Paper presented in 28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China

An Edge Marking Technique for Dynamic Slicing of Object-Oriented Programs

Durga Prasad Mohapatra, Rajib Mall and Rajeev Kumar
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur - 721302, India
{durga, rajib, rkumar}@cse.iitkgp.ernet.in
dpmohapatra@nitrkl.ac.in

Abstract

We propose a new dynamic slicing technique for object-oriented programs that is more efficient than the related algorithms. We use an extended system dependence graph (ESDG) as the intermediate program representation. Our dynamic slicing algorithm is based on marking and unmarking the edges in the ESDG as and when dependencies arise and cease during run-time.

1 Introduction

The concept of a program slice was introduced by Weiser [6]. A static program slice consists of those parts of a program that affect the value of a variable v selected at some program statement s. The pair $< s, v >$ is referred to as a slicing criterion. A dynamic program slice contains only those statements that actually affect the value of v at a program statement s for a given input.

Slicing object-oriented programs, presents new challenges which are not encountered in traditional program slicing. To slice an object-oriented program, features such as classes, dynamic binding, encapsulation, inheritance and polymorphism need to be considered carefully. Larson and Harrold were the first to consider these aspects in their work [3].

Efficiency is an especially important concern in slicing object-oriented programs, since the size of object-oriented programs is often very large. With this motivation, in this paper we propose a new dynamic slicing algorithm that is more time and space efficient than the existing dynamic slicing algorithms [5, 7]. We have named our algorithm *edge-marking dynamic slicing* (EMDS) algorithm for object-oriented programs.

The rest of the paper is organized as follows. In section 2, we review the related works. In section 3, we present the edge-marking dynamic slicing algorithm. In section 4, we compare our algorithm with related algorithms. Section 5 concludes the paper.

2 Related Work

Horwitz developed the *system dependence graph* (SDG) as an intermediate program representation and proposed a two-phase graph reachability algorithm on the SDG to compute inter-procedural slice [2]. Larson and Harrold extended the SDG to represent object-oriented programs [3]. Their extended SDG can be used to represent many object-oriented features such as classes and objects, inheritance and polymorphism. After constructing the SDG, they used the two-phase algorithm to compute the static slice. Liang used a more comprehensive intermediate representation by taking the *parameter object* as a tree in which the root of the tree represents the object itself and the leaves represent the data members of the object [4].

All the work discussed above deal with static slicing of object-oriented programs. But research results on dynamic slicing of object-oriented programs have scarcely been reported in the literature [5, 7]. To represent object-oriented programs, Zhao extended the DDG of Agrawal and Horgan [1], to *dynamic object-oriented dependence graph* (DODG) [7]. After constructing the DODG, he used a two-phase algorithm to compute the dynamic slices. A disadvantage of this approach is that the number of nodes in a DODG is equal to the number of executed statements, which may be unbounded for programs having loops.

Song proposed a method to compute forward slices of object-oriented programs using *dynamic object relationship diagram* (DORD) [5]. In this method, the dynamic slices for the variables in each statement is computed immediately after the statement is executed.

```
1:  class Elevator{
        public:
2:        Elevator(int 1_top_floor)    /* initialization for Elevator */
3:           { current_floor = 1;
4:              current_direction = UP;
5:              top_floor = 1_top_floor;  }     /* end of Elevator */
6:         virtual  ~Elevator()  { }
7:         void up()
8:            { current_direction = UP; }
9:         void down()
10:           {  current_direction = DOWN;  }
11:         int which_floor()
12:            { return current_floor;  }
13:         Direction direction()
14:             { return current_direction;  }
15:         virtual void go(int floor)   /* declaration for method go() */
16:             { if  (current_direction = UP )
17:                   { while (current_floor != floor)
                          &&  (current_floor <= top_floor)
18:                      add(current_floor, 1); }
                   else
19:                    { while (current_floor != floor)
                           && (current_floor  > 0 )
20:                      add(current_floor, −1);  } /* end if */
               };
           private:
21:          add(int &a, const int &b) /* This method computes value of current_floor */
22:              {  a = a+b; } ;
           protected:
              int current_floor;
              Direction current_direction;
               int top_floor;
             };
23:         class AlarmElevator: public Elevator {  /* AlarmElevator is derived from  Elevato
           public:
24:            AlarmElevator(int top_floor);
25:                Elevator(top_floor)
26:                    {alarm_on = 0;   }
27:             void set_alarm()
28:                     {alarm_on = 1;  }
29:             void reset_alarm()
30:                      {alarm_on = 0;  }
31:             void go(int floor)
32:                 { if (! alarm_on)
33:                    Elevator :: go(floor);
                       };
            protected:
                 int alarm_on;
            };
34:  main(int argc, char  **argv)  {
           Elevator *e_ptr;
35:        if (argv[1])
36:                e_ptr = new Elevator(10);
             else
37:                e_ptr = new AlarmElevator(10);
38:        e_ptr −> go(3);      /* polymorphic method call */
39:        cout << "\n currently on floor:"
                    << e_ptr −> which_floor();
             } /* end of main */
```

Figure 1: An Example Program

When the last statement is executed, the dynamic slices of all executed statements are obtained. However, such run time slice computation is essential for only some special statements in the loops. For other statements, this approach incurs unnecessary run time overheads. So, computation of dynamic slices using this technique is unnecessarily expensive.

3 EMDS Algorithm

Before presenting our algorithm, we first introduce a few definitions that would be used in the algorithm. In the following definitions and throughout the rest of the paper, we use the terms statement, node and vertex interchangeably.

Definition 1. *Precise Dynamic Slice.* A dynamic slice is said to be *precise* if it includes only those statements that actually affect the value of v at s for the given execution.

Definition 2. *Def(var).* Let *var* be a variable in a class in the program P. A node u of the ESDG of P is said to be a *Def(var)* node if u represents a definition (assignment) statement that defines the variable *var*.

In the ESDG of Fig. 2, nodes 3 and 22 are the Def(current_floor) nodes.

Definition 3. *RecentDef(var).* For each variable *var* in a class, *RecentDef(var)* represents the node (the label number of the statement) corresponding to the most recent definition of *var* with respect to some point s in an execution.

In Fig. 1, RecentDef(current_floor) at statement 17 during the first iteration of the while loop represents the statement 3, where the variable *current_floor* is initialized while during the second iteration it represents statement 22.

3.1 Extended System Dependence Graph(ESDG)

First, we construct the extended system dependence graph (ESDG) of the object-oriented program statically as in [3]. The extended system dependence graph (ESDG) is an extension of the SDG proposed by Horowitz [2]. In addition, the ESDG can model the object-oriented features like inheritance, polymorphism etc. We discuss ESDG in the context of C++ programs. But the ESDG can be easily modified to handle other object-oriented languages such as Java. ESDG models the *main* program together with all non-nested methods. Each class in a given program is represented by a *class dependence graph.* Each method in a class dependence graph is represented by a *procedure dependence graph* as in [2]. Each method has a *method entry node* that represents the entry into the method. The class dependence graph contains a *class entry vertex* that is connected to the method entry node for each method in the class by a special edge known as *class member edge.* To model parameter passing, the class dependence graph associates each method entry node with *formal-in* and *formal-out* vertices. A formal-in vertex is added corresponding to each formal-parameter in the method, and a formal-out vertex is added corresponding to each formal reference parameter that is modified by the method. The class dependence graph uses a *call vertex* to represent method call. At each call vertex, there are *actual-in* and *actual-out* vertices to match the formal-in and formal-out vertices present at the entry to the called method. For example, in Fig. 2 , vertex 18 and vertex 20 represent calls to method *add()*.

To represent inheritance, we construct representations for each method defined by the *derived class*, and reuse the representations of all methods that are inherited from the *base class* [3]. In Fig. 1, the constructor for *AlarmElevator* calls the constructor for *Elevator*. Thus in Fig. 2, the ESDG connects call vertex 25 in *AlarmElevator* to entry vertex 2 in *Elevator* by call edge (25, 2). Virtual method *go()* of *Elevator* is not directly called in *AlarmElevator*; it is redefined in *AlarmElevator* and calls *Elevator::go()*. Thus, our ESDG connects the *call vertex* 33 in *AlarmElevator::go()* to entry vertex 15 in *Elevator::go()*. For call sites 25 and 33, parameter-in and parameter-out edges are added to the ESDG.

A polymorphic method call using dynamic binding occurs when the address of the method to be bound during a method call is unknown at compile time. To represent the polymorphic method call, the ESDG uses a *polymorphic choice vertex* [3]. This *polymorphic choice vertex* represents the dynamic choice among the possible destinations. A call vertex corresponding to a polymorphic call has a call edge incident on a *polymorphic choice vertex*. A *polymorphic choice vertex* has call edges incident on subgraphs that represent calls to each possible method to be bound. In Fig. 2 the vertex *P1* is a *polymorphic choice vertex* that represents a dynamic choice between calls to *Elevator::go* and *AlarmElevator::go*.

3.1.1 Construction of ESDG

ESDG of a complete program is constructed by first creating a partial system dependence graph for the function *main* and then connecting the calls in the partial system dependence graph to methods in the class dependence graph for each class. To do this, we need to connect call vertices to method entry nodes by *call edges* , actual-in vertices to formal-in vertices by *parameter-in edges* and formal-out vertices to actual-out vertices by *parameter-out edges*. The summary edges are added between the actual-in and actual-out vertices at call sites. Fig. 2 shows the ESDG for the example program given in Fig. 1. The ESDG clearly represents the derived class *AlarmElevator* which is inherited from the base class *Elevator*. Also, the ESDG represents the polymorphic method call *go()* at statement 38 in Fig. 1.

3.2 Overview of EMDS Algorithm

Before execution of an object-oriented program P, its extended system dependence graph (ESDG) is constructed statically. During execution of program P, the algorithm marks an edge of the ESDG when its associated dependence exists, and unmarks an edge when its associated dependence ceases to exist. To handle method calls, when a statement invokes a method, the algorithm marks the corresponding call edge between the *call vertex* and the method entry node. Simultaneously, the algorithm marks the corresponding parameter edges between the actual parameter vertices and the formal parameter vertices. Then, the summary edges are marked if the value associated with the actual-in vertex affects the value associated with the actual-out vertex. After the method call is completed and the dynamic slice is recorded at the invoking node, all the marked edges associated with the method entry node are unmarked.

Let *dslice(u)* denote the dynamic slice with respect to the most recent execution of the node u. Let $(x_1, u), \ldots, (x_k, u)$ be all the marked incoming edges of u in the ESDG after an execution of the statement corresponding to node u. Then, it is clear that the dynamic slice with respect to the present execution of the node u is given by:

$$dslice(u) = \{x_1, x_2, \ldots, x_k\} \cup dslice(x_1) \cup dslice(x_2) \cup \ldots \cup dslice(x_k).$$

Once a slicing criterion is specified, the EMDS algorithm computes the dynamic slice with respect to any given slicing criterion by looking up the corresponding *dslice* computed during run time. We now present our EMDS algorithm for object-oriented programs in pseudocode form.

Edge-Marking Dynamic Slicing (EMDS) Algorithm.

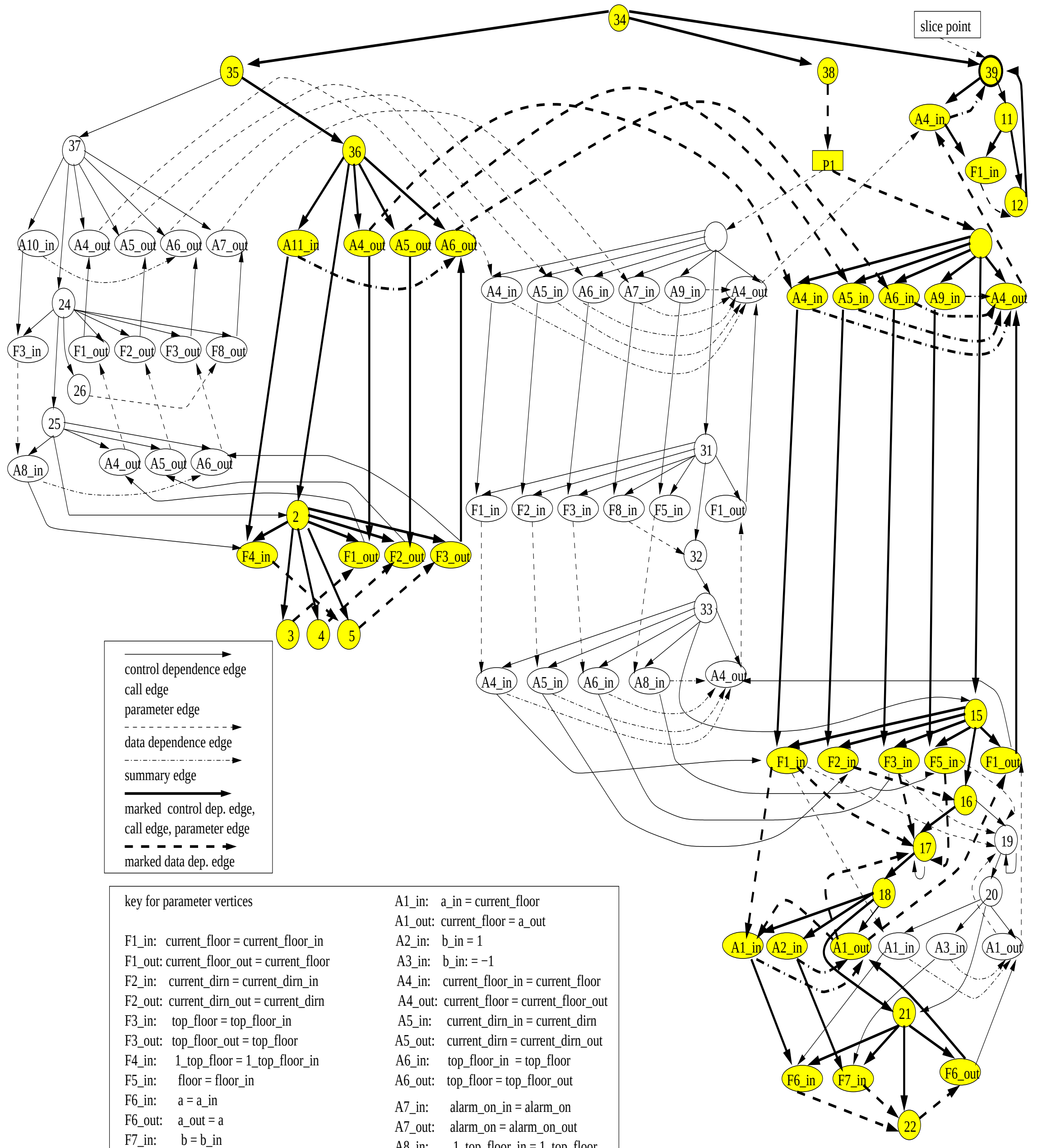
1. **E**SDG Construction: Construct the extended SDG of the object-oriented program P before execution starts.

2. **I**nitialization: Do the following before execution of the program P

   (a) unmark all the edges.

   (b) Set *dslice(u)* $= \phi$ for every node u.

   (c) Set *RecentDef(var)* = NULL for every variable *var* of the program P.



key for parameter vertices

F1_in:	current_floor = current_floor_in	A1_in:	a_in = current_floor
F1_out:	current_floor_out = current_floor	A1_out:	current_floor = a_out
F2_in:	current_dirn = current_dirn_in	A2_in:	b_in = 1
F2_out:	current_dirn_out = current_dirn	A3_in:	b_in: = −1
F3_in:	top_floor = top_floor_in	A4_in:	current_floor_in = current_floor
F3_out:	top_floor_out = top_floor	A4_out:	current_floor = current_floor_out
F4_in:	1_top_floor = 1_top_floor_in	A5_in:	current_dirn_in = current_dirn
F5_in:	floor = floor_in	A5_out:	current_dirn = current_dirn_out
F6_in:	a = a_in	A6_in:	top_floor_in = top_floor
F6_out:	a_out = a	A6_out:	top_floor = top_floor_out
F7_in:	b = b_in	A7_in:	alarm_on_in = alarm_on
F8_in:	alarm_on = alarm_on_in	A7_out:	alarm_on = alarm_on_out
F8_out:	alarm_on_out = alarm_on	A8_in:	1_top_floor_in = 1_top_floor
		A9_in:	floor_in = 5
		A!0_in:	top_floor = 10
		A11_in:	1_top_floor = 10

Figure 2: The updated ESDG of Fig. 1.

3. **R**untime Updations: At run-time, until the program ends or a slicing command is given, carry out the following after each statement s of the program P is executed. Let the node u in ESDG correspond to the statement s.

    (a) For every variable *var* used at node u do the following:

    - i. Unmark the marked dependence edges, if any, associated with the variable *var*, which may have been marked by the previous execution of the node u.
    - ii. Mark the dependence edge (x, u) where x = *RecentDef(var)*.

    (b) Update *dslice(u)* to $dslice(u)\ =\{x_1, x_2, \ldots, x_k\} \cup dslice(x_1) \cup dslice(x_2) \cup \ldots \cup dslice(x_k)$

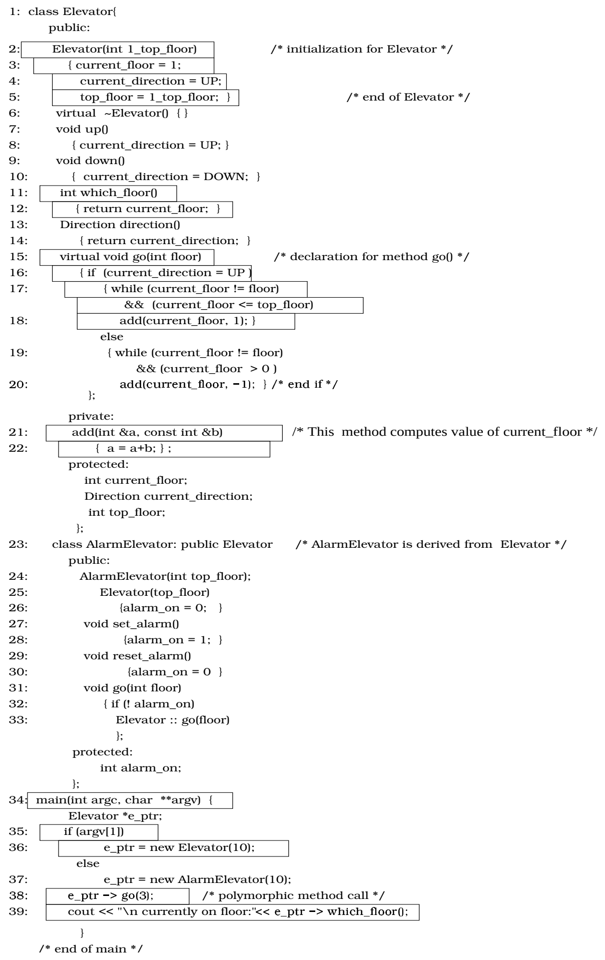
    (c) If u is a *Def(var)* node, then update *RecentDef(var)* = u.

    (d) If u is a method entry node, then do:

    - i. unmark all the marked edges including call edge, parameter edges and summary edges, corresponding to the previous execution of the node u.
    - ii. mark the call edge between the method entry node and the call vertex corresponding to the present execution of the node u.
    - iii. mark the corresponding parameter edges between the actual parameter vertices and the formal parameter vertices.
    - iv. mark the corresponding summary edges between the actual-in and actual-out vertices.

    (e) If u is a node representing the operator *new*, then do:

    - i. unmark the marked call edge between u and the method entry node of it's constructor method, as well as the associated parameter edges and summary edges, corresponding to the previous execution of the node u.
    - ii. mark the call edge between u and the method entry node of it's constructor method corresponding to the present execution of the node u.
    - iii. mark the associated parameter edges between the actual parameter vertices and the formal parameter vertices.



```
1:  class Elevator{
       public:
2:        Elevator(int 1_top_floor)               /* initialization for Elevator */
3:          { current_floor = 1;
4:             current_direction = UP;
5:             top_floor = 1_top_floor;  }                    /* end of Elevator */
6:        virtual  ~Elevator()  { }
7:        void up()
8:           { current_direction = UP; }
9:        void down()
10:          {  current_direction = DOWN;  }
11:        int which_floor()
12:           { return current_floor;  }
13:        Direction direction()
14:           { return current_direction;  }
15:        virtual void go(int floor)            /* declaration for method go() */
16:           { if  (current_direction = UP )
17:               { while (current_floor != floor)
18:                   &&  (current_floor <= top_floor)
18:                  add(current_floor, 1); }
                 else
19:               { while (current_floor != floor)
                      && (current_floor  > 0 )
20:                  add(current_floor, −1);  } /* end if */
           };
       private:
21:       add(int &a, const int &b)          /* This  method computes value of current_floor */
22:          {  a = a+b; } ;
       protected:
          int current_floor;
          Direction current_direction;
           int top_floor;
        };
23:    class AlarmElevator: public Elevator      /* AlarmElevator is derived from  Elevator */
       public:
24:       AlarmElevator(int top_floor);
25:           Elevator(top_floor)
26:               {alarm_on = 0;   }
27:        void set_alarm()
28:               {alarm_on = 1;  }
29:        void reset_alarm()
30:               {alarm_on = 0  }
31:        void go(int floor)
32:            { if (! alarm_on)
33:               Elevator :: go(floor)
               };
       protected:
           int alarm_on;
       };
34: main(int argc, char  **argv)  {
       Elevator *e_ptr;
35:     if (argv[1])
36:          e_ptr = new Elevator(10);
        else
37:          e_ptr = new AlarmElevator(10);
38:     e_ptr −> go(3);           /* polymorphic method call */
39:     cout << "\n currently on floor:"<< e_ptr −> which_floor();
          }
   /* end of main */
```

Figure 3: The dynamic slice of example program of Fig. 1 on slicing criterion (39, current_floor)

    - iv. mark the associated summary edges between the actual-in and actual-out vertices of u.

4. **S**lice Look Up:

    (a) If a slicing command $< s, V >$ is given, carry out the following:

    - i. Look up *dslice(u)* for variable V for the content of the slice. // node u corresponds to statement s.
    - ii. Display the resulting slice.

    (b) If the program has not terminated, go to step 3.

Working of the EMDS Algorithm. We illustrate the working of the algorithm with the help of an example. Consider the C++ program of Fig. 1. Its ESDG is shown in Fig. 2. During the initialization step, EMDS algorithm first unmarks all the edges of the ESDG and sets $dslice(u) = \phi$ for every node u of the ESDG. Now for the input data *argv[1]=3*, the program will execute the statements 34, 35, 36, 2, 3, 4, 5, 38, 15, 16, 17, 18, 21, 22, 17, 18, 21, 22, 17, 39, 11, 12 in order. So, EMDS algorithm marks the edges (34, 35), (35, 36), (36, 2), (2, 3), (2, 4), (2, 5), (34, 38), (38, 15), (15, 16), (16, 17), (17, 18), (18, 21), (21, 22), (34, 39), (39, 11), (11,12). EMDS algorithm also marks the corresponding parameter-in edges and parameter-out edges associated with the actual parameter vertices and formal parameter vertices. All the marked edges in Fig. 2 are shown in bold lines.

Let us assume that a slicing command $<$ 39, *current_floor* $>$ is given at statement 39. This command requires us to find the backward dynamic slice for the variable *current_floor* with respect to call to *which_floor()* at statement *39*. According to EMDS algorithm, the dynamic slice at statement 39 after the second iteration of the while loop, is given by the expression dslice(39) = {39→A4_in, 12, 34} ∪ dslice(39→A4_in) ∪ dslice(12) ∪ dslice(34). During run-time, the slice for each statement is computed immediately after the execution of the statement. So, we are able to get the final dynamic slice at statement 39 by performing a table look up on *dslice(u)*. The statements included in the dynamic slice are shown as shaded vertices in Fig. 2. The dynamic slice is the statements in rectangular boxes in Fig. 3.

3.3 Complexity Analysis

Space complexity. The worst case space complexity of the EMDS algorithm is $O(N^2)$, where N is the number of nodes in the ESDG.
Time complexity. The worst case time complexity of the EMDS algorithm is *O(mN)* where *m* is an upper bound on the number of variables used at any statement.

4 Comparison with Related Work

Larson and Harrold have computed the static slice of an object-oriented program based on the SDG, using a two-pass algorithm [3]. They have not considered dynamic slicing. Zhao [7] computed the dynamic slice of an object-oriented program based on the dynamic object-oriented dependence graph (DODG) [7]. But for programs having loops, the size of the DODG becomes unbounded. Also the worst case space complexity of this algorithm is $O(2^n)$. But the worst case space complexity of our EMDS algorithm is $O(N^2)$, where N is the number of nodes in the ESDG.

The dynamic slicing algorithm of Zhao [7] finds dslice(u) for each occurrence of node u. So, the time complexity becomes exponential in the number of statements for programs having loops. But the worst case time complexity of our EMDS algorithm is $O(mN)$, where m is an upper bound on the number of variables used at any statement. Thus, it is clear that our EMDS algorithm is more space and time efficient than the existing algorithms.

5 Conclusions

Our EMDS algorithm does not require any new nodes to be created and added to the ESDG at run-time nor does it require to maintain any execution trace in a trace file. Besides run-time efficiency of the algorithm, since costly file operations are eliminated, substantial improvement in response time is obtained. Although we have presented our slicing technique using C++ examples, the technique can easily be adapted to other object-oriented languages such as Java. We are now extending this approach to compute the dynamic slice of concurrent object-oriented programs.

References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programmimg Lanuages Design and Implementation, SIGPLAN Notices, Analysis and Verification*, volume 25, pages 246–256, White Plains, NewYork, 1990.

[2] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[3] L. D. Larson and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, German, March 1996.

[4] D. Liang and L. Larson. Slicing objects using system dependence graphs. In *Proceedings of International Conference on Software Maintenance*, pages 358–367, November 1998.

[5] Y. Song and D. Huynh. *Forward Dynamic Object-Oriented Program Slicing, Application Specific Systems and Software Engineering and Technology (ASSET'99)*. IEEE CS Press, 1999.

[6] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[7] J. Zhao. Dynamic slicing of object-oriented programs. Technical report, Information Processing Society of Japan, May 1998.