ELSEVIER

Information and Software Technology 44 (2002) 123–132

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

An efficient dynamic program slicing technique

G.B. Mund, R. Mall*, S. Sarkar

Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721 302, India

Received 13 June 2001; revised 11 December 2001; accepted 13 December 2001

Abstract

An important application of the dynamic program slicing technique is program debugging. In applications such as interactive debugging, the dynamic slicing algorithm needs to be efficient. In this context, we propose a new dynamic program slicing technique that is more efficient than the related algorithms reported in the literature. We use the program dependence graph as an intermediate program representation, and modify it by introducing the concepts of *stable and unstable edges*. Our algorithm is based on marking and unmarking the unstable edges as and when the dependences arise and cease during run-time. We call this algorithm *edge-marking algorithm*. After an execution of a node x, an unstable edge (x, y) is marked if the node x uses the value of the variable defined at node y. A marked unstable edge (x, y) is unmarked after an execution of a node z if the nodes y and z define the same variable *var*, and the value of *var* computed at the node y does not affect the present value of *var* defined at the node z. We show that our algorithm is more time and space efficient than the existing ones. The worst case space complexity of our algorithm is $O(n^2)$, where n is the number of statements in the program. We also briefly discuss an implementation of our algorithm.

Keywords: Program slicing; Static slicing; Dynamic slicing; Program dependence graph; Control flow graph; Program debugging

1. Introduction

The concept of a program slice was first introduced by Weiser [1]. A static backward program slice consists of those parts of a program that directly or indirectly 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 $\langle s, V \rangle$ specifies a location (statement s) and a set of variables (V).

The program slices introduced by Weiser [1,2] are now called static program slices because they are computed as the solution to a static analysis problem that does not consider the program input. To find a static slice, we analyze the source code and compute the slice, which is valid for all possible input values. Therefore, we need to make conservative assumptions, which often lead to relatively larger slices. To overcome this difficulty, Korel and Laski [3] 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, program understanding, maintenance, testing, etc. [4–14]. Excellent surveys on the existing slicing algorithms and their applications are available in Tip [15] and Binkley and Gallagher [16]. A comprehensive survey on the existing dynamic program slicing algorithms is available in Korel and Rilling [17].

A major goal of any dynamic slicing technique is efficiency since the results are used during interactive applications such as program debugging. To meet this goal, in this paper, we propose a new dynamic slicing algorithm that is more time and space efficient than the existing dynamic slicing algorithms. We name our algorithm as *edge-marking dynamic slicing algorithm*. Our algorithm uses a modified program dependence graph [18] as the intermediate representation of the program. We introduce the concepts of a *stable edge* and an *unstable edge*. Our algorithm is based on marking and unmarking the unstable edges appropriately as and when dependences arise and cease at run-time. After an execution of the node x at run-time, an unstable edge (x, y) is marked if the node x uses the value of the variable defined at node y. A marked unstable edge (x, y) is unmarked after an execution of a node z if the nodes y and z define the same variable *var*, and the value of *var* computed at the node y does not affect the present value of *var* defined at the node z. We show that our algorithm is more time and space efficient than the existing ones. The worst case space complexity of

* Corresponding author. Tel.: +91-3222-83482; fax: +91-3222-778985.
E-mail addresses: mund@cse.iitkgp.ernet.in (G.B. Mund), rajib@cse.iitkgp.ernet.in (R. Mall), sudeshna@cse.iitkgp.ernet.in (S. Sarkar).

0950-5849/02/$ - see front matter
PII: S0950-5849(01)00224-5

our algorithm is $O(n^2)$, where n is the number of statements in the program.

The rest of the paper is organized as follows. In Section 2, we review the related work. In Section 3, we introduce some preliminary concepts and definitions. In Section 4, we present our edge-marking dynamic slicing algorithm, show that the algorithm always finds *precise dynamic slices*, and that it is more time and space efficient than the existing algorithms. Section 5 concludes our paper.

2. Review of related work

Much of the literature on program slicing is concerned with improving the algorithms for slicing both in terms of reducing the size of the slice and improving the efficiency of slice computation. These works address computation of precise dependence information and the accuracy of the computed slices. The approach of Weiser [1] for intra-procedural static program slicing was based on iteratively solving data-flow equations representing inter-statement influences. Weiser used the control flow graph (CFG) as the intermediate representation for his static slicing algorithm. Later in Ref. [2], Weiser presented a two-phase algorithm for computing inter-procedural slices. Ottenstein and Ottenstein [18] presented a linear time solution to intra-procedural static slicing in terms of graph reachability in the program dependence graph (PDG). Horwitz et al. [19] extended the PDG representation to system dependence graph (SDG) for inter-procedural static slicing. Hwang et al. [20] presented an inter-procedural static slicing algorithm based on replacing recursive calls by instances of the procedure body. The inter-procedural slicing algorithm of Bergeretti and Carré [21] is based on the concepts of information flow relations. Note that inter-procedural slicing algorithms can be used to extract intra-procedural slices as well.

Korel and Laski [3] extended Weiser's static slicing algorithm to the dynamic case. They computed dynamic slices by solving the associated data-flow equations. Their method 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 during the run of the program. Note that for programs containing loops, N may be unbounded. Several works based on these basic methods for computing intra- and inter-procedural dynamic slices of programs have been reported [3,22–30].

Agrawal and Horgan [23] were the first to present algorithms for finding dynamic program slices using PDG. Their first approach to compute dynamic slices uses the PDG as the intermediate representation and marks the nodes of this graph as the corresponding parts of the program are executed for a given set of input values. A dynamic slice is computed by applying the static slicing algorithm of Ottenstein and Ottenstein [18] to the subgraph of the PDG induced by the marked nodes. This approach is imprecise because it does not consider the situations where there exits an edge in the PDG from a marked node u to a marked node v but the definition at v is not used at u. We illustrate this imprecision through an example. Consider the program of Fig. 1 and its PDG of Fig. 2. Let the input value of the variable m be 2, and the input values of x in the first and second iterations of the while loop be 0 and 2, respectively. In the first iteration of the while loop, statement 8 defines a value for y. In the second iteration of the loop, statement 9 defines a value for y without using its previous value, and destroys the previous definition of y. Therefore, the dynamic slice for the slicing criterion $\langle 10, z\rangle$ in the second iteration of the while loop should contain the statement 9 and it should not contain the statement 8. Let us find out the dynamic slice using the first approach of Agrawal and Horgan [23]. We mark the node 8 in first iteration of the loop and node 9 in the second iteration. As node 10 has outgoing dependence edges to both the nodes 8 and 9 in the PDG, both the statements 8 and 9 get included in the dynamic slice, which is clearly imprecise.

The second approach of Agrawal and Horgan [23] marks the edges of the PDG as and when the corresponding dependences arise during program execution. A dynamic slice is computed by applying the static slicing algorithm of Ottenstein and Ottenstein [18] and traversing the PDG only along the marked edges. This approach finds accurate dynamic slices of programs having no loops. In the presence of loops, the slices may sometimes include more statements than necessary because this approach does not consider the fact that execution of the same statement during different iterations of a loop may be (transitively) dependent on

```
   /* This program reads m numbers one by one into the variable x */
   /* and assigns values to the variable y, z, a and b in each iteration */
  /* of the while loop depending on the input value of x in the iteration.*/
       integer m, a, i, b, x, y, z;
1.     read(m);
2.     a=0;
3.     i=1;
4.     b=2;
5.     while (i<=m) do
6.        read(x);              /* Read a number x */
7.        if (x<=0) then   /* Test the value of x and assign value to y */
8.           y=x+5;
          else
9.           y=x-5;
10.       z=y+4;
11.       if (z>0) then  /* Test the value of z and assign value to a or b */
12.          a=a+z;
          else
13.          b=a+5;
14.       i=i+1;    /* Increment the count of numbers read */
       endwhile
15.    write(a);    /* Output the value of a */
16.    write(b)     /* Output the value of b */
```

Fig. 1. Example program 1.

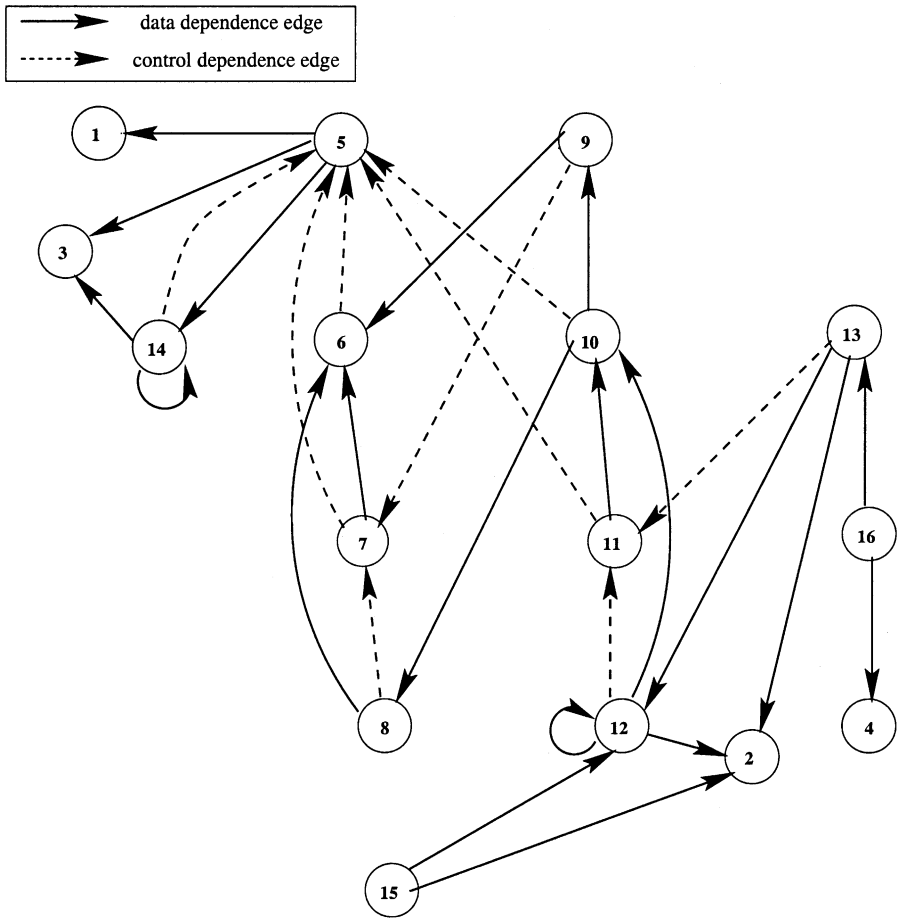

Fig. 2. The PDG of the program of example 1.

different sets of statements. To illustrate this point, consider the example program of Fig. 1 and its PDG of Fig. 2, with the input values $m = 2$, $x = 0$ in the first iteration of the while loop, and $x = 2$ in the second iteration. As already discussed, the dynamic slice for the slicing criterion $\langle 10, z\rangle$ in the second iteration of the while loop should contain the statement 9 and it should not contain the statement 8. Using the second approach of Agrawal and Horgan [23], we mark the outgoing dependence edge (10, 8) in the first iteration of the while loop. In the second iteration, we mark the outgoing dependence edge (10, 9). Traversing along the marked edges, we include both the statements 8 and 9 in the dynamic slice, which clearly results in an imprecise slice.

Agrawal and Horgan [23] pointed out that their second approach to compute dynamic slices produces results identical to that produced by the algorithm of Korel and Laski [3]. Note that the PDG of a program having n statements requires only $O(n^2)$ space. So, the space requirement of Agrawal and Horgan's second algorithm [23] is $O(n^2)$. But the algorithm of Korel and Laski [3] may use unbounded space in the worst case.

The shortcomings of the second approach of Agrawal and Horgan [23] motivated their third approach: construct a dynamic dependence graph (DDG) creating a new node for each occurrence of a statement in the execution history along with the associated dependence edges. The disadvantage of using the DDG is that the number of nodes in a DDG is equal to the number of executed statements (length of execution), which may be unbounded for programs having loops.

In their fourth approach, Agrawal and Horgan [23] proposed to reduce the number of nodes in the DDG by merging nodes whose transitive dependences map to the same set of statements. In other words, a new node is introduced only if it can create a new dynamic slice. This check incurs run-time overhead. The resulting reduced graph is called the reduced dynamic dependence graph (RDDG). The size of the RDDG is proportional to the number of dynamic slices that can arise during execution of the program. Tip [15] has shown that the number of dynamic slices of a program having n statements is $O(2^n)$ in the worst case. The example of Fig. 4 (adopted from Ref. [15]) shows a program having $O(2^n)$ different dynamic slices. The program reads n distinct numbers x_1, x_2,… x_n. For every possible subset $S \subseteq \{x_1, x_2, \ldots, x_n\}$, it finds the sum of all the elements of the subset S. Note that in each iteration of the outer loop, the slice with respect to write(y) will contain

exactly the statements read(x_i) for $x_i \in S$. Since a set with n elements has 2^n subsets, the example of Fig. 4 has $O(2^n)$ different dynamic slices. Thus, the worst case space complexity of the RDDG based algorithm of Agrawal and Horgan [23] is exponential in the number of statements in the program.

All dynamic slicing algorithms discussed in this section manipulate run-time information and need at least $O(N)$ time, where N is the number of executed statements, in order to store the execution history of the program or to update the dependence graphs.

3. Preliminary concepts and definitions

Let *var* be a variable in a program P. In the program P, several statements may define the variable *var*. Let u be a statement that uses the value of the variable *var*. In the PDG G_P of the program P, the node u will have an outgoing dependence edge corresponding to each of the definitions of the variable *var*. For example, in the program of Fig. 1, the statements 8 and 9 define the same variable *y*. In Fig. 2 (the PDG of the program of Fig. 1), node 10 has two outgoing dependence edges (10, 8) and (10, 9) corresponding to the definitions of the variable *y*. Any iteration of the while loop executes exactly one of the statements 8 and 9, and skips the other. Each of the statements 8 and 9 defines the variable *y* afresh without using its previous value directly or indirectly. Therefore, to get a precise dynamic slice for the slicing criterion $\langle 10, z\rangle$ in any iteration of the while loop, we should consider only the edge (10, 8) or the edge (10, 9) depending on whether the statement 8 or the statement 9 is executed in that iteration. Before proceeding any further, we need to define what is meant by a precise dynamic slice. We consider a *dynamic slice 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*. Now we introduce a few definitions that would be used in our dynamic slicing algorithm. In these definitions and throughout the rest of the paper, we use the terms node and statement interchangeably.

Definition 1. *Unstable edge*. Let S be a statement in a program P. An outgoing dependence edge (S, S_i) in the PDG G_P of P is said to be an unstable edge if there exists an outgoing dependence edge (S, S_j) with $S_i \neq S_j$ such that the statements S_i and S_j both define the same variable used at S.

Definition 2. *Stable edge*. An edge of the PDG of a program P is said to be a stable edge if it is not an unstable edge.

Consider the program of Fig. 1 and its MPDG of Fig. 3. The MPDG of a program does not distinguish data and control dependence edges. It contains two types of edges: stable edges and unstable edges. In the example of Fig. 3, the unstable edges are (5, 3), (5, 14), (10, 8), (10, 9), (12, 2), (12, 12), (13, 2), (13, 12), (14, 3), (14, 14), (15, 12), (15, 2), (16, 4), and (16, 13). All other edges are stable edges. If (x, y) is an unstable edge, then the dependence of node x on node y keeps on changing as execution proceeds. That is, it may exist at some point during execution and may not exist at some other point. In such a situation, the edge (x, y) does not represent a stable dependence of node x on node y. If (x, y) is a stable edge, then at every point of execution, node x has dependence on node y.

Definition 3. *Def*(*var*), *DefSet*(*var*). Let *var* be a variable in a program P. A node u of the PDG G_P is said to be a Def(*var*) node if u is a definition (assignment) statement that defines the variable *var*. The set DefSet(*var*) denotes the set of all the Def(*var*) nodes.

Definition 4. *Use*(*var*), *UseSet*(*var*). Let *var* be a variable in a program P. A node u of the PDG G_P is said to be a Use(*var*) node if the statement u uses the value of the variable *var*. The set UseSet(*var*) denotes the set of all the Use(*var*) nodes.

In the example of Fig. 1, the nodes 8 and 9 are the Def(y) nodes, and DefSet(y) = {8, 9}. The node 10 is the only Use(y) node, and UseSet(y) = {10}. The nodes 2 and 12 are the Def(a) nodes, and DefSet(a) = {2, 12}. The nodes 12, 13 and 15 are the Use(a) nodes, and UseSet(a) = {12, 13, 15}.

Definition 5. *Free execution*. Let *var* be a variable such that $|\text{DefSet}(var)| \geq 1$. An execution of a Def(*var*) node u is said to be a free execution with respect to a Def(*var*) node v if node v does not affect the computation of the value of the variable *var* in the execution of node u.

Definition 6. *Free*(*var*). Let *var* be a variable in a program P with $|\text{DefSet}(var)| \geq 1$. Before execution of the program, the set Free(*var*) = ϕ. Let u be a Def(*var*) node that is not a Use(*var*) node. After execution of the node u, the updated set Free(*var*) = {v:v is a Def(*var*) node and the execution of node u is a free execution with respect to the node v}.

To intuitively analyze what is meant by Free execution and Free(*var*), consider the example program of Fig. 5. The nodes 3, 7, 9, 11 and 16 are Def(x) nodes. Consider input values $m = 2$, $y = 36$ in the first iteration of the while loop, and $y = 31$ in the second iteration. In the first iteration, execution of node 16 is not a free execution with respect to the node 3 as it affects the execution of node 16, and Free(x) = $\{7, 9, 11, 16\}$. In the second iteration, the execution of node 16 is a free execution with respect to every Def(x) node as its execution is not affected by any Def(x) node, and Free(x) = $\{3, 7, 9, 11, 16\}$. After having discussed

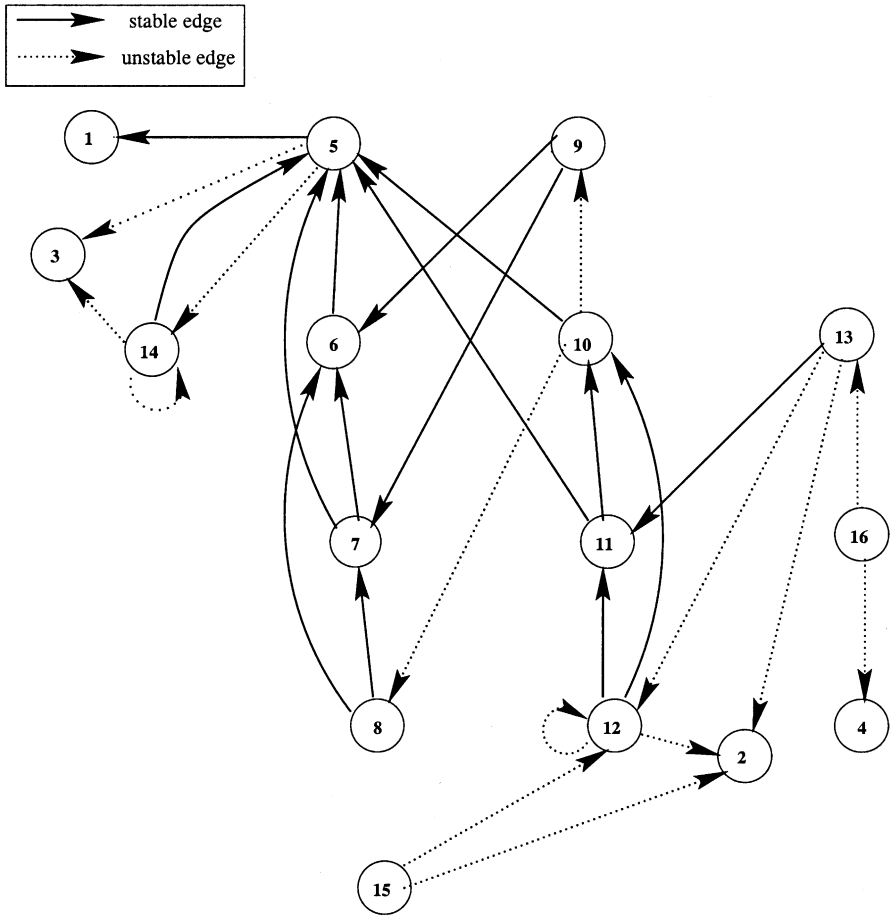

Fig. 3. The MPDG of the program of example 1.

the basic concepts, we now introduce our edge-marking algorithm in Section 4.

4. The edge-marking dynamic slicing algorithm

Before execution of a program P, its MPDG G_P is constructed statically. The type of each of its edges is appropriately recorded either as unstable or as stable. The sets DefSet(*var*) and UseSet(*var*) for all the variables in the program P are computed during construction of the MPDG G_P. The sets DefSet(*var*) and UseSet(*var*) of the variable *var* do not change during run-time of the program P. Note that the dependence associated with a stable edge exists at every point of execution. The dependence associated with an unstable edge keeps on changing as execution proceeds. That is, it may exist at some point and may not exist at some other point during execution of the program. We mark an unstable edge when its associated dependence exists, and unmark when its associated dependence ceases to exist. To compute dynamic slices we need to construct the MPDG of a program statically only once. Each stable edge is marked and each unstable edge is unmarked at the time of constructing the MPDG G_P. Note that during execution of the program, we mark and unmark the unstable edges as and when the dependences arise and cease, and a stable edge is never unmarked. During execution of the program, let *dslice*(u) denote 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 MPDG G_P after an execution of the node u. It is clear that the dynamic slice with respect to the present execution of the node u is $\text{dslice}(u) = \{x_1,\ldots,x_k\} \cup \text{dslice}(x_1) \cup \ldots \cup \text{dslice}(x_k)$. After statically constructing the MPDG G_P of a given program P, our dynamic slicing algorithm can compute dynamic slices with respect to any given slicing criterion. We now present our edge-marking dynamic slicing algorithm.

Algorithm edge-marking dynamic slicing:

1. Do the following before execution of the program P.
   (a) Unmark all the unstable edges.
   (b) Set $\text{dslice}(u) = \phi$ for every node u of G_P.
   (c) Set $\text{Free}(var) = \phi$ for every variable *var* of P with $|\text{DefSet}(var)| \geq 1$.
2. Carry out the following after each statement u of the program P is executed.

```
/* This program reads n numbers and prints the sum of various */
 /* subsets of the set of the read numbers.*/
        integer y, i, x1, .. , xn;
        moreSubsets, finished : boolean;
        read(x1);
         .....
        read(xn);
        moreSubsets=true;
        while (moreSubsets) do
              finished=false;
              y=0;
              while not(finished) do    /*  this loop finds the  */
                    read(i);            /*  sum of the elements  */
                    case(i) of          /*  of a subset of { x1, .., xn}.  */
                       1:  y=y+x1;

                           .....

                       n: y=y+xn;
                    endcase;
                    read(finished);
              endwhile;
              write(y);                 /* Output sum of the current subset */
              read(moreSubsets);        /* Find out whether there are any more subsets . */
        endwhile.
```

Fig. 4. A program having exponential (in the number of statements) number of dynamic slices.

(a) For every variable *var* used at u, mark the unstable edge corresponding to its most recent definition.
(*Node u uses the value computed by the most recent definition of variable *var* in each of its execution*)
(b) Update dslice(u).
(*The set dslice(u) represents the dynamic slice corresponding to the most recent execution of the node u.*)
(c) If u is a Def(*var*) node such that $|\text{DefSet}(var)| \geq 1$ and u is not a Use(*var*) node, then do the following
(i) update Free(*var*) = DefSet(*var*) − dslice(u).
(*The set Free(*var*) consists of all the Def(*var*) nodes that do not affect the present definition of the variable *var*.*)
(ii) unmark every marked unstable edge (v, w) with $v \in$ UseSet(*var*) and $w \in$ Free(*var*).
(*Node w does not affect the present definition of the variable *var*. Therefore, the marked unstable edge (v, w) representing the dependence of node v on node w in the previous execution of node v must not continue to represent the same dependence in the next execution of the node v.*)

We illustrate the working of the algorithm with the help of the following two examples. Consider the example program of Fig. 1 and its MPDG shown in Fig. 3. Consider the input values $m = 2$, $x = 2$ in the first iteration of the while loop, and $x = -9$ in the second iteration. Note that the Def(y) nodes 8 and 9 always have free executions with respect to each other. The first iteration of the while loop executes the statements 9, 10 and 12 among other statements. When the node 10 is executed, we mark the unstable edge (10, 9). When the statement 12 is executed, we mark the unstable edge (12, 2), and then find dslice(12) which includes node 9 among other nodes and does not include node 8, and store it. The second iteration of the while loop executes the statements 8 and 10 among other statements and skips the statements 9 and 12. When the statement 8 is executed in the second iteration of the while loop, we unmark the unstable edge (10, 9). When the statement 10 is executed, we mark the unstable edge (10, 8) corresponding to the most recent definition of the variable y. Therefore, the dynamic slice for the statement 10 in the second iteration of the while loop contains the statement 8 and does not contain the statement 9. When statement 15 is executed after completion of the while loop, we mark the unstable edge (15, 12). As statement 12 was executed in the first iteration and not in the second iteration of the while loop, and the statement 15 is dependent only on the statement 12 of the while loop, the dynamic slice for this execution of the statement 15 should include statement 9 and it should not include the statement 8. Using our edge-marking algorithm, we get the dynamic slice for this execution of 15 as $\{12\} \cup$ dslice(12). As dslice(12) includes 9 and does not include 8, we get a precise dynamic slice.

Now consider the example program of Fig. 5 with the input value $m = 8$ and let the values of y in the first to eighth iterations be -4, 15, -3, 14, 37, 19, 34 and 18. The Def(x) nodes executed in the first to eighth iterations are 7, 11, 7, 11, 16, 11, 16 and 11, respectively. Fig. 7 represents the updated MPDG of Fig. 6 after the sixth iteration of the while loop. In the sixth iteration, the dynamic slice for the slicing criterion $\langle 17, w\rangle$ contains the nodes 16, 13, 11, 7 and 3 which is clearly precise. In the seventh iteration, the Def(z) node 14 has a free execution with respect to the other Def(z) node 13, and the Def(x) node 16 has a free execution with respect to each of the other Def(x) nodes. When node

```
/* This program reads  m  numbers  one by one into  the variable  y    */
/*    and assigns values to the variables x,  z  and  w   in each iteration */
/*  of the while loop depending on the input value of y in the iteration */
      integer m, i, x, y, z, w;
1.    read(m);
2.    i = 1;
3.    x = 10;
4.    while ( i <= m )  do
5.        read(y)                        /* reads a  number in each iteration  */
6.        if ( y <= 0 )  then
7.            x = x+9;
          else
8.            if ( 0 < y  and  y < 5 )  then  /* Test the read number and assign  */
                                              /* value  to the variables x,  z  and  w. */
9.                x =  100;
          else
10.           if  ( 10<y and y<20 )  then
11.               x = x +1;
12.       if ( y<=0 or y>35 )    then
13.           z=x+99;
          else
14.           z=99
15.       if ( 30<y and y<40 )  then
16.           x=z+1;
17.       w=x+10;
18.       i=i+1;                 /* Increment  the count  of  numbers  read. */
      endwhile
```

Fig. 5. Example program 3.

14 is executed, we unmark the unstable edge (16, 13). When the node 16 is executed, we mark the unstable edge (16, 14) corresponding to the most recent definition of the variable z, and unmark each of the unstable edges in the set $\{(7,3), (7,11), (11,7), (11,16), (13,7), (13,11), (17,7), (17,11), (17,16)\}$. When the node 17 is executed, we mark the unstable edge (17, 16) corresponding to the most recent definition of the variable x. When node 11 is executed in the eighth iteration, we mark the unstable edge (11, 16) corresponding to the most recent definition of the variable x. When node 17 is executed, we mark the unstable edge (17, 11) corresponding to the most recent definition of the variable x. Fig. 8 represents the updated MPDG of Fig. 6 after the eighth iteration of the while loop. In the eighth iteration, the dynamic slice corresponding to the slicing criterion $\langle 17, w\rangle$ contains the nodes 16, 14 and 11, and it

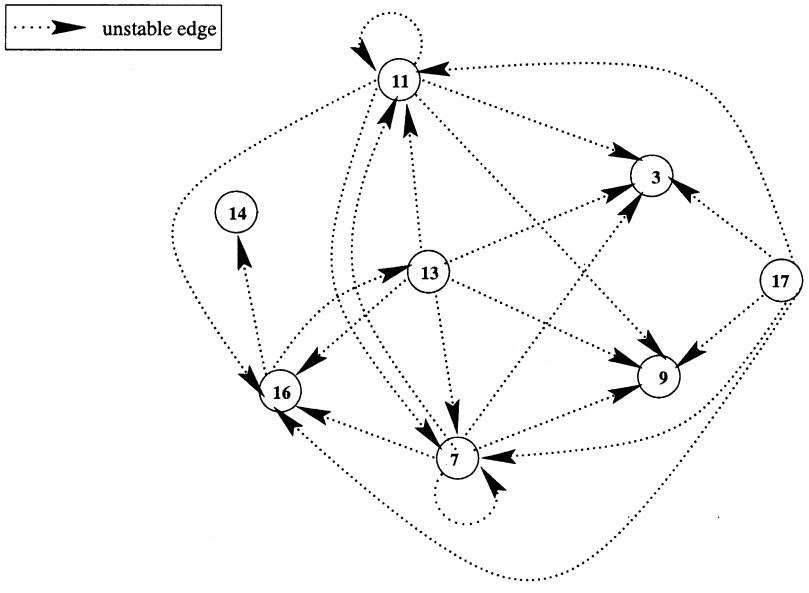

Fig. 6. The induced subgraph of the MPDG of the example program in Fig. 5 on the Node Set{3,7,9,11,13,14,16,17}.

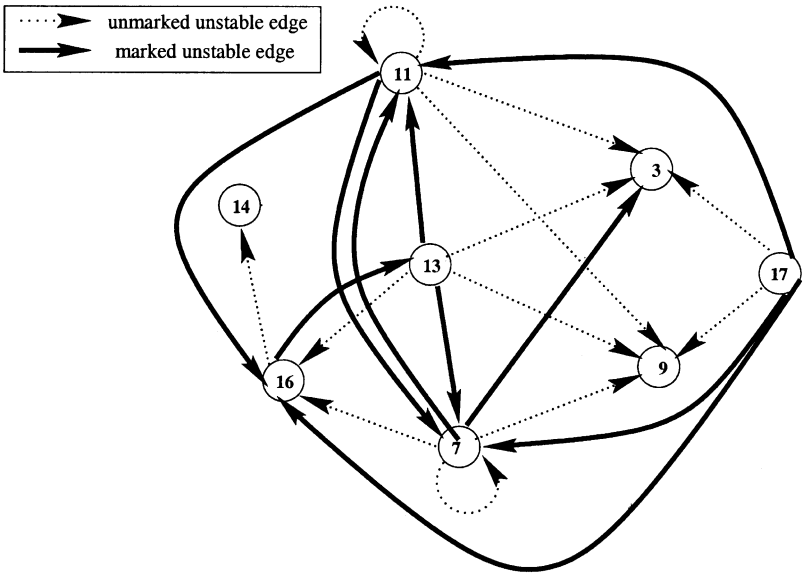

Fig. 7. Updated MPDG of Fig. 6 after sixth iteration.

contains none of the nodes 13, 7 and 3, which is clearly precise.

4.1. Correctness and complexity

In this section, we sketch the proof of correctness of our algorithm. We also show that our algorithm is more efficient than the existing algorithms.

Theorem 1. *The edge-marking algorithm always finds a precise dynamic slice with respect to a given slicing criterion.*

Proof. Let us consider a variable *var* with $|\text{DefSet}(var)| \geq 1$, and u be a Use(*var*) node. Let reachNode(u,*var*) denote the dynamic slice for the slicing criterion $\langle u, var \rangle$, for the most recent execution of the node u. Clearly, reachNode(u, *var*) consists of only those nodes which affect the computation of the most recent definition of the variable *var*. If a node v uses the variable *var* after an execution of a Def(*var*) node s, and no Def(*var*) has a free execution with respect to the node s in between the executions of the nodes s and v, then node v must be (transitively) dependent on node s. Therefore, reachNode(u,*var*) contains a Def(*var*) node s if and only if, either (i) s is the most recently executed Def(*var*) node, or (ii) no Def(*var*) node has a free execution with respect to the node s in between the most recent execution of node s and the present execution of the node u. The steps 2(a), 2(b) and 2(c) of the edge-marking algorithm take

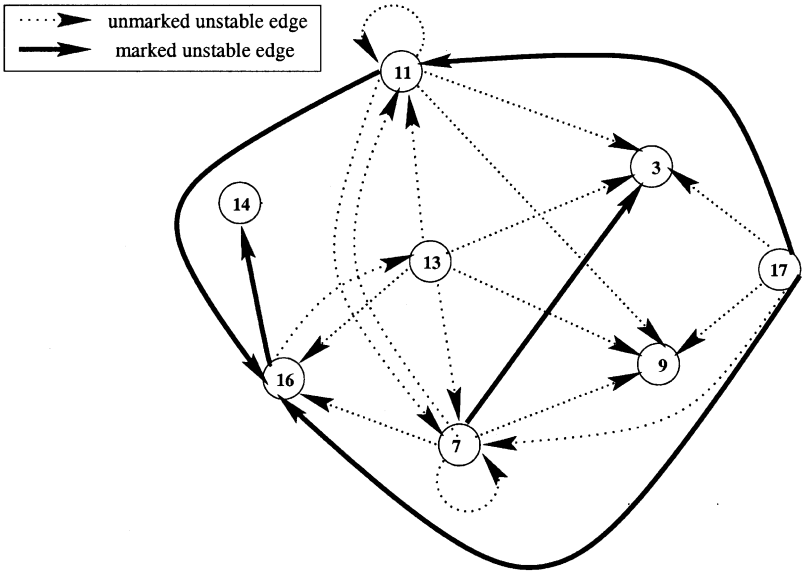

Fig. 8. Updated MPDG of Fig. 6 after eighth iteration.

care of the facts discussed above precisely. Note that, for a Def(*var*) node v of the form $y = f(x_1, \ldots, x_n)$, reachNode(v,y) = reachNode(v,x_1) ∪ … ∪ reachNode(v,x_n). Therefore, it is clear that the edge-marking algorithm finds precise dynamic slices. □

Corollary. *From Theorem 1 the following is easily established. Every node that affects a variable of the slicing criterion is included in the dynamic slice. If a node has not affected any variable of the slicing criterion, then it would not be included in the dynamic slices. This establishes the correctness of the algorithm.*

Theorem 2. *The space complexity of the edge-marking dynamic slicing algorithm is $O(n^2)$, where n is the number of statements in the program.*

Proof. For every node u, the edge-marking algorithm finds dslice(u) and stores it in between successive executions of u. To store dslice(u) for a node u, we need at most $O(n)$ space, where n is the number of statements in the program. Thus to store the sets dslice(u), for all the nodes in the program P, we need at most $O(n^2)$ space. To store the sets DefSet(*var*), UseSet(*var*) and Free(*var*) for all the variables in the program P, we need at most $O(n^2)$ space. Therefore, the worst case space complexity of the algorithm is $O(n^2)$. □

The worst case space complexity of our edge-marking algorithm is $O(n^2)$, whereas the worst case space complexity of RDDG based algorithm of Agrawal and Horgan is $O(2^n)$. The intra- and inter-procedural dynamic slicing algorithms of Agrawal and Horgan [23], Agrawal et al. [25], Kamkar [12] and Kamkar et al. [28] use DDGs with distinct vertices for different occurrence of statements in the execution history. Hence, the worst case space complexity of all these methods is either exponential in the number of statements or unbounded. The dynamic slicing algorithm of Gopal [24] based on information flow relations requires $O(N)$ space to store the execution history, where N is the number of executed statements. The algorithm of Korel and Laski [3] requires $O(N)$ space to store the execution history, and $O(N^2)$ space to store dynamic flow concepts. Thus, it can be easily verified that our edge-marking dynamic slicing algorithm is more space efficient than the existing algorithms.

The dynamic slicing algorithm of Agrawal and Horgan [23] based on the RDDG finds dslice(u) for each occurrence of the node u. If the dynamic slice dslice(u) corresponding to the present execution of the node u is different from all the dynamic slices corresponding to its previous executions, then it creates a new node for this execution along with its required dependence edges and stores the associated dynamic slice. When the number of stored (different) dynamic slices for a node u becomes very large, e.g. the program of Fig. 3, the time required for performing comparisons also increases substantially. In the worst case, time complexity becomes exponential in the number of statements of the program. Our edge-marking algorithm remembers dslice(u) in between successive executions of the node u, and it does not need to compare the dynamic slices. Our edge-marking dynamic slicing algorithm needs at most $O(n^2)$ time, where n is the number of statements in the program, for computing and storing dslice(u) and updating other information corresponding to an execution of a node u. The algorithm of Korel and Laski [3] needs $O(N^2)$ time, where N is the number of executed statements (length of the execution), to compute and store the dynamic flow concepts. The algorithm of Gopal [24] needs $O(N^2v^2)$ time, where N is the number of executed statements and v is the number of variables in the program, to compute the information flow relations. Thus, our edge-marking algorithm is more time efficient than the existing algorithms. It is therefore clear that our algorithm is more space and time efficient than the existing algorithms.

4.2. Implementation

We have implemented our algorithm and also the RDDG based algorithm of Agrawal and Horgan [23]. Our dynamic slicing tools was coded in C and uses the compiler writing tools LEX and YACC. The sample programs for which we computed the dynamic slices were written in a language that is a subset of *C*. The MPDG G_P of a given program P can be constructed statically following the approach of Ottenstein and Ottenstein [18] for constructing the PDG. We store the following additional information.

- The type (unstable or stable) of each edge.
- The set DefSet(*var*) and UseSet(*var*) for each variable *var* in the program P.
- For every variable *var*, pointer to the node (label number of the node) corresponding to its most recent definition at run-time.
- For every node u, the dynamic slice dslice(u) corresponding to its most recent execution at run-time.
- For every variable *var*, the set Free(*var*) at run-time.

We use an adjacency list to store the MPDG of a given program. Every node u points to an array of variables used at the node. Every variable *var* of the array points to a linked list. The linked list represents the set of pointers of Def(*var*) nodes adjacent to the node u. Each element (structure) of the linked list contains a member field for marking and unmarking the corresponding edge, and a member field for storing the type (stable or unstable) of edge. The sets dslice(u), Free(*var*), UseSet(*var*) and DefSet(*var*) are stored using arrays.

We studied the space and time requirements of our edge-marking algorithm and the RDDG based algorithm of Agrawal and Horgan [23] for several sample programs. The performance results of our implementation agree with the

theoretical analysis given in this Section. The performance results show that our algorithm is more time and space efficient than the RDDG based algorithm of Agrawal and Horgan [23]. Further, the space saving of our algorithm compared to Ref. [23] becomes especially marked for programs having a high number of branching statements.

5. Conclusions

In this paper, we have proposed a new algorithm for computing intra-procedural dynamic slices of sequential programs. We have named this algorithm edge-marking algorithm as it is based on marking and unmarking the unstable edges as and when the dependences arise and cease during run-time. We have shown that our algorithm is more time and space efficient than the existing algorithms. We are now extending our approach to handle inter-procedural slicing. For this, we plan to use the SDG as an intermediate representation of the program as proposed in Horwitz et al. [19]. For this, all the *call, parameter-in, parameter-out, and summary* edges of an SDG can be considered as unstable edges. These unstable edges can be marked and unmarked as and when the corresponding dependences arise at run-time. After executing a procedure call and recording the dynamic slice at the calling node, all types of unstable edges associated with the procedure may be unmarked.

Acknowledgements

We acknowledge the constructive suggestions of the anonymous referees which helped us improve the presentation of this paper and simplify the algorithm.

References

[1] M. Weiser, Programmers use slices when debugging, Communications of the ACM 25 (7) (1982) 446–452.
[2] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (4) (1984) 352–357.
[3] B. Korel, J. Laski, Dynamic program slicing, Information Processing Letters 29 (3) (1988) 155–163.
[4] J. Lyle, Evaluating variations on program slicing for debugging, PhD Thesis, University of Maryland, College Park, December 1984.
[5] J.-D. Choi, Parallel program debugging with flowback analysis, PhD Thesis, University of Wisconsin, Madison, August 1989.
[6] A. Podgurski, L.A. Clarke, A formal model of program dependences and its implications for software testing, debugging, and maintenance, IEEE Transactions on Software Engineering 16 (9) (1990) 965–979.
[7] N. Shahmehri, Generalized algorithmic debugging, PhD Thesis, Linkoping University, Sweden, 1991.
[8] K. Gallagher, J. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering SE-17 (8) (1991) 751–761.
[9] H. Agrawal, Towards automatic debugging of computer programs, PhD Thesis, Purdue University, 1992.
[10] T. Shimomura, The program slicing technique and its application to testing, debugging, and maintenance, Journal of IPS of Japan 9 (9) (1992) 1078–1086.
[11] H. Agrawal, R.A. DeMillo, E.H. Spafford, Debugging with dynamic slicing and backtracking, Software—Practice and Experience 23 (6) (1993) 589–616.
[12] M. Kamkar, Interprocedural dynamic slicing with applications to debugging and testing, PhD Thesis, Linkoping University, Sweden, 1993.
[13] R. Mall, Fundamentals of Software Engineering, Prentice Hall, India, 1999.
[14] A.D. Lucia, A.R. Fasolino, M. Munro, Understanding function behaviors through program slicing, Proceedings of the Fourth IEEE Workshop on Program Comprehension, Berlin, Germany, March 1996, IEEE Computer Society Press, Los Alamitos, CA, 1996 pp. 9–18.
[15] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.
[16] D. Binkley, K.B. Gallagher, in: M. Zelkowitz (Ed.), Program slicing, Advances in Computers, vol. 43, Academic Press, San Diego, CA, 1996, pp. 1–50.
[17] B. Korel, J. Rilling, Dynamic program slicing methods, Information and Software Technology 40 (1998) 647–659.
[18] K. Ottenstein, L. Ottenstein, The program dependence graph in software development environment. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, vol. 19, no. 5, 1984, pp. 177–184.
[19] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 26–61.
[20] J. Hwang, M. Du, C. Chou, Finding program slices for recursive procedures, Proceedings of the Twelveth Annual International Computer Software and Applications Conference, COMPSAC'88, Chicago, October 1988, pp. 220–227.
[21] J.-F. Bergeretti, B. Carré, Information-flow and data-flow analysis of while-programs, ACM Transactions on Programming Languages and Systems 7 (1) (1985) 37–61.
[22] B. Korel, J. Laski, Dynamic slicing of computer programs, Journal of Systems and Software 13 (1990) 187–195.
[23] H. Agrawal, J. Horgan, Dynamic program slicing. Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation, SIGPLAN Notices, Analysis and Verification, White Plains, New York, vol. 25, no. 6, 1990, pp. 246–256.
[24] R. Gopal, Dynamic program slicing based on dependence relations, Proceedings of the IEEE Conference on Software Maintenance, 1991, pp. 191–200.
[25] H. Agrawal, R.A. Demillo, E.H. Spafford, Dynamic slicing in the presence of unconstrained pointers, Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV 4), 1991, pp. 60–73.
[26] B. Korel, R. Ferguson, Dynamic slicing of distributed programs, Applied Mathematics and Computer Science 2 (2) (1992) 199–215.
[27] M. Kamkar, N. Shahmehri, P. Fritzson, Interprocedural dynamic slicing, in: M. Bruynooghe, M. Wirsing (Eds.), Proceeding of the Fourth International Conference on Programming Language Implementation and Logic Programming, LNCS 631, vol. 631, Springer, Berlin, 1992, pp. 370–384.
[28] M. Kamkar, N. Shahmehri, P. Fritzson, Three approaches to interprocedural dynamic slicing, Microprocessing and Microprogramming 38 (1993) 625–636.
[29] M. Kamkar, P. Fritzson, N. Shahmehri, Interprocedural dynamic slicing applied to interprocedural data-flow testing, Proceedings of the Conference on Software Maintenance, Montreal, Canada, 1993, pp. 386–395.
[30] B. Korel, Computation of dynamic program slices for unstructured programs, IEEE Transactions on Software Engineering 23 (1) (1997) 17–34.