Dynamic Graph Based Slicing for Object-oriented Programs

Swatee Rekha Mohanty[#1], Prafulla Kumar Behera[*2], Durga Prasad Mohapatra[#3]

[#]*PhD Scholar, Utkal University, Vani Vihar, Bhubaneswar, Odisha, India, Pin-751004*

[#]*Reader, Dept. of CSA, Utkal University, Vani Vihar, Bhubaneswar, Odisha, India, Pin-751004*

[#]*Associate Professor, Dept. of CSE, National Institute of Technology, Rourkela, Odisha, India, Pin-769008*

Abstract: *This paper proposes a Dynamic Graph (DG) traversal slicing algorithm for computing dynamic slices of object-oriented programs in presence of inheritance. The computed dynamic slice can facilitate various software engineering activities like program comprehension, testing, debugging, reverse engineering maintenance etc. This paper first proposes an intermediate program Dynamic Graph (DG) to represent the execution trace of an object-oriented program. Then the proposed slicing algorithm is applied on the intermediate program representation to compute the dynamic slices. The advantage of this approach is that, the intermediate program representation is manageable as it is created on the execution trace, hence needs less memory to store and less time to traverse. The proposed algorithm is space as well as time efficient and computes precise dynamic slices.*

Keywords: *Dynamic slice, Dynamic Graph, Object-oriented program, System Dependence Graph, Execution Trace.*

I. INTRODUCTION

Slicing is a program analysis technique. This concept was originally developed by Mark Weiser [1].Slicing has a remarkable contribution to the field of software engineering, as it facilitates various software engineering activities, as, program comprehension, testing, debugging, maintenance and reverse engineering etc. Generally, program slices can be computed with respect to the slicing criterion <S, V >, Where S is the statement number and V is the set of variables used or defined at S. Object-oriented programs are enriched with many additional features to best represent and implement the real world problems.

Object-oriented technique modularizes the programs, but at the same time it is very complex and difficult to debug, test and maintain those products. Slicing technique extracts the set of statements program, which is relevant to a particular computation. Such strategies are usually called filtering techniques. The most important filtering technique is program slicing [2]. This paper computes the dynamic slices of object-oriented programs in presence of inheritance.

Inheritance is one of the features amongst those offered by object-oriented programming languages. This is the way to promote reusability, where the features of the super classes can be extended to the newly created subclasses. It may introduce difficulties to get a suitable representation of the object-oriented programs that possess inheritance. This paper focuses on the above cited issue and proposes the best way to represent the object-oriented program having the concepts of inheritance through an intermediate program representation called dynamic graph (DG). Dynamic graph is designed to represent the execution trace of the object-oriented program. We are concerned with the dynamic slice, which contains the set of statements that actually affects the slicing criterion for a particular execution of the program. The execution trace plays a vital role as it is the set of statements those are actually executed for a specific input to the program.

After the construction of dynamic graph, we apply a dynamic graph traversal slicing algorithm to compute the dynamic slices of object-oriented programs. This found to be more precise and correct.

The rest of the paper is organized as follows. An exclusive review of literature is presented in Section 2. Some basic concepts and definitions are discussed in Section 3. Section 4 presents the proposed algorithm for computing dynamic slices of an object oriented program in presence of inheritance. We also discuss the working of the proposed algorithm along with the correctness and complexity analysis in this section. The implementation of the proposed algorithm and tool architecture is presented in Section 5. In this section the proposed tool architecture is discussed. The comparison of the proposed work with the existing approaches is discussed in Section 6 followed by some of the limitation of our proposed algorithm. Section 7concludes the paper and presents the future work.

II. LITERATURE REVIEW

Mark Weiser [1] was the first to introduce the concepts of Program slicing. According to him, slicing is a method of decomposing a program based on the data and control dependence analysis. He introduced the concept of approximate slicing and stated that the slice must be a small executable program.

Ottenstein et al. [3] used Program Dependence Graph (PDG) to capture the data and control dependence presents within a single procedure. They stated that many software engineering applications can be made in an optimized way by traversing the

Program Dependence Graph. Their slice can be used specifically in program transformation.

Horwitz et al. [4] introduced System dependence Graph (SDG) for representing program that spans multiple procedures. They proposed a better way to find the slices in case of the inter-procedural programs. In their representation, they introduced many new edges like call edge, parameter- in, parameter- out and summary edges etc.

Mund et al. [7] extended their approach of computing dynamic slices of intra-procedural program to compute the dynamic slices of inter-procedural program. For intermediate representation of the program, they have used the Control Flow Graph (CFG). In their representation they included the inter-procedural calls. They claimed that their approach was efficient than that of the existing approaches.

Larsen and Harrold [9] introduced the concept of Class Dependence Graph (ClDG) to represent the classes in object-oriented programs. They have introduced some new edges like class member edge, inheritance edge etc. They have correctly represented the object-oriented features like classes, inheritance, polymorphism, message passing etc in their proposed graph. Ultimately, they constructed System Dependence Graph (SDG) for the whole object-oriented program. After the SDG is being constructed they applied the two pass graph traversal algorithm [4] for computing the dynamic slices of object-oriented programs.

Wang et al. [16] described the utility of the program slicing in various software engineering applications. At the same time they stated that the computed slices were too large and bit difficult to inspect by the software engineer. So to have a manageable dynamic slice they divided the execution trace of the program under consideration into several phases. They have generated intermediate program representation of individual phases by considering the data and control dependencies. Then they applied the dynamic slicing algorithm at those different levels to compute the dynamic slice.

Korpi et al. [17] stated that slicing has not widely been applied in software engineering field. They did a survey by considering 12 dynamic program slicer. From their survey they identified many issues, which should be the concern and should consider for improvement. Some of these issues are: limitations regarding supported programming languages, virtualization and navigation features and limitation of empirical studies. They claimed that paying attention to these issues will help in faster scientific progress and will help in more practical implementation of program slicing in the field of software engineering.

Treffer et al. [18] introduces the Abstract Slicing. They stated that abstract slicing is an efficient technique, which helps in understanding the interaction between the different parts of the program. They also explained that dynamic slices are more precise as they used the run time information for their construction.

Chebaro et al. [19] proposed a technique to reduce the source code by program slicing before test generation took place. In this paper they presented optimized and adaptive usages of program slicing. They proposed an algorithm which was implemented in a tool named as SANTE (Static ANalysis and TEsting). They did program simplification to easy detect and analysis of error.

III. BASIC CONCEPTS AND DEFINITIONS

This section is organized into two parts. The first part describes the intermediate program representations and the second part describes the basic concepts and definitions used in the proposed algorithm.

A. Intermediate program representations

This paper introduces a new intermediate program representation named dynamic graph (DG). This is exclusively based on the execution trace of the program. Execution trace is the set of those statements that are executed when the program starts execution after getting a specific input value from the user. The idea behind constructing such intermediate program representations is that, we found in literature many intermediate program representations proposed by the researchers are based on the programs under consideration and they are quite unmanageable due to the size of the programs. Hence, this paper tries to minimize the nodes in the intermediate program representation by focusing on the execution trace, as the dynamic slice contains the statements that belong to the execution trace.

Program Dependence Graph: Ferrante et al. [14] proposed Program Dependence Graph (PDG) to represent the intra-procedural programs. Program dependence graph represents the data dependencies and control dependencies within the statements of the programs. But PDG, is unable to represent the inter-procedural dependencies.

System Dependence Graph: Horwitz et al. [4] proposed System Dependence Graph (SDG) to represent the inter-procedural dependencies of a program that span multiple procedures. SDG can correctly represent inter-procedural programs by adding some additional edges like call edge, parameter-in and parameter-out edges, summary edge etc.

Class Dependence Graph: Larsen and Harrold [9] extended the System Dependence Graph [4] to represent the object-oriented features. They introduced a new representation called Class Dependence Graph (ClDG) to represent the classes in an object-oriented program. They have introduced a new edge called class membership edge to connect the class entry node with the class members. They have nicely represented the object-oriented features like inheritance, dynamic binding and message passing etc. in their approach.

Dynamic Graph: To represent the execution trace of the object-oriented programs in presence of inheritance, this paper introduces dynamic graph (DG) as the intermediate program representation. DG is a subset of SDG. DG can be defined as Gh = (Nd, Ed). Where Nd is set of nodes and Ed is the set of edges in the dynamic graph .Nd = {n |n represents the statements of a program}, Ed = {e |e represents data and/or control dependencies between ni and nj where (ni, nj) ϵ Nd}.

Dynamic graph (DG) basically represents two types of dependencies. Those are data dependencies and control dependencies. Dynamic graph replaces the call edge (an edge between the method call nodes to the method entry node) by control dependence edge. Objective of doing so is that, when a node calls a method, ultimately the control is transferred from the calling node to the called node.

Algorithm for constructing Dynamic Graph: This section presents an algorithm for constructing the dynamic graph. In this algorithm, n represents the number of statements in the execution trace.

Gh represents a two dimensional array containing the dependency information within the statements of the execution trace. During the implementation, 0 represents that there is no dependence between the nodes, 1 represents data dependence between the nodes and 2 represents a control dependence between the nodes.

Consider the example program given in Fig.1.

The example program given in Fig. 1 defines four different classes. They are class person, student, teacher and InheritanceDemo. InheritanceDemo contains the main() methods and it creates the objects of other classes within it. We have considered this program as it nicely implements the concepts of inheritance, message passing and method overloading (method polymorphism). Class person is the base class, student and teacher are the derived classes.

Let our program runs with the input value i=1. We will have the execution trace as shown in Fig. 2.

As we have executed the program with the input set i=1, the else part of the if construct have executed.

We proposed our idea of constructing dynamic graph (DG) instead of constructing the System Dependence Graph (SDG) to represent the execution trace of our example program under consideration. As mentioned earlier, the matrix represents the dependence information of the execution trace which is given in Table 1.

Table 1. Dependence information for the execution trace of the example program given in Fig.1.

	Dependence information for the execution trace of the example program given in Fig.1													
	S2	S3	SME4	S5	S6	ME7	S8	S9	MME39	S40	S41	S42	SMC43	MC44
S2	0	0	0	1	0	0	1	0	0	0	0	0	0	0
S3	0	0	0	0	1	0	0	1	0	0	0	0	0	0
SME4	0	0	0	2	2	0	0	0	0	0	0	0	0	0
S5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ME7	0	0	0	0	0	0	2	2	0	0	0	0	0	0
S8	0	0	0	0	0	0	0	0	0		0	0		0
S9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MME39	0	0	0	0	0	0	0	0	0	2	2	2	0	0
S40	0	0	0	0	0	0	0	0	0	0	1	1	0	0
S41	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S42	0	0	0	0	0	0	0	0	0	0	0	0	2	2
SMC43	0	0	2	0	0	0	0	0	0	0	0	0	0	1
MC44	0	0	0	0	0	2	0	0	0	0	0	0	0	0

Below, we present our algorithm for constructing the dynamic graph (DG) in pseudo code.

Algorithm for DG Construction

Input: Execution trace
Output: Dynamic Graph

1. For each statement of the execution trace draw a circle, representing the node
2. For i= 1 to n
3. For j= 1 to n
4. If Gh[j] data dependence on Gh[i]
5. Add data dependence edge from Gh[i] to Gh[j]
6. ElseIf Gh[j] control dependence on Gh[i]
7. Add control dependence edge from Gh[i] to Gh[j]
8. EndIf
9. EndIf
10. EndFor
11. EndFor

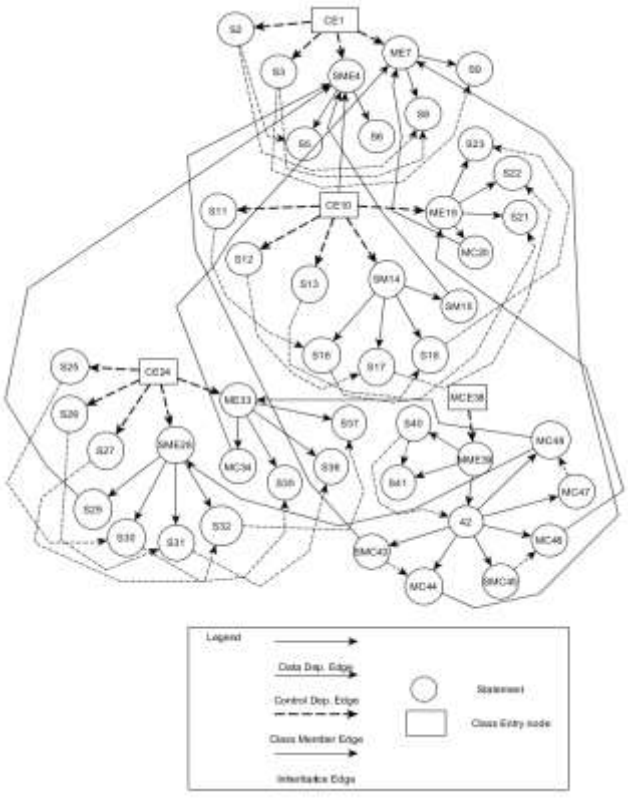

Fig. 3. SDG of the example program given in Fig.1

Description of the DG Construction algorithm: First we draw circles to represent the statements of the execution trace. Then, we verify various dependencies like data dependence and control dependence of a particular node with rest of the node. If node Gh[j] data dependent on Gh[i], we add data dependence edge we add data dependence edge from Gh[i] to Gh[j]. If node Gh[j] control dependent on Gh[i], we add control dependence edge from Gh[i] to Gh[j].

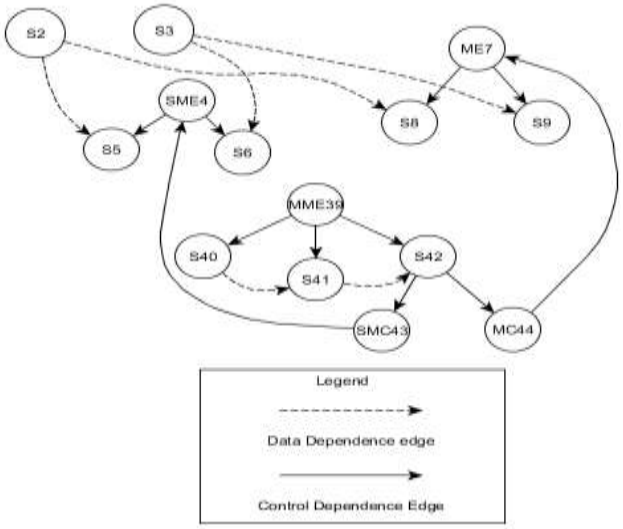

Fig 4. DG of the example program given in Fig. 1.

B. Basic concepts and definitions

This section presents some definitions used in the proposed algorithm. In the rest of the paper, we use the terminologies statement, node and vertex interchangeably.

Control Dependence: Let x and y are two different nodes in the SDG. Node y depends on x if there is a directed path from x to y, y post-dominates every node z on directed path D, excluding node x and y and y does not post-dominate x. In the example program nodeSMC43 is control dependent on node S42.

Data Dependence: Let x and y are two nodes in the system dependence graph. Then node y is data dependence on node x, if a variable var defined at x is used at y. There is a directed path exist from x to y along which there is no intervening definition of var. In the example program node S42 is data dependent on node S40.

Def(var): Let var be a variable in a program P. Then a node u is said to be Def(var) node if node u defines variable var. In our example program Def(pObj) = SMC43.

Use(var): Let var be a variable in the program P . Then a node u is said to be Use(var) node, if node u uses the variable var. Use(pObj) = MC44.

DefVarSet(u): Let u and var be the node and variable ,respectively. Then, DefVarSet(u) = {var|var is a variable of the program P and u is a Def(var) node}.DefVarSet(SMC43) = pObj.

UseVarSet(u): Let var be a variable of a program P.And u be a node. Then UseVarSet(u) = {var | var is a variable of the program P and u is a Use(Var) node}.UseVarSet(MC44) = pObj.

Let the example program be executed with the input value i = 1. According to the predicate condition, object of class Person is created and it called the display( ) method. The execution trace for the input i = 1, is shown in Fig. 2. The dynamic graph (DG) of the example program with respect to the execution trace given in Fig. 2 is shown in Fig 4.

ActiveDataSlice: Let P be a program and var be a variable. Before execution of the program P, ActiveDataSlice(var) = ϕ. Let u be a def(var) node and UseVarSet(u) = var1,var2, ...,varK. Let programP be executed with a given set of input values. Then,ActiveDataSlice(var) = u U ActiveDataSlice(var1) U …U ActiveDataSlice(vark) U ActiveDataSlice(vart) U ActiveControlSlice(t), where t is the most recently executed node.

ActiveControlSlice: Let s be the test node in the SDG of the program P and UseVarSet(s) = {var1,var2,...,varK}. Before execution of the program P, ActiveControlSlice = ϕ. After each execution of the node s in an actual run of the program,ActiveControlSlice(s) = {s} U ActiveDataSlice(var1) U....U ActiveDataSlice(vark) U ActiveDataSlice(vart) U ActiveControlSlice(t) where t is the most recently executed predicate node.

DyanSlice(s, var): Let s be a node in the program P, and the variable var be in set DefVarSet(s) U UseVarSet(s).Before execution of the program P DyanSlice(s,var)= ϕ. For each execution of the statement s, DyanSlice(s,var) = ActiveDataSlice(var) U ActiveControlSlice(t), where t is the most recently executed predicate node of s.

ActiveCallSlice: For a call node u, ActiveCallSlice(ucall) = ActiceDataSlice(var) U ActiveControlSlice(ucall),where var is the variable or object used to call the method.

DS: During the implementation of the proposed algorithm we have used a one dimensional array named as DS.

IV. PROPOSED ALGORITHM

This section proposes the *DG traversal slicing algorithm* to compute the dynamic slices of object-oriented programs in presence of inheritance.

The proposed algorithm is given below.

Algorithm- DG traversal slicing algorithm :

Input: An object-oriented program
Output: Dynamic slice

1) Consider the program P.

2) ***Initialization:*** Before execution of the program do the followings:

a) For each node u of the program P do the followings:

If u is a predicate node, then set Active Control Slice(u) = ϕ.

b) For every variable var of the program P, set ActiveDataSlice(var) = ϕ.

c)For each variable var ϵ DefVarSet(u) U UseVarSet(u), set DyanSlice(u, var) = ϕ

d) For each call node uof the program P Set ActiveCallSlice(u)= ϕ.

3) Run the program P with given set of input values.

4) Get the execution trace of the program with respect to the given input values and then construct the dynamic graph (DG).

5) ***Dynamic slice computation:*** Enter the slicing node u, Slicing node is the node, on which the slice has to computed.

a) If u is a Def(var) node and not a call node, then DyanSlice(u, var) = ActiveDataSlice(var). Compute ActiveDataSlice(var) by traversing DG through the incoming data dependence edges and list the reached nodes in DS.

b) If u is a call node, DyanSlice(u,var) = ActiveCallSlice(u). Compute ActiveCallSlice(u) by traversing DG through the outgoing control dependence edges and incoming data dependence edges and list the reached nodes in DS.

c) If u is a test node,DyanSlice(u,var) = ActiveControlSlice(u). Compute ActiveControlSlice(u) by traversing DG through all incoming control dependence edges and incoming data dependence edges and list the reachednodes in DS.

d) If u is a Def(var) and Use(var) node, DyanSlice(u,var) = ActiveDataSlice(var) U ActiveControlSlice(t),where t is the most recent executed predicate node.

6) ***Slice look up:***

Extract the nodes of DS, which are reached during the traversal of the dynamic graph. Those nodes will constitute the dynamic slice.

Description of the Algorithm:

This section explains our proposed algorithm. At first an object-oriented program is being executed with a given set of input values, and the execution trace is found out. Based on the execution trace, the dynamic graph (DG) is constructed. Then the slicing node is fetched to compute the dynamic slice at a particular node. Now four different conditions may arise.

If the slicing node (which is a node in the dynamic graph) is a Def(var) node, then ActiveDataSlice(var) is computed. This is computed by traversing the dynamic graph (DG) through the incoming data dependence edges and the reached nodes are listed. If the slicing node is a call node, ActiveCallSlice(u) is computed by traversing DG through the outgoing control dependences edges and incoming data dependence edges and the reached nodes are listed. If the slicing node is a test node, then the ActiveControlSlice(u) is computed by traversing DG through all incoming control dependence edges and incoming data dependence edges and the reached nodes are listed. If the slicing node is a Def(var) and Use(var) node, then ActiveDataSlice(var) U ActiveControlSlice(t) is computed.

Working of the proposed algorithm:

In this section, we explain the working of our proposed algorithm by taking the example program given in Fig. 1. The example program runs with input i = 1. The slicing node is <SMC43>.The node SMC43 is a call node. So, Activecallslice (SMC43) is computed by traversing through the outgoing control dependence edges and incoming data dependence edges and the reached nodes are listed.

```
CE1 class Person Author
{
S2 String FirstName;
S3 String LastName;
SME4 Person(String fName, String lName)
{
S5 FirstName = fName;
S6 LastName = lName;
}
ME7 void Display()
{
S8 System.out.println("First Name : " + FirstName);
S9 System.out.println("Last Name : " + LastName);
}
}
CE10 class Student extends Person
{
S11 int id;
S12 String standard;
S13 String instructor;
SME14 Student(String fName, String lName, int nId, String stnd, String instr)
{
SMC15 super(fName,lName);
S16 id = nId;
S17 standard = stnd;
S18 instructor = instr;
}
ME19 void Display()
{
MC20 super.Display();
S22 System.out.println("Standard : " + standard);
S23 System.out.println("Instructor: " + instructor);
}
}
CE24 class Teacher extends Person
{
S25 String mainSubject;
S26 int salary;
S27 String type; //Primary or Secondary School teacher
SME28 Teacher(String fName, String lName, String sub, int slry, String sType)
{
S29 super(fName,lName);
```

```
S30 mainSubject = sub;
S31 salary = slry;
S32 type = sType;
}
ME33 void Display()
{
MC34 super.Display();
S35 System.out.println("MainSubject : " + mainSubject);
S36 System.out.println("Salary : " + salary);
S37 System.out.println("Type:"+type);
}
}
MCE38 class InheritanceDemo
{
MME39 public static void main(String args[])
{
S40 int i;
S41 i=Integer.parseInt(in.readLine());
S42 if(i==2)
{
SMC43 Person pObj = new Person("Rayan","Miller");
MC44 pObj.Display();
}
Else
{
SMC45Student sObj = new Student("Jacob","Smith",1,"1 - B","Roma");
MC46 sObj.Display();
SMC47 Teacher tObj = new Teacher("Daniel","Martin","English","6000","Primary Teacher");
MC48 tObj.Display();
} } }
```

Fig. 1.An example program

```
CE1 class Person
{
S2 String FirstName;
S3 String LastName;
SME4 Person(String fName, String lName)
{
S5 FirstName = fName;
S6 LastName = lName;
}
ME7 void Display()
{
S8 System.out.println("First Name : " + FirstName);
S9 System.out.println("Last Name : " + LastName);
}
}
CE10 class Student extends Person
{
S11 int id;
S12 String standard;
S13 String instructor;
SME14 Student(String fName, String lName, int nId, String stnd, String instr)
{
SMC15 super(fName,lName);
S16 id = nId;
S17 standard = stnd;
S18 instructor = instr;
}
ME19 void Display()
{
MC20 super.Display();
S22 System.out.println("Standard : " + standard);
S23 System.out.println("Instructor: " + instructor);
}
}
CE24 class Teacher extends Person
{
S25 String mainSubject;
S26 int salary;
S27 String type; //Primary or Secondary School teacher
SME28 Teacher(String fName, String lName, String sub, int slry, String sType)
{
S29 super(fName,lName);
S30 mainSubject = sub;
S31 salary = slry;
S32 type = sType;
}
ME33 void Display()
{
MC34 super.Display();
S35 System.out.println("MainSubject : " + mainSubject);
S36 System.out.println("Salary : " + salary);
S37 System.out.println("Type:"+type);
}
}
MCE38 class InheritanceDemo
{
MME39 public static void main(String args[])
{
S40 int i;
S41 i=Integer.parseInt(in.readLine());
S42 if(i==2)
Else
{
SMC45Student sObj = new Student("Jacob","Smith",1,"1 - B","Roma");
MC46 sObj.Display();
SMC47 Teacher tObj = new Teacher("Daniel","Martin","English","6000","Primary Teacher");
MC48 tObj.Display();
}
}
}
```

Fig.2 Execution trace of the example program given in Fig.1

Table 2 also shows the slice for node MC44, SME4, ME7 S41 and S42.

Table 2. Dynamic slices of the example program at different statements

Slice computation at different nodes		
Slice Node	**Types of Node**	**Dynamic Slice**
SMC43	Call Node	SMC43, SME4, S5, S6, S2, S3
MC44	Call Node	MC44, ME7, S8, S9, S2, S3, SMC43, SME4, S5, S6
SME4	Call Node	SME4, S5, S6, S2, S3
ME7	Call Node	ME7, S8, S9, S2, S3
S41	Def(var)node	S41, S40
S42	Test node	MME39, S41, S42

Correctness of DG traversal slicing algorithm:

This section presents the correctness proof of our algorithm.

Theorem 4.1:

Our algorithm computes correct and precise dynamic slices for a given slicing criterion.

Proof:

Our proposed algorithm works on dynamic graph, which is constructed on the execution trace of an object-oriented program. Dynamic graph is an arc classified graph represented as follows:

Gh = (Nd, Ed), where *Nd* is set of nodes and Ed is the set of edges in the dynamic graph. *Nd* = {*n* | *n* represents the statements of a program}, *Ed* = {*e* | *e* represents data and/or control dependencies between *ni* and *nj*, where *(ni , nj) ϵ Nd* }.

Here, | *Nd* | is finite; hence our algorithm will execute for finite number of times and terminate safely without entering into an infinite loop.

Now, we will prove the correctness of our algorithm. Our algorithm works on the control and data dependence analysis among the nodes in the dynamic graph. During the traversal of the dynamic graph, it lists those nodes in the dynamic slice which really affects the slicing node. Let us consider that dynamic graph has only one statement *s1*. The computed dynamic slice for *s1* will contain only the statement *s1*. Then the dynamic slice for *s1* is correct. Suppose there are two statements *s1* and *s2* in the execution trace. Than the dynamic slice with respect to s2 will contain *s1* and *s2* if there is a control and/or data dependence between the statements *s2* and *s1*. Otherwise it will contain statement *s2*. So, dynamic slice at statement *s2* is correct. Let us assume that the dynamic slice of all the statements before statement *su* is correct. It can be proved that the dynamic slice at statement *su* must contain the statements that actually affect the slicing node i. e. statement *su.*

Hence this proves that our algorithm computes correct and precise slice for every statement of the execution trace.

Complexity Analysis of DG traversal slicing algorithm`

This section presents the complexity of our proposed algorithm in terms of space and time.

Space complexity analysis:

Our algorithm works on dynamic graph which contains n numbers of nodes. If the execution trace contains n numbers of statements, then the space complexity of our proposed algorithm is $O(n^2)$. Further, we have used a one dimensional array, named DS which contains the nodes in the dynamic slice list during the traversal of the dynamic graph. To store those nodes, we require at most $O(n)$ space if there are n number of nodes in our dynamic graph.

So the space complexity of our DG traversal slicing algorithm is $O(n^2)$.

Time complexity analysis :

Let n numbers of statements are there in the execution trace of our program under consideration. $O(n^2)$ time is required to construct the dynamic graph. For traversal of the dynamic graph, the time required is $O(En)$, where E is the total numbers of edges between n number of nodes. Substituting the value of E, we get the time complexity as $O((n-1)n)$, which is $O(n^2)$. So, total time complexity is $O(n^2) + O(n^2) = 2O(n^2)$. Ignoring the constant term, we have the time complexity of our proposed algorithm is found to be $O(n^2)$.

V. IMPLEMENTATION OF THE PROPOSED ALGORITHM

In this section we discuss the architecture of our tool which will implement our algorithm and then we discuss some of the experimental results during implementing our algorithm.

A. Architecture of the proposed tool

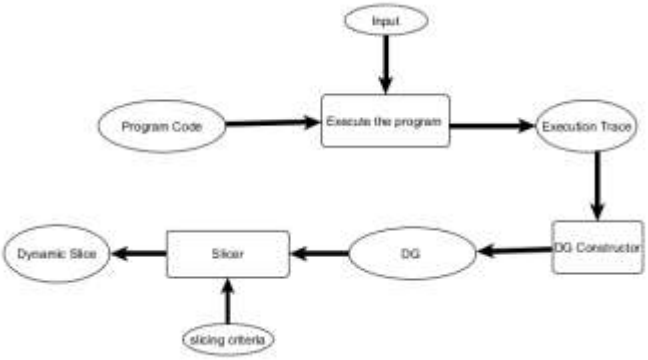

Fig 5. Architecture of the proposed **tool DSOOP**

We have developed a tool in Java for implementing our proposed algorithm. We have named our tool *"Dynamic Slicer for Object-Oriented Programs (DSOOP)"*.

In this section, we discuss the tool architecture of our proposed tool *DSOOP*, which can be used to compute the dynamic slice of object-oriented programs. Fig 5 shows the architecture of our proposed tool *DSOOP*.

Our proposed tool *DSOOP* takes an object-oriented program written in Java as an input. Then the program runs with a set of input values and the execution trace is found out. Now based on the execution trace of the program, the *DG* constructor constructs the *dynamic graph (DG).* Then a slicer component computes the dynamic slice by traversing the dynamic graph provided with a slicing node. Basically, slicing node is a node on which the dynamic slice has to be computed.

B. Experimental Results

The proposed DG traversal slicing algorithm is implemented using Java. During the implementation, several object-oriented programs are considered for computing the dynamic slices. It is observed that the proposed algorithm works efficiently and computes correct and precise dynamic slices. The proposed algorithm do not traverse the whole System Dependence Graph (SDG), rather it traverses the dynamic graph (DG) only, which is constructed by taking into consideration the dependence information within the statements of the execution trace of the program under consideration

The experimental results of our implementation are shown in Table 3.

Table 3. Experimental results

Sl No	Prog Name	Prog. Size (# stmts)	Execution Trace Size (# stmts)	No. of dependences	Avg slice computation time(in milli Sec)
1	*Java program to print student details*	45	14	17	3.5013
2	*Emp. detail*	81	30	17	3.5615
3	*Implementing sorted double linked list*	210	150	119	21.0120
4	*Implement Graham Scan algorithm*	320	200	120	21.0121
5	*Queue Imp*	400	300	300	52.6710
6	Tree implementation	500	400	300	52.6710
7	*Pubman Game*	600	550	500	87.785
8	*Tic-tac-toe Game*	750	600	500	87.785
9.	*Snake Game*	800	600	550	96.5635
10	*Library management system*	1000	800	600	105.342

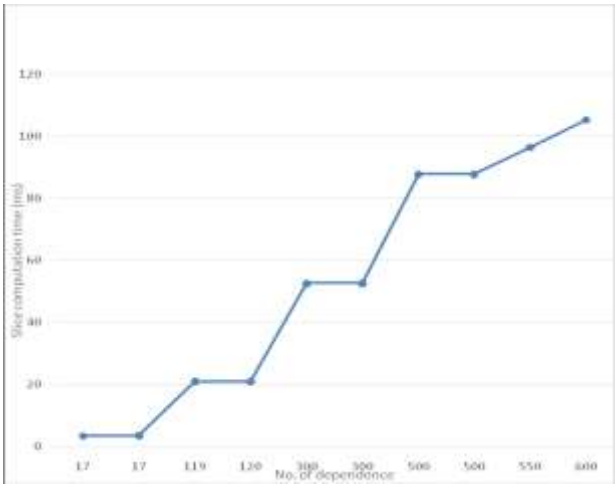

Fig. 6. Graph showing No. of Dep.Vs Dynamic Slice Computation Time in Milli Sec.

In Fig 6, we have shown a comparison between the number of dependence and the slice computation time in milli second. During the implementation, it is observed that the time for computing the dynamic slices is independent of the number of nodes in the execution trace. However the slice computation time increases when the number of dependencies increases among the nodes in the execution trace.

VI. COMPARISON WITH RELATED WORKS

This work has been influenced with the work done by Mund et al. [7]. Mund et al. [7] have not considered data dependence in their work. They have considered only the control dependence in their intermediate program representation. But we have considered data dependence along with the control dependence in our intermediate program representation. So our proposed intermediate program representation is a better way to represent the dependencies present in an object-oriented program.

Jain et al. [12] proposed d-u chain which will be large for the large program. As this paper computes dynamic slice based on the execution trace of the program under consideration, thus the slice generated is more precise and correct.

Du et al. [11] used System Dependence Graph as the intermediate program representation to compute the dynamic slices of a program. This paper proposes an intermediate representation called dynamic graph (DG), which contains the required dependence edges only i.e. data and control dependence. Hence, dynamic graph is simple and can be traversed faster for generating precise dynamic slices for various software engineering applications.

Moreover, although number of nodes increases, the time of computing slices does not increase, rather it depends on the number of dependencies (control

and/or data) in the dynamic graph (DG). This is another advantage of the proposed algorithm.

While comparing our work and work done by Wang et al. [16], we found that our intermediate representation is more applicable because in their work although they have focused on data and control dependencies, they are being suppressed inside a phase. But focused on the dependencies at inter-phase level. But it is essential to address those dependencies, which is done in our work. Our intermediate program representation in a more compact way of representing the dependencies.

Threats to validity:

 Our proposed algorithm can compute dynamic slice of the object-oriented programs in presence of inheritance. But it does not address the other object-oriented features like polymorphism, Dynamic binding, message passing etc.

Further, it cannot handle the situation where the objects are passed as parameters when a methods is being called.

 The dynamic graph may be complicated and difficult to handle, when the number of statements in the execution trace is very large. So our algorithm can handle only moderate sized applications and cannot handle large sized industrial applications.

VII. CONCLUSIONS AND FUTURE WORK

This paper presented an algorithm called *DG traversal slicing algorithm* to compute the dynamic slices of object-oriented programs in presence of inheritance. This paper used an intermediate representation called *dynamic graph (DG)* to represent the data and control dependencies present among the statements of the execution trace.

The proposed algorithm is not influenced by the number of nodes in the dynamic graph but it is influenced by the number of data and control dependencies in the intermediate program representation. This paper does not consider polymorphism, dynamic binding and message passing, which are also very important features of object-oriented programs. So in future, our work will focus on the above said features and computing the dynamic slice. Also, we will be focusing on computing slices of concurrent object-oriented and distributed object-oriented programs.

REFERENCES

[1] ,M. Wieser : *Program slicing*. In Proceedings of the 5th international conference on Software engineering.IEEE Press.(1981)439–449.

[2] J. Zhao: *Dynamic Slicing of Object-oriented Programs*.Journal of natural sciences. 6(2001)391-397.

[3] K., Ottenstein, L. Ottenstein : *The program dependence graph in software.Symposium on Practical Software Development Environments.*19(1984)177-184.

[4] S. Horwitz, T. Reps, D. Binkley : *Interprocedural slicing using dependence graphs*.ACM Transactions on Programming Languages and Systems.12(1990)26-61.

[5] B. Korel, J. Lask : *Dynamic Program Slicing.Information* Processing Letters.29(1988)155-163.

[6] H. Agrawal, J. Horgan : *Dynamic program slicing*. In Proceeding of the ACM SIGPLAN Conference on programming Languages Design and Implementation.25 (1990)246-256.

[7] G.B. Mund : *An efficient inter-procedural dynamic slicing method.* The Journal of Systems and software.79(2006)791-806.

[8] G.B. Mund, R. Mall, S. Sarkar : *Computation of intra-procedural dynamic program slices*. Information and Software Technology.45(2003)499-512.

[9] L. Larsen, M. J. Harrold : *Slicing Object-oriented Software.* In proceeding of ICSE.(1996)495-505.

[10] L. DU, G. Xiao, Y. Yu : *Research on algorithm for object-oriented program slicing*.Journal of Convergence Information Technology.6(2011).

[11] P. Jain, N. Garg : *A Novel Approach for Slicing of Object oriented programs*.ACM SIGSOFT Software Engineering Notes.38(2013)1-4.

[12] A. G. Beszedes : *Graph-less dynamic dependence-based dynamic slicing algorithm*. ixth IEEE International Workshop on Source Code Analysis and Manipulation.6(2006)21–30.

[13] J. D. Ferrante, J. Ottenstein, K. J.: *The program dependence graph and its use in optimization*.ACM Transaction On Program Languages And Systems.9(1987)319-349.

[14] Tao Wang, Abhik Roychoudhury : *Hierarchical dynamic slicing.* Proceedings of the 2007 international symposium on Software testing and analysis.2(2007)228-238.

[15] J. Korpi, J. Koskinen : *Constructive Dynamic Program Slicing Research*. Int. J. Adv. Comp. Techn.2(2010)7-23.

[16] A. Treffer, M. Uflacker : *Dynamic slicing with soot*. Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis. (2014)1-6.

[17] O. Chebaro, N. Kosmatov, A. Giorgetti, J. Julliand : *Program slicing enhances a verification technique combining static and dynamic analysis*. Proceedings of the 27th Annual ACM Symposium on Applied Computing.(2012)1284-1291.

[18] G. Sai Raghunath , Bhaludra Raveendranadh Singh , Moligi Sangeetha:*"Perpetuate Data Report based on the Slicing Approach"*. International Journal of Computer Trends and Technology (IJCTT) V16(2):68-72, Oct 2014. ISSN:2231-2803. Published by Seventh Sense Research Group.

[19] D. Mohanapriya , Dr. T.Meyyappan :"*Slicing Technique For Privacy Preserving Data Publishing"International Journal of Computer Trends and Technology (IJCTT)*,V4(5):1355-1361 May Issue 2013 .ISSN 2231-2803.www.ijcttjournal.org. Published by Seventh Sense Research Group.