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

*International Journal of Advanced Research in Computer and Communication Engineering*
*Vol. 2, Issue 4, April 2013*

# AN AUTOMATIC METHOD OR STATEMENT LEVEL PLAGIARISM DETECTION IN SOURCE CODE USING ABSTRACT SYNTAX TREE

D.Poongodi[1], G.TholkkappiaArasu[2]

Research Scholar, Manonmaniam Sundaranar University, Tirunelveli[1]

Principal, AVS Engineering College, Salem[2]

**ABSTRACT**: Plagiarism detection plays an important role in software security protection and license issues. Source-code plagiarism detection method can be classified as string-based, token-based, parse-tree-based and program-dependency-based. All of these approaches have certain limitations and can not meet the requirements when the source code is large and may produce false positives. But, parse-tree based detection improves the detection ability and efficiency. This paper describes method based source code detection, which detect the simple plagiarized code like exact match, near exact match and longest common sequence. And also proposes the agent based detection which will perform the detection automatically. Automatic plagiarism detection will be helpful for code clone detection in software industry.

**Keyword**: abstract syntax tree, plagiarism detection, source code plagiarism detection, parse tree, code clone.

## I. INTRODUCTION

Source code plagiarism refers to "the act of 'copying others code' without giving credit to the author of the code". Plagiarism increases in the number of resources available in the electronic form. The easy access to the internet has also been increased. Manual detection of plagiarism is not very easy and is time consuming due to the vast amount of contents available. As the amount of programming code created is increasing, different techniques are available to detect plagiarism in source code. Intentional plagiarism is used knowingly from others work without any acknowledgement. Unintentional plagiarism becomes identical when different authors unknowingly use the same logic. Intentional Plagiarisms are classified into two different categories, they are

1.1 Low-Level Plagiarism and
1.2 High-Level Plagiarism

### *1.1 LOW LEVEL PLAGIARISM*

- Copy-paste (with some spacing and comments & modification)
- Plagiarism with renaming
  - Methods, fields, classes
- Reordering of the code (that does not affect the final state)
- Addition of redundant lines of code

### *1.2 HIGH LEVEL PLAGIARISM*

- Control structures Change
- Mixing of Different sources
  - Procedures
  - Classes

Mixing of own and others' code

### 1.3 *PLAGIARISM DETECTION APPROACHES*

According to Roy and Cordy, [1] source-code similarity detection algorithms can be classified as based on the following

1. String Based Detection
2. Token Based Detection
3. Parse Tree(AST) Based Detection
4. PDG Based Detection
5. Metric Based Detection
6. Hybrid Based Detection

- Strings – look for exact textual matches of segments, for instance five-word runs. Fast, but can be confused by renaming identifiers.
- Tokens – as with strings, but using a lexer to convert the program into tokens first. This discards whitespace, comments, and identifier names, making the system more robust to simple text replacements. Most academic plagiarism detection systems work at this level,

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

*International Journal of Advanced Research in Computer and Communication Engineering*
*Vol. 2, Issue 4, April 2013*

using different algorithms to measure the similarity between token sequences.

- Parse Trees – build and compare parse trees. This allows higher-level similarities to be detected. For instance, tree comparison can normalize conditional statements, and detect equivalent constructs as similar to each other.
- Program Dependency Graphs (PDGs) – a PDG captures the actual flow of control in a program, and allows much higher-level equivalences to be located, at a greater expense in complexity and calculation time.
- Metrics – metrics capture 'scores' of code segments according to certain criteria; for instance, "the number of loops and conditionals", or "the number of different variables used". Metrics are simple to calculate and can be compared quickly, but can also lead to false positives: two fragments with the same scores on a set of metrics may do entirely different things.
- Hybrid approaches – for instance, parse trees + suffix trees can combine the detection capability of parse trees with the speed afforded by suffix trees, a type of string-matching data structure.

### 1.4 *Advantages Of AST Based Plagiarism Detection Algorithms*

The Abstract syntax tree based linear representations of source code is efficient than the other comparison algorithms, because the other comparison approach hide the structure of the source code. They may identify parts of code between two different programs as plagiarism which are not relevant, for example in between source code {break, continue} and so. Large source code plagiarism detection can not deal without manipulation of structured representations like AST or PDG. PDG (Program Dependency Graph) is much costlier than AST.

The parse tree or AST contains the complete information about the source code. Although the variable names and literal values of the source are discarded in the tree representation, more sophisticated methods for the detection of plagiarism.

One of the benefits of the working on the lexical level is that the lexical stream better reflects the “structure" of a program. The parse tree or derivation tree built from the lexical for a program also exhibits structure for the underlying program. Furthermore, ASTs offer syntactic knowledge which can be leveraged to filter certain types of plagiarism.

The existing detection algorithm can meet either the detection ability or detection efficiency. 50% of the plagiarism detection algorithm based on AST using the string based detection after parsing the source code. These type of algorithm might have high efficiency but lacking on detection ability. Remaining plagiarism detection algorithm based on AST using the AST node directly to compare the source code and find the plagiarism. These types of algorithm might have high detection ability but lacking on detection efficiency.

This paper introduce an automatic component to detect plagiarism in source code either method level or statement level using abstract syntax tree. In the beginning developed an algorithm to compare the source code and detect the plagiarism in less time. Followed by developed the component to detect the plagiarism. So this component will be efficient and capable to detect the plagiarism in great manner.

## II. ALGORITHM FOR CODE DETECTION METHOD

In the tree-based approach a program is parsed to a parse tree or an Abstract Syntax Tree (AST) with a parser of the language of interest. Similar sub trees are then searched in the tree with some tree matching techniques and the corresponding source code of the similar sub trees are returned as plagiarism classes.

In part, programming languages are defined by their grammars, which describe the set of all possible strings that represent programs (called a language). During the compilation process, a compiler builds a parse tree which represents the program and uses this tree to guide compilation.

Traverse the parse tree of different parts of source code to identify the plagiarism between the programs.

Steps of algorithm are given below:

1. Parse the source code into a AST using AST Parser
2. Compare the Parse trees, based on the methods as follows

a. Count the number of children nodes that matches for both the methods.

b. If the number of children nodes matches with two different methods and if it is greater than or equal to three then do the comparison

c. Find the number of children which is matched with children for both statements.

d. Find the threshold value using the following formula.

$$\text{Ratio}=\frac{nm}{\text{Min}(\sum m1,\sum m2)!}\ x\ \frac{\text{Min}(\sum_{i=0}^{nm} nmc(m1),\sum_{i=0}^{nm} nmc(m2))}{\text{Min}(\sum_{i=0}^{nm} nmc(m1),\sum_{i=0}^{nm} nmc(m2))}$$

Where nm is number of node matches in between method1 and method2.

Where nmc is number of children count for the node matched.

e. Ratio of threshold can be configured with 0.75 , 0.9 or any value greater than 0.5

3. Compare the tree based on statements as

a. Count the number of children node match between two different Statements.

b. Find the number of children of matched children for both statements.

c. Find the threshold value using the following formula.

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

*International Journal of Advanced Research in Computer and Communication Engineering*
*Vol. 2, Issue 4, April 2013*

$$Ratio = \frac{nm}{Min(\sum sl1, \sum sl2)!} \times \frac{Min(\sum_{i=0}^{nm} nmc(sl1), \sum_{i=0}^{nm} nmc(sl2))}{Min(\sum_{i=0}^{nm} nmc(sl1), \sum_{i=0}^{nm} nmc(sl2))}$$

Where nm is number of node matches in between statement list1 and statement list2.

Where nmc is number of children count for the node matched.

d. Ratio of threshold can be configured as 0.75 , 0.9 or any value greater than 0.5

### *2.1 METHOD OF COMPARISON*

The proposed approach of comparison is different from the existing algorithms.

After parsing the source code into parse tree, if the comparison is method level then comparing is as follows

1. Collecting all the methods and its child node up to leaf node.
2. Count the number of nodes in each method
3. Based on the number of nodes compare with other source code, if the count difference is less than or equal to 3 then do the node match as follows

a. Take the first node or statement from the given list of code1

b. Compare to the first node or statement of the other list of code

c. If both the nodes match then compare the next statement of both the list of code. Else compare the first node of list1 to compare the second node of list2 until to find the match node or compare with all the nodes in the list.

d. While continuous matching (like continuously 3 nodes are matched), if the next node is not matched then that node will be compared from the first node including matched node.

e. Steps c and d will be repeated until the end of both the list or all nodes are comparing with all the nodes.

4. Depends upon the number of child node matches, the threshold value using the ratio is calculated.
5. If the threshold value is between 0.1 and 0.9 then there is a similarity between both the codes.

Statement level comparison is as follows

1. Collecting all the statements from different source codes.
2. Take the first statement node from the given source code and compare with the similar program first statement node.
3. If both the statements are equal or match with each other then take the second node of both the source code, if it matches then compare the next as same until there is a match.
4. In the continuous matching , if there is any mismatch is found then start comparing the similar program list mismatch node with the first statement node of the given source code.
5. If first statement node is not matched, take second node and compare with the mismatch node.
6. If matched then next node of both the source codes are compare like step 3.
7. Matched nodes are stored in file and this report is the output of the algorithm
8. This algorithm finds the exact match and near exact match like longest common sequence.

For Example,

i) *Exact match or no change Comparison*.

If the exact match codes are as follows

| | | |
|---|---|---|
| 1. | int i; | int k; |
| 2. | int j; | int m; |
| 3. | for( i=0;i<10;i++) | for( k=0;k<10;k++) |
| 4. | for ( j=0;j<10;j++) | for (m=0;m<10;m++) |
| 5. | System.out.println(i+j); | System.out.println(k+ m); |

TABLE1. SAMPLE SOURCE CODE FOR EXACT MATCH

In the above source code both the list are same, the only difference is identifier name has been changed. This type of plagiarism is exact match or no change plagiarism.

The proposed algorithm compares the given source code with similar type of program and it is represented as the below diagram for the exact match. First it takes the statement node1 (s1) from both the parsed source code and compares. If it is matches then compares the next node of both the list. The process continuous until the end of both the parsed source code is reached.


Match List1 – s1, s2, s3, s4, s5
Match List2 – s1, s2, s3, s4, s5

FIGURE1. SAMPLE COMPARISON FOR EXACT MATCH

Finally it gives the report as text file which contains matched node in the given source code and the similar program source code.

ii) *Near exact match or Longest Common Sequence (LCS) Comparison*

"Near exact match" is like copying part of the source code from others and adding own code or including unnecessary codes. If the plagiarizer includes some code then the source code might looks like different from the original code.

Some of the plagiarizer may divide the copied code and paste in different manner without affecting the final result of the source code. That is changing the order of the program like first line as third or fourth line, fourth line as first or second. Example for near exact match as follows

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

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 4, April 2013

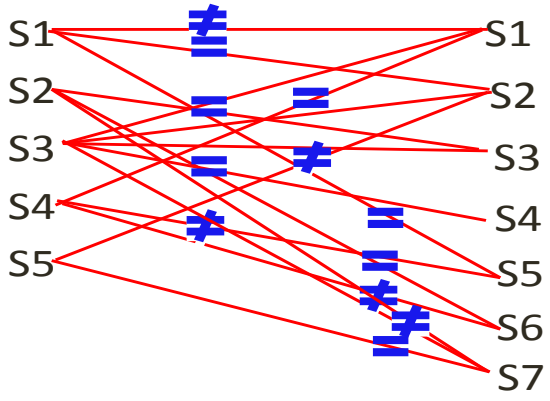
```
1.  i=f=1;
2.  for(i=1;i<=n;i++)
3.  f=f*i;
4.  System.out.println
    (“Factorial”);
5.  System.out.println(f);
```

```
1.  System.out.println(“Factorial”);
2.      i=f=1;
3.  for(i=1;i<=n;i++)
4.  f=f*i;
5.  i=s=1;
6.      for(i=1;i<=n;i++);
7.  System.out.println(f);
```

TABLE2. SAMPLE SOURCE CODE FOR NEAR EXACT MATCH

In the above source code, first and second line is repeated and the fourth line is pasted as first line of other program. This example code contains the “Near exact match” and the “Longest common sequence”. Comparison of this kind of source code as follows.

1. Compare the first node with the other entire nodes until match.
2. Once matched then compare the next node which is in given source code and similar program source code.
3. If mismatch occurred, it has to start comparing from the first node until matches. If first node not matched then second node will be compare until match occurs.
4. Once matched then repeat step2 and step3 until source code ends.




Match List1 – s1, s2, s3, s1, s2, s4, s5
Match List2 – s2, s3, s4, s5, s6, s1, s7

FIGURE1. SAMPLE COMPARISON FOR NEAR EXACT MATCH

Comparison of this source code compares the first node of similar program , not matching so comparing with the next node, it is matched (s1 of left side and s2 of right side), then both the list next nodes are comparing (s2 of left side and s3 of right side) they are matching , then same as before (s3 of left side and s4 of right side) matching, then the next nodes (s4 of left side and s5 of right side) are not matching. Once mismatch occur then algorithm start to compare from the first node to find out the repeated code. So, it is comparing first node of left side with the next node (s5 of right node) matching, then next taking next nodes (s2 of left side and s6 of right side) they are matching, next nodes are (s3 of left side and s7 of right side) not matching. Now s1, s2 and s3 of left side are compared with all the nodes of right side. So taking the next nodes as which are not compared. As per this example taking next node as s4 of left side and s1 of right side matching then taking next node as s5 of left side and compare with not compared nodes. S5 and s7 are matching.

Finally it gives the report as matched node as longest common sequence and repeated node in the given source code and similar program source code as text file.

This algorithm reduces the time of comparison and detect the maximum possible plagiarized or cloned code in the given source code and the similar source code of various programs.

## III. RESULTS

The above method of comparison algorithm is implemented as agent based plagiarism detection using JADE[12] (Java Agent Development Environment) framework.

Developed the component based on multi agent system, because it uses agents with their own actions and behaviors. The main characteristic is to control their own behavior and interact with the environments and other agents. Some properties of agents are

a. The agents are able to decide their own without the human or other interventions.
b. The agents perceive their environments and to respond for the change occurs with them.
c. The agent has initiative and do not act only in response to their environment.

The developed component has the following agents, which helps to perform the task in easy manner.

1. Main agent which helps to get the type of code detection either method level or statement level. Depends on the type of detection it moves to the other agents as
   a. Stmt_Agent gets the source code file name which is going to compare with various similar programs.
   b. Plag_agent used to collect the various similar programs in the collection using size and comments or Meta data of the program.
   c. Detect_agent will compare the given source code with the various similar programs matched with it and produce the result as text file. Which has the filename and its location and similarities between the program
   d. Method_agent gets the file name which is going to compare with the various similar programs. This agent compare the methods with in the given source code.

The developed algorithm will provide a heuristic approach to identify plagiarized code detection in the source code based on Method and statement comparison using ratio threshold between the Methods or statements and efficiently find plagiarism in less time. Some of the key benefits of the system were:

- **Reduce Software Maintenance**

Detection of code that gives the same result, promises decreased software maintenance costs corresponding to the reduction in code size.

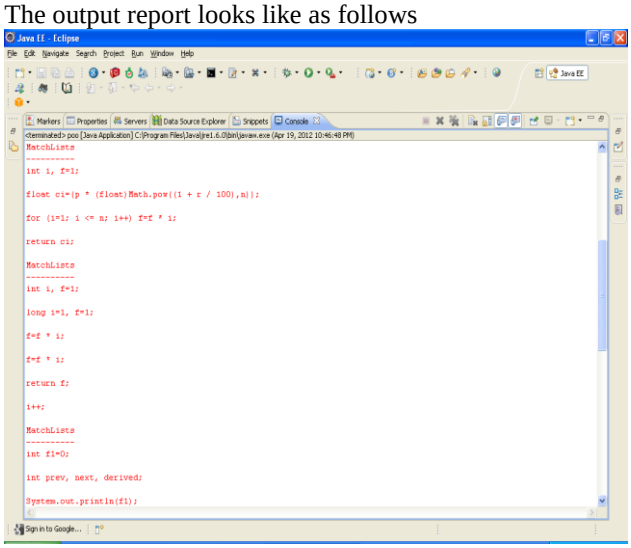
- **Recover Licensing Issues**

Plagiarisms are detected then the code will not get any problem to get copyrights.

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

*International Journal of Advanced Research in Computer and Communication Engineering*
*Vol. 2, Issue 4, April 2013*

The output report looks like as follows



## IV. LITERATURE CITED

This paper is based on the Baojiang Cui [7] algorithm based on rehash classification, which enhances the node storage structure of the syntax tree, and greatly improves the efficiency.

In the progress of traversal, calculate the hash value and the number of child nodes of each node in the syntax tree, and record the start and end line number of each node in the source file. Then store all the nodes into a chained list.

By searching into the syntax tree, find the node with the maximum number N of child nodes, to create an array according to the number N. After the traversal of the chained list set up before, do the first classification, save nodes into the nth position of the array based on its child node number n, so at every position of the array it creates a chained list where stores the same kind of nodes. For instance, if have a node -A, and "A" is the root of n nodes, then save the information of "A" into the chained list at position of Array.

Finally, set up an array Long Array, which has the length of t*N according to the empiric value t. By traversing the linked list Array got from previous step, do the modulus calculation to the hash value of the nodes in Array. We classify the nodes in the chained list into t categories by the modulus value, and the according to the modulus result h (h from 0 to k-1), save the node into the (nt+h) th position of Long-Array [nt+h]. In this way, finish the rehash classification.

Studied some of the Detection algorithms are classified based on the approach and comparison method. They are sequence, finger print, hashing, suffix tree and so.

### *4.1 SIMPLE / SEQUENCE ALGORITHM*

Baxter [2] a used three main algorithms to detect the plagiarisms, they are.

*I. Basic algorithm is to detect sub-tree clones*

*II. Sequence detection algorithm, is concerned with the detection of variable-size sequences of sub-tree clones, and is used essentially to detect statement and declaration sequence clones.*

*III. Complex near-miss clones by attempting to generalize combinations of other clones. The resulting detected clones can then be pretty printed. Clone removal is not carried out.*

### *4.2 FINGER PRINT ALGORITHM*

Michel [4] focuses on exact tree matching retrieval through the use of a good hash function minimizing collisions between false-positives uses suffix tree clone detection on AST node types, our system uses AST node fingerprints that reflect the whole underl ying sub-tree.

A double index is maintained on the fingerprint database: fingerprints are first sorted according by decreasing weight, then by hash value, and also by parent linked node. Implement these indexes using a B+-k-tree.

Since comparing nm sub-trees of m projects of size n (in terms of nodes) for exact equality detection would require $O((nm)^2)$ comparisons with a naive approach, all sub-trees are rather fingerprinted and put in buckets according to their hash value.

### *4.3 BIJECTION / MAPPING ALGORITHM*

Iulian Neamtiu [5] analyzes the bodies of functions of the same name and matches their abstract syntax trees structurally. During this process, compute a bijection between type and variable names in the two program versions. Then use this information to determine what changes have been made to the code. This approach allows us to report a name or type change as single difference, even if it results in multiple changes to the source code.

Traverse the ASTs of the function bodies of the old and new versions in parallel, adding entries to a LocalNameMap and GlobalNameMap to form mappings between local variable names and global variable names, respectively. Two variables are considered equal if we encounter them in the same syntactic position in the two function bodies.

LocalNameMap will help us detect functions which are identical up to a renaming of local and formal variables, and GlobalNameMap is used to detect renamings for global variables and functions.

### *4.4 HASHING ALGORITHM*

Lingxiao Jiang Ghassan [6] algorithm is based on a novel characterization of sub-trees with numerical vectors in the Euclidean space Run and an efficient algorithm to cluster these vectors w.r.t. the Euclidean distance metric. Sub-trees with vectors in one cluster are considered similar. DECKARD is both scalable and accurate. It is also language independent, applicable to any language with a formally specified grammar

The main idea of the algorithm is to compute certain characteristic vectors to approximate structural information within ASTs and then adapt Locality Sensitive Hashing (LSH) to efficiently cluster similar vectors.

### *4.5 STRUCTURE BASED ALGORITHM*

Young-Chul Kim [8] system can check whether or not it is structurally similar or not structurally without regard to

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

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 2, Issue 4, April 2013

modification of the programs source code and can perform a syntax error check.

The value of similarity between two programs is as follows: $0 \leq$ Similarity(program1, program2) $< 1$ Compare strings continues until max-match is bigger than min-length. Set (total match string) is defined as a function that stores all substring which is found in two node string. Set (total_match_string) is defined as a function that stores all match substring. Length(X) is defined as a function showing the length of node string X. Length(X) function which is used for a similarity evaluation is a function which calculates the length of node string. Grouping is performed on assignments which have a high similarity among their programs

*4.6 GREEDY STRING TILING ALGORITHM*

Matt G. Ellis [9] algorithm first, parses each program. Next, for each pair of parse trees, convert each parse tree to a string using some coding method. Using these strings, construct Greedy String Tiling to obtain a metric of similarity. Report this as the similarity between the two programs.

It should be noted that the whole parse tree need not be converted to a string. An intermediate stage in our algorithm could transform the original parse tree into a "degenerate" parse-tree by removing nodes. For example, dropping the nodes dealing with the looping conditions is a for node, replacing the different types of looping constructs (for, while, do, etc) with a general loop construct node may provide better results. These techniques can be used to combat some common attacks.

*4.7 PATTERN MATCHING ALGORITHM*

William S. Evans Christopher [10] structural abstraction prototype is called Asta. Asta accepts a single AST represented as an XML string. It has been used with ASTs created by JavaML from Java code. Asta produces a series of patterns that represent cloned code in a given abstract syntax tree S. It first generates a set of candidate patterns that occur at least twice in S and have at most H holes Asta also generates a pattern called the full cap for v, which is the full sub-tree rooted at v. Asta finds the occurrences of every cap by building an associative array called the clone table, indexed by pattern. Asta performs a greedy version of pattern specialization, called best-pair specialization that attempts to produce large patterns that occur at least twice.

*4.8 SUFFIX TREE ALGORITHM*

Rainer Koschke [11] et al. Using Abstract Syntax Suffix Trees algorithm consists of the following steps:

1. Parse program and generate AST
2. Serialize AST
3. Apply suffix tree detection
4. Decompose resulting cloned token sequence into complete syntactic units

The original suffix tree clone detection is based on tokens. In our application of suffix trees, the AST node type plays the role of a token.

Procedure emit is used to report clones based on the representative. It may filter clones based on various additional criteria such as length, type of clone, syntactic type. (Basic Algorithm)

This provides an efficient clustering mechanism for exact match of sequences of sibling sub-trees. Direct access to indexed ASTs allows further analysis and manipulations to extend neighboring matches into larger near-miss similarities

According to Young-Chul Kim [8] an evaluation algorithm for program similarity and a grouping algorithm for the sake of reducing the count of comparisons. The experiment and estimation proves that a grouping algorithm can reduce a lot of counts of comparison.

Baxter [4] et al. To find clones in the AST, in principal to compare each sub-tree to each other sub-tree in the AST. This approach would not scale, and use a hash function that first partitions the AST into similar sub-trees.

Hash function cannot be perfect (there is an infinite number of possible combinations of AST nodes), it is necessary to compare all sub-trees within the same partition in a second step. This comparison is a tree match, where use an inexact match based on a similarity metric. The similarity metric measures the fraction of common nodes of two trees. Cloned sub-trees that are themselves part of a complete cloned sub-tree are combined to larger clones. Special care is taken of chained nodes that represent sequences in order to find cloned subsequences.

## V. DISCUSSION

The proposed system is based on multi-agent system using Abstract Syntax tree. It is implemented with the help of JADE framework and Eclipse.

*5.1 EVALUATION*

To evaluate the effectiveness of proposed algorithm, collected various similar programs and compared.

Once the source code is converting into the parse file, comparison process is easy cause of the algorithm approach.

Java was used to parse the source code into abstract syntax tree. Each statement of source codes is converted into AST based node and each node contains full information about the statement.

Then the number of node matches is finding based on program level or method level of the source code.

The output file is like report about the statement or method matches in various similar programs.

## CONCLUSION

Today, Plagiarism detection in source code is an active research area. In this paper presents how the plagiarism detection can be handled using the new algorithm based on the AST. The proposed algorithm reduced the time of comparison. It might take minimum O(n) comparison time to detect the plagiarism in source code. It is developed using agent oriented programming, so man power also reduced. Agent can control their own behaviors, actions and communicate with other agents. The component is based on multi-agent system, so it is helpful to control their own behavior and interact with the environment and

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

*International Journal of Advanced Research in Computer and Communication Engineering*
*Vol. 2, Issue 4, April 2013*

other agents. This study may help the plagiarism detection users to detect the plagiarized code.

*5.3FUTURE ENHANCEMENT*

This proposed approach support only for the java based source code the same approach may be used to compare with cross programming language, which is language independent comparison.

This algorithm helps to detect the plagiarism and cloning in source code in effective manner, it might be give false positives. However, still some of the algorithms lacking to avoid the false positives. In future these algorithms may be improved to avoid false positives and detect all type of plagiarism to affect success plagiarism detection using AST. So it may enrich to avoid false positive with efficient manner.

General approaches are using like Meta data to find the similar program or logic of source code, find the similar logic of source code without using Meta data.

## REFERENCES

1. Roy, Chanchal Kumar;Cordy, James R.."*A Survey on Software Clone Detection Research*". School of Computing, Queen's University, Canada. (September 26, 2007)
2. Baxter,I.D, Yahin,A.; Moura, L.; Sant'Anna,M; Bier, L. *"Clone detection using abstract syntax trees"*, International conference on software maintenance 1998.
3. Kevin Greenan, *"Method-Level Code Clone Detection on Transformed Abstract Syntax Trees Using Sequence Matching Algorithms"* University of California - Santa Cruz , 2005
4. Michel Chilowicz, Etienne Duris and Gilles Roussel *"Syntax tree fingerprinting: a foundation for source code similarity detection"*, University Paris-Est.
5. Iulian Neamtiu;Jeffrey S. Foster;Michael Hicks *"Understanding Source Code Evolution Using Abstract Syntax Tree Matching*" MSR '05
6. Lingxiao Jiang Ghassan and St´ephane Glondu *"DECKARD: Scalable and Accurate Tree-based Detection of Code Clones"* Software Engineering, 2007. ICSE 2007
7. Baojiang Cui, Jun Guan, Tao Guo, Lifang Han, Jianxin Wang and Yupeng J *"Code Syntax-Comparison Algorithm based on Type-Redefinition-Preprocessing and Rehash Classification"* , *Journal of Multimedia*, Vol 6, No 4 (2011), 320-328, Aug 2011
8. Young-Chul Kim and Jaeyoung Choi *"A Program Plagiarism Evaluation System"* , ICCSA 2005 Volume 3483, 2005.
9. Matt G. Ellis, Claude W. Anderson *"Plagiarism Detection in Computer Code"* March 23, 2005
10. William S. Evans Christopher W. Fraser Fei Ma *"Clone Detection via Structural Abstraction"* 14th Working Conference on Reverse Engineering (WCRE 2007)
11. Rainer Koschke, Raimar Falke, Pierre Frenzel "Clone Detection Using Abstract Syntax Suffix Trees" 13th Working Conference on Reverse Engineering (WCRE 2006), October 2006.
12. http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf
13. http://jade.tilab.com/doc/administratorsguide.pdf
14. http://www.intechopen.com/books/multi-agent-systems-modeling-control-programming-simulations-and-applications/principles-of-agent-oriented-programming
15. http://www.informatik.uni-freiburg.de/~ki/teaching/ws0910/imap/01_Introduction.pdf