International Journal of Software Engineering and Technology (IJSET)
Vol. 1, No. 2, August 2016, pp. 69~82
ISSN: 2032-4038

Property Based Dynamic Slicing of Object Oriented Programs

Santosh Kumar Pani, G.B.Mund
School of Computer Engineering, KIIT University

Article Info

Article history:

Received Jun 12th, 2015
Revised Aug 20th, 2015
Accepted Aug 26th, 2015

Keyword:

Dynamic Slice
Abstract Properties
Object Reference
Concrete Value
Dynamic Binding
Property Based Slicing

ABSTRACT

Slicing is used for program analysis. It the process of extracting the statements of a program that are relevant to a given computation. Static slicing generates slices for all possible execution of a program helping in program understanding, verification, and maintenance and testing. Dynamic slices are smaller in size as they extract slices for a given execution of a program and helps in interactive applications like debugging and testing. With the wide spread use of object oriented software, there are many papers on Dynamic Slicing of object oriented programs but few papers only address in details about the most basic features of Object Oriented Programming that is class definition, Object creation, accessing object through reference, invoking methods of a class, polymorphism, inheritance etc. From last three decades many algorithms have been designed to slice a program with respect to the syntax of the program. The real worlds object oriented programs consist of thousands of lines of code. Traditional Syntax based slices for program variables used at many places in a program are generally large even for dynamic slices. Recently, some work has been done to get slices based on abstract/concrete properties of program variables. For smooth debugging and testing, the slice will be small if any particular property is being considered (semantics based). 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 program variables is scarcely reported in literature. In this paper we present an algorithm for generating dynamic abstract slices of object oriented programs addressing all key object oriented features.

Corresponding Author:

Santosh Kumar Pani,
School of Computer Engineering,
KIIT University,
Patia, Bhubaneswar, Odisha, India.
Email: spanifcs@kiit.ac.in

1. INTRODUCTION

Program slicing is a well known decomposition technique for extracting statements of a program related to a particular computation. A slicing criterion is a tuple <s, V> where s is a program point of interest and V is a subset of the program's variables used or defined at s. A static slice of a program P with respect to a slicing criterion <s, V> is the set of all the statements of program P that might affect the slicing criterion for every possible input to the program. A dynamic slice of program P contains only those statements that affect the slicing criterion for a particular set of inputs. Hence a dynamic slice is smaller in size and more useful for interactive application like program testing and debugging.

Program Slicing has been studied primarily in the context of procedural programming language. In such languages, slicing is typically performed by using a control flow graph or a dependency graph. Weiser [1] introduced the concept of a static program slice and presented the first intraprocedural static slicing

algorithm. His method used a Control Flow Graph (CFG) as the intermediate representation of the program, and was based on iteratively solving data-flow equations representing inter-statement influences. This algorithm did not handle programs having multiple procedures. Korel and Laski [2] extended Weiser's [1] CFG based static slicing algorithm to compute dynamic slices. Their method computes dynamic slices by solving the associated dataflow equations. The method of Korel and Laski needs O (N) space to store the execution history, and O (N^2) space to store the dynamic flow data, where N is the number of statements executed (length of execution) during the execution of the program.

Mund et al. [4] present an efficient interprocedural dynamic slicing algorithm for structured programs. They proposed an intraprocedural dynamic slicing algorithm, and subsequently extend it to handle interprocedural calls. The interprocedural dynamic slicing algorithm uses a collection of control dependence graphs (one for each procedure) as the intermediate program representation, and computes precise dynamic slices.

Larson and Harrold [7] were the first to consider object orientation aspects in their work. They introduced the class dependence graph which can 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.

Larson and Harrold have reported only a static slicing technique for sequential object-oriented programs, and did not address the concurrency and dynamic slicing aspects. Zhao [11], Huynh and Song [13] , Wang et al. [14] and Xu and Chen [12] have addressed the issues of dynamic slicing of object-oriented programs.

Now-a-days most of the application programs contain thousands of lines of code. Traditional 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 x in P to have a particular property ρ. If we realize that, at a fixed program point, x does not have that desired property, we may like to understand which statements affect the computation of property ρ of x, in order to find out more easily where the computation was wrong. In this case we are not interested in the exact value of x; hence we may not need all the statements that a standard slicing algorithm would return. Therefore, the traditional bulky value based static slicing is not adequate in this case. In this situation we would need a technique that returns the minimal amount of statements that actually affect the computation of a desired property of x. since, properties propagate less than values, and some statements might affect the values but not the property. This can make debugging and program understanding tasks easier, since a smaller portion of the code has to be inspected.

In traditional PDGs, the notion of dependency between statements is based on syntactic presence of a variable in the definition of another variable or on a conditional expression. Therefore, the definition of slices at semantic level creates a gap between slicing and dependencies.

Mastroeni and Zanardini in [15] introduced a semantics-based dependency which fills up the existing gap 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 concrete 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 that is 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 [17] 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 orientred features.The semantic relevancy of a statement is also caputed as soon as it is executed in actual run of the program. We define a slicing criterion 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 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.

The rest of the paper is organized as follows. In next section, we describe some basic definitions that are used by our 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 DEFINATION

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.

In this section, we present a few basic concepts, notations and terminologies associated with our Algorithm. Some of the concepts and definitions are available in Mund et al [4] and R Halder and A cortesi [17]. However, we present them here for the sake of completeness. Further, we introduce some new concepts and definitions which are used in the rest of the paper.

2.1. Control Dependency

Control Flow Graph: The control flow graph (CFG) G of an object oriented 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.

*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.

ActiveControlSlice(s): Let s be a test statement (predicate statement) of a program P and UseVarSet(s) = {var1. . . vark}. Before execution of the program P, ActiveControlSlice(s) = Φ. After each execution of the statement s in an actual run of the program, ActiveControlSlice(s) = {s} U ActiveDataSlice (var1) U …U ActiveDataSlice (vark) U ActiveControlSlice (t), where ControlDependentOn(s) = t. If s is a loop control statement, and the present execution of s corresponds to exit from the loop, then ActiveControlSlice(s) = 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 its member definition. The class member can be data or methods. We define the following data structure to process classes in an object oriented program.

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

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. Object

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.

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)

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*, then before execution of the program *P*, *ActiveDataSlice(var) =Φ.*

Let *u* be a *Def(var)* node, and *UseVarSet(u)={var1,var2,…,vark}*.After each execution of the node *u* in the actual run of the program, *ActiveDataSlice (var) ={u} ∪ ActiveDataSlice (var1) ∪…∪ ActiveDataSlice (vark)∪ ActiveControlSlice (t),* where ControlDepend*entOn*(u)=t.

If *var* is a data member of a statically created object *obj*. Before execution of the program *P*, *ActiveDataSlice (obj.var) =Φ.*For all var 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 *u* be a *Def (obj.var)* node, and *UseVarSet (u)={var1,var2,..vark}*. After each execution of the node *u* in the actual run of the program, *ActiveDataSlice(obj.var)={u}∪ActiveDataSlice(var1)∪…∪ ActiveDataSlice(vark)∪ActiveControlSlice(t)* where ControlDepend*entOn*(u)=t

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) ∪ActiveControlSlice(t) where ControlDependentOn(s)=t

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

Title must be in 24 pt Regular font. Author name must be in 11 pt Regular font. Author affiliation must be in 10 pt. Email address must be in 10 pt Regular font the data structures defined for handling control dependency are as follows.

CallSliceStack: This stack is maintained to keep track of the ActiveCallSlice during the actual run of the program.

*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. Let f be a formal parameter of member function m1 and its corresponding actual parameter at the calling node x be a. We define Formal(x, a) = f and Actual(x, f) = a

*ActiveReturnSlice**:*** Let P be an Object Oriented Program. Before each execution of program P, ActiveReturnSilce= Φ. Let x be a RETURN statement in P, and UseVarSet(x) = {var$_1$,...,var$_k$}. Then, before each execution of the RETURN node x, ActiveReturnSlice={x}∪ActiveCallSlice ∪ ActiveDataSlice(var$_1$) ∪... ∪ActiveDataSlice(var$_k$) ActiveControlSlice(t), where ControlDependentOn(x)=t.

Let u be a call node. After each execution call node u, we do the following: use ActiveReturnSlice to compute and update relevant run-time information corresponding to the execution of u. Update ActiveCallSlice=Φ. Let f be a formal parameter of the method m$_1$and its corresponding actual parameter at the calling node x be a. Then ActiveDataSlice (f) =ActiveDataSlice (a)∪ ActiveCallSlice until an execution of a Def (f) node in the method m$_1$ takes place. Thus for each var used or defined at an execution node z, DyanSlice (z, var) = ActiveCallSlice∪ActiveDataSlice (var) ∪ActiveControlSlice (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 (varx) node where varx is any variable, then ActiveDataSlice (varx) =ActiveReturnSlice after each 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.

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.

*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;
}

1. *ABC ref1;*
2. *ref1=new ABC( );*
3. *ABC ref2;*
4. *ref2=ref1;*
5. *ABC ref3;*
6. *ref3=new ABC( );*
7. *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

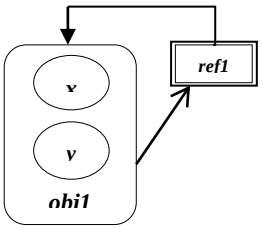

Figure 1. After execution of statement -2

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

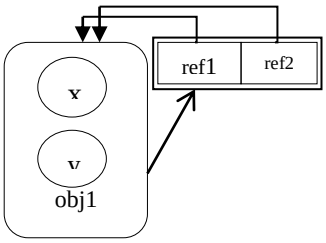

Figure 2. After execution of statement -4

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}

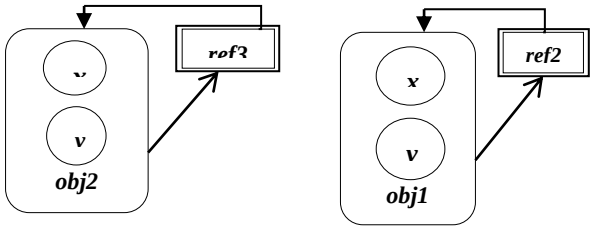

Figure 3. After execution of statement -7

2.6. Inheritance

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 does not exist 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. Polymerphism

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 are chosen for denoting properties over concrete domains, since their mathematical structure guarantees, for each concrete element the existence of the *best correct approximation* in the abstract domain. This is due to 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. An uco ρ: C →C on a poset C is monotone, idempotent, and extensive. UCOs are distinctively calculated by the set ρ(C) of their fix-points. We have used the abstract domain SIGN containing ($\top$), ($\bot$) and the abstract values [neg] ≡ Z^- (negative number) and [pos] ≡ Z^+ (positive numbers including 0). For parity, PAR = {[$\top$], [even], [odd], [$\bot$]} models parity of number. Lastly PARSIGN = PAR ⊓ SIGN it is the smallest domain which is more precise then the others (example: poseven).

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 ρ o f o ρ = ρ o f. In other words, computing f in the abstract domain corresponds precisely to abstracting the concrete computation of f, without further loss of information. E.g., PAR is complete for +, but SIGN is not.

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)*={+, $\bot$,$\bot$}.
After execution of statement 2: *AbstractState(2)*={+, -,$\bot$}.
After execution of statement 3: *AbstractState(3)*={+, - , +}.

Where $\bot$ 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

∀ ε ∈ ∑ρ : $P[[s]]_\rho^\rho(\varepsilon) = \varepsilon$, 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 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. ActiveSemanticSlice(v, ρ)

ActiveSemanticSlice holds only those statements which influence the variable v semantically on the basis of an abstract propert 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, ActiveControlSlice (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.

Let u be a Def (v) statement in program p, the node ActiveCallSlice and ActiveReturnSlice 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.

2.12. DynamicSemanticSlice(v, s, ρ)

Let s be a statement of Program P,v be a variable in the set DefVarSet(s) U UseVarSet(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 ActiveControlSlice(t)(if s is semantically relevant to v), where ControlDepenedentOn(u) = t.

2.13. SemanticReturnSlice

Before each execution of program P, *SemanticReturnSilce= Φ.* Let *x* be a *RETURN* statement in *P*, and *UseVarSet(x)={var$_1$,...,var$_k$}.* Then, before each execution of the *RETURN* node *x*, *SemanticReturnSlice={x}*$\cup$*ActiveCallSlice* $\cup$*ActiveDataSlice(var$_1$)* $\cup$... $\cup$*ActiveDataSlice(var$_k$)* $\cup$ *ActiveControlSlice(t),* $\forall$var$_i$: var$_i$ 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:

(i) use *SemanticReturnSlice* to compute and update relevant run-time information corresponding to the execution of *u*

(ii) update *ActiveCallSlice=Φ*

Let *f* be a formal parameter of the method $m_1$and its corresponding actual parameter at the calling node *x* be *a.* Then *ActiveSemanticSlice (f) =ActiveDataSlice (a)* until an execution of a *Def (f)* node in the method m_1 takes place. Thus for each *var* used or defined at an execution node z, *DyanSemanticSlice (z,var,*ρ*) =ActiveCallSlice*$\cup$*ActiveSemanticSlice(var)* $\cup$*ActiveControl Slice(t),* where z is semantically relevant to var with respect to abstract property ρ and *ControlDependentOn(z)*=t.

Execution of the member method m_1 ends with a *RETURN* node iff its corresponding method call node *y* is a *Def (varx)* node where *varx* is any variable, then *ActiveSemanticSlice(varx) =SemanticReturnSlice* after each execution of the node *y*.

3. ALGORITHM

In this section, we present an efficient property based dynamic slicing algorithm for an Object Oriented program. Let P be an oriented program. To compute slices, we first construct the control flow graph (CFG) of the program P. The algorithm uses the CFG for extracting the control dependency. It is based on computing and updating the run time data structures during execution of the program P.

3.1. Property based dynamic slicing Algorithm for Object Oriented programs

1. Construct the CFG GP of the program P statically only once.
2. Do the following before each execution of the program.
For each statement u of program P do the following
If u is a test (predicate) statement, then ActiveControlSlice (u) = Φ.
Update ControlDependentOn (u)
For each variable var Є DefVarSet (u) U UseVarSet (u) do
DyanSlice (u, var) = Φ.
DynamicSemanticSlice (var, u, ρ) = Φ.
For each variable var of the program P do
ActiveDataSlice (var) = Φ.
ActiveSemanticsSlice (var, ρ) = 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. Run the program P with the given set of input values, and repeat steps 4, 5 and 6 until the program terminates.
4. Do the following before execution of each call statement u.
Let u be a call statement to a method Q.
(a) Update CallSliceStack and ActiveCallSlice.
(b)For each actual parameter var in the procedure call Q does
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. Do the following before execution of each RETURN statement u.
Update ActiveReturnSlice.
Update SemanticReturnSlice
If the return value is an object reference then
Update RefSet ( ) and CurrentRefOf ()
6. Do the following after each statement u of the program P is executed.
(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 does
ActiveDataSlice (var) = Φ.
ActiveSemanticSlice (var) = Φ

Update CallSliceStack and ActiveCallSlice.
Set ActiveReturnSlice = Φ.
SemanticReturnSlice= Φ.
(c) For every variable var Є DefVarSet (u) U UseVarSet (u) do
Update DyanSlice (u, var).
Update DynamicSemanticSlice (var,u, ρ)
(d) If u is a test statement, then update ActiveControlSlice (u).
7. Exit when execution of the program P terminates.

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;
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,ref ∪ 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:

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

int add(int a, int b)
{
11. a=a+b;
12. return(a);
}
int  calculate(int y, int z)
{
int x;
15. x=4y%2 + 2z +6;
16. return(x);
}
int inc(int s)
{
13. t=add(s,1);
14. return(t);
}
```

Now let us consider an actual run of the program (Figure 1) with some inputs such as p= -5, i= 1 and r= 7. Here we are calculating our slices in terms of property ρ= "SIGN". We have to find out the corresponding slices on the basis of given property. We consider the program has not been executed and we are executing it with our set of inputs and property.

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

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

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

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

After execution of node 4: AbstractState (4) ={-,+,+,⊥}. ActiveControlSlice(4) ={2,4},DyanSemanticSlice (i,4,SIGN)={2}.

Before execution of node 5: AbstractState (5) = {-,+,+,⊥}. CallSliceStack=[{-,+,+,⊥ },{2,4,5}],ActiveCallSlice={2,4,5},ActiveSemanticSlice(a,SIGN)={1,2,4,5},ActiveSemanticSlice(b,SIGN)= {2,3,4,5}.

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

After execution of node 11: AbstractState (11)={+,+}. ActiveSemanticSlice(a,SIGN) ={1,2,3,4,5,11},DyanSemanticSlice(a,11,SIGN)={1,2,3,4,5,11},DyanSemanticSlice(b,11,SIGN)={,2,3,4, 5 }

Before execution of node 12 :AbstractState (12)={+,+}.SemanticReturnSlice ={1,2,3,4,5,11,12}

After execution of node 5 : AbstractState (5)={+,+,+,⊥}. CallSliceStack=Φ, ActiveCallSlice=Φ, ActiveSemanticSlice(p,SIGN) ={1,2,3,4,5,11,12}, Dyan SemanticSlice(p,5,SIGN)={1,2,3,4,5,11,12}, DyanSemanticSlice(r,5, SIGN) ={2,3,4}.

After execution of node 6: AbstractState (6)={+,+,+,⊥}. ActiveSemanticSlice(p,SIGN)={1,2,3,4,5,11,12}, DyanSemanticSlice(p,6,SIGN) = {1,2,3,4, 5, 11, 12} (here the statement 6 is not added as it is not changing the abstract property of p) .

Before execution of node 7: AbstractState (7)={+,+,+,⊥}. CallSliceStack=[{+,+,+,⊥ },{1,2,3,4,5,7,11,12}], ActiveCallSlice={1,2,3,4,5,7,11,12}, ActiveSemanticSlice(y,SIGN)={ 1,2,3,4,5,7,11,12}, ActiveSemanticSlice(z, SIGN) ={2,3,4,7}.

Before execution of node 15: AbstractState {+,+,⊥}.

After execution of node 15: AbstractState (15)={+,+,+}. ActiveSemantic Slice(x,SIGN)={2,3,4,7,15},DyanSemanticSlice(x,15,SIGN)={2,3,4,7,15},DyanSemanticSlice(z,11,SIGN)= {2,3,4,7},DyanSemanticSlice(y,11,SIGN)={ 1,2,3,4,5,7,11,12} (as the variable y is not relevant to x as the value of 4y%2=0 so it is discarded)

Before execution of node 16: AbstractState (16)={+,+,+}. SemanticReturnSlice= {2,3,4,7,15,16}

After execution of node 7: AbstractState (7)={+,+,+,+}. CallSliceStack =Φ,ActiveCallSlice=Φ,ActiveSemanticSlice(q,SIGN)={2,3,4,7,15,16},DyanSemanticSlice(q,7,SIGN)={2,3, 4,7,15,16},DyanSemanticSlice(r,7,SIGN)= {2,3,4}.

Before execution of node 8 : AbstractState (8)={+,+,+,+}.CallSliceStack =[{+,+,+,+},{2,4,8}],ActiveCallSlice={2,4,8},ActiveSemanticSlice(s,SIGN)={2,4,8}.

Before execution of node 13 : AbstractState (13)={+,⊥}. CallSliceStack =[{+,+,+,+},{2,4,8},{+,⊥ },{2,4,8,13}],ActiveCallSlice={2,4,8,13} ActiveSemanticSlice (a,SIGN) = {2,4,8,13}.

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

After execution of node 11: AbstractState (11)={+,+}. ActiveSemanticSlice(a,SIGN)={2,4,8,13}, DyanSemanticSlice (a,11,SIGN) ={2,4,8,13}, DyanSemanticSlice (b,11,SIGN) =Φ (as statement 11 does not change the abstract property of a, it is discarded)

Before execution of node 12: AbstractState (12)={+,+}. SemanticReturnSlice ={2,4,8,12,13}

After execution of node 13: AbstractState (13)={+,+}. Call Slice Stack=[{+,+,+,+},{2,4,8}],ActiveCallSlice={2,4,8},ActiveSemanticSlice(t,SIGN) = {2,4, 8,12,13}, DyanSemanticSlice(t,13,SIGN)={2,4,8,12,13} DyanSlice(s,13,SIGN) = {2,4,8}

Before execution of node 14 : AbstractState (14)={+,+}. SemanticReturnSlice ={2,4,8,12,13,14}

After execution of node 8: AbstractState (8)={+,+,+,+}. CallSliceStack=Φ, ActiveCallSlice=Φ,ActiveSemanticSlice(i,SIGN)={2,4,8,12,13,14},DyanSemanticSlice(i,11,SIGN) ={2,4,8,12,13,14}.

After execution of node 9 : AbstractState (9)={+,+,+,+} DyanSemanticSlice(p,9,SIGN)= ={1,2,3,4,5,11,12}

After execution of node 10 : AbstractState (10)={+,+,+,+} DyanSemanticSlice(q,10,SIGN)= {2,3,4,7,15,16}.

3.3. Complexity analysis

The space complexity of our algorithm is mainly due to the space requirement for storing the CFG G of the local part of the program P. If program P has n number of statements then maximum O (n^2) space is required to store the graph G. It can be easily shown that the other data structure used by our algorithm requires maximum O (n^2) space with disposal of the runtime data structures when not required. Even the graph G can be disposed once the control dependency information is captured in the ControDependentOn ( ) data structure. 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. Comarision with related work

The advantage of this algorithm is that, it does not use a trace file to store the execution history. It does not traverse a dependency graph and uses some data structures to capture the runtime dependencies that exist in an object oriented system. Initially our algorithm constructs a Control Flow Graph to capture the control dependencies, used variable set and defined variable sets of a node. As the Control Flow Graph is not traversed by our algorithm even it can be disposed after the information is extracted from it. Our algorithm uses some run time disposable data structures which are updated with execution of each statement in the program. Hence the slices are available before the it is asked for resulting in fast response time.

A pure dynamic slicing algorithm on abstract properties of variable 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 algorithm reported in [16,17] 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.

4. CONCLUSION

In this paper we have proposed an algorithm to extract dynamic slices from an object oriented program. We have addressed almost all the features of an object oriented program and stored some of them statically before the execution of the program. Most of the features are captured dynamically in runtime disposable data structures.

Any slicing algorithm mainly intends to make the slice as small as it can be depending upon slicing criterion. Because the compact size programs are more convenient in terms of practical use In this paper we have used the concepts of Mund and Mall [4] as well as Halder and Cortesi [17]. We have defined new data structures for different purpose. Then we have purposed a new algorithm which finds the minimal slice of in terms of semantics of object oriented programs.

We have unveiled that our algorithms get meticulous dynamic slices on the abstract property. Or we can further say that our calculated slices include only those statements which have impact on slicing criterion. The future scopes of this paper lies in designing a testing tool for the proposed slicing algorithm and investigate the scope for finding dynamic slices with respect to abstract properties in Object Oriented and distributed systems.

REFERENCES

[1] M.Weiser, *Programmers use slices when debugging*, Communications of the ACM, vol.25, no. 7, pp. 446–452, July 1982.
[2] B.Korel and J.Laski, *Dynamic program slicing, Information Processing Letters*, vol. 29, no.3, pp 155–163, October 1988.
[3] G.B.Mund and R.Mall, *Program Slicing, in the Compiler Design Hand Book*, Optimization and Machine Code Generation, pp. 14.1-14.35, CRC Press, 2008.
[4] G.B.Mund and R.Mall, *An efficient interprocedural dynamic slicing method*, The Journal of Systems and Software, vol.79, pp. 791–806, 2006.
[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*, Informationand Software Technology, vol. 44 pp. 123–132, 2002.
[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. 1995. *A survey of program slicing technique*, Journal of Programming Languages vol.3 pp. 121–189, 1995.
[9] R.Halder and A.Cortesi *Abstract slicing of dependence condition graphs*, Science of Computer programming, vol.78 pp.1240-1263, 2013.
[10] S. K. Pani, P. Arundhati and M. Mohanty. *An Effective Methodology for Slicing C++ Programs*. International Journal of Computer Engineering and Technology, vol. 1 pp.72–82, 2010.
[11] J. Zhao. *Dynamic slicing of object-oriented programs*, Technical Report SE-98-119, Information Processing Society of Japan, pp.17–23, 1998.
[12] Z. Chen and B. Xu. *Slicing objected-oriented java programs*, ACM SIGPLAN Notices.vol.36 pp.33–40, 2001.

[13] D. Huynh and Y. Song. *Forward computation of dynamic slices in the presence of structured jump statements*, Proceedings of ISACC'.vol.97 pp. 73–81, 1997.
[14] T.Wang and A. Roy Choudhury. *Using compressed bytecode traces for slicing Java programs*, Proceedings of the IEEE International Conference on Software Engineering. pp. 512–521., 2004.
[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, pp. 125-134, 2008.
[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 pp. 96-121, 2010.
[17] R.Halder and A.Cortesi, *Abstract slicing of dependence condition graphs*, Science of Computer programming, vol.78 pp.1240-1263, 2013.

BIOGRAPHIES OF AUTHORS

Santosh Kumar Pani belongs to School of Computer Engineering, KIIT University, Bhubaneswar Odisha,INDIA. Research area includes program slicing, data flow analysis.

Ganga Bishnu Mund belongs to School of Computer Engineering, KIIT University, Bhubaneswar Odisha,INDIA. Research area includes software Engineering.