An Efficient Technique for Dynamic Slicing of Concurrent Java Programs

D.P. Mohapatra, Rajib Mall, and Rajeev Kumar

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, WB 721 302, India
{durga,rajib,rkumar}@cse.iitkgp.ernet.in

Abstract. Program slice has many applications such as program debugging, testing, maintenance and complexity measurement. We propose a new dynamic program slicing technique for concurrent Java programs that is more efficient than the related algorithms. We introduce the notion of *Concurrent Program Dependence Graph* (CPDG). Our algorithm uses CPDG as the intermediate representation and is based on marking and unmarking the edges in the CPDG as and when the dependencies arise and cease during run-time. Our approach eliminates the use of trace files and is more efficient than the existing algorithms.

1 Introduction

The concept of a program slice was introduced by Weiser [1]. A static backward program slice consists of those parts of a program that affect the value of a variable selected at some program point of interest. The variable along with the program point of interest is referred to as a slicing criterion. More formally, a slicing criterion $< s, V >$ specifies a location (statement s) and a set of variables (V).

The program slices introduced by Wesier [1] are now called *static slices*. A static slice is valid for all possible input values. Therefore conservative assumptions are made, which often lead to relatively larger slices. To overcome this difficulty, Korel and Laski introduced the concept of *dynamic program slicing*. A dynamic program slice contains only those statements that actually affect the value of a variable at a program point for a given execution. Therefore, dynamic slices are usually smaller than static slices and have been found to be useful in debugging, testing and maintenance etc.

Object-oriented programming languages present new challenges which are not encountered in traditional program slicing. To slice an object-oriented program, features such as classes, dynamic binding, inheritance, and polymorphism need to be considered carefully. Larson and Harrold were the first to consider these aspects in their work [2].

Many of the real life object-oriented programs are concurrent which run on different machines connected to a network. It is usually accepted that understanding and debugging of concurrent object-oriented programs are much harder compared to those of sequential programs. The non-deterministic nature of concurrent programs, the lack of global states, unsynchronized interactions among processes, multiple threads of control and a dynamically varying number of processes are some reasons for this difficulty.

S. Manandhar et al. (Eds.): AACC 2004, LNCS 3285, pp. 255–262, 2004.

An increasing number of resources are being spent in debugging, testing and maintaining these products. Slicing techniques promise to come in handy at this point. However research attempts in the program slicing area have focussed attention largely on sequential programs. But research reports dealing with slicing of concurrent object-oriented programs are scarce in literature [3].

Efficiency is especially an important concern for slicing concurrent object-oriented programs, since their size is often large. With this motivation, in this paper we propose a new dynamic slicing algorithm for computing slices of concurrent Java programs. Only the concurrency issues in Java are of concern, many sequential Object-Oriented features are not discussed in this paper. We have named our algorithm *edge-marking dynamic slicing* (EMDS) algorithm.

The rest of the paper is organized as follows. In section 2, we present some basic concepts, definitions and the intermediate program representation: *concurrent program dependence graph* (CPDG). In section 3, we discuss our edge-marking dynamic slicing (EMDS) algorithm. In section 4, we briefly describe the implementation of our algorithm. In section 5, we compare our algorithm with related algorithms. Section 6 concludes the paper.

2 Basic Concepts and Definitions

We introduce a few definitions that would be used in our algorithm. In the following definitions we use the terms statement, node and vertex interchangeably.We also describe about the intermediate representation.

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 a variable at a program point for the given execution.

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

In Fig. 2, nodes 2, 9 and 17 are the Def(a2) nodes.

Definition 3. *Use(var) node*. Let *var* be a variable in a class in the program P. A node *x* is said to be a *Use(var)* node iff it uses the variable *var*.

In Fig. 2, the node 4 is a Use(a3) node and nodes 2, 6 and 12 are Use(a2) nodes.

Definition 4. *RecentDef(var)*. For each variable *var*, *RecentDef(var)* represents the node (the label number of the statement) corresponding to the most recent definition of the variable *var*.

Definition 5. *Concurrent Control Flow Graph (CCFG)*. A *concurrent control flow graph* (CCFG) *G* of a program *P* is a directed graph *(N, E, Start, Stop)*, where each node $n \in N$ represents a statement of the program *P*, while each edge $e \in E$ represents potential control transfer among the nodes. Nodes *Start* and *Stop* are unique nodes representing entry and exit of the program *P* respectively. There is a directed edge from node *a* to node *b* if control may flow from node *a* to node *b*.

Definition 6. *Post dominance*. Let *x* and *y* be two nodes in a CCFG. Node *y post dominates* node *x iff* every directed path from *x* to *stop* passes through *y*.

Definition 7. *Control Dependence*. Let *G* be a *CCFG* and *x* be a *test (predicate node)*. A node *y* is said to be *control dependent* on a node *x iff* there exists a directed path *Q* from *x* to *y* such that

- y *post dominates* every node $z \neq x$ in *Q*.
- *y* does not *post dominate x*.

Definition 8. *Data Dependence*. Let *G* be a *CCFG*. Let *x* be a *Def(var)* node and *y* be a *Use(var)* node. The node *y* is said to be *data dependent* on node *x iff* there exists a directed path *Q* from *x* to *y* such that there is no intervening *Def(var)* node in *Q*.

Definition 9. *Synchronization Dependence*. A statement *y* in one thread is *synchronization dependent* on a statement *x* in another thread if the start or termination of the execution of *x* directly determines the start or termination of the execution of *y* through an inter thread synchronization.

Let *y* be a *wait()* node in thread t_1 and *x* be the corresponding *notify()*node in thread t_2. Then the node *y* is said to be *synchronization dependent* on node *x*.
For example, in Fig. 2, node 5 in Thread1 is *synchronization dependent* on node 10 in Thread2.

Definition 10. *Communication Dependence*. Informally, a statement *y* in one thread is *communication dependent* on statement *x* in another thread if the value of a variable defined at *x* is directly used at *y* through inter thread communication.

Let *x* be a *Def(var)* node in thread t_1 and *y* be a *Use(var)* node in thread t_2. Then the node *y* is said to be *communication dependent* on node *x*. For example, in Fig. 2, node 6 in Thread1 is *communication dependent* on nodes 9 and 13 in Thread2.

Definition 11. *Concurrent Program Dependence Graph (CPDG)*. A concurrent program dependence graph (CPDG) G_C of a concurrent object-oriented program *P* is a directed graph (N_C, E_C) where each node $n \in N_C$ represents a statement in P. For *x, y* $\in N_C$, (y,x) $\in E_C$ *iff* one of the following holds:

- *y* is *control dependent* on *x*. Such an edge is called a *control dependence edge*.
- *y* is *data dependent* on *x*. Such an edge is called a *data dependence edge*.
- *y* is *synchronization dependent* on *x*. Such an edge is called a *synchronization dependence edge*.
- *y* is *communication dependent* on *x*. Such an edge is called a *communication dependence edge*.

2.1 Construction of the CPDG

A CPDG of a concurrent Java program captures the program dependencies that can be determined statically as well as the dependencies that may exist at run-time. The dependencies which dynamically arise at run-time are data dependencies, synchronization

dependencies and communication dependencies. We will use different types of edges defined in Definition 11 to represent the different types of dependencies. We use *synchronization dependence* edge to capture dependence relationships between different threads due to inter-thread synchronization. We use *communication dependence edge* to capture dependence relationships between different threads due to inter-thread communication. A CPDG can contain the following types of nodes: (i) *definition (assignment)* (ii) *use* (iii) *predicate* (iv) *notify* (v) *wait*. Also, to represent different dependencies that can exist in a concurrent program, a CPDG may contain the following types of edges: (i) *control dependence edge* (ii) *data dependence edge* (iii) *synchronization dependence edge* and (iv) *communication dependence edge*. We have already defined these different types of edges earlier. Fig. 2 shows the CPDG for the program segment in Fig. 1.

3 EMDS Algorithm

We now provide a brief overview of our dynamic slicing algorithm. Before execution of a concurrent object-oriented program P, its CCFG and CPDG are constructed statically. During execution of the program P, we mark an edge when its associated dependence exists, and unmark when its associated dependence ceases to exist. We consider all the control dependence edges, data dependence edges, synchronization edges and communication edges for marking and unmarking.

Let *Dynamic_Slice (u, var)* with respect to the slicing criterion $< u, var >$ denotes the dynamic slice with respect to the most recent execution of the node *u*. Let $(u, x_1), \ldots, (u, x_k)$ be all the *marked* outgoing dependence edges of *u* in the updated CPDG after an execution of the statement *u*. Then, it is clear that the dynamic slice with respect to the present execution of the node *u*, for the variable *var* is given by :
Dynamic_Slice(u, var) =$\{x_1, x_2, \ldots, x_k\} \cup Dynamic\_Slice(x_1, var) \cup$
$Dynamic\_Slice(x_2, var) \cup \ldots \cup Dynamic\_Slice(x_k, var)$.

Let *var_1*, *var_2*, ..., *var_k* be all the variables used or defined at statement *u*. Then, we define the dynamic slice of the whole statement *u* as :

dyn_slice(u) =$Dynamic\_Slice(u, var\_1) \cup Dynamic\_Slice(u, var\_2)$
$\cup \ldots \cup Dynamic\_Slice(u, var\_k)$.

Our slicing algorithm operates in three main stages:

- Constructing the concurrent program dependence graph statically
- Managing the CPDG at run-time
- Computing the dynamic slice

In the first stage the CCFG is constructed from a static analysis of the source code. Also, in this stage using the CCFG the static CPDG is constructed. The stage 2 of the algorithm executes at run-time and is responsible for maintaining the CPDG as the execution proceeds. The maintenance of the CPDG at run-time involves marking and unmarking the different edges. The stage 3 is responsible for computing the dynamic slice. Once a slicing criterion is specified, the dynamic slicing algorithm computes the dynamic slice with respect to any given slicing criterion by looking up the corresponding *Dynamic_Slice* computed during run time.

```
// In this example, SyncObject is a class, in which there are two synchron
  methods  Swait( ) and Snotify( ).  Swait( ) invokes a wait( ) method and
   Snotify( ) invokes a notify method . CompObject is a class which provide
  a method  mul(CompObject, CompObject).  If a1.mul(a2, a3) is invoked t
  a1 = a2 * a3. The detail codes are not listed here.
    class Thread1 extends Thread {
            private SyncObject  O;
            private CompObject  C;
            void Thread1(SyncObject O, CompObject a1,
                              CompObject a2, CompObject a3);
             {
                this.O=O;
                this.a1= a1;
                this.a2= a2;
                this.a3= a3;
             }
1           public void run( ) {
2            a2.mul(a1, a2); // a2= a1 * a2
3            O.Snotify( );
4            a1.mul(a1, a3); // a1= a1 * a3
5            O.Swait( );
6            a3.mul(a2, a2); // a3= a2*a2
             }
         }
     class Thread2 extends Thread {
            private SyncObject  O;
            private CompObject  C;
            void Thread1(SyncObject O, CompObject a1,
                              CompObject a2, CompObject a3);
             {
                this.O=O;
                this.a1= a1;
                this.a2= a2;
                this.a3= a3;
             }
7            public void run( ) {
8                 O.Swait( );
9                 a2.mul(a1, a1); // a2 = a1 * a1
10                O.Snotify( );
11           if (a1 = a2)
12                     a3 . mul(a2, a1); // a3 = a2 * a1
             else
13                     a2. mul(a1, a1); // a2 = a1 * a1

                  }

           }
14  class example {
15     public static void main(mstring[ ] argm) {
                CompObject a1, a2, a3;
                 SyncObject o1;
                       o1.reset( ); // reset ( ) is a function for initializing Syr
16                      a1 = new  CompObject(Integer.parseInt( argm[0] );
17                      a2 = new  CompObject(Integer.parseInt( argm[1] );
18                      a3 = new  CompObject(Integer.parseInt( argm[2] );
19               Thread1 t1 = new  Thread ( o1, a1, a2, a3);
20                Thread2 t2 = new Thread (o1, a1, a2, a3);
21                 t1.start( );
22                 t2.start( );
                     }
                  }
```

Fig. 1. An Example Program

Working of the EMDS Algorithm: We illustrate the working of the algorithm with the help of an example. Consider the Java program of Fig. 1. The updated CPDG of the program is obtained after applying stage 2 of the EMDS algorithm and is shown in Fig. 3. We are interested in computing the dynamic slice for the slicing criterion $< 6, a3 >$. For the input data *argm[0]=1, argm[1]=1 and argm[2]=2*, we explain how our algorithm computes the slice. We first unmark all the edges of the CPDG and

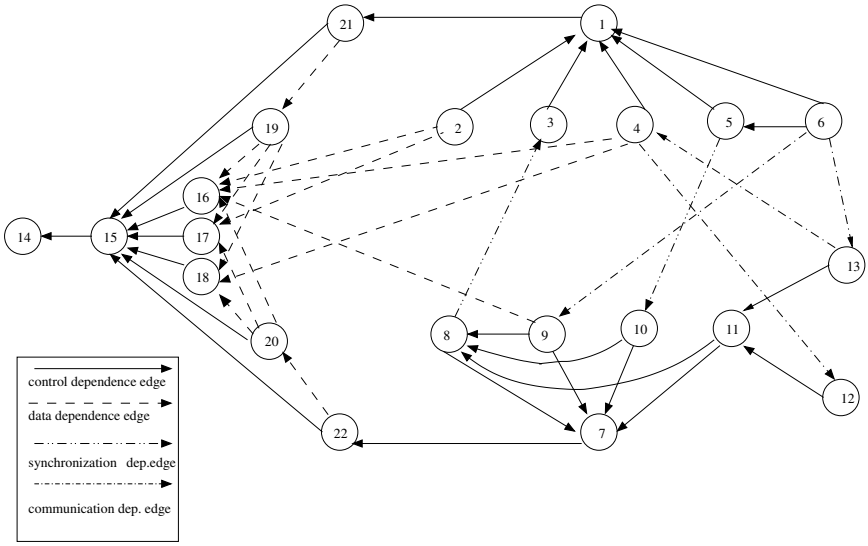

Fig. 2. The CPDG of Fig. 1

set$Dynamic_Slice(u, var)$ =ϕ for every node u of the CPDG. The figure shows all the *control dependence edges* as marked. The algorithm has marked the *synchronization dependence edges* (5, 10) and (8, 3) as *synchronization dependency* exists between statements 5 and 10, and statements 8 and 3. For the given input values, statement 6 is *communication dependent* on statement 9. So, the algorithm marked the *communication dependence edge* (6, 9). All the marked edges in Fig. 3 are shown in bold lines.

Now we shall find the backward dynamic slice computed with respect to the slicing criterion $< a3, 6 >$. According to our edge marking algorithm, the dynamic slice at statement 6, is given by the expression Dynamic_Slice(6, a3) = $\{1, 5, 9\}$ $\cup$ dyn_slice(1) $\cup$ dyn_slice(5) $\cup$ dyn_slice(9). Evaluating the expression in a recursive manner, we get the final dynamic slice at statement 6. The statements included in the dynamic slice are shown as shaded vertices in Fig. 3. Although statement 12 can be reached from statement 6, it can not be included in the slice. Our algorithm successfully eliminates statement 12 from the resulting slice. Also, our algorithm does not include statement 2 in the resulting slice. But by using the approach of Zhao [3], the statements 2 and 12, both would have been included in the slice which is clearly imprecise. So, our algorithm computes precise dynamic slices.

3.1 Complexity Analysis

Space complexity. The space complexity of the EMDS algorithm is $O(n^2)$, where n is the number of executable statements in the program.

Time complexity. The worst case time complexity of our algorithm is *O(mn)*, where *m* is an upper bound on the number of variables used at any statement.

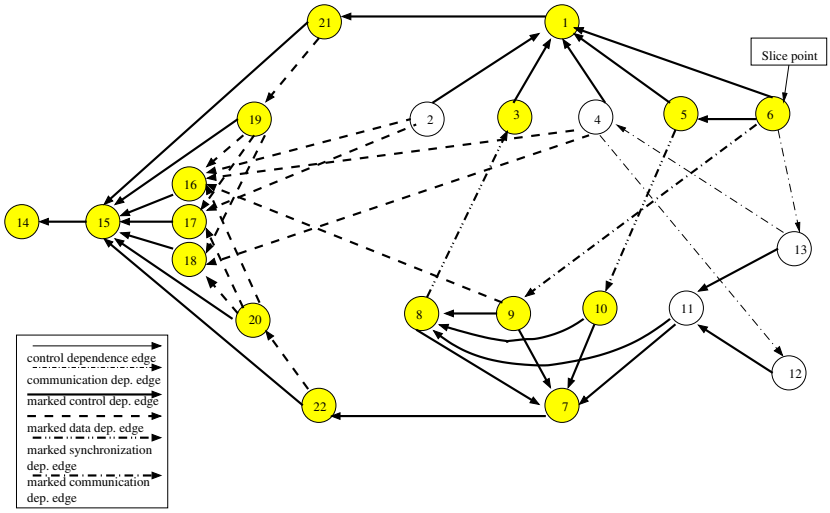

Fig. 3. The updated CPDG of Fig. 1

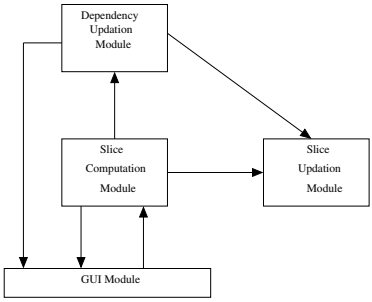

Fig. 4. Module Structure of the Slicer

4 Implementation

The *lexical analyzer* component has been implemented using *lex*. The *semantic analyzer* component has been implemented using *yacc*. The following are the major modules which implement our slicing tool. The module structure is shown in Fig. 4.

- Dependency Updation Module
- Slice Computation Module
- Slice Updation Module
- GUI Module

5 Comparison with Related Works

Zhao computed the static slice of a concurrent object-oriented program based on the *multi-threaded dependence graph* (MDG) [3]. He did not take into account that depen-

dences between concurrently executed statements are not transitive. So, the resulting slice is not precise. Again, he has not addressed the dynamic aspects. Since our algorithm marks an edge only when the dependence exists, so this *transitivity* problem does not arise at all. So, the resulting slice is precise.

Krinke introduced an algorithm to get more precise slices of concurrent object-oriented programs [4]. She had handled the *transitivity* problem carefully. But she has not considered the concept of *synchronization* in her algorithm. But, synchronization is widely used in concurrent programs and in some environment it is necessary. So, krinke's algorithm can not be used in practice. We have considered the synchronization dependence in our algorithm. So, our algorithm can be practically used to compute dynamic slices of most concurrent object-oriented programs like Java.

Chen and Xu developed a new algorithm to compute static slices of concurrent Java programs [5]. To compute the slices, they have used *concurrent control flow graph* (CCFG) and *concurrent program dependence graph* (CPDG) as the intermediate representations. Since they have used static analysis to compute the slices, so the resulting slices are not precise. But, we have performed dynamic analysis to compute the slices. So, the slices computed by our algorithm are precise.

6 Discussion and Conclusions

We have proposed a new algorithm for computing dynamic slices of concurrent java programs. We have named this algorithm *edge-marking dynamic slicing* (EMDS) algorithm. It is based on marking and unmarking the edges of the CPDG as and when the dependences arise and cease at run-time. The EMDS algorithm does not require any new nodes to be created and added to the CPDG at run time nor does it require to maintain any execution trace in a trace file. This saves the expensive node creation and file I/O steps. Further, once a slicing command is given, our algorithm produces results through a mere table-lookup and avoids on-demand slicing computation. Although we have presented our slicing technique using Java examples, the technique can easily be adapted to other object-oriented languages such as C++. We are now extending this approach to compute the dynamic slice of object-oriented programs running parallely in several distributed computers.

References

1. Weiser, M.: Programmers use slices when debugging. Communications of the ACM **25** (1982) 446–452
2. Larson, L.D., Harrold, M.J.: Slicing object-oriented software. In: Proceedings of the 18th International Conference on Software Engineering, German (1996)
3. Zhao, J.: Slicing concurrent java programs. In: Proceedings of the 7th IEEE International Workshop on Program Comprehension. (1999)
4. Krinke, J.: Static slicing of threaded programs. ACM SIGPLAN Notices **33** (1998) 35–42
5. Chen, Z., Xu, B.: Slicing concurrent java programs. ACM SIGPLAN Notices **36** (2001) 41–47