Journal of Theoretical and Applied Information Technology
20th November 2014. Vol. 69 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

PROPERTY BASED DYNAMIC SLICING OF OBJECT ORIENTED PROGRAMS

[1]SANTOSH KUMAR PANI, [2]G.B.MUND
[1] School of Computer Engineering, KIIT University
[2] School of Computer Engineering, KIIT University
E-mail: [1]spanifcs@kiit.ac.in , [2]mundgb@yahoo.com

ABSTRACT

Slicing is the process of extracting the statements of a program affecting a given computation.In contrast to static slices, Dynamic slices are smaller in size as they extract statements for a given execution of a program and helps in interactive applications like debugging and testing. From last three decades, many algorithms have been designed to slice a program with respect to the syntax of the program. Traditional bulky Syntax based slices for program variables used at many places in a program are generally large even for dynamic slices. Most of the semantics based slicing algorithms extract slices by storing an execution trace of a program. To the best of our knowledge generating dynamic slices based on abstract/concrete properties of program variables is scarcely reported in literature. We present here an algorithm for generating dynamic syntax based as well as property slices of object oriented programs addressing all key object oriented features.

Keywords: *Dynamic slice, ReferenceSet, Property based Slice, Polymerphism, Abstract state.*

1. INTRODUCTION

Program slicing is an analysis method. It is the process of extracting the statements of a program affecting a given computation. A slicing criterion <s, V> is a tuple where s is a program statement and V is a subset of the program's variables used or defined at s. A dynamic slice of program P contains the statements that has an effect on the slicing criterion for a given execution. Hence a dynamic slice is smaller in size and more useful for interactive application like program testing and debugging.

Weiser [1] was the first to introduce static program slicing. In his intraprocedural static slicing algorithm he constructed a Control Flow Graph (CFG) for intermediate representation. Inter-statement influences were represented by means of data-flow equations. Weiser's method generated static slices based on iteratively solving these data-flow equations. Korel and Laski [2] were first to compute dynamic slices. The space requirement of Korel and Laski was O(N) for storing the execution history, and $O(N^2)$ for storing the dynamic flow data, where N is the length of execution.

Mund et al. [4] present an efficient dynamic slicing algorithm for intraprocedural environment, and then extend it to handle interprocedural calls. A collection of control dependence graphs were used by their methods as the intermediate program representation.

Larson and Harrold [7] were the first to consider object orientation aspects in their work. They introduced the class dependence graph .They represent a class hierarchy, data members, inheritance and polymorphism. This paper describes the construction of system dependence graphs for object-oriented software on which efficient slicing algorithms can be applied.

The concurrency and dynamic slicing aspects were not addressed by Larson and Harrold, Wang et al. [14], Huynh and Song [13] , Xu and Chen [12] and Zhao [11] have addressed these issues of object-oriented programs.

Now-a-days most of the application programs contain thousands of lines of code. Traditional bulky syntax based slices for program variables used at many places in a program are generally large even for dynamic slices.

While analyzing a program P, suppose it is required for a variable v in P to have a particular property ρ. If we find at a fixed program point, v does not have the desired property ρ, then it is needed to know which statements affect the computation of property ρ of v. Here we are not interested in the exact value of v, hence all the statements that a standard syntax based slicing algorithm would extract are not required. Therefore, the traditional bulky value based static slicing is not

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

adequate in this case. Since, properties propagate less than values, some statements might affect the values but not the property. Due to this the debugging and program understanding can be easier, as a relatively smaller portion of the code has to be inspected.

Mastroeni and Zanardini in [15] introduced a semantics-based dependency which act as a bridge between syntax and semantics. Based on this semantic dependency, a more precise PDG can be obtained by removing the false dependencies from the traditional syntactic PDG. The semantic dependency can also be lifted to an abstract domain where dependencies are computed with respect to some specific properties of interest rather than syntax influenced values. This (abstract) semantic dependency is computed at expression level over all possible (abstract) states appearing at program points.

Sukumaran et al.[16] introduced Dependence Condition Graph (DCG), a refinement of PDGs based on the notion of conditional dependency. This is obtained by adding the annotations which encode the condition under which a particular dependence actually arises in a program execution.

There are many papers on Dynamic Slicing of object oriented programs but few papers address in details about the most basic features of Object Oriented Programming i.e. class definition, object creation, accessing object through reference, invoking methods of a class, polymorphism, inheritance, dynamic binding etc. Most of the semantics based slicing algorithms have focused on finding static slices on the abstract properties by using SSA as intermediate representation and extract slices by storing an execution trace of a program. To the best of our knowledge generating dynamic slices based on abstract/Concrete properties of variables/objects in object oriented programs addressing all key features of object oriented programming is scarcely reported in literature.

We combine the concepts of Mund et al [4] and R Halder and A cortesi [9] to design an algorithm to generate dynamic slices on abstract properties of object oriented program variables rather than syntax based concrete values. In our approach we first maintain some additional data structures to capture all the object oriented features. The semantic relevancy and semantic dependency is also captured as soon as it is executed in actual run of the program. We define a slicing criteria as <s,V,P> where s is a program statement, V is the variable of interest and P is the examined property of interest. We modify the algorithm of mund et. al. [4] to extract the syntax based slice of object oriented program and simultaneously add it to the semantic slice if the statement to be added to the syntax data slice is semantically relevant. Since the syntax based data dependencies and control dependencies are already addressed and the syntax slice is always a super set of semantic slice, the generated slice will be smaller and useful in interactive applications. Our algorithm also not required to store any execution trace as it immediately updates the required data structures. The slices on defined properties of program variables are already available before a slice is asked for.

Next section describes some basic definitions that are used by our proposed algorithm. The property based dynamic slicing algorithm is discussed in the next section followed by the analysis of the algorithm and comparison with related work. The next section concludes the paper.

.

2. BASIC CONCEPTS AND DEFINITIONS

Object oriented programs are much similar to procedural programs except the restriction in access to data. The dependencies that exist in an object oriented program are the static control dependency and the dynamic data dependency. The other features of object oriented programming like inheritance, polymorphism, dynamic binding etc can be captured by using runtime disposable data structures.

We present here few basic concepts and definitions associated with our Algorithm. Some of the concepts and definitions are available in Mund et al [4] and R Halder and A cortesi [9].

2.1 Control Dependency

2.1.1 Control Flow Graph

The control flow graph (CFG) 'G' of a program P is a graph G = (N,E), where each node n Є N represents a basic block of statements in the program P. For any pair of nodes x and y, (x,y) Є E iff there is possible flow of control from x to y. This Control Flow Graph can be used to extract control dependency that can exist among statements in a program..

2.1.2 ControlDependentOn(u)

Let u be a statement of the program P. ControlDependentOn(u) = s iff the statement u is control dependent on s.

2.1.3 ActiveContrlSlice(s)

If s is a predicate statement of a program P and UseVarSet(s) = $\{v_1 . . . v_k\}$. Before execution of the program P, ActiveContrlSlice(s) is set ti Φ. After

Journal of Theoretical and Applied Information Technology
20th November 2014. Vol. 69 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

each execution of the statement s in an actual run of the program, ActiveContrlSlice(s) = {s} U ActiveDataSlice(v_1) U …U ActiveDataSlice(v_k) U ActiveContrlSlice(t), where ControlDependentOn(s)= t. Let s is a loop control statement, if the current execution of s corresponds to exit from the loop, then ActiveContrlSlice(s) is set to Φ.

2.2 Class

Any Object Oriented Programming must supports classes. A class has a definition which includes the definition of its data members and methods. Different Object Oriented Programming languages support different types of access to use these class members. A programmer defined class has to be defined with all it's member definition. The class member can be data or methods. We define the following data structure to process classes in an object oriented program.

2.2.1 DMemberSet()

Let C be a class then, DMemberSet(C) is the set of all data members of the class C.

2.2.2 MMemberSet()

Let C be a class then, MMemberSet(C) is the set of all method members of the class C.

Whenever a class is defined, the DMemberSet( ) and MMemberSet( ) data structures are updated.

2.3 Objects

The classes in Object Oriented Programming are made useable by creating objects. Objects can be created statically (C++) or dynamically (C++ & JAVA). Most Object Oriented Programming languages access objects through reference variable. Again the reference variables may be permanently (C++) or it may be temporarily (JAVA) attached to an object.We define the following data structures to process object creation and accessing a class member through object reference in an object oriented program.

2.3.1 InstanceOf(obj)

Let obj be an object or object reference of a class C, then InstanceOf(obj)=C.

The InstanceOf( ) data structure is updated with creation of each object(static creation) or object reference (dynamic creation).

2.3.2 ActiveDataSlice(var)

Let var denotes a data variable or a member variable or a reference variable of an Object Oriented Program P.

If var is a data variable of basic data type like int, char, float, double or a reference variable in Object Oriented Program P, Initially, ActiveDataSlice(var) =Φ.

Let x be a Def(var) node, and UseVarSet(x)={v_1,v_2,…,v_k}.ActiveDataSclice is updated after execution of each statement u in the following way:

ActiveDataSlice (var) ={x} ∪ ActiveDataSlice (v_1) ∪ ActiveDataSlice (v_2)… ∪ ActiveDataSlice (v_k)∪ ActiveContrlSlice(t), where ControlDependentOn(x)=t.

If dv is a data member of a statically created object obj.Initially, ActiveDataSlice(obj.dv) =Φ.For all dv Є DMemberSet(InstanceOf(obj)). For dynamically created object obj the ActiveDataSlice(obj.var)= Φ for all var Є DMemberSet(InstanceOf(obj)) dynamically whenever the object creation statement is executed.

Let x be a Def(obj.dv) node, and UseVarSet(x)={v_1,v_2,..v_k}. After each execution of the node x in the actual run of the program,ActiveDataSlice(obj.dv)={x}∪ ActiveDataSlice(v_1)∪ ActiveDataSlice(v_2)…∪ ActiveDataSlice(v_k)∪ ActiveContrlSlice(t) where ControlDependentOn(x) = t.

2.3.3 DyanSlice(s,var)

Let s be a statement of an object oriented program P, var (may be a data variable, member variable or reference variable) be a variable in the set i.e. var∈ DefVarSet(s) ∪ UseVarSet(s).

Before execution of the program P, DyanSlice(s,var) = Φ. After each execution of the node s in an actual run DyanSlice(s,var)=ActiveDataSlice(var) ∪ ActiveContrlSlice(t), whereControlDependentOn(s) = t.

2.3.4 DyanSlice(obj)

Let obj be an object in Object Oriented Program P. Before execution of program P, DyanSlice(obj)=Φ. Let the DMemberSet(InstanceOf(obj)) ={$mvar_1$.$mvar_2$,…,$mvar_n$} then DyanSlice(obj) =DyanSlice(obj.$mvar_1$) ∪ DyanSlice(obj.$mvar_2$) ∪ … ∪ DyanSlice(obj.$vbar_n$).

2.4 Method Call

2.4.1 CallSliceStack

This stack is maintained to keep track of the ActiveCallSlice during the execution of the program.

2.4.2 Formal(x,var), Actual(x,var).

Let m1 be a member method of a class in an Object Oriented Program P and x be a calling node to the member function m1. The formal and actual parameter of member function m1 be f and a

Journal of Theoretical and Applied Information Technology
20th November 2014. Vol. 69 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

respectively, then Formal(x,a) = f and Actual(x,f) = a.

2.4.3 ActiveReturnSlice

Let P be an Object Oriented Program. Initially, ActiveReturnSilce= Φ. If x is a *return* statement in program P and UseVarSet(x)={v_1,v2,...,v_k}. Then, before execution of x, ActiveReturnSlice={x} ∪ ActiveDataSlice(v_1) ∪ ActiveDataSlice(v_2) ∪ ... ∪ ActiveDataSlice(v_k) ∪ ActiveCallSlice ∪ ActiveContrlSlice(t),where ControlDependentOn(x)=t.

Followings are done after execution of each call node u:

- ActiveReturnSlice is used to compute and update necessary run-time information .
- ActiveCallSlice is set to Φ.

If the formal and actual parameter of a method m_1 be f and a respectively, then ActiveDataSlice(f)=ActiveDataSlice(a) ∪ ActiveCallSlice.

For each variable *var* used or defined at an execution node z, DyanSlice(z,*var*)= ActiveDataSlice(*var*) ∪ ActiveCallSlice ∪ ActiveContrlSlice(t), where ControlDependentOn(x)=t.

Execution of the member method m_1 ends with a *return* node iff its corresponding method call node y is a Def(v) node where v is a variable, then ActiveDataSlice(v) =ActiveReturnSlice after execution of the node y.

2.5 Object Reference

In OOP language it is possible that a reference of a class can refer to one or more objects of that class at different instance of time. We propose to maintain a list for each object that contains all the references which are referring to that object. This list may contain a reference of its own class or a reference of its base class.

2.5.1 RefSet(obj)

Let obj be an object of class ABC and var_1,var_2…$var_k$are the referencesof class ABC or its base class referring to the object obj. Then RefSet(obj)={var_1,var_2,…,var_k}ControlDependentOn(u)):Let u be a statement of the object oriented program P. ControlDependentOn(u)= s iff the statement u is control dependent on s.

Whenever a reference variable var changes its reference from obj_1 to obj_2, it will be removed from RefSet(obj_1) and inserted into RefSet(obj_2).Whenever a member function is called with object(s) obj_1, obj_2, . . ., obj_n as reference arguments then RefSet(obj) should be updated for each obj in the argument list of the member function. RefSet(obj) = RefSet(obj) ∪ Formal(x,obj) where x is the calling node to the member function.

2.5.2 CurrentRefObj(var)

Let var is a reference of a class is referring to an object obj of that class or of its any derived class. Then CurrentRefObjrvar) = obj iff obj is the current object to which var is referring to. If RefSet(obj)={ref1,ref2,…,refk) then for each var in RefSet(obj) , CurrentRefObj(rvar)=obj.

e.g.
```
class ABC{
   int m;
   int n;
   }
ABC ref1;
ref1=new ABC( );
ABC ref2;
ref2=ref1;
ABC ref3;
ref3=new ABC( );
ref1=ref3;
```

For each execution of the Constructor of a class, the slicer can assign a unique name (like obj1, obj2,. . . ,objn) to newly created objects for identifying them uniquely.

After execution of *statement-2*
RefSet(obj1)={ref1}
CurrentRefObj(ref1)=obj1

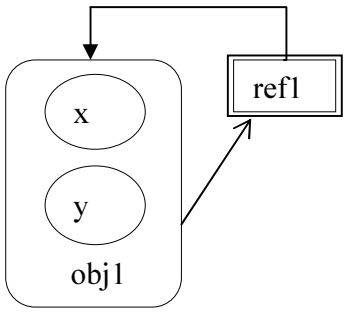

Figure. 1: After Execution Of Statement -2

After execution of *statement-4*
RefSet(obj1)=RefSet(obj1) ∪ {ref2}={ref1,ref2}
CurrentRefObj(ref2)=obj1

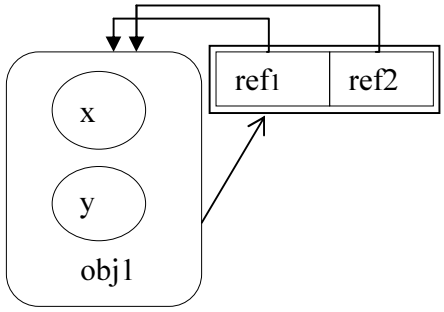

Figure 2: After execution of statement -4

Journal of Theoretical and Applied Information Technology
20th November 2014. Vol. 69 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

After execution of *statement-6*
RefSet(obj2)={ref3}
CurrentRefObj(ref3)=obj2
On execution of *statement-7*
RefSet(CurrentRefObj(ref1))=
RefSet(CurrentRefObj(ref1))
i.e. RefSet(obj1)=RefSet(obj1)− {ref1}={ref1,ref2} − {ref1}={ref2}

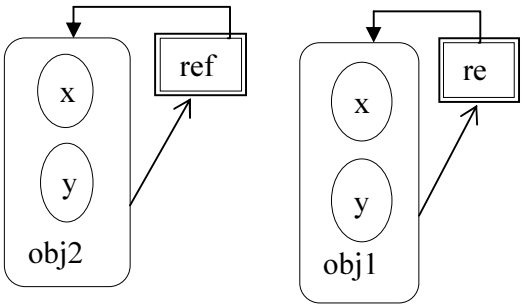

Figure 3: After execution of statement -7

2.6 Inheritance

2.6.1 Predecessorof( )

Let D be a class inherited from a class B then Predecessorof(D)=B , if D is not inheriting from any class in the source code program then Predecessorof(D)= Φ. Inheritance is captured by the Predecessorof() data structure. If a method which is not existing in the derived class is called with the derived class reference then the method can be recursively searched in the MMemberSet(predecessorof(derived class)) and the required update can be made before calling the method.

2.7 Polymephism

Polymerphism basically achieved in object oriented program in two ways i.e. method overloading and through method overriding and dynamic binding. In method overloading each overloaded method has an unique signature so it can be uniquely renamed by the slicer and the required update can be made before calling an overloaded method.In the other polymorphic behaviour we allow the base class references to stay in the RefSet of an object. On that base class reference whenever an overriding method is called ,by looking into the CurRef(obj) and MMemberSet(InstanceOf(obj))The required update can be made before calling an overloaded method.

2.8 Abstract interpretation

Abstract domains represent properties of variables over concrete domains. Their mathematical structure guarantees, for each concrete element there exists the best correct approximation in the abstract domain. This is because the property of abstract domains of being closed under greatest lower bound. The lattice of abstract interpretation of C is isomorphic to the lattice UCO(C) of all the upper closure operators(uco) on C .UCOs are distinctively calculated by the set ρ(C) of their fix-points. We have used the abstract domain SIGN containing (⊤), (⊥) and the abstract values [neg] ≡ Z^- (negative number) and [pos] ≡ Z^+ (positive numbers including 0).

Completeness in abstract interpretation is a property of abstract domains relative to a fixed computation. An abstract domain ρ is complete for f if it is optimally precise for calculation. Generally ρ is complete for f if $\rho \circ f \circ \rho = \rho \circ f$. In other words, computing f in the abstract domain corresponds precisely to abstracting the concrete computation of f, without further loss of information.

2.9 AbstractState(u)

AbstractState (u) represents the abstract state associated with each program variable at statement u of program P. This is updated after each execution of program statement u.

e.g. 1.p=10;
2.q=-6;
3.r=p+2q

After execution of statement 1: AbstractState(1)={+, ⊥,⊥}.
After execution of statement 2: AbstractState(2)={+, -,⊥}.
After execution of statement 3: AbstractState(3)={+, - , +}.
Where ⊥ represents the abstract state of uninitialized variables and is the least upper bound for the lattice for abstract domain for abstract property sign.

2.10 Semantic Relevancy

$\forall \varepsilon \in \sum\rho : P[[s]]_p^{\rho}(\varepsilon) = \varepsilon$, the statement s is not semantically relevant with respect to the abstract domain ρ.statement s at program point p is semantically irrelevant if no changes take place in the abstract state ε (abstract state(s)) occurring at p, when s is executed over ε. The statements which do not contribute to any change in the states occurring at that program point are considered semantically irrelevant. The atomicity of the abstract value for

Journal of Theoretical and Applied Information Technology
20th November 2014. Vol. 69 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

each variable in the abstract state ε with respect to property ρ is one of the crucial requirements during computation of ρ-relevancy of the statements. These atomic abstract values are obtained from induced partitioning. The following example shows how to compute the semantic relevancy for the statements by using covering techniques.

e.g. 1.x=y+0
2.x=y+1

If we consider a property ρ =sign then statement 1 is irrelevant with respect to property ρ as for any of the value of y the statement will not change the state. In statement 2 if y= -1 then the statement changes the value of y from negative to zero. Similarly if y=0 then the statement changes the value of y from zero to positive. So statement 2 becomes relevant with respect to the property ρ

2.11 Semantic Dependancy

A variable vi is said to be have semantic dependency on variable v, if excluding vi from u and re-executing u does not change the abstract state of v with respect to ρ.

2.12 ActiveSemanticSlice(v, ρ)

ActiveSemanticSlice holds only those statements which influence the variable v semantically on the basis of an abstract property ρ. It is updated in the following ways:

- Let u be a Def(v) statement in program p, the node u is included in ActiveSemanticSlice(v, ρ) if execution of node u changes the abstract state of v for current set of inputs with respect to ρ.
- Let UseVarSet(u)={v1,v2.....vk}, the ActiveDataSlice(vi) is included in ActiveSemanticSlice(v, ρ) if there is semantic dependency of vi on v(where v is defined in statement u).A variable vi is said to be have semantic dependency on variable v, if excluding vi from u and re-executing u does not change the abstract state of v with respect to ρ.
- Let ControlDependentOn(u) = t, ActiveContrlSlice(t) is included in ActiveSemanticSlice(v, ρ) if execution of node u changes the abstract state of v for current set of inputs with respect to ρ.

Before execution of node u ActiveSemanticSlice(var, ρ)=Φ. ActiveSemanticSlice(var, ρ) is updated appropriately after execution of u.

2.13 DynamicSemanticSlice(v, s, ρ)

Let s be a statement of Program P, v be a variable in the set UseVarSet(s) U DefVarSet(s) and ρ is the abstract property of interest. Before execution of the program P, DyanSlice(v, s, ρ) = Φ. After each execution of the node s in the actual run of the program, the dynamic slice DyanSemanticSlice(v, s, ρ) with respect to the slicing criterion <v,s, ρ > is updated as DyanSemanticSlice(v, s, ρ) = ActiveSemanticSlice(v) U ActiveContrlSlice(t)(if s is semantically relevant to v), where ControlDepenedentOn(u) = t.

2.14 SemanticReturnSlice

Initially, SemanticReturnSilce= Φ. Let x is a *return* statement in program P, and UseVarSet(x)={v_1,...,v_k}. Before execution of node x, SemanticReturnSlice={x} ∪ ActiveDataSlice(v_1) ∪ ActiveDataSlice(v_2)... ∪ ActiveDataSlice(v_k) ∪ ActiveCallSlice ∪ ActiveContrlSlice(t), ∀v_i : v_i has semantic dependency on result of x with respect to abstract property ρ and ControlDependentOn(x)=t.

Let u be a call node. After each execution call node u, we do the following:

- Use SemanticReturnSlice to compute and update relevant run-time information corresponding to the execution of u
- Update ActiveCallSlice=Φ

Let the formal and actual parameter of the method m_1 at the calling node x be f and a respectively.. Then ActiveDataSlice(f)=ActiveDataSlice(a)U ActiveCallSlice and ActiveSemanticSlice(f)= ActiveSemanticSlice(a).

Thus for each var used or defined at an execution node z inside method m1, then DyanSlice (z,var,ρ) =ActiveCallSlice ∪ ActiveDataSlice(var) ∪ ActiveControl Slice(t) where ControlDependentOn(z)=t. and DyanSemanticSlice (z,var,ρ) =ActiveCallSlice ∪ ActiveSemanticSlice(var) ∪ ActiveContrlSlice(t), if z is semantically relevant to var with respect to abstract property ρ.

Execution of the member method m_1 ends with a RETURN node iff its corresponding method call node y is a Def(v) node where v is a variable, then

- ActiveDataSlice(v) =ActiveReturnSlice.
- ActiveSemanticSlice(v) =SemanticReturnSlice.

3. ALGORITHM

we present here an efficient property based dynamic slicing algorithm for an Object Oriented program. To compute slices, we first construct the

Journal of Theoretical and Applied Information Technology
20th November 2014. Vol. 69 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

control flow graph (CFG) of the program P statically once. The algorithm uses the CFG for extracting the control dependency. The run time data structures are updated by the algorithm during execution of the program P.

3.1. Property based dynamic slicing Algorithm for Object Oriented programs

1. Construct the Control Flow Graph G_P of the program P.
2. Before each execution of the program do the followings for each statement u.
Set ActiveContrlSlice(u) = Φ If u is a predicate statement..
Update ControlDependentOn(u)
For each variable var Є DefVarSet(u) U UseVarSet(u) do
DynamicSemanticSlice(var,u, ρ) = Φ.
DyanSlice(u, var) = Φ.
Initialize the followings for each variable var of the program P
ActiveSemanticsSlice(var, ρ) = Φ
ActiveDataSlice(var) = Φ.
CallSliceStack = NULL.
ActiveCallSlice = Φ
For definition of each class C
Update DMemberSet(C)
Update MMemberof(C)
For each member m of the class
If the class is inhering from a class D
Update
Predeccesorof(C)
For each object or reference variable r
Update
Instanceof(r)
For abstract property ρ set AbstractState(u)={ ⊥, ⊥,⊥,…,⊥ },where u is the first statement to be executed by program P.
3. Repeat steps 4, 5 and 6 with given set of input values until the program terminates.
4. Do the following before execution of a call statement u.
If u is a call statement to a method Q, then
(a) Update CallSliceStack and ActiveCallSlice.
(b)For each actual parameter var in the procedure call Q do
ActiveDataSlice(Formal(u,var))= ActiveDataSlice(var) U ActiveCallSlice.
ActiveSemanticSlice(Formal(u,var)) = ActiveSemanticSlice(var).
If the parameters are object references then
Update RefSet( ) and CurrentRefOf()
5. before execution of a *return* statement u,
Update ActiveReturnSlice and SemanticReturnSlice.
If the return value is an object reference then
Update RefSet( ) and CurrentRefOf()
6. After execution of statement u of the program P, do the following
(a) If u is a Def(var) statement and not a call statement then
Update ActiveDataSlice(var).
Update ActiveSemanticSlice(var)
(b) If u is a call statement to a procedure Q then do
For every formal reference parameter var in the procedure Q do
ActiveDataSlice(Actual(u,var))= ActiveDataSlice(var).
ActiveSemanticSlice(Actual(u,var))= ActiveSemanticSlice(var).
if u is a Def(var) statement then
ActiveDataSlice(var) = ActiveReturnSlice.
ActiveSemanticSlice(var) = SemanticReturnSlice.
for every local variable var in the procedure Q do
ActiveDataSlice(var) = Φ .
ActiveSemanticSlice(var) = Φ
Update CallSliceStack and ActiveCallSlice.
Set ActiveReturnSlice = Φ.
SemanticReturnSlice= Φ.
Updtae AbstractState(u).
(c) For every variable var Є DefVarSet(u) U UseVarSet(u) do
Update DyanSlice(u, var).
Update DynamicSemanticSlice(var,u, ρ)
(d) Update ActiveContrlSlice(u) where u is a predicate statement.
7. Exit

3.2. Working of the Algorithm

For better understanding of the working of the algorithm and updation of the run-time data structures, we consider the following two example programs.Example program-1 illustrates the updation of data structures in dealing with all object oriented features and generates syntax based slices. Example program-2 illustrates the property based slicing to generate both syntax and semantic based slices.

Example program-1:

```
class Number
      {
      int x;
      int y;
      Number(int p, int q)
      {
15.   x=p;
16.   y=q;
      }
void getData(int p, int q)
      {
17.   x=p;
```

Journal of Theoretical and Applied Information Technology
20th November 2014. Vol. 69 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
18.    y=q;
       }
 void showData( )
       {
19.System.out.println("x = "+x+"\t
y = "+y);
       }
 void getIncrement( )
       {
20.    x = x + 1;
21.    y = y + 1;
       }

Number addNumber(Number ref5)
       {
       Number temp;
22.    temp=new Number( );
23.    temp. x = x + ref5. x;
24.    temp.y = y + ref5.y;
25.    return temp;
       }
       }
public class Main
       {
Public   static   void   main(String
s[])
       {
       Number ref1;
1.     ref1=new Number( );
       Number ref2;
       int a, b;
2.     a=4;
3.     b=5;
4.     ref1.getData(a,b);
5.     ref1.getIncrement( );
6.     ref1.showData( );
7.     ref2=ref1;
8.     ref2.getIncrement( );
9.     ref2.showData( );
       Number ref3,ref4;
10.    a=10;
11.    b=20;
12.    ref3=new Number( a,b);
13.    ref4=ref1. addNumber(ref3);
14.    ref4. showData( );
       }
       }
```

After execution of node-1: ActiveDataSlice(obj1. x)= Φ, DyanSlice(1,obj1. x)= Φ, ActiveDataSlice(obj1. y)= Φ, DyanSlice(1,obj1. y)= Φ, DyanSlice(obj1)= DyanSlice(1,obj1. x) ∪ DyanSlice(1,obj1. y)= Φ∪ Φ = Φ, ActiveDataSlice(ref1)= {1}∪ Φ = {1}, DyanSlice(1,ref1)= {1}, RefSet(obj1)={ref1}, CurrentRefObj(ref1)=obj1.

After execution of node-2: ActiveDataSlice(a)={2}, DyanSlice(2,a)={2}.

After execution of node-3: ActiveDataSlice(b)={3}, DyanSlice(3,b)={3}.

Before execution of node-4: ActiveCallSlice = {4} ∪ Φ = {4}, CallSliceStack = [{4}], Formal(4,a)=p, Formal(4,b)=q,ActiveDataSlice(p)=ActiveDataSlice(a) ∪ ActiveCallSlice={2}∪{4}={2,4}, ActiveDataSlice(q)=ActiveDataSlice(b) ∪ ActiveCallSlice ={3}∪{4}={3,4}.

After execution of node-17: ActiveDataSlice(ref1.x)={17}∪ActiveDataSlice(p) ={17}∪{2,4} = {2,4,17},DyanSlice(17,ref1.x)=ActiveDataSlice(ref1.x)={2,4,17},DyanSlice(17,p)=ActiveDataSlice(p) ={2,4},DyanSlice(CurrentRefObj(ref1))=DyanSlice(obj1)=DyanSlice(17,ref1.x)∪DyanSlice(17,ref1.y) = {2,4,17} ∪ Φ={2,4,17}.

After execution of node-18: ActiveDataSlice(ref1.y)={18}∪ActiveDataSlice(q) ={18}∪{3,4} = {3,4,18},DyanSlice(18,ref1.y)=ActiveDataSlice(ref1.y)={3,4,18},DyanSlice(18,q)=ActiveDataSlice(q) ={3,4},DyanSlice(CurrentRefObj(ref1))=DyanSlice(obj1)=DyanSlice(18,ref1.x∪DyanSlice(18,ref1.y) ={2,4,17}∪{3,4,18}={2,3,4,17,18}.

After execution of node-4: DyanSlice(4, ref1)= {1}, ActiveCallSlice= Φ, CallSliceStack= Φ.

Before execution of node-5: ActiveCallSlice = {5} ∪Φ = {5}, CallSliceStack = [{5}].

After execution of node-20: ActiveDataSlice(ref1.x)={20}∪ActiveDataSlice(ref1.x)={20}∪{2,4,17} ={2,4,17,20},DyanSlice(20,ref1.x)=ActiveDataSlice(ref1.x)={2,4,17,20},DyanSlice(CurrentRefObj(ref1))=DyanSlice(obj1)=DyanSlice(20,ref1.x) ∪ DyanSlice(20,ref1.y)={2,4,17,20}∪{3,4,18} ={2,3,4,17,18,20}.

After execution of node-21: ActiveDataSlice(ref1.y)={21}∪ActiveDataSlice(ref1. y)={21}∪{3,4,18} ={3,4,18,21},DyanSlice(21,ref1.y)=ActiveDataSlice(ref1.y)={3,4,18,21},DyanSlice(CurrentRefObj(ref1))=DyanSlice(obj1)=DyanSlice(20,ref1.x) ∪DyanSlice(21,ref1.y) = {2,3,4,17,18,20,21}.

After execution of node-5: DyanSlice(5, ref1)= ActiveDataSlice(ref1)={1}, ActiveCallSlice= Φ, CallSliceStack= Φ.

Example program-2:

Journal of Theoretical and Applied Information Technology
20th November 2014. Vol. 69 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
void  main()
 {
 int p,q,i,r;
 1. read(p);
 2. read(i);
 3. read(r);
 4. q=r;
 5. while(i<2)
 {
 6. p=add(p,r);
 7. p=2*(p+2);
 8. q=calculate(p,r);
 9. i=i+1;
}
 10.  write(p);
 11. write(q);
 }

int add(int a, int b)
 {
 12. a=a+b;
 13. return(a);
 }

int  calculate(int y, int z)
 {
  14. return(4*y%2 + 2*z +6);
 }
```

Before execution of node 1: AbstractState(1) = { ⊥, ⊥, ⊥, ⊥}.

After execution of node 1:ActiveDataSlice(p)={1}, DynaSlice(1,p)={1}, AbstractState (1)={-, ⊥,⊥,⊥}, ActiveSemanticSlice(p,SIGN)= ActiveSemanticSlice(p, SIGN) U {1} = {1}, DyanSemanticSlice (p,1,SIGN)={1}.

After execution of node 2 : ActiveDataSlice(i) ={2}, DynaSlice(2,i) = {2}, AbstractState(2)= {-,+,⊥,⊥}, ActiveSemanticSlice(i, SIGN) = ActiveSemanticSlice(i, SIGN) U {2} = {2}, DyanSemanticSlice(i,2,SIGN)={2}.

After execution of node 3 : ActiveDataSlice(r)= {3}, DynaSlice(3,r)= {3}, AbstractState(3)= {-,+,+,⊥}, ActiveSemanticSlice(r, SIGN) = ActiveSemanticSlice(r, SIGN) U {3} = {3}, DyanSemanticSlice(r, 3, SIGN)= {3}.

After execution of node 4: ActiveDataSlice(q)={4} U ActiveDataSlice(r)={3,4}, DynaSlice(3,r)= {3\},DynaSlice(3,q)= {3,4}, AbstractState(3)= {-,+,+,+\}, ActiveSemanticSlice(q, SIGN) = ActiveSemanticSlice(q,SIGN)U{4}UActiveDataSlice(r) = {4} U {3}={3,4} , DyanSemanticSlice(r, 3, SIGN)= {3}.

After execution of node 5 :ActiveControlSlice(5)={5} U ActveDataSlice(i) = {2,5}, DynaSlice(5,i) = ActiveDataSlice(i)= {2}, AbstractState(5) = {-,+,+,+}, DyanSemanticSlice(i, 5, SIGN) = {2}.

Before execution of node 6 : AbstractState (6)= {-,+,+,+},ActiveCallSlice={6}UActiveCallSliceU ActiveControlSlice(5)={6}U{2,5}={2,5,6}, CallSliceStack=[{2,5,6}], ActiveDataSlice(a) = ActiveDataSlice(p) U ActiveCallSlice = {1} U {2,5,6}={1,2,5,6},ActiveDataSlice(b)=ActiveDataSlice(r) UActiveCallSlice= {3} U {2,5,6} = {2,3,5,6},ActiveSemanticSlice(a,SIGN)=ActiveSemanticSlice(p,SIGN)={1},ActiveSemanticSlice(b,SIGN)= ActiveSemanticSlice(r,SIGN)={3}

Before execution of node 12 :AbstractState(11) ={-,+}

After execution of node 12 : ActiveDataSlice(a)={12\} U ActiveDataSlice(a) UActiveDataSlice(b)={12} U {1,2,5,6} U{2,3,5,6} ={1,2,3,5,6,12},DynaSlice(12,a)=ActiveDataSlice(a) U AciveCallSlice={1,2,3,5,6,12} U {2,5,6}={1,2,3,5,6,12},DynaSlice(12,b)=ActiveDataSlice(b) U AciveCallSlice={2,3,5,6} U {2,5,6}={2,3,5,6},AbstractState(12)={+,+}, ActiveSemanticSlice(a,SIGN)={12}UActiveDataSlice(b) U AciveCallSlice = {12} U {2,3,5,6} U{2,5,6}={2,3,5,6,12},DyanSemanticSlice(a,12,SIGN)=ActiveSemanticSlice(a,SIGN)UAciveCallSlice ={2,3,5,6,12} U {2,5,6} = {2,3,5,6,12}, DyanSemanticSlice(b,12,SIGN)=ActiveSemanticSlice(b,SIGN) ={3}.

12 is added to ActiveSemanticSlice(a,SIGN) since it changes the SIGN of a ActiveDataSlice(b) is added to ActiveSemanticSlice(a,SIGN) since the value of b changes the SIGN of result of expression at 12. Finally ActiveCallSlice is added to ActiveSemanticSlice(a,SIGN) since statement 12 changes the SIGN property of a.

Before execution of node 13 :AbstractState (13)={+,+},ActiveReturnSlice={13}UActiveDataslice(a) U ActiveCallSlice={13} U {1,2,3,5,6,12} U {2,5,6}={1,2,3,5,6,12,13}, SemanticReturnSlice = {13} U ActiveDataslice(a) U ActiveCallSlice ={13}U{1,2,3,5,6,12}U {2,5,6}={1,2,3,5,6,12,13}.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

After execution of node 6 : ActiveDataSlice(p)=ActiveReturnSlice={1,2,3,5,6, 12,13},DynaSlice(5,p) = ActiveDataSlice(p) ∪ ActiveControlSlice(5)={1,2,3,5,6,12,13}∪{2,5}={ 1,2,3,5,6,12,13}, AbstractState (5) = {+,+,+,+}, ActiveSemanticSlice(p,SIGN)=ActiveSemanticSlic e(p,SIGN) ∪ SemanticReturnSlice=={1} ∪ {1,2,3,5,6,12,13}={1,2,3,5,6,12,13},DyanSemantic Slice(p,5,SIGN)=ActiveSemanticSlice(p,SIGN) ∪ ActiveControlSlice(5) = {1,2,3,5,6,12,13} ∪ {2,5} ={1,2,3,5,6,12,13},CallSliceStack=Φ,ActiveCallSli ce=Φ,ActiveReturnSlice= Φ, SemanticReturnSlice= Φ.

After execution of node 7 :ActiveDataSlice(p)={7} ∪ ActiveDataSlice(p) ∪ ActiveControlSlice(5)= {7}∪{1,2,3,5,6,12,13}∪{2,5}={1,2,3,5,6,7,12,13}, DynaSlice(7,p)=ActiveDataSlice(p)∪ ActiveControlSlice(5)={1,2,3,5,6,7,12,13}∪ {2,5}={1,2,3,5,6,7,12,13},AbstractState(7)={+,+,+, +},ActiveSemanticSlice(p,SIGN)={1,2,3,5,6,12,13 },DyanSemanticSlice(p,7,SIGN)=ActiveSemanticS lice(p,SIGN)={1,2,3,5,6,12,13}.

Statement 7 is not added as it is not changing the abstract property of p.

3.3. Complexity analysis

The space requirement of our algorithm is $O(n^2)$ and is mainly due to the storage of the Control Flow Graph G_p. It can be easily shown that the other data structures used by our algorithm requires maximum O(n) space . Some of the data structures we use are reused or disposed when not required(e.g.RefSet, Active CallSlice[4]) resulting in efficient use of space.

The time complexity of finding dynamic syntax slices is linear in terms of the number of statements of the program[4].The complexity for finding semantic relevancy only depends on the comparison between the abstract state of the currently executed statement and the previous executed statement, which depends on the no of variables in a program and no of abstract states possible for a given property which is a constant. Calculation of semantic dependency depends on the maximum no of operators possible in an expression.

3.4. Comparison with related work

The advantage of this algorithm is that, it does not use a trace file for storing the execution history. Our algorithm extract slices of Object Oriented Programs by capturing all features of object Oriented Programming in some run time reusable and disposable data structures which are updated with execution of each statement in the program resulting in efficient use of space.

A pure dynamic slicing algorithm on abstract properties of variable in Object Oriented Programs is scarcely reported in literature. The dynamic slicing algorithm in [4] only finds out the slices based on the underlying syntax of the program. The static slicing algorithms reported in [9,11] has main disadvantage of maintaining trace files for storing the execution trace which may be unbounded in presence of loops. Our algorithm does not use trace file as the recent semantic slice is already captured in the data structures. Our algorithm can find both syntax based and property based dynamic slices and can be more helpful in interactive application like debugging and testing

4. CONCLUSION

In this paper we present an algorithm for property based dynamic slicing of Object Oriented Programs. We first use the basic concepts of the inter-procedural dynamic slicing algorithm [4] and remodel it to extract slices of Object Oriented Programs with introduction of some additional data structures.

Since the syntax based data dependencies and control dependencies are already addressed and the syntax slice is always a super set of semantic slice, the generated slice will be useful in interactive applications like debugging and testing. Our algorithm also not required to store any execution trace as it immediately updates the required data structures. The slices on defined properties of program variables are already available before a slice is asked for resulting in faster response time. As opposed to any slicing algorithms our algorithm generated both syntax and property based dynamic slices of object oriented programs.

The future scope of this paper lies in investigating property based dynamic slicing in distributed system and designing a testing tool for the proposed slicing algorithm.

REFRENCES:

[1] M.Weiser, "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 25, No. 7, July 1982, pp. 446 - 452.

[2] B.Korel and J.Laski, "Dynamic Program Slicing", *Information Processing Letters,* Vol. 29, No. 3, October 1988, pp. 155 - 163.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[3] G.B.Mund and R.Mall, "Program Slicing", *The Compiler Design Hand Book, Optimization and Machine Code Generation*, CRC Press, 2008, pp. 14.1 - 14.35.

[4] G.B.Mund and R.Mall, "An efficient interprocedural dynamic slicing method", *The Journal of Systems and Software*, Vol. 79, 2006, pp. 791 - 806.

[5] G.B.Mund,R.Mall and R.S.Sarkar, "Computation of intraprocedural dynamic program slices," Information and Software Technology, vol. 45, no. 8, pp 499 - 512, 2003.

[6] G.B.Mund, R.Mall and S.Sarkar, "An efficient dynamic program slicing technique", *Information and Software Technology*, Vol. 44, 2002, pp. 123 - 132.

[7] L.D.Larson and M.J.Harrold, "Slicing objectoriented software," In Proceedings of the 18th InternationalConference on Software Engineering, German, March 1996.

[8] F. Tip., "A survey of program slicing technique", *Journal of Programming Languages*, Vol. 3, 1995, pp. 121 - 189.

[9] R.Halder and A.Cortesi, "Abstract slicing of dependence condition graphs", *Science of Computer programming*, Vol.78, 2013 pp.1240 - 1263.

[10] S. K. Pani, P. Arundhati and M. Mohanty, " An Effective Methodology for Slicing C++ Programs", *International Journal of Computer Engineering and Technology*, 2010 Vol. 1 pp.72 - 82.

[11] J. Zhao, "Dynamic slicing of object-oriented programs", *Technical Report SE-98-119, Information Processing Society of Japan,* 1998 pp.17 - 23.

[12] Z. Chen and B. Xu., "Slicing objected-oriented java programs", *ACM SIGPLAN Notices,* Vol.36, 2001, pp. 33 - 40.

[13] D. Huynh and Y. Song., "Forward computation of dynamic slices in the presence of structured jump statements", *Proceedings of ISACC,* Vol.97, 1997, pp. 73 - 81.

[14] T.Wang and A. RoyChoudhury, "Using compressed bytecode traces for slicing Java programs", *Proceedings of the IEEE International Conference on Software Engineering*, 2004, pp. 512 - 521.

[15] I.Mastroeni and D. Zanardini, " Data dependencies and program slicing: from syntax to abstract semantics", *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '08, ACM Press,* 2008, pp. 125 - 134.

[16] S. Sukumaran, A. Sreenivas and R. Metta, "The dependence condition graph: precise conditions for dependence between program points", *Computer Languages, Systems & Structures*, Vol. 36, , 2010, pp. 96 - 121.