An approach to find dynamic slice for C++ Program

Santosh Kumar Pani and Priya Arundhati

Abstract—Object-oriented programming has been considered a most promising method in program development and maintenance. An important feature of object-oriented programs (OOPs) is their reusability which can be achieved through the inheritance of classes or reusable components. Dynamic program slicing is an effective technique for narrowing the errors to the relevant parts of a program when debugging. Given a slicing criterion, the dynamic slice contains only those statements that actually affect the variables in the slicing criterion. This paper proposes a method to dynamically slice object-oriented (00) programs based on dependence analysis. It uses the Control Dependency Graph for object program and other static information to reduce the information to be traced during program execution. In this paper we present a method to find the dynamic Slice of object oriented programs where we are finding the slices for object and in case of function overloading.

Index Terms— Program Slicing, Dynamic Slicing, Control dependency Graph.

1 INTRODUCTION

Program slicing is an effective technique for narrowing the focus of attention to the relevant parts of a program. The slice of a program consists of those statements and predicates that may directly or indirectly affect the variables computed at a given program point [1]. The program point s and the variable set V, denoted by <s, V>, is called a slicing criterion. The basic idea of program slicing is to remove irrelevant statements from source codes while preserving the semantics of the program such that at the specified point the variable produces the same value as its original program. Program slicing has been widely used in many software activities, such as software analyzing, understanding, debugging, testing, maintenance, and so on [2, 3, 4]. Slicing algorithm can be classed according to whether they only use static information (static slicing) or dynamic execution information for a specific program input (dynamic slicing). This paper focuses on the dynamic slicing methods for finding the slices for class, object and in case of function overloading.

2 REVIEW OF RELATED WORKS

Agrawal proposes a dynamic slicing method by marking nodes or edges on a static program dependence graph during execution [5]. The result is not precise, because some dependencies might not hold in dynamic execution. Agrawal also proposes a precise method based on the dynamic dependence graph (DDG) [5], and Zhao applies it to slice object-oriented programs [6]. The shortcoming is that the size of the DDG is unbound.

Korel [8], Song [7] and Tibor [9] propose forward dynamic slicing methods and Song also proposes method to slice OOPs using dynamic object relationship diagram (DORD). In these methods, they compute the dynamic slices for each statement immediately after this statement is executed. After the last statement is executed, the dynamic slices of all statements executed have been obtained. However, only some special statements in loops need to compute dynamic slices. In this paper we represent the object oriented program in control dependence Graph (CDG).

3 BASIC CONCEPT & DEFINITION

Slicing object oriented program is quite difficult as it comprises of many features like Class, Object, Polymorphism, inheritance etc. In this paper we are trying to finding the slices for object and in case of function overloading.

In object oriented programming language it is possible to define more then one function that share a common name this feature is called function overloading. Hence our aim is to find the Slice where the function is overloaded. In that case the Slicer should know that which version of the function to call. For this, the Slicer uses the following information about the function Call.

1. The type of the argument.
2. The no.of argument.
3. The name of the function.

After getting these information from the calling node the unique function will be called and the slice of the function will be found out in normal method. The following Data Structures are used for this purpose.

- Control Dependence Graph: The control dependence graph (CDG) G of an object oriented program P is a graph G=(N,E), where each node nεN represents a statement of the program P. For any
pair of nodes \( x \) and \( y, (x,y) \in E \) if node \( x \) is control dependent on node \( y \).

2. \( \text{Def}(dvar) \): If \( dvar \) is a variable in a program \( P \), a node \( u \) of CDG \( Gp \) is said to be a \( \text{Def}(dvar) \) node if \( u \) is a definition statement that defines the variable \( dvar \).

3. \( \text{Use}(dvar) \): If \( dvar \) is a variable in a program \( P \), a node \( u \) of CDG \( Gp \) is said to be a \( \text{Use}(dvar) \) node if \( u \) is an execution statement that uses the value of the variable \( dvar \).

4. \( \text{DefVarSet}(u)\): Let \( u \) be a node in the CFG \( Gp \) of a program \( P \). The set \( \text{DefVarSet}(u) = \{dvar: dvar \text{ is a variable in the program } P \text{ and } u \text{ is a } \text{Def}(dvar) \text{ node} \} \).

5. \( \text{UseVarSet}(u)\): Let \( u \) be a node in the CFG \( Gp \) of program \( P \). The set \( \text{UseVarSet}(u) = \{dvar: dvar \text{ is a variable in the program } P \text{ and } u \text{ is a } \text{Use}(dvar) \text{ node} \} \).

6. \( \text{ActiveControlSlice}(s)\): Let \( s \) be a test node (predicate statement) in the CDG \( Gp \) of a program \( P \). After each execution of the node \( s \) in the actual run of the program, \( \text{ActiveControlSlice}(s) = \emptyset \).

7. \( \text{ActiveDataSlice}(var)\): Let \( var \) be a data variable in the object oriented program \( P \). Before each execution of the program \( P \), \( \text{ActiveDataSlice}(obj, var) = \emptyset \). Let \( u \) be a \( \text{Def}(obj, var) \) node, and \( \text{UseVarSet}(u) = \{var1, var2, ..., vark\} \). Before execution of the node \( u \) in the actual run of the program, \( \text{ActiveDataSlice}(obj, var) = \{s\} U \text{ActiveDataSlice}(var1) U ... U \text{ActiveDataSlice}(dvark) \) where \( t \) is most recently executed successor node of \( s \) in \( Gp \). If \( s \) is a loop control node and the present execution of the node \( s \) corresponds to exit from loop, then \( \text{ActiveControlSlice}(s) = \emptyset \).

8. \( \text{DyanSlice}(obj)\): Let \( obj \) be an object of the program \( P \). Before execution of program \( P \), \( \text{DyanSlice}(obj) = \emptyset \). Let the object contains the data member \( \text{mem1}. \text{mem2}, ..., \text{memn} \) then \( \text{DyanSlice}(obj) = \text{DyanSlice}(obj, \text{mem1}) U \text{DyanSlice}(obj, \text{mem2}) U ... U \text{DyanSlice}(obj, \text{memn}) \).

9. \( \text{DyanSlice}(s, var)\): Let \( s \) be a node of the CDG of a program \( P \) and \( var \) be a variable in the set \( \{\text{DefVarSet}(s) U \text{UseVarSet}(s)\} \). Before execution of the program \( P \), \( \text{DyanSlice}(s, var) = \emptyset \). After each execution of the node \( s \) in an actual run \( \text{DyanSlice}(s, var) = \text{Active Data Slice}(var) U \text{Active Control Slice}(t) \).

10. \( \text{Active Call Slice}\): Let \( Gp \) represents the CDG of the multiprocedure program \( P \). Before execution of the program \( P \), \( \text{Active Call Slice} = \emptyset \). Let \( u \) be an active Call node then \( \text{Active Call Slice} = \{\text{Active}\} U \text{Active Call Slice} U \text{Active Control Slice}(t) \).

11. \( \text{Call Slice Stack}\): This stack is maintained to keep track of the call nodes to procedures during the actual run of the program.

12. \( \text{Formal}(x, var), \text{Actual}(x, var)\): Let \( p1 \) be a procedure of a program \( P \) having multiple procedures, and \( x \) be a calling node to the procedures, and \( x \) be a calling node to the procedure \( P1 \). Let \( f \) be a formal parameter of procedure \( P1 \) and its corresponding actual parameter at the calling node to the procedure \( P1 \) and its corresponding actual parameter at the calling node \( x \) be \( a \). We define \( \text{Formal}(x, a) = f \) and \( \text{Actual}(x, f) = a \).

### 4 AN ALGORITHM FOR FINDING DYNAMIC SLICE FOR OBJECT ORIENTED PROGRAMS IN CASE OF FUNCTION OVERLOADING.

**Step 1:** Start

**Step 2:** At first Control Dependence Graph \( G \) of object oriented program \( P \) is statically constructed.

**Step 3:** Before each execution of the program do step 4 to step 7.

**Step 4:** For each node \( u \) of \( G \) do step 5 to step 6.

**Step 5:** If \( u \) is a test node, then \( \text{ActiveControlSlice}(u) = \emptyset \).

**Step 6:** For each variable \( var \in \text{DefVarSet}(u) U \text{UseVarSet}(u) \) do \( \text{DyanSlice}(u, var) = \emptyset \).

**Step 7:** Call Slice Stack = NULL.

**Step 7a:** For each data variable \( var \) of the program do

\[ \text{ActiveDataSlice}(var) = \emptyset. \]

**Step 7b:** If \( var \) is a data member of an object, then do

\[ \text{ActiveDataSlice}(obj, var) = \emptyset. \]

**Step 8:** Run the program \( P \) with the given set of input values and repeat steps 9 to 19 until the program terminates.

**Step 9:** Before execution of each call node \( u \) do step 10 and step 11.

**Step 10a:** Let \( u \) be a call node to a procedure \( Q \), update CallSlice Stack and ActiveCallSlice.

/*In case of function having same name the signature of the function is found out as each function have unqiue signature hence the slicer will know that which version of the function to call.*/

**Step 10b:** For each actual parameter \( var \) in the procedure
call Q do ActiveDataSlice (Formal(u,var)) = ActiveDataSlice(var) U ActiveCallSlice.

//Here when a member function of a class is called with an object then the actual parameter of the function is individually copied to the formal parameter of the member function.//

**Step 11:** Update ActiveReturnSlice before execution of the return node.

**Step 12:** After execution of each node u of the program P do step 13 to step 18.

**Step 13:** If u is a Def(var) node and not a call node then Update ActiveDataSlice(var).

//var can be a data variable or data member of an object then Update ActiveDataSlice(var) will be done accordingly //

**Step 14:** If u is a call node to a procedure Q then for every formal reference parameter var in the procedure Q do ActiveDataSlice(Actual(u,var))= ActiveDataSlice(var).

**Step 15:** If u is a Def(var) node then ActiveDataSlice(var)=ActiveReturnSlice.

**Step 16a.** For every variable var if declared as automatic local in the procedure Q do ActiveDataSlice= Ø.

//For local static variable and global variable the ActiveDataSlice remains unchanged

**Step 16b.** Update CallSliceStack and ActiveCallSlice and Set ActiveReturnSlice= Ø.

**Step 17:** For every variable var ε DefVarSet(u) U UseVarSet(u) Update DyanSlice(u,var).

**Step 18:** If u is a test node then update ActivecontrolSlice(u).

**Step 19:** Exit

## 5 SAMPLE PROGRAM AND WORKING OF THE ALGORITHM

Sample Program

The following object oriented Program where only the concept of class and object are used is discussed to illustrate our proposed Algorithm.

```cpp
class test{
  int a;
  int b;
public:
  void get(int x,int y)
  {17.a=x;
   18.b=y;
  }
  void display()
  {19.Cout<<a;
   20.Cout<<b;
  }
  Test add(test tp1,test tp2)
  {21.a=tp1.a+tp2.a;
   22.b=tp1.b+tp2.b;
  }
}
test add(test tp3,int s)
{23.a=tp3.a +s;
  24.b=tp3.b +s;
}
}

void main()
{
  test T1,T2,T3,T4;
  int p,q;
  1.Cout<<"Enter the value of p";
  2.Cin>>&p;
  3.Cout<<"Enter the value of q";
  4.Cin>>&q;
  5.T1.get(p,q);
  6.T1.display();
  7.Cout<<"Enter the value of p";
  8.Cin>>&p;
  9.Cout<<"Enter the value of q";
  10.Cin>>&q;
  11.T2.get(p,q)
  12.T2.display()
  13.T3.add(t1,t2);
  14.T3.display();
  15.T4.add(T3,5);
  16.T4.display();
}
```

### Control Dependency Graph

After Execution of node-2
ADS(p)={2}
DyanSlice(2,p)={2}

After Execution of node-4
ADS(q)={4}
DyanSlice(4,q)={4}

**After Execution of node-5**

Here one of the object of test class is T1.get() is a member function of the test class. The formal parameter of get() is x & y as well as these are the local variable to get(). The actual parameter is p & q. So, ADS(x)=ADS(p) U Active Call Slice
\[ =2 \] U \[ 5=[2,5] \]

ADS(y)=ADS(q) U Active Call Slice
\[ =4 \] U \[ 5=[4,5] \]

Again T1 is calling the get() so here a & b are T1's variable ADS(T1.a)=ADS(x)U[15]=[2,5] U[15]=[2,5,17]
DyanSlice(15,T1.a)=[2,5,17]

ADS(T1.b)=ADS(y)U[16]=[4,5,18]
DyanSlice(16,T1.b)=[4,5,18]

**After Execution of node-6**

Now T1 is calling display() So,DyanSlice(17,T1.a)=ADS(T1.a)={2,5,17}
DyanSlice(18,T1.b)=ADS(T1.b)={4,5,18}
DyanSlice(17,T1.b)={2,5,17} U {4,5,18}={2,4,5,17,18}

**After Execution of node-7**

Here T3 is calling the add() where the formal parameter are tp1 & tp2 but here the object T1 & T2 are passed as the actual parameter. and there are two version of add() function. Now the Slicer will decide which function to call by checking the signature of the function. when this node will be executed the control will go to the first add() function where the argument are the objects. So, ADS(tp1.a)=ADS(T1.a) U Active Call Slice
\[ ={2,5,17} \] U \[ 13 \]
ADS(tp1.b)=ADS(T1.b) U Active Call Slice
\[ ={4,5,18} \] U \[ 13 \]

ADS(tp2.a)=ADS(T2.a) U Active Call Slice
\[ ={8,11,17} \] U \[ 13 \]
ADS(tp2.b)=ADS(T2.b) U Active Call Slice
\[ ={8,11,17} \] U \[ 13 \]

**After Execution of node-8**

ADS(tp3.a)=ADS(T3.a) U Active Call Slice
\[ ={2,5,17,13} \] U \[ 15 \]
ADS(tp3.b)=ADS(T3.b) U Active Call Slice
\[ ={2,5,17,13} \] U \[ 15 \]

**Analysis of the Algorithm**

The improved InterSlice Algorithm also uses a collection of control dependence graphs as the intermediate program representation & computes precise dynamic slice. The space complexity of the improved Interslice algorithm is also O(n^2) The time complexity of this algorithm is linear as no extra data structures are required for this enhancement. This algorithm does not require to store the past execution histo-
7 Conclusion

In this paper of the dynamic Slicing of Object Oriented Programs is discussed. The Object Oriented Programs has various features like Class, Object, Dynamic Binding, Polymorphism, Inheritance etc. In this proposed Algorithm only the Class, Object and function overloading are discussed. Finding the Slices for Inheritance, Dynamic binding and other features are the future work of this Research.

References

[1] M. Weiser, Program Slicing, IEEE Trans. Software Engineering, 1984, 16(5): 498-509.
[2] H. Agrawal, et al. Debugging with Dynamic Slicing and Backtracking. Software – Practice and Experience, 1995, 23(6): 589-616.
[3] B. Korel. Application of Dynamic Slicing in Program Debugging. Third International Workshop on Automated Debugging, 1997: 59-74.
[4] F. Tip. A Survey of Program Slicing Techniques, J. Programming Languages, 1995, 3(3): 121-189.
[5] H. Agrawal, J. Horgan. Dynamic Program Slicing. Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation, 1990: 246-256.
[6] J. Zhao, Dynamic Slicing of Object-Oriented Programs. Technical Report SE-98-119, Information Processing Society of Japan, May 1998: 17-23.
[7] Y. Song, D. Huynh, Forward Dynamic Object-Oriented Program Slicing, Application-Specific Systems and Software Engineering and Technology (ASSET ‘99), IEEE CS Press, 1999: 230 – 237.
[8] B. Korel, S. Yalamanchili. Forward Derivation of Dynamic Slices. Proceedings of the International Symposium on Software Testing and Analysis, 1994, pp. 66-79.
[9] G. Tibor, et al. An Efficient Relevant Slicing Method for Debugging. Software Engineering Notes, Software Engineering Notes – ISEC/FSE’99 Springer ACM SIGSFT, 1999: 303-321.
[10] G.B. Mund, R. Mall, S. Sarkar, 2002a. An efficient dynamic program slicing technique. Information and Software Technology 44(2),123-132.