffix trees to find clones in abstract syntax trees. That new approach was able to find syntactic clones in linear time and space. It reports the results of several large case studies in which we empirically compare the new technique to other techniques using the Bellon benchmark for clone detectors.

Stephane Ducasse *et al*. [16] proposed that, duplicated code is known to pose severe problems for software maintenance, it is difficult to identify in large systems. Many different techniques have been developed to detect software clones, some of which are very sophisticated, but are also expensive to implement and adapt. Lightweight techniques based on simple string matching are easy to implement, but how effective are they? They presented a simple string-based approach which they have successfully applied to a number of different languages such COBOL, JAVA, C++, PASCAL, PYTHON, SMALLTALK, C and PDP-11 ASSEMBLER. In each case the maximum time to adapt the approach to a new language was less than 45 minutes. In that paper, they investigate a number of simple variants of string-based clone detection that normalize differences due to common editing operations, and assess the quality of clone detection for very different case studies. Their results confirm that that inexpensive clone detection technique generally achieves high recall and acceptable precision. Overzealous normalization of the code before comparison, however, can result in an unacceptable numbers of false positives.

R. R. Brooks *et al*. [17] proposed that, in cloning attacks, an adversary captures a sensor node, reprograms it, makes multiple copies, and inserts these copies, into the network. Cloned nodes subvert sensor network processing from within. In a companion paper, they

shown how to detect and remove clones from sensor networks using random key pre distribution security measures. Keys that are present on the cloned nodes are detected by using authentication statistics based on key usage frequency. For consistency with existing random key pre distribution literature, and ease of explanation, the network in that paper used an Erdos-Renyi topology. In the Erdos-Renyi topology, the probability of connection between any two nodes in the network is uniform. Since the communications ranges of sensor nodes were limited, this topology is flawed. This article applies the clone detection approach from to more realistic network topologies. Grid and ad hoc topologies reflect the node connectivity patterns of networks of nodes with range limits. They provided analytical methods for choosing detection thresholds that accurately detect clones. They used simulations to verify our method. In particular they found the limitations of that approach, such as the number of nodes that can be inserted without being detected.

Shinji Kawaguchi *et al*. [18] proposed that, code clones decrease the maintainability and reliability of software programs, thus it is being regarded as one of the major factors to increase development/maintenance cost. They have introduced SHINOBI, a novel code clone detection/modification tool that was designed to aid in recognizing and highlighting code clones during software maintenance tasks. SHINOBI was implemented as an add-in of Microsoft Visual Studio that automatically reports clones of modified snippets in real time.

Kodhai. E *et al*. [19] proposed that, clone detection has considerably evolved over the last decade, leading to approaches with better results but with increasing complexity. Most of the existing approaches were limited to finding program fragments similar in their syntax or semantics, while the fraction of candidates that were actually clones and fraction of actual clones identified as candidates on the average remain similar. In that paper, a metric-based approach combined with the textual comparison of the source code for the detection of functional Clones in C source code has been proposed. Various metrics had been formulated and their values were utilized during the detection process. Compared to the other approaches, this method was considered to be the least complex and to provide a more accurate and efficient way of Clone Detection. The results obtained had been compared with the two other existing tools for the open source project Weltab.

Nam H. Pham *et al*. [20] proposed that, Model-Driven Engineering (MDE) has become an important development framework for much large-scale software. Previous research has reported that as in traditional code-based development, cloning also occurs in MDE. However, there has been little work on clone detection in models with the limitations on detection precision and completeness. That paper presented the ModelCD, a novel clone detection tool for Matlab/Simulink models that is able to efficiently and accurately detect both exactly matched and approximate model clones. The core of ModelCD is two novel graph-based clone detection algorithms that are able to systematically and incrementally discover clones with a high degree of completeness, accuracy, and scalability.

3. TEXTUAL & METRIC ANALYSIS

In textual comparison line by line comparison is done. That is whole lines are compared to each other textually using hashing for strings. This comparison is done by string matching algorithm. The result can be plotted in a dot plot and each dot indicates a pair of cloned lines. Uninterrupted diagonals or displaced diagonals which occur in the dot plot indicate the consecutive duplicated lines.

Metric based technique gathers different metrics from a particular code fragments, such as, a function or a class, then groups these metric together into a metrics vector. After that it compares these metric vector instead of actual code directly [LPM+97, KDM+96], because this method is focused on a specific type of code fragments, it can only detect an type of high level clone, e.g. duplicated function.

Here in metric computation each code fragments are given different metric values. During comparison these metric vectors are compared instead of comparing code directly. As a hint for similar code an allowable distance can be used for these metric vectors. Text based technique is the oldest and simplest way to detect clone, which takes each line of source code as code representation. In order to increase the performance, lines are often transformed by a hash function and uninterested code, such as comments and white spaces are filtered. The result of comparison is presented in a dot plot graph, where each dot indicates a pair of cloned lines.

A clone pair can be determined as a sequence of uninterrupted diagonals line of spot. Because text based technique does not perform any syntactical or semantically analysis on source code, it's one of the fastest clone detection approaches. It can easily deal with type-1 clone, and with additional data transformation, the type-2 can also be taken care. However without information of syntactical or semantically level support, the third type of clone cannot be detected at all.

4. AN EFFICIENT CLONE DETECTION PROCESS

A clone detector must try to find pieces of code of high similarity in a system's source text. The main problem is that it is not known beforehand which code fragments can be found multiple times. The detector thus essentially has to compare every possible fragment with every other possible fragment. Such comparison is very expensive from a computational point of view and thus, several measures are taken to reduce the domain of comparison before performing the actual comparison. Once potential cloned fragments are identified further analysis is carried out to detect actual clones. In our proposed method, a hybrid technique based on textual and metric analysis is used to detect all types of clones present in the source code.

Text based clone detection technique uses the transformation such as comments removal, whitespace removal. Because text based technique does not perform any syntactical or semantically analysis on source code, it is one of the fastest clone detection approach. It can easily deal with type-1 clone, and with additional data transformation, the type-2 can also be taken care. In Metric based technique, instead of comparing the code directly,

different metric of code are gathered and these metrics were compared to detect clones. Many clone detection techniques today use metrics for detecting similar codes. The proposed method is implemented as a tool in java. The system architecture of the tool is as shown in Fig. 1.

Our proposed approach use metric based and text based technique to detect clones and divided into two stages. In the first stage metric based technique is used for the selection on potential clone. Potential clones are selected on the basis of metric match and after this potential clones are further processed with text based technique. The potential clones are compared line by line to determine whether two potential clones really are clones of each other. The tool developed initially parses through the given input source code and identifies the various methods present.

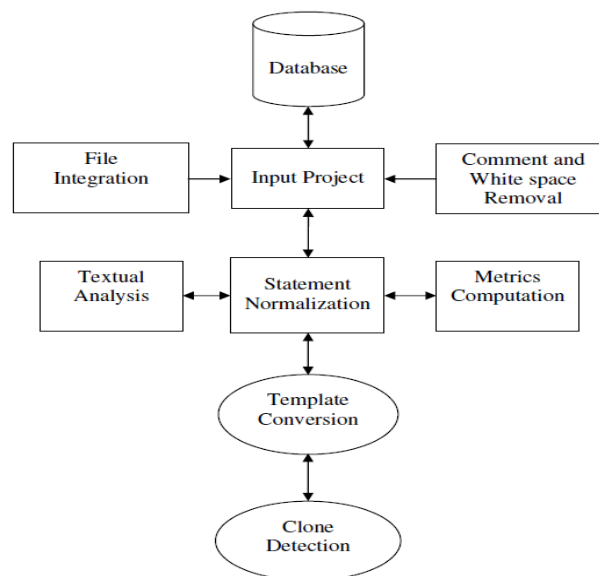

fig.1: Clone Detection Architecture

Clone detection process has been divided into number of phases. As shown in the fig.1 the phases include input and pre-processing, template conversion, metrics computation and finally detecting the clone types. The pairs that show similar in textual comparison are listed as the clones. The detection tool thus developed does not employ any external parsers. It requires only less overhead compared to other methods.

4.1 Preprocessing and input Selection

All the source code uninteresting to the comparison phase is filtered out in this phase. This phase also includes file integration, source code standardization and the normalization. File integration involves the grouping of all the files of the same project into a single large file for external parsing. This phase includes file integration, source code standardization and the normalization. File integration involves the concatenation of all the files of the same project into a single large file for external parsing. Here it includes the removal of whitespaces, comments and pre-processor statements. After

removing the uninteresting code, the remaining source code is partitioned into a set of disjoint fragments called source units. These units are the largest source fragments that may be concerned in direct clone relations with each other. Source units can be at any level of granularity, for example, files, classes, functions/methods, begin-end blocks, statements, or sequences of source lines.

Source units may require to be further partitioned into smaller units depending on the comparison technique used by the tool. For example, source units may be subdivided into lines or even tokens for comparison. Comparison units can also be derived from the syntactic structure of the source unit. For example, an if-statement can be further partitioned into conditional expression, then and else blocks. The order of comparison units within their corresponding source unit may or may not be important, depending on the comparison technique. Source units may themselves be used as comparison units. For example, in a metrics based tool, metrics values can be computed from source units of any granularity and therefore, subdivision of source units is not required in such approaches. The source code is re-structured to a standard format to establish the similarity between the cloned fragments.

These steps are very similar to normalization procedures and produces gain in the recall. Almost all approaches disregard whitespace, although line-based approaches retain line breaks. Some metrics-based approaches however use formatting and layout as part of their comparison. Most approaches remove and ignore comments in the actual comparison. Most approaches apply identifier normalization before comparison in order to identify parametric Type-2 clones. In general, all identifiers in the source code are replaced by the same single identifier in such normalizations.

4.2 Template Conversion

Template conversion is the process of transformation of the input source code into a pre-defined set of statements or conversion into a standard intermediary form. For example, renaming of data types, variables, function names etc as shown in fig. 2. This type of format used in textual analysis is called 'template'. The textual comparison of the selected candidates while detecting the type-2 cloned methods where as per the definition, function identifiers, variable names, types etc., are edited during the cloning process and mere textual comparison would not suffice. Once the template conversion is over, the source file and the template file is stored in the database for applying metrics. This transformation can vary from very simple e.g., just removing the white space and comments to very complex e.g., generating PDG representation and/or extensive source code transformations. Metrics-based methods usually compute an attribute vector for each comparison unit from such intermediate representations.

SOURCE CODE	***TEMPLATE***
int templconv(ptra, buff1,leng, buff2)	*DAT FUN_NAME(S,S,S,S)*
char buff1[];	*DAT S;*
int leng;	*DAT S;*
int ptra;	*DAT S;*
char buff2[];	*DAT S;*
{	*{*
int i;	*DAT S;*
int j;	*DAT S;*
While(i<=leng)	*LOOP*
{	*{*
If(buff1[ptra+j]!=buff2[ptrb+j])	*IF*
return TRUE;	*RETURN;*
};	*};*
i++;	*ASSIGNMENT STATEMENT;*
j++;	*ASSIGNMENT STATEMENT;*
tembuf[ptra]='\0';	*ASSIGNMENT FROM FUNCTION CALL*
return TRUE;	*RETURN;*
}	*}*

fig.2: Example for template conversion

4.3 Metric Computation

A set of 12 existing method level metrics are used for the detection of type-1, type-2, type-3 and type-4 clone methods. They are as follows:

1. No. of effective lines of code in each method : Get the number of lines of code, Subtract white space lines, Subtract comment lines, Subtract the lines that contains only block constructs (for example in C# begin block construct is the character '{' while end block construct is the character '}'.
2. No. of arguments passed to the method: Calling the function involves specifying the function name, followed by the function call operator and any data values the function expects to receive. These values are the arguments for the parameters defined for the function, and the process just described is called passing arguments to the function.
3. No. of function calls in each method: A function call is an expression containing a simple type name and a parenthesized argument list. The argument list can contain any number of expressions separated by commas. It can also be empty.
4. No. of local variables declared in each method: A variable declared as local is one that is visible only within the block of code in which it appears. It has local scope. In a function, a local variable has meaning only within that function block.
5. No. of conditional statements in each method: In computer science, conditional statements, conditional expressions and conditional constructs are features of a programming language which perform different computations or actions depending on whether a programmer-specified Boolean condition evaluates to true or false.

6. No. of looping statements in each method: A looping statement is one in which you want to execute a statement (or many) as many number of times you want. It is useful when you want to check some constraints with a specific value.
7. No. of return statements in each method: A return statement ends the processing of the current function and returns control to the caller of the function. A value-returning function should include a return statement, containing an expression.
8. No. of function calling in each method: Once a function has been declared and defined, it can be called from anywhere within the program: from within the main function, from another function, and even from itself. Calling the function involves specifying the function name, followed by the function call operator and any data values the function expects to receive.
9. No. of inheritance in each method: Inheritance is a way to compartmentalize and reuse code by creating collections of attributes and behaviors called objects that can be based on previously created objects.
10. No. of virtual functions in each method: A virtual function or virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature.
11. No. of overloading constructor in each method: Overload constructor is multiple constructors which differ in number and/or types of parameters.
12. No. of overriding functions in each method: Function over loading means two functions will have same name but they differ in the number or type of arguments.

For each of the methods identified the metrics are computed and the corresponding values are stored in a database Table I shows the metric values for the code fragment in fig 2. After computing the metric values, the method pairs with equal or similar set of values are identified by comparing the records in the database. The short-listed set of candidates is then textually compared to be confirmed as clone pairs.

Table I: Metric values for fig. 2

Sl. No.	Metrics	Value
1.	No. of lines of code	18
2.	No. of arguments passed	4
3.	No. of local variables declared	6
4.	No. of function calls	1
5.	No. of conditional statements	1
6.	No. of looping statements	1
7.	No. of return statements	2

4.4. Finding Clone Types and Clone Pairs

By taking up a line by line comparison of the standardized and normalized source code for type-1 clone method the identification of the potential clone pairs is done. That is identical code fragments are selected except for variations in whitespace, layout and comments. For type-2 clone comparison of templates are done. Here syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout

and comments are taken. In the fragments there is some modifications except there is some similarities means it must be declared as type-3 by matching template with the exact code. Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments can be said as type-3 clones.

It's declared as type-4 clone when the fragments are completely different but produce similar output**.** If the functionalities of the two code fragments are identical or similar and referred as Type IV clones. That is when two or more code fragments that perform the same computation but are implemented by different syntactic variants are said to be type-4 clone. The identified cloned methods are then clustered separately for each type and the clusters are uniquely numbered. Clustering gives a clear image of how the methods were cloned and helps to provide an easier review process.

5. RESULTS AND DISCUSSION

The proposed software clone detection system has been implemented in the working platform of JAVA (version JDK 1.6). Here we use the source code with more than 500 LOC. The main aim of the proposed method is to identify all the four clone types in the source code. This can be achieved by the combining both textual analysis and metrics. The step by step results obtained from the proposed method is described as follows.

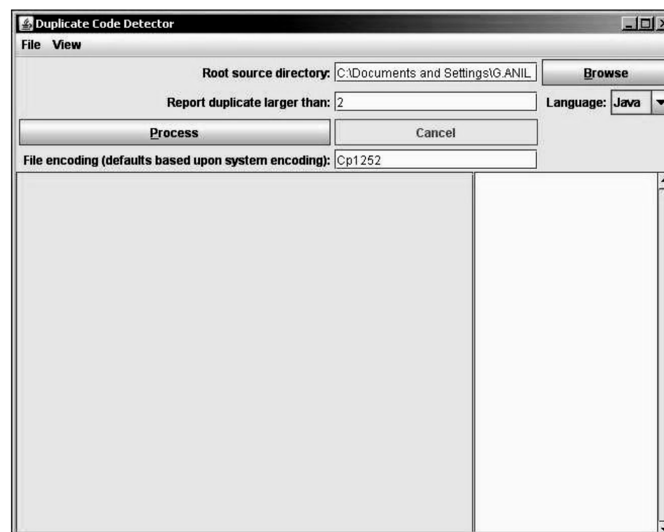

fig. 3: Initial Process

Fig. 3 represents the initial screen obtained in the clone detection process to Load the database (set of source programs). After loading the database, select the input files to detect the clones.

For detecting the clones in the input files, initially, the textual analysis is performed in the preprocessed codes. The textual analysis finds 2 types of clones such as type-1 and type-2. It is presented in Fig. 4. Then, metric computations are performed to detect the remaining clones in the source files. The metric analysis finds the remaining clones which are described in fig. 5.

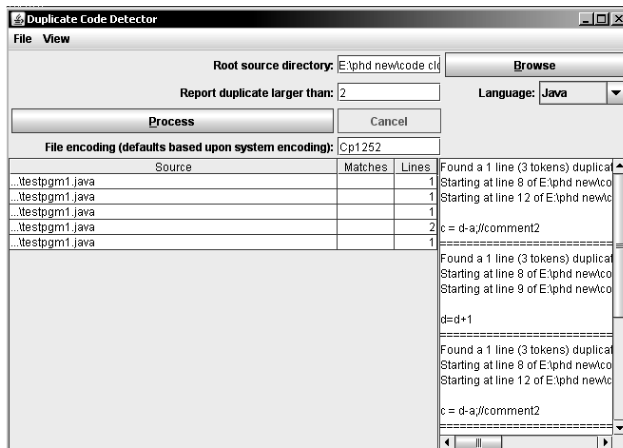

fig. 4: Textual Analysis

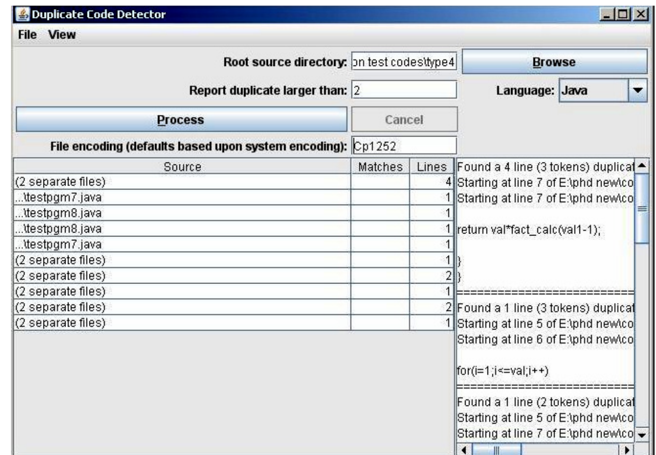

fig. 5: Metrics computation

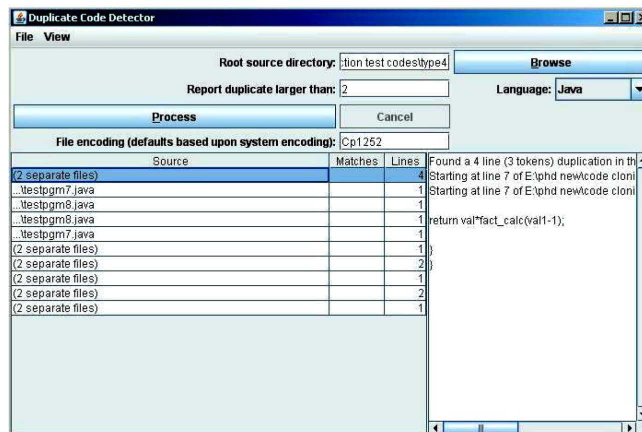

fig. 6: Clone Detection Process

Finally, the clones available in the source files are detected in the efficient manner and the final output is presented in fig.6.

Performance Measure

Detection result accuracy refers to a combination of both precision and recall. Precision denotes the probability that a randomly chosen candidate clone group is relevant. Recall denotes the probability that a relevant clone group, chosen from the hypothetical set of all relevant clone groups, is contained in a detection result.

$$\text{Precision},\ P = \frac{\text{Number of clones correctly found}}{\text{Total Number of clones found}}$$

$$\text{Recall},\ R = \frac{\text{Number of clones found correct}}{\text{Total number of clones in the source code}}$$

The precision and recall of the proposed method will evaluate the proposed system's efficiency. The following graph describes the comparison of performance measure.

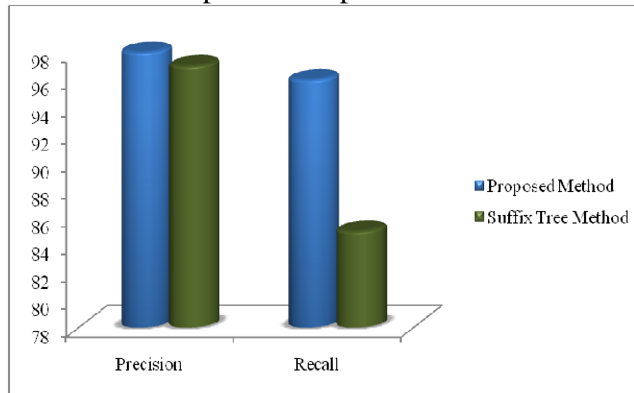

fig. 7: Comparison of Precision and Recall

From the fig.7, we observe that our proposed method detects the clones available in the source files in an efficient mannar. We compare the proposed work with the already existing clone detection tool which uses suffix tree method that will give less precision and recall rate when compared to our proposed method. The measures for the above graph is given in Table II.

Table II: Performance Measure

Methods	Performance Measure	
	Precision	**Recall**
Proposed method	98	96
Suffix Tree method	97	85

6. CONCLUSION

The paper has proposed a light-weight technique to detect functional clones with the computation of metrics combined with simple textual analysis technique. With the usage of metrics the existing exponential rate comparison is an overhead. Since the string matching/textual comparison is performed over the short listed candidates, a higher amount of recall could be obtained. Proposed work is divided into two stages, selection of potential clones and comparing of potential clones. The proposed technique detects exact clones on the basis of metric match and then by text match. Potential clones are compared line-by-line to determine whether two potential clones really are clones of each other. The early experiments prove that this method can do at least as well as the existing systems in finding and classifying the function clones in Java. The Precision and Recall plot describes the efficiency of the proposed work.

REFERENCES

1. Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna and Lorraine Bier, "Clone Detection Using Abstract Syntax Trees," In Proc. of the International Conference on Software Maintenance, Bethesda, MD, pp. 368 - 377, Nov 1998.

2. Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue, "CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code," IEEE Transactions on Software Engineering, Vol. 28, No. 7, pg. Software Engineering, Jul 2002.
3. Bruno Laguë, Daniel Proulx, Ettore M. Merlo, Jean Mayrand and John Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," In Proc of the 1997 International Conference on Software Maintenance (ICSM '97), Washington, DC, 1997.
4. J Howard Johnson, "Substring Matching for Clone Detection and Change Tracking,," In Proc of the International Conference on Software Maintenance (ICSM), Victoria, British Columbia, pp. 120–126, Sep 1994.
5. Magiel Bruntink, Arie van Deursen, Remco van Engelen and Tom Tourwe, "An Evaluation of Clone Detection Techniques for Identifying Cross-Cutting Concerns," In Proc. of the 20th IEEE International Conference on Software Maintenance, Washington, DC, 2004..
6. Elizabeth Burd and John Bailey, "Evaluating Clone Detection Tools for Use during Preventative," In Proc. of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02), Montreal, Canada, Oct 2002.
7. Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwe, "On the Use of Clone Detection for Identifying Crosscutting Concern Code," IEEE Transactions on Software Engineering, Vol. 31, No. 10, pp. 804 - 818, Oct 2005.
8. Chanchal K. Roy, James R. Cordy and Rainer Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," Science of Computer Programming, Vol. 74, No. 7, Feb 2009.
9. Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia, "Problems Creating Task-relevant Clone Detection Reference Data," In Proc. of the 10th IEEE Working Conference on Reverse Engineering, Victoria, Canada, Nov 2003.
10. Vera Wahler, Dietmar Seipel, Jurgen Wolff V. Gudenberg, and Gregor Fischer, "Clone Detection in Source Code by Frequent Itemset Techniques," In Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation, Chicago, IL, pp. 128 - 135, Sep 2004.
11. Robert Tairas and Jeff Gray, "Phoenix-Based Clone Detection Using Suffix Trees," In Proc. of the 44th annual southeast regional conference, New York, NY, 2006.
12. Chanchal K. Roy and James R. Cordy, "Scenario-Based Comparison of Clone Detection Techniques," In Proc. of the 16th IEEE International Conference on Program Comprehension, Washington, DC, 2008.
13. Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstrac, "Finding Software License Violations Through Binary Code Clone Detection," In Proc. of the 8th working conference on Mining software repositories, New York, NY, May 2011.
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. Rainer Koschke, Raimar Falke and Pierre Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," In Proc. of the 13th Working Conference on Reverse Engineering, Benevento, pp. 253 - 262, Oct 2006.

16. Stephane Ducasse, Oscar Nierstrasz and Matthias Rieger, "Research On the effectiveness of clone detection by string matching," Journal of Software Maintenance and Evolution: Research and Practice, Vol. 18, No. 1, pp. 37-58, 2006.
17. R. R. Brooks, P. Y. Govindaraju, M. Pirretti, N. Vijaykrishnan and M. Kandemir, "Clone Detection in Sensor Networks with Ad Hoc and Grid Topologies," International Journal of Distributed Sensor Networks, Vol. 5, pp. 209–223, 2009.
18. Shinji Kawaguchi, Takanobu Yamashinay, Hidetake Uwanoz, Kyhohei Fushida, Yasutaka Kamei, Masataka Nagura and Hajimu Iida, "SHINOBI: A Tool for Automatic Code Clone Detection in the IDE," In Proc. 16th Working Conference on Reverse Engineering, pp. 313 - 314, Oct 2009.
19. Kodhai. E, Kanmani. S, Kamatchi. A, Radhika. R and Vijaya Saranya. B, "Detection of Type-1 and Type-2 Code Clones Using Textual Analysis and Metrics," In Proc. of the 2010 International Conference on Recent Trends in Information, Telecommunication and Computing, Washington, DC, pp. 241-243, 2010.
20. Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi and Tien N. Nguyen, "Complete and Accurate Clone Detection in Graph-based Models," In Proc. of the 31st International Conference on Software Engineering, Washington, DC, 2009.