AST in a refactoring; the annotated grammar was more than a request of extent smaller than the generated code, and the overhead of concretizing ASTs was very reasonable.

Andrew Yahin et al [6] proposed Clone detection using abstract syntax trees. A functional system for recognizing close-miss and sequence clones on scale has been introduced. The methodology is taking into account varieties of strategies for compiler common sub expression elimination using hashing. The method is implemented directly by standard parsing technology which identifies clones in arbitrary language constructs, and computes macros that permit evacuation of the clones without influencing the operation of the program. The method is applied to a genuine use of moderate scale, and affirmed past appraisals of clone density of 7-15%, suggesting there is a "manual" software engineering process "redundancy" consistent. Automated methods can recognize and remove such clones, lowering the value of this constant, at concomitant savings in software engineering or maintenance costs. Clone discovery tools additionally have good potential for supporting domain analysis.

Pavitdeep singh et al [7] proposed a software quality tool for measuring the different code metrics for C# source code using Abstract syntax tree. Nfactory libraries are used to generating abstract syntax tree of the source code.

Harjot Singhvirdi and Balraj Singh [8] proposed different types of coupling i.e. static and dynamic coupling. These metrics performed under the different environments and calculate the mean and the standard. The value of standard deviation is useful in judging the representativeness of the mean and quality of software system.

III. PROPOSED APPROACH

The proposed approach consists of various steps from C# source code to syntax tree creation. Once the syntax parse is generated it is resolved to using the Type system to generate the semantic tree, which is further utilized to construct the Software model graph refer to Fig.4.

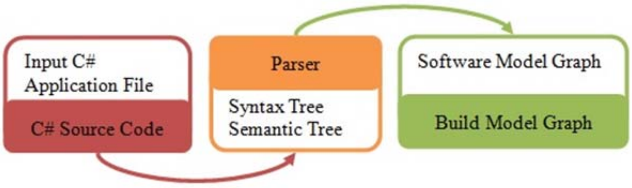

Fig. 4 Approach for constructing Software Model Graph

Algorithm: Constructing a software model graph
Input: C# Source Code
Output: Directed graph (Software Model Graph)
Step 1: Source code (object oriented code) as input
Step 2: Source code samples are passed into the language parse as input.
Step 3: Parser analyses the base class of the syntax tree as the AST (Abstract Syntax Tree)
Step 4: AST method is used to determine the semantics of any node classes within the syntax tree.
Step 5: generating software model graph

a. Class ← node / vertex
b. Relationship ← edge
c. Labeling the node/vertices and placing the edge weights to edges.

Step 6: The output is Directed graph of Software model graph

A. *C# Source Code*

The proposed system at first takes a single file as input and afterward peruses all the tasks inside of that single file (Solution file) and afterwards parses the task files to discover all the source files inside of the activities. During Amid traversal of different files present it will likewise shift through the files which are checked for prohibition during parsing. When all the obliged files are read by the system they are passed to the language parser for syntax tree creation. Fig.5 shows the sample source code example.

```
using System;
using System.Linq;
class Test
{
    public void Main(string[] args)
    {
        Console.WriteLine("Hello, World");
    }
}
```

Fig.5 Source Code Example

B. *Language parser*

1) Syntax Tree: C# source code is just a string. Parsing the string into a syntax tree informs that it is an invocation expression, which has a member reference as target. The syntax tree is shown in Fig.6. A syntax tree doesn't give the complete information regarding object. Some object most in all likelihood is a case technique, and some object seems to be a local variable, parameter or a field of the current class. It might be that some object could be a class name.

Fig. 6 Syntax tree example

2) Semantic Tree: The semantic tree gives the information with respect to these attributes. The semantic tree constructed from the above syntax tree shown in Fig.7.

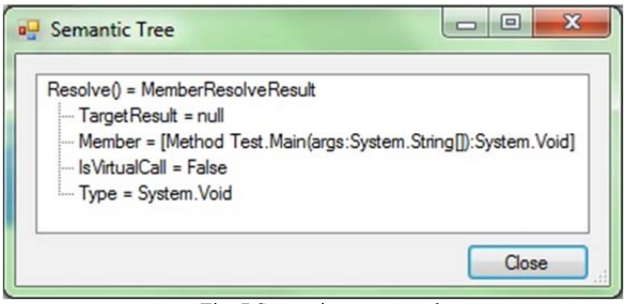

Fig .7 Semantic tree example

C. Software Model Graph

The constructed software model graph represents the high - level view of software architecture as a simple directed and labeled graph (G). The vertices of this graph are classes, abstract classes, and interface classes. The edge of the graph represents directed relations between these entities (node/ vertices).

Graph: Let G = (N, E, L_n, L_e, n, e) be a directed software model graph, where N is a set of vertices, $E \subseteq N \times N$ is a set of edges, L_n is a set of labels for the vertices, L_e is a set of labels for the edges, n: $N \rightarrow L_n$ is a function that assigns a label to the vertices, e: $E \rightarrow L_e$ is a function that assigns a label to the edges.

Relation types in the software model graph are based on UML-like [27] relations. At this point, especially consider class and sequence diagrams of the UML. Moreover, to handle some important relations that is visually hidden in the UML diagrams. For example, if a method of a class has the same signature with a method of the parent class, then there is an "override" relation between these classes that is Number of visually observable in UML class diagrams. We also include some important high-level relations from UML sequence diagrams, such as the "create" and "method call" relations between entities. Possible entity types, relation types and their labels are given in Table 1.

TABLE I LABELS OF NODES AND EDGES IN SMG

Node Label	**Node Type**
C	Class
I	Interface
A	Abstract class
Edge Label	**Relation Type (Edges are directed from A to B)**
X	source class extends target class
I	source class implements target class
A	source class has field type of target class
T	source class uses target class in generic type declaration
L	source class method has a local parameter of target class
P	source class uses target class in its methods parameter
R	source class has methods has been return type of target class
M	source class has method call to target class
F	source class access field of target class
C	source class creates target class
O	source class overrides methods of Class B

The nature of the object oriented design is that, there can be more than one relation between the vertices (classes and interfaces). To build a simple and understandable graph, we collect all of the labels of parallel edges between two vertices into a solitary set of labels, such that 'L_{ij}' is a set of labels of directed edge 'e_{ij}' that contains all relation labels from vertex 'v_i' to vertex 'v_j'. For example, if two entities have both method call {M} and method parameter {P} relations in the same direction, then the combined label set for this edge becomes $\{M\} \cup \{P\} = \{M,P\}$. In our approach for detecting identical design-level clones, the edges are compared during their set of the labels in such a way that, when comparing two non- empty edge label sets, 'L_u' and 'L_n' are considered to be equal if and only if 'L_u' $\subseteq$ 'L_n' and 'L_n' $\subseteq$ 'L_u'. Fig. 1 demonstrates the UML class diagram of an observer design pattern example, and Fig.8 represents the related software graph model that we constructed. Fig.8 shows that the software model graph includes additional information compared to the UML class diagram, such as the "type field" (A) , "method call" (M), "override" (O), "methods parameter" (P), and "extend" (X) relations.

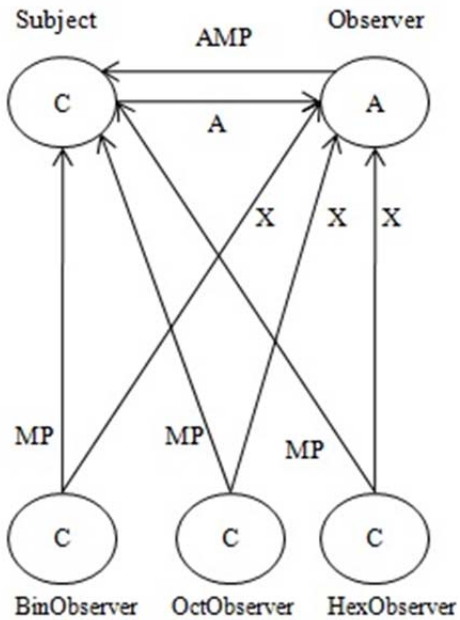

Fig. 8 Software model graph of example observer pattern

D. Degree of Coupling

The degree of coupling [9] is calculated as the ratio of number of message received to the number of message sending. The degree of coupling is given in equation 1. Where DC is degree of coupling, MRC is message received coupling and MSC is message sender coupling. The MRC measures the complexity of the message received by the classes, as MRC is the number of messages received by a class from the other classes. The MSC is the number of message sender coupling among the objects of the classes; it is low level coupling that is achieved through the communication between the components.

$$\text{Degree of Coupling (DC)} = \frac{\text{MRC}}{\text{MSC}} \quad (1)$$

IV. Results and Discussion

The results are analysed from sample source code. The outcome of software model graph is shown in Fig.9 which consists of 7 classes, one abstract class, one interface class and five normal classes. The directed edge weight is referred as communication message between the various classes (nodes/ vertices) as described in Table I. Fig.10 show common design structures that are identified manually from Fig. 9(C). Fig.11a) gives information about BinObserver class. Here BinObserver class is communicating to subject class with two different message, one is method (M) other is parameter (P) and to observer class with Extend (X). It performed coupling is MSC. Similarly rest of the classes shown in Fig. 11 (b) (c) (d) (e). In Fig. 11(d) communication of MSC there is multiple labels (AMP) on edge which refers three different communications are performing. Table II describes about the total numbers of MSC and MRC of various classes and their degree of coupling.

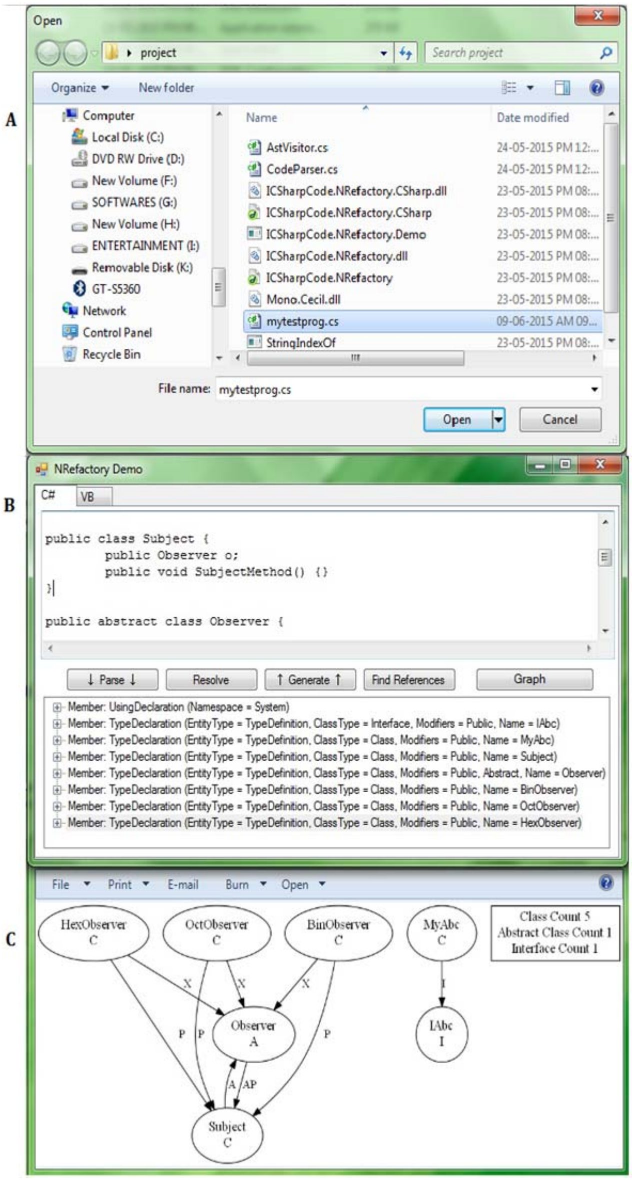

Fig. 9(A) Selection of C# source file (B) Abstract Syntax tree, Semantic (C) Software model graph

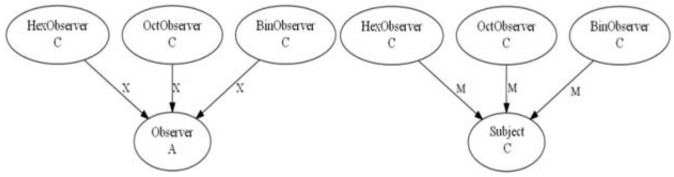

Fig. 10 Common design structures

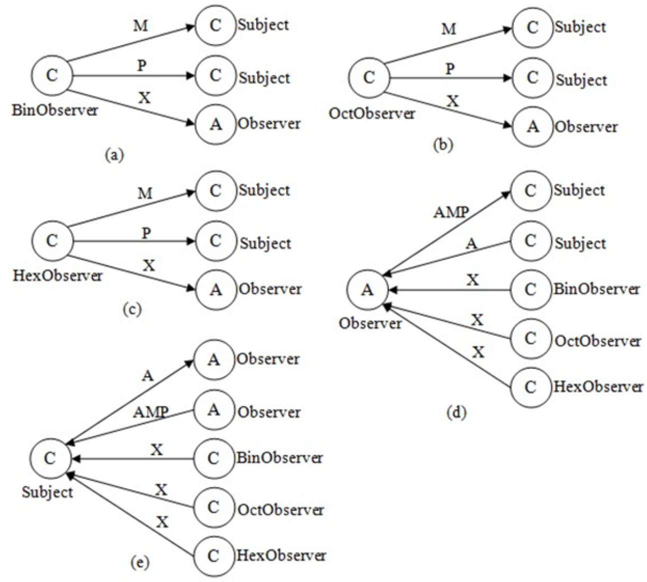

Fig. 11 Degree of Coupling

TABLE III

COUPLING METRICS

Class	Object Oriented		
	MSC	MRC	DC
BinObserver	3	0	0/3
OctObserver	3	0	0/3
HexObserver	3	0	0/3
Observer	3	4	4/3= 1.3
Subject	3	4	4/3= 1.3

V. Conclusion and Future Work

In this paper an approach is proposed to generate a software model graph and to analyse the source code using abstract syntax tree method. In suggested approach, the solution file for the C# application is taken as an input to the system. It loads the file into memory and reads one by one to load all the source code in order to construct the syntax tree. Once the creation of abstract syntax tree is completed, it is ready for analysis. The Nfactor library [10] is utilized to generate syntax tree. Generated syntax tree is used for finding refactoring of similar source code and for finding a patterns design. The construction of software model graph provides in-depth information about a system. Based on SMG, common design structures, coupling and substructures can be found in result section. These structured graphs can help develop in understanding the architecture of the object oriented system.

The future enhancement for this work is to apply graph partitioning technique on software model graph to identify specific-domain structures, commonly used design structures; copy-past activity and design patterns of object oriented systems. These structures can assist for software developers to improve the quality and design of the software.

REFERENCES

[1] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-Miner: Finding copy–paste and related bugs in large scale software code," *IEEE Trans. Softw. Eng*, vol. 32, pp. 172-192, 2006.

[2] P. Gandhi and P. K. Bhatia, "Optimization of Object- Oriented Design Using Coupling Metrics," *International Journal of Computer Applications*, Vol. 27(10), 2011, pp. 41-44.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[4] Vipin Saxena and Santosh Kumar, "Impact of Coupling and Cohesion in Object-Oriented Technology," *Journal of Software Engineering and Applications,* vol. 5, pp. 671-676, 2012.

[5] Jeffrey L. Overbey and Ralph E. Johnson, "Generating Rewritable Abstract Syntax Trees," *Software Language Engineering, Lecture Notes in Computer Scienc*. Springer- Verlag Berlin Heidelberg, vol. 5452, pp 114-133, 2009.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance*, 1998, IEEE Computer Society, Washington, DC, USA , p. 368.

[7] Pavitdeep Singh, Satwinder Singh, and Jatinder Kaur, "Tool for generating code metrics for C# source code using abstract syntax tree technique." *SIGSOFT Softw. Eng*. Notes 38, vol. 5, p.1-6, August 2013. DOI=10.1145/2507288.2507312 http://doi.acm.org/10.1145/2507288.2507312

[8] Harjot Singh Virdi and Balraj Singh, "Study of the Different Types of Coupling Present in the Software Code," *International Journal of computer Science and Information Technology,* vol. 3(3), pp. 4153-4156, 2012.

[9] Jehad Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and software Technology,* vol. 58, pp 231-249, 2015.

[10] Daniel Grunwald, Using NRefactory for analyzing C# code (http://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code)