IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 9, September 2017

Code Clone Detection based on Logical Similarity

Ashish N. Runwal[1], Mr. Akash D. Waghmare[2]

P.G. Student, Department of Computer Engineering, SSBT's College of Engineering & Technology, Jalgaon, India[1]

Assistant Professor, Dept of Computer Engineering, SSBT's College of Engineering & Technology, Jalgaon, India[2]

Abstract: Code Clones are the entities in software ecosystems which can be unavoidable. Demand of software based clone detection has risen in industries day by day. Due to code duplication means the copy and paste activities, such pattern is recurrent thereby developers can reduce effort and time of rewriting similar code fragment by editing already written code. Code duplication may affect on quality, consistency, maintainability and comprehensibility. The trial is variety of syntax, compiler dependent language, and various coding patterns to resolve a single problem. There is lots of software tools, code clone detection algorithms exist, but they have some restrictions to detect perfect cloning. Earlier research and tools developed till now can find only Type-1, Type-2 and some part of Type-3 clones. Some tools are very slow and time consuming for comparing codes and with low in precision. Type-4 clone detection represents a challenge in current scenario. Type-4 is the Code with similar functionality that may be syntactically different but logically similar referred as semantic clones. Proposed system presents an algorithm for clone detection based on comparing parts of abstract syntax tree (AST) of programs and finding semantic coding styles.

Keywords: Type-1, Type-2, Type-3, Type-4, AST.

1. INTRODUCTION

Software industry highlights the need of reuse of code and library. As repositories of industrial projects shows significant quantity of duplication. The duplicity is mostly caused by copy-pasting or trying to cover multiple variants of the same processes. Thus, the software maintenance is a complex task. The software code can have presence of large quantity of similar code fragments which increase the code length. It is necessary to correct the error in corresponding cloned code fragments if any fault is occurs in some code. The detection and correction in large scale software is difficult and tough task. Code clones are the similar code content that is knowingly or accidently occurs in code development process. Code fragments that are resembling or equivalent in terms of text, structure, logically or semantically equal. Clones do harm and provide undesired impact on software maintainability, also cloned code may be less error prone than non-cloned code. Risks are indeed associated with existence of clone in due course [3] [4] [5].

I. Basic of Code Cloning

Code cloning is a code fragment similar to one another in the form of semantics and syntax. It is a similar code structures present in different forms in software systems. Researchers have shown that many programs contain remarkable amount of duplicated code. Code clones do not directly cause errors and faults, but the inconsistent changes to them can lead to unexpected behavior of program. The general view to code clones is that the less duplicates the software has, the better it is. Also, a large scale study has shown that nearly every second unintentional and inconsistent change to a clone results in a fault [6].

a) Categories of Code Clones

Code clones are section of source code that is doubled in numerous localities due to replica or even mirroring activity, or content copy paste by program writer. Basically, clones have the functional similarity. Clone upsurges maintenance budget of software system, major standard methodologies to categorize the code clone in account that brings four benchmarks [4].

- *Type-1 Exact clones*: Same code blocks except for differences in whitespace, design, and developer comments.
- *Type-2 Renamed clones*: Code with similar functionality and also syntactically identical fragment. Modification is made in replicated code, such as renaming identifiers.
- *Type-3 Gapped clones*: Code with Type-I and Type-II with additional insertion or deletion of statements are referenced as gapped clones.
- *Type-4 Semantic or logical clones*: Code that may be syntactically different but logically similar referred as semantic clones.

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 9, September 2017

2. LITERATURE SURVEY

Over the last few decades the amount and need for software have increased rapidly. Therefore, there is a pressure to develop software faster, which has led to bad code quality. That, in turn, results in slower future development, higher chances of producing faults, bigger maintenance gaps, and may end up in the need for rewriting the whole project. Research has shown that many programs contain remarkable amount of duplicated code. Code clones do not directly cause errors and faults, but the inconsistent changes to them can lead to unexpected behavior of program.

I. Background

Execution-semantic code-clone detection has been introduced in it, where TWO code fragments are defined to be equivalent to each other when their execution sequences include the same method invocations in the same order, in any possible executions of the target program. Such execution sequences were generated with a kind of static analysis, by constructing an entire call tree from branches and method calls in the target program. Dynamic dispatches of many Object-Oriented programming languages were resolved with called object type information and argument objects. The detection algorithm generates n-grams of an execution sequence and reports the same n-grams as clone classes. Code fragments of a clone class do not include gaps in terms of execution sequence, but include gaps on source code, because a branch or a procedure call in source code results in a kind of skip in some lines of code or a jump to another procedure, such code clones are regarded as a kind of type-3 clone.

a) List of Clone Detection Techniques

- *Text-based Comparison*
- *Token based Comparison*
- *AST Based Comparison*
- *Program Dependency Graph Comparison*
- *Metrics Based Comparison*

b) Details about G++

The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU tool chain and the standard compiler for most Unix-like Operating Systems. The Free Software Foundation distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example. Originally named the GNU C Compiler, when it only handled the C programming language, GCC 1.0 was released in 1987. It was extended to compile C++ in December of that year. Front ends were later developed for Objective-C, Objective-C++, Fortran, Java, Ada, and Go among others. By default, the current version supports gnu++14, a superset of C++14 and gnu11, a superset of C11, with strict standard support also available. It also provides experimental support for C++17 and later.

c) Overview on AST

An AST is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The code is "abstract" in not representing every detail appearing in the real syntax. For example, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if<condition>then expression may be denoted by means of a single node with three siblings, that distinguishes AST from concrete syntax trees, traditionally designated parse trees, which are many times built by a parser during the source code translation and compiling process. Once built, extra information is added to the AST by means of subsequent processing, e.g., contextual analysis. ASTs are also used in program analysis and program transformation system which helps in code clone detection.

3. RELATED WORK

Kaur et al., in [1], presented the analysis on the code clone detection techniques. In code clone there are TWO types of similarities, one is if the program text matches also known as syntactic similarity and the other is if the code matches logically also known as semantic similarity. The syntax similarity is easily detected as compare to semantic similarity. Clone needs high maintenance because if change is made in one clone then that change has to be done in all the respective clones. From previous studies it is clear that 7% to 23% of code clone present in source code of the software. Code detection can be done manually. In manual code detection, code clone is identify by comparing lines one by one, but for the large program it is very time consuming and tedious. Various tools are proposed to detect code clone in the software, all tools follow their own algorithm. The code clone detection tools must provide precise and crucial information about clone.

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 9, September 2017

Jiang et al., in [2], proposed an efficient algorithm for identifying similar sub trees and apply it to representations of tree of source code. The algorithm is based on a novel characterization of sub trees with numerical vectors in the Euclidean space Rn and an efficient algorithm to cluster these vectors with respect to the Euclidean distance metric. With vectors, sub trees in one cluster are considered similar. They have implemented a tree similarity algorithm as a clone detection tool called DECKARD and evaluated it on large code bases written in C and Java including the Linux kernel and JDK. On the base of experiments show that DECKARD is both scalable and accurate, language independent and applicable to any language with a formally specified grammar.
Keivanloo et al., in [3], discussed k-means clustering algorithm to replace the threshold-based cutoff step in the clone detection process. Earlier studies on clone detection have addressed the scalability challenge for clone detection. Due to which they propose an algorithm that helps practitioners in using scalable Type-3 clone detection algorithms across software systems. Specifically, they focus on improving the performance and the ease of use. Use of k-means to determine the number of expected clusters as part of the configuration. The experimental result shows that use of k-means algorithm improves the performance significantly by 12%.
Patil et al., in [4], proposed a multi-model learning technique to detect various types of code clone. Detecting dissimilarity is due to operator or function overloading. Since, it is essential feature of a good object oriented language. It also discusses key techniques that save time in retrieval and comparison of data, by extracting and arranging code that is mined from code document. Experiment shows that effort of comparing the code line by line between TWO files is eliminated.
Chodarev et al., in [5], implemented a pattern recognition algorithm for clone detection based on comparing parts of abstract syntax tree of programs and finding repeating patterns. The algorithm is implemented in the prototype tool that allows detecting clones in programs written in Haskell programming language. The main contribution is believed to be proposal and experimental verification of the new technique for identifying exact and syntactical clones in Haskell code. Results of proposed algorithm found promising even on large code bases.
Kamiya in [6], presented code clone detection and its analysis method, based on an execution semantic and arbitrary granularity model of code fragments. Code clones detected with the proposed method are a kind of type-3 clone, where code fragments exist across boundaries of procedures or modules. The model also seems useful as clone metrics based on the contents and contexts of code fragments in a clone class and extensible to a unified method of code-clone detection and code search. These experiments show that code-clone detection and its analysis method suitable for programming languages.
Kumar et al., in [7], discussed a simple and practical method for detecting exact and near miss clones over arbitrary program fragments in program source code by using AST. Code clone detection is not only helpful in producing more structured code fragment but also in discovering domain concepts and their idiomatic implementations.
Nappa et al., in [8], the first systematic study of patch deployment in client-side vulnerabilities is presented. The patching rates differ considerably among applications; many hosts patch the vulnerability in ONE application but not in the other one. They demonstrate TWO novel attacks that enable exploitation by invoking old versions of applications that are used infrequently, but remain installed. These experiment shows that the median fraction of vulnerable hosts patched when exploits are released is at most 14%.
Siim Karus and Karl Kilgi in [9] presented a set of wavelets-based code clone detection approach for detecting code clones. In order to effectively manage code clones, it is important to know where the clones are and how they relate to each-other. Wavelet analysis has been found to be extremely useful for clone detection in image processing and financial market analysis. Wavelets have the benefit of allowing comparisons than span different scales and strength. The experiment mental evaluation shows that approach is able to effectively identify more clones than alternative algorithms.
Kamiya in [10], discussed a code-clone detection tool named Agec, based on a model of execution of multi-level invocation, which finds code segments equivalent in terms of method invocation, but not equivalent in terms of code structured. Refactoring is an important application of code clone detection. A various number of code-clone detection tools have been designed to detect the code segments that need to be refractor. Moreover, such clone detection tools should be able to find a code clone which includes both not-yet-refractor code fragments and the refractor code fragments equivalent to them.
Lin et al., in [11], an automatic clone summarization technique that presented the extract commonalities of syntactic contexts in which code clones occur as syntactic patterns and cluster code clones with regard to these patterns for developers to locate and maintain code clones relevant to certain syntactic contexts in a top-down manner. The first aim is to builds an ontology model that captures code clones, program elements, and their relations. On the basis of the ontology model, the aim then uses sequence matching and clustering techniques to cluster code clones by their common syntactic contexts. Finally, it abstracts commonalities of syntactic contexts in which code clones of syntactic patterns occurred. Clonepedia have developed a proof-of-concept tool which has been integrated with Eclipse IDE. The Clonepedia tool implements a Graphical User Interface GUI to support developers to interactively explore and analyze code clones by their syntactic patterns.

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 9, September 2017

Crussell et al., in [12], implemented scalable approach to detecting similar Android apps based on their semantic information. Implementation of this approach in a tool called AnDarwin takes place and evaluate it on 265,359 apps collected from 17 markets including Google Play store and various third-party markets. As compare to earlier approaches, AnDarwin has four advantages: it avoids comparing apps pair wise, thus greatly improving its scalability; that tool analyzes only the app code and does not rely on other information such as market of the app's, signature, or description thus greatly increasing its reliability; it can detect both full and partial app similarity; and it can detect automatically library code and remove it from the similarity analysis. Additionally, AnDarwin detects similar code that is injected into many apps, which may indicate the spread of malware. Evaluation demonstrates AnDarwins ability to accurately detect similar apps on a large scale.
Gode and Koschke, in [13], presented an incremental clone detection algorithm, based on the results of the previous revision's analysis which detects clones. Also, it creates a mapping between clones of one revision to the next, supplying information about the addition and deletion of clones. Results demonstrate that the incremental technique considerably less time consuming than a non-incremental approach if the changes do not exceed a certain fraction of the source code. For detection and evolutionary clone analysis, an incremental analysis is useful.
Rilling et al., in [14], conducted a set of empirical studies on a large open source code corpus to gain insight about its characteristics and used these results to design and optimize a multi-level indexing approach using based on hash table and binary search to improve internet scale real-time code clone search response time. Finally, Rilling et al. performed an evaluation on an Internet scale corpus. Their approach maintains a response time for 99.9% of clone searches in the microseconds range, while supporting the requirements.
Harman et al., in [15], introduced desktop and parallelized cloud-deployed versions of a search based solution that finds suitable configurations for empirical studies. Harman et al. evaluate their approach on 6 widely used clone detection tools applied to the Bellon suite of 8 subject systems. And evaluation reports the results of 9.3 million total executions of a clone tool.
Livieri et al., in [16], proposed a novel approach to distributed large scale clone analysis. The approach has been implemented in their prototype D-CCFinder, and it has been applied to a vast collection of open source programs. One of the objectives is the exploration of the application of a distributed approach at software engineering in the context of Mega Software Engineering. Livieri et al. illustrates that D-CCFinder as a fairly cheap and practical method for large scale code clone analysis employing commodity hardware, freely available software, an already existing network infrastructure, and a trivial but efficient implementation.
Priyambadha and Rochimah in [17], presented a concern on how to find an input, output and effect in void and non-parameterized method. Detection of input, output and effect has done using PDG. The result is used to reconstruct void and non-parameterizes method. Using random input data, trial is performed on each method to get the behavior methods. Trial is done using small size source code from jDraw. This method can also detect type-1, type-2 and type-3 clones beside the semantic clone itself.
Mondal et al., [18], presented a tool called SPCP-Miner which is the pioneer one to automatically identify and rank the important refactoring as well as important tracking candidates from the whole set of clones in a software system. SPCP-Miner implemented the technique that is used to conduct a large scale empirical study on SPCP clones. SPCP-Miner can help researchers in better management of code clones by suggesting important clones for refactoring or tracking.
Soh et al., [19], proposed a technique of finding Android applications clones based on the analysis of User Interface data collected at runtime. By leveraging on the multiple entry points feature of Android applications, the UI data collected easily without the need to generate relevant inputs and execute the application. Evaluated approach on a set of real-world dataset.
Roy et al., in [20], presented a comparison and evaluation of the current state in clone detection techniques and tools and organize the large amount of information into a coherent conceptual framework. Begin with generic clone detection, background concepts and an overall taxonomy of current techniques and tools. After that classify, compare and evaluate the techniques and tools in TWO different dimensions. First, classify and compare approaches based on a number of facets, each of which has a set of attributes. Second, 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, Chanchal K. Roy et al. provide examples of how one might use the results of this to choose the most appropriate clone detection tool or technique in the context of a particular set of goals and constraints. The primary contributions are a schema for classifying clone detection techniques and tools and a classification of current clone detectors based on the schema, and taxonomy of editing scenarios that produce different clone types and a qualitative evaluation of current clone detectors based on the taxonomy.
Koschke et al., in [21], described how programmer can make use of suffix trees to find clones in abstract syntax trees. This new approach is able to find syntactic clones in linear time and space. The paper reports the results of several large case studies in which they empirically and quantitatively compare the new technique to nine other techniques using the Bellon benchmark for clone detectors. As a side effect of case study, they extend the Bellon benchmark by adding

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 9, September 2017

reference clones. There are additional reasons for AST-based clone detection beyond better precision. Because most refactoring tools are based on ASTs, they need to access clones using nodes in the AST if they support clone removal. Also, ASTs offer syntactic knowledge which can be helpful to filter certain types of clones. For example, one could exclude clones in declarative code or strictly sequential assignments as in constructors, which are many times unavoidable. From a research point of view, it would be also meaningful to categorize and see where mostly the redundancy occurs in syntactic terms. It could also help to identify programming language deficiencies.
Stefan Bellon et al., in [22], presented an experiment that evaluates six clone detectors. The selected techniques cover the whole spectrum of the state of art in clone detection. The techniques work on text, lexical and syntactic information, software metrics, and program dependency graphs.
Sajnani et al., in [23], aimed is to find the activity of reuse of code within open source Java development, and to identify the criteria under which files are reused in this manner. Sajnani et al., developed a novel method of file level code clone detection that is scalable to millions of files and applied the method to the Source Repository, which contains over 13,000 Java projects aggregated. Results of method detected that in excess of 10% of files are clones, and over 15% of all projects contain at least one cloned file. Also, manually examine a large number of the reported clones. Results found the most commonly cloned files to be Java extension classes, both large and small.
Saini and Verma, in [24], an implementation of association rule FP-growth algorithm is done on similar android applications, which is a novel area itself. It described a framework which mines association rules from android application clones. The analysis of the dependencies and rules proved fruitful and gave meaningful results. From the code interesting relations are extracted, which can help the developers in performing future operations like modification of the code easily.
Hummel et al., in [25], presented a novel, index-based clone detection algorithm for type-1 and 2 clone detections that is both incremental and scalable. For very large software, it enables a new generation of clone management tools that provide real-time cloning information. Discusses several case studies that show both its suitability for real-time clone detection and its scalability on 42 MLOC of Eclipse codes, average time to retrieve all clones for a file was below 1 second; on 100 machines, detection of all clones in 73 MLOC was completed in 36 minutes.
Svajlenko et al., in [26], presented a big data clone detection benchmark that finds true and false positive clones in a big data inter-project Java repository. By mining and manually checking clones of ten common functionalities, the benchmark was built. The benchmark contains six million true positive clones of different clone types: Type-1, Type-2, Type-3 and Type-4, including various strengths of Type- 3 similarity. It shows how the benchmark can be used to measure the recall and precision of clone detection techniques.
Baxter et al., in [27], presented simple and practical methods for detecting Type-1 and Type-2 clones over arbitrary program fragments using abstract syntax trees in program source code. This method operate in terms of the program structure, clones could be removed by mechanical methods producing in-lined procedures or standard preprocessor macros. Joshi et al., in [28], the code reduction and decentralized system with multiple smart nodes are implemented. The code reduction method designates clusters for faster detection of clones. The proposed method detects duplicate code in efficient way by using decentralized computing and code reduction. Significant efforts are applied to address Type-3 and various other clones that set challenging aspects in the research. These experiment evaluates as code complexity, weighted graphs enhances precision in detection. Schwarz et al., in [29], proposed for very large amounts of source code, a set of lightweight techniques in the presence of multiple versions. The idea behind these techniques is to use bad hashing to get a quick answer. Give a report on a case study for thousands of software projects, the Squeak source ecosystem, with more than 40 million versions of methods, across more than seven years of evolution. Finally provide estimates for the prevalence of type-1, type-2, and type-3 clones in Squeak source.

Schneider et al., in [30], investigated the effectiveness of simhash, a fingerprint based data similarity measurement technique for detecting both Type-1 and Type-2 clones in large scale systems. And from the experimental data show that simhash is indeed effective in identifying various types of clones in a software system despite wide variations in experimental circumstances. It is also suitable for building other tools as a core capability, such as tools for: incremental clone detection, code searching, and clone management. Neamtiu et al., in [31], presented a tool for quickly comparing the source code of different versions of a C program. This is based on partial abstract syntax tree matching, and can track simple changes to global variables, types and functions. These changes can characterize aspects of software evolution useful for answering higher level questions. Give report results based on measurements of various versions of popular open source programs, including BIND, OpenSSH, Apache, Vsftpd and the Linux kernel.

4. PROPOSED SOLUTION

Using the help of reference on literature survey, some type of code clones used AST to solve the problem of code cloning. Use of AST found very helpful in solving issues related to code cloning based on logical similarity which decreases the maintenance cost, increase the efficiency, comprehensibility, etc. The proposed solution introduces a new approach of using AST for meaningful Code Clone Detection based on Logical Similarity.

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 6, Issue 9, September 2017

I. Proposed Approach

The proposed approach describes the solution to the Type-4(Semantic) clone detection. An existing system consists of drawback of lack of techniques and proposed solution overcomes the drawback by using AST. AST based clone detection is a technique that is said to be exist if the Type-4 clone taken into the consideration. Semantic code clone detection means finding the clone from the syntactically different codes and logically similar codes.

a) Architecture

The Partial Object Oriented programming languages or purely OOP's for the comparison to find the meaningful code clone. Applying AST to C++ and Java are easy as compare to other OOP languages. Due to which, C++ and Java languages are used. Figure 3.1 shows the Architecture of Cloning system. Because the code clone is detecting in between the TWO syntactically different but logically similar languages, there is the need of entering the TWO different codes of TWO different languages as input. Before entering that code, apply the THREE approve conditions for selecting the file of code. Out of that THREE conditions -1 is use if any error is happen during selecting program file, 0(ZERO) is use if file select and open properly and if the cancel button is click on the Open Dialogue box then 1 is use. Equation 3.1 shows process as follows,

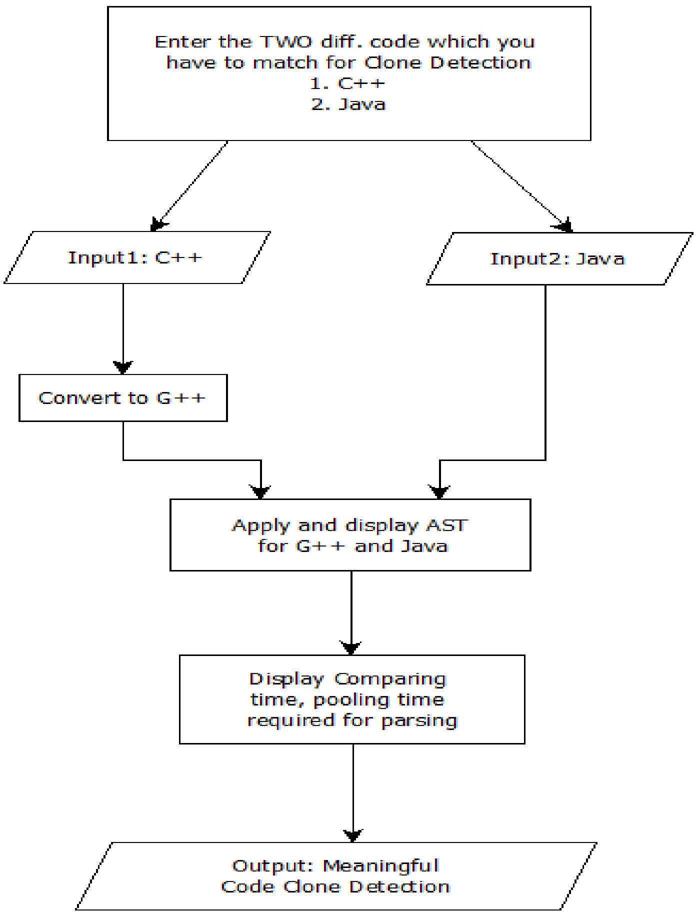
Enter the TWO diff. code which you have to match for Clone Detection
1. C++
2. Java
Input1: C++
Input2: Java
Convert to G++
Apply and display AST for G++ and Java
Display Comparing time, pooling time required for parsing
Output: Meaningful Code Clone Detection

Figure 3.1: Architecture of Code Clone Detection

Let, A be the APPROVE Condition,
Hence,

$$A = \begin{cases} -1, & \text{error} \\ 0, & \text{file open} \\ 1, & \text{cancel} \end{cases}$$

After giving the approve condition as 0, P1 becomes,

$$P1 = \{F1, F2\}$$

$$P1 \ni F1 \qquad (3.1)$$
$$P1 \ni F2$$

Where, P1: Phase 1
F1: code of C++
F2: code of Java

Due to inconvenience in applying the AST to the C++ code enter, convert that C++ code in the G++ code for properly apply the AST to the C++ code. From equation 3.1, the next step is given as follows,

Let, $F1 = \text{code of C++}$ (3.2)

After converting that code into the G++, from equation 3.2, the equation 3.3 is as follows,

$\therefore$ $F1 \rightarrow G1$ (3.3)

Where, F1: code of C++
G1: code of G++

Now, apply the AST to the newly made G++ code and Java code for finding the better result in code clone detection. Equation 3.4 becomes,

$$Å = \{ÅG, ÅJ\}$$
$$G1 \rightarrow ÅG$$
$$F2 \rightarrow ÅJ \qquad (3.4)$$

Where, Å: Set of AST
ÅG: AST for G++
ÅJ: AST for Java

Parsing is done in converting the programming from code to the AST. During the conversion, various token like Modifier type, Data type, Variable, Variable Declaration, etc. are parsed, as shown in the Figure 2.1 with coding example. After creating the AST of both the code, now, there is time to do the matching and comparison between the TWO AST's. Equation 3.5 given below shows the comparison between TWO different AST's.

Hence, $ÅG == ÅJ$ (3.5)

For doing the matching of the every token and comparing them with each and every other token, need to find different timing like comparing timing, filtering time, pooling time, etc. and other details as number of node before pruning, number of node after pruning, etc. Equation 3.6 shows the timing require and other details.

$$\therefore \quad P3 = \{T, P, Pr, C, A\}$$
$$T \ni \{T1, T2, T3\}$$
$$P \ni \{PBP, GBP, GAP\}$$
$$Pr \ni \{NBPr, NAPr\}$$
$$C \ni \{LC, NC\}$$
$$A \ni \{CA, ACA, EA, IA, MA, CRA, FA\} \quad (3.6)$$

Where, P3: Phase Third
T: Set of required time like Comparison pooling time, etc.
P: Set of similarity pair and group before and after pruning
Pr: Set of node before and after preprocessing
C: Set of Count
A: Set of different amount as Enum amount, Interface amount, etc.

As discuss before, AST contains the token separated by parsing method. With that parsing tokens also contain the line number in the code and the position in that line. In the result, it shows the different set of Similarity Group build by string made by the different tokens which are match with each other in AST comparison. The following equation 3.7 and 3.8 shows the results as given below,

$$P4 \ni R \qquad (3.7)$$

And $R = \{SG1, SG2 \ldots\}$ (3.8)

Where, P4: Phase 4
R: Result
SG: Similarity Group i.e. SG1, SG2...

5. DESIGN

The section presents the design algorithm of the proposed system.

a) Proposed System Algorithm

Figure 3.2 shows the detail flow chart of the algorithm in semantic code clone detection system. The following Semantic Code Clone Detection algorithm is focuses on the FOUR steps, namely inputs, tree structure, time and count details and the result. The first step takes TWO different codes as an input to the proposed system. In next step, apply AST. But, due to the incompatibility between C++ and AST, AST can't apply directly to the C++.

Thus, C++ needs to convert in G++ first. G++ is the GNU Compiler Collection which is run on various operating systems such as Mac OS, Linux, Windows, etc. After conversion, apply AST to both the code and convert them into the tree structure. On the basis of tree structure comparison and matching is to be done. During the matching step, different required times and count are noted which are given in next step. Finally in the last step, if matching is found then result is display with the line numbers and position in the line. Otherwise give the output as "No similar group is found". All discuss steps are given in the algorithmic format in algorithm 1 as follows,

Let, AC = APPROVE Condition
Input = Input files
FT = Filtering Time;
CT = Comparing Time;
PT = Pooling Time
Let, CA = Class Amount;
EA = Enum Amount;
IA = Interface Amount;
MA = Method Amount
Let, NC = Node Count
N = End of program
n = End of file
NCi and NJi = Node of C++ and Node of Java

Algorithm 1 Semantic Code Clone Detection Algorithm

```
Input: TWO different languages program.
Output: Code Clone Detection Result.
Require: C++ and JAVA program.
1: Start
2: Enter input
3: if AC == 0 then
4: Input read and writes
5: if Input == ".CPP" then
6: Convert to G++
7: end if
8: Convert Input to tree structure
9: Calculate FT, CT, PT, CA, FA, IA, MA, NC N
10: Match tree of both inputs
11: for i = 0; i ≤ n; i++ do
12: if NCi == NJi then
13: Display match line no. and position in output
14: else
15: No Similarity group found.
16: end if
17: end for
18: else if AC == -1 then
19: Error encounter.
20: else
21: Cancel by user.
22: end if
23: End
```

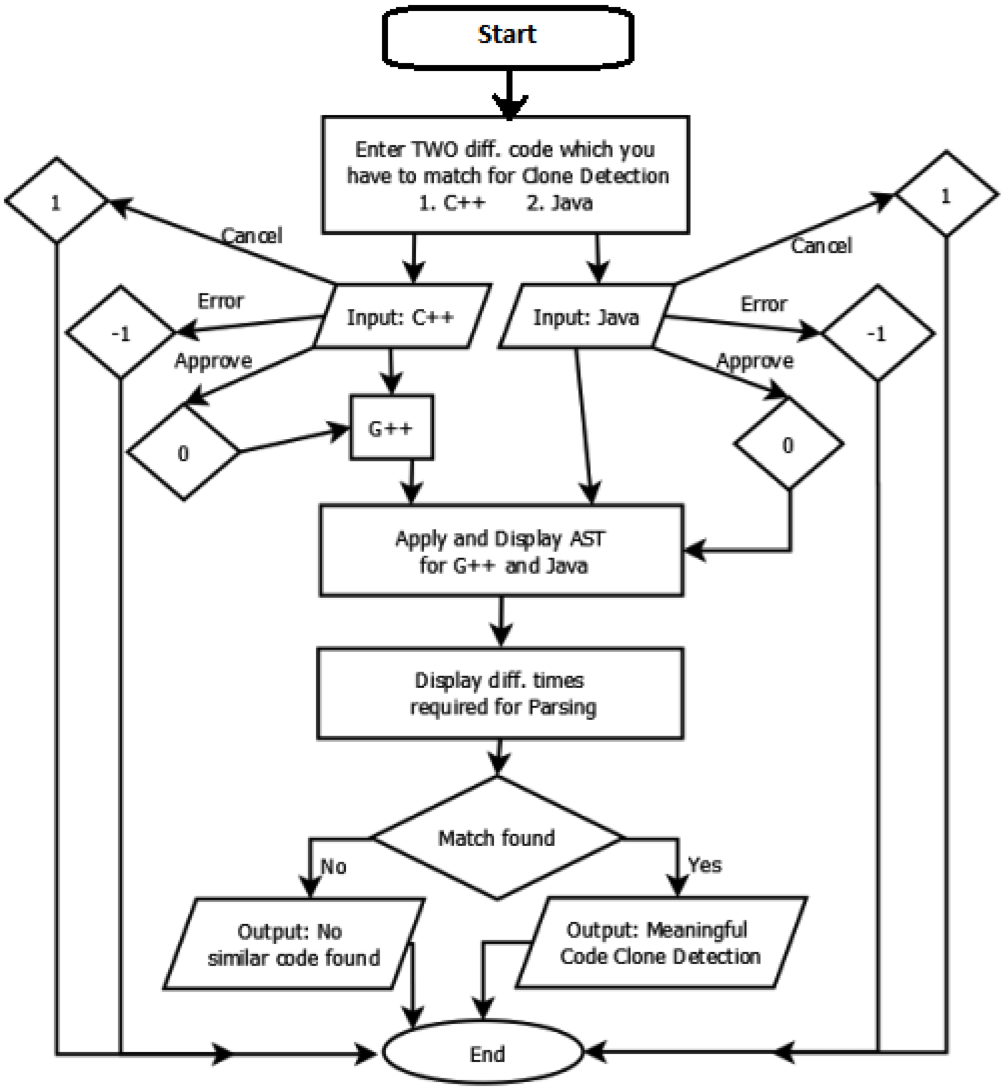
Start
Enter TWO diff. code which you have to match for Clone Detection 1. C++ 2. Java
Cancel
1
Error
-1
Approve
0
Input: C++
Input: Java
G++
Apply and Display AST for G++ and Java
Display diff. times required for Parsing
Match found
No
Yes
Output: No similar code found
Output: Meaningful Code Clone Detection
End

Figure 3.2: Flow Chart

6. RESULT AND DISCUSSION

Result and discussion section is an important part of research work. Evaluation of the proposed approach versus existing approach is carried out in the result and discussion section. Result section presents the experimental results of the proposed approach as well as the existing approach. Evaluation of the both the approaches are carried out in the discussion section on the basis of obtained results. In some cases performance metrics are used to evaluate the system, in order to decide which ONE is the best.

I. Experimental Setup

Experimental setup is the part of research in which the experimenter analyzes the effect of contribution on existing system. Importance and unique aspect of setup used in experiments are introduced in experimental setup section.

a) Simulation Results

Parameters are the various numerical or other measurable factors forming one of a set that defines a system or sets the conditions for operation. Necessary minimum hardware and software requirements to run the experimental setup are as follows.

- *Operating System: Windows 7 (64 Bit)*
- *RAM: 1 GB Minimum*
- *Processor: Intel Core i3 (64 Bit)*
- *Front End Tool: Net beans IDE 8.1 (64 Bit)*

- *Back End Tool: Files (.cpp and .java)*
- *Other Requirements: Java 8 (64 Bit), G++*

b) Implementation Result

In the implementation result section, performance evaluation of the proposed algorithms is presented. Table 4.1 shows clone detection values in terms of number of clone exist and number of clone found. The evaluation is very useful for the programmer. Using the evaluation programmer can give better improvements in software building.

Table 4.1: Clone Detection Values

Input set for cloning	Number of Clone exist	Number of Clone found
Input 1	2	2
Input 2	1	1
Input 3	3	3
Input 4	4	3
Total	10	9

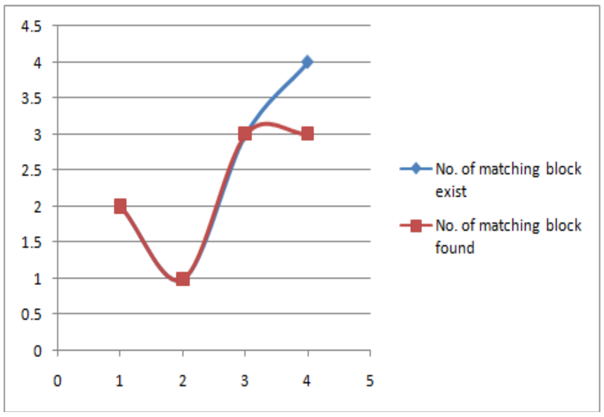

Figure 4.2: Semantic Clone Detection Graph

7. DISCUSSION

The purpose of the discussion section is to state interpretations and opinions, explain the implications of findings in the experimental evaluation. Main function of discussion section is to answer the questions posed in the Introduction, explain how the results support the answers and, how the answers fit in with existing knowledge on the topic. The discussion section is important to know the detail advantage and need about proposed solution.

After giving the input to the system in C++ and in Java, apply the AST. But in the case of C++, AST can't apply directly. To apply there is a need to convert C++ into the G++. G++ is a GNU based compiler which is run on Windows, Linux and MAC operating system also. The detail of G++ is given in the literature survey. After the conversion of C++ to G++, to overcome the problem of finding Type-4 code cloning, AST is applying in the proposed system. AST give the details about the code which is use as the input. It provides the detail tree structure of the code. AST parse the every literals and token in the code such as separating the modifiers, data types, variables, etc. Comparison, filtering, node counting has done after parsing of literals and token.

The required time for comparison, filtering, etc. is given in the next part of the system. The result is given in the last phase of the system. The output shows that Type-4 code clone is detected in TWO different language codes. In short AST is use to solve the many problem up to Type-3 and gives the high performance rate as compare to other techniques as given in the literature survey. Also, it shows that only AST and PDG come under the category of Semantic Code Clone Detection. In the integrity, PDG is medium dependant where as AST is algorithm dependent. AST doesn't require the medium to give the high performance. AST gives the high efficiency with low complexity. Due to detection of the semantic code clone the maintenance of the software is getting easier and the required cost for the maintenance is going to be reducing. Thus, it increases the performance, quality, consistency, maintainability and comprehensibility of the software.

8. CONCLUSION AND FUTURE WORK

From the discussion it is clear that code cloning detection is a technique of finding the semantically and syntactically similar clones. The code which is syntactically different, but semantically similar remains hidden from code clone

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering

ISO 3297:2007 Certified

Vol. 6, Issue 9, September 2017

detection techniques. AST can be used to overcome the challenge. Usage of AST will give better and superior results than the traditional approach.
In the future, Semantic code clone detection can be further improve using supervised or unsupervised learning algorithm or the algorithm build by combination of both learning.

REFERENCES

[1] M. Kaur and M. Lal, Review on various code clone detection techniques," International Journal of Advanced Research in Computer Science and Software Engineering, vol. 5, May 2015.

[2] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, Deckard: Scalable and accurate tree-based detection of code clones," in Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007, pp. 96-105.

[3] I. Keivanloo, F. Zhang, and Y. Zou, Threshold-free code clone detection for a large-scale heterogeneous java repository," in 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2015, pp. 201-210.

[4] R. V. Patil, S. D. Joshi, S. V. Shinde, and V. Khanna, An effective approach using dissimilarity measures to estimate software code clone," in International Conference on Electrical, Electronics, Signals, Communication and Optimization (EESCO). IEEE, 2015, pp. 1-6.

[5] S. Chodarev, E. Pietrikova, and J. Kollar, “Haskell clone detection using pattern comparing algorithm," in 13th International Conference on Engineering of Modern Electric Systems (EMES). IEEE, 2015, pp. 1-4.

[6] T. Kamiya, “An execution semantic and content and context based code-clone detection and analysis”, in 9th International Workshop on Software Clones (IWSC). IEEE, 2015, pp. 1-7.

[7] G. Anil, C. Reddy, and A. Govardhan, “Software code clone detection using ast” International Journal of P2P Network Trends and Technology, pp. 33-39, June 2014.

[8] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, “The attack of the clones: A study of the impact of shared code on vulnerability patching," in Symposium on Security and Privacy (SP). IEEE, 2015, pp. 692-708.

[9] K. Kilgi, “Code clone detection using wavelets," Master's Thesis, 2014.

[10] T. Kamiya, Agec: An execution-semantic clone detection tool," in 21st International Conference on Program Comprehension (ICPC). IEEE, 2013, pp. 227-229.

[11] Y. Lin, Z. Xing, X. Peng, Y. Liu, J. Sun, W. Zhao, and J. Dong, Clonepedia: summarizing code clones by common syntactic context for software maintenance," in International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2014, pp. 341-350.

[12] J. Crussell, C. Gibler, and H. Chen, Andarwin: Scalable detection of android application clones based on semantics," IEEE Transactions on Mobile Computing, vol. 14, no. 10, pp. 2007-2019, Oct 2015.

[13] N. Gode and R. Koschke, “Incremental clone detection”, in 13th European Conference on Software Maintenance and Reengineering, CSMR. IEEE, 2009, pp. 219-228.

[14] I. Keivanloo, J. Rilling, and P. Charland, “Internet-scale real-time code clone search via multilevel indexing," in 18th Working Conference on Reverse Engineering (WCRE). IEEE, 2011, pp. 23-27.

[15] T. Wang, M. Harman, Y. Jia, and J. Krinke, Searching for better configurations: a rigorous approach to clone evaluation," in Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 455-465.

[16] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007, pp. 106-115.

[17] B. Priyambadha and S. Rochimah, “Case study on semantic clone detection based on code behavior,” in 2014 International Conference on Data and Software Engineering (ICODSE). IEEE, 2014, pp. 1-6.

[18] M. Mondal, C. K. Roy, and K. A. Schneider, Spcp-miner: A tool for mining code clones that are important for refactoring or tracking," in 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), March 2015, pp. 484-488.

[19] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, “Detecting clones in android applications through analyzing user interfaces," in Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension. IEEE Press, 2015, pp. 163-173.

[20] C. K. Roy, J. R. Cordy, and R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Sci. Computer Program., vol. 74, no. 7, pp. 470-495, May 2009.

[21] R. Koschke, R. Falke, and P. Frenzel, Clone detection using abstract syntax suffix trees," in 13th Working Conference on Reverse Engineering (WCRE). IEEE, 2006, pp. 253-262.

[22] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, Comparison and evaluation of clone detection tools," IEEE Transactions on software engineering, vol. 33, no. 9, 2007.

[23] J. Ossher, H. Sajnani, and C. Lopes, File cloning in open source java projects: The good, the bad, and the ugly," in 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, 2011, pp. 283-292.

[24] U. Saini and S. Verma, “Generation & analysis of association rules from android application clones," in International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, 2015, pp. 1255-1262.

[25] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, Index-based code clone detection: incremental, distributed, scalable," in IEEE International Conference on Software Maintenance (ICSM). IEEE, 2010, pp. 1-9.

[26] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, Towards a big data curetted benchmark of inter-project code clones," in IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2014, pp. 476-480.

[27] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, Clone detection using abstract syntax trees," in International Conference on Software Maintenance Proceedings. IEEE, 1998, pp. 368-377.

[28] R. V. Patil, S. D. Joshi, S. V. Shinde, D. A. Ajagekar, and S. D. Bankar, Code clone detection using decentralized architecture and code reduction," in International Conference on Pervasive Computing (ICPC). IEEE, 2015, pp. 1-6.

[29] N. Schwarz, M. Lungu, and R. Robbes, On how often code is cloned across repositories," in Proceedings of the 34th International Conference on Software Engineering. IEEE Press, 2012, pp. 1289-1292.

[30] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in 18th Working Conference on Reverse Engineering. IEEE, 2011, pp. 13-22.

[31] I. Neamtiu, J. S. Foster, and M. Hicks, Understanding source code evolution using abstract syntax tree matching," ACM SIGSOFT Software Engineering Notes, vol. 30, no. 4, pp. 1-5, 2005.