An Efficient Algorithm for Computing all Program Static Slices

JEHAD AL DALLAL
Department of Information Sciences
Kuwait University
P.O. Box 5969, Safat 13060
KUWAIT

Abstract: - Program slicing is the task of finding all statements in a program that directly, or indirectly, influence the value of a variable occurrence. The set of statements that can affect the value of a variable at some point in a program is called a program backward slice. In several software engineering applications, such as program debugging and measuring the program cohesion and parallelism, several slices are computed at different program points. The existing algorithms for computing program slices are introduced to compute a slice at a program point. In these algorithms, the program, or the model that represents the program, is traversed completely or partially once. To compute more than one slice, the same algorithm is applied for every point of interest in the program. Thus, the same program, or program representation, is traversed several times.
In this paper, an algorithm is introduced to compute all static slices of a computer program by traversing the program representation graph once. Therefore, the introduced algorithm is useful for software engineering applications that require computing program slices at different points of a program. The program representation graph used in this paper is called Program Dependence Graph (PDG).

Key-Words: - Program slicing, static slicing, backward slicing, program dependence graph.

1 Introduction

At a program point *p* and a variable *x*, the slice of a program consists of all statements and predicates of the program that might affect the value of *x* at point *p*. Program slicing can be static or dynamic. In the static program slicing (e.g., [1]), it is required to find a program slice that involves all statements that may affect the value of a variable at a program point for any input set. In the dynamic program slicing (e.g., [2]), the slice is found with respect to a given input set. Many algorithms have been introduced to find static and dynamic slices. These algorithms compute the slices automatically by analyzing the program data flow and control flow. Computing slices of a given procedure is called intra-procedural slicing [1]. Computing slices of a multi-procedure program is called inter-procedural slicing [3]. This paper focuses on computing intra-procedural static slices.

The basic algorithms for computing static intra-procedural slices follow three main approaches. The first approach uses data flow equations (e.g., [1,4]), the second approach uses information-flow relations (e.g., [5]), and the third approach uses program dependence graphs (e.g., [6]). Dependency graph-based slicing algorithms are in general more efficient than the algorithms that use the data flow equations or information-flow relations [7].

Depending on the slicing purpose, slicing can be backward or forward [3]. In the backward slicing, it is required to find the set of statements that may affect the value of a variable at some point in a program. This can be obtained by walking backward over the PDG to find all the nodes that have an affect on the value of a variable at the point of interest. In the forward slicing, it is required to find the set of statements that may be affected by the value of a variable at some point in a program. This can be obtained by walking forward over the PDG to find all the nodes that can be affected by the value of the variable. In this paper, we are interested in backward slicing.

Program slicing is used in several software engineering applications including program debugging [8], regression testing [9], maintenance [10], integration [11], and measuring program cohesion and parallelization [12]. Some of these applications such as program debugging, regression testing, and measuring program cohesion and parallelization require computing slices at different program points.

In program debugging, when an error is detected, it is required to slice the statements that can affect the program point at which the error is detected. In a typical programming, several errors are detected in

each module in the system. Therefore, several slices at different points have to be calculated.

In regression testing, it is required to check that the modifications performed on the system have not caused unintended effects. Each modification might require changes at different program points and it is required to test the slices computed at each of these program points.

Different algorithms that use program slicing are introduced to measure the cohesion of a module in a program. Weiser [1] suggested computing slices for each variable at all program output statements. Longworth [13] suggested computing a slice for each variable in the module. Finally, Ott and Thuss [12] suggested computing a slice for each output variable in the module. The computed slices are used to find different metrics including cohesion and parallelism. As a result, to compute the cohesion and parallelism of a module, it is required to compute several slices of the module.

The above program slicing applications are considered important in the software development process and, therefore, they need an efficient slicing algorithm to speed them up. Unfortunately, no special algorithm is introduced in the literature to serve the above program slicing applications. In this case, the same single-point-based slicing algorithms have to be applied several times and, as a result, the dependency graph has to be traversed several times. This introduces the need for a slicing algorithm that computes all the required slices in more efficient way.

In this paper, an algorithm is introduced to compute all possible static intra-procedural slices of a program. The algorithm requires traversing the PDG once only.

The paper is organized as follows. Section 2 overviews the program dependence graph. In Section 3, the algorithm for computing all static slices is introduced. Section 4 illustrates how to apply the algorithm using a sample example. Finally, Section 5 provides conclusion and discussion of future work.

2 The Program Dependence Graph

The program dependence graph (PDG) consists of nodes and direct edges. Each program simple statement and control predicate is represented by a node. Simple statements include assignment, read, and write statements. Compound statements include conditional and loop statements and they are represented by more than one node. There are two types of edges in a PDG: data dependence edges and control dependence edges. A data dependence edge between two nodes implies that the computation performed at the node pointed by the edge directly depends on the value computed at the other node. This means that the pointed node has the definition of the variable used in the other node. A control dependence edge between two nodes implies that the result of the predicate expression at the node pointed by the edge decides whether to execute the other node or not. Figure 1 shows a C function example. The function computes the sum, average, and product of numbers from 1 to *n* where *n* is an integer value greater than or equal to 1. Figure 2 shows the PDG of the C function example given in Figure 1. The number associated with each PDG node is called node identifier. For simplicity, in this paper, the node identifier indicates the line numbers of the statements that are represented by the node. Solid and doted direct edges represent the control and data dependency edges, respectively.

```
1   void NumberAttributes(int n, int &sum,
            double &avg, int &product) {
2       int i=1;
3       sum=0;
4       product=1;
5       while (i<=n) {
6          sum=sum+i;
7          product=product*i;
8          i=i+1;
9       }
10      avg=static_cast<double>(sum)/n;
11  }
```

Figure 1: C function example

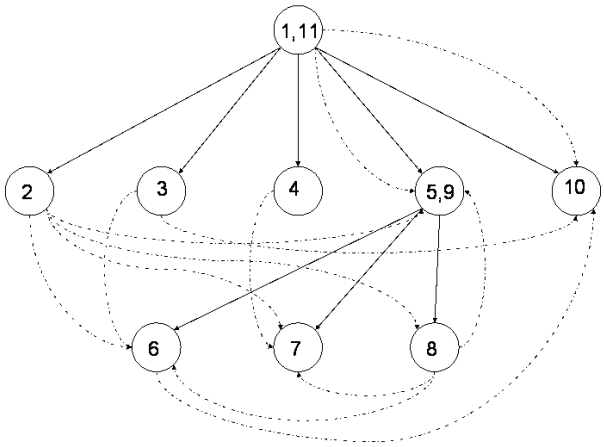

Figure 2: PDG of the C function example given in Figure 1

Using the PDG shown in Figure 2, we can obtain the backward slices. For example, to obtain the backward slice of variable *i* at line 5 of the C

function given in Figure 1, we first add the node that represents line 5 to the slice. This implies adding lines 5 and 9 to the slice. Then, we traverse the incoming edges to node 5 backwards and add lines represented by the nodes attached to the incoming edges to the slice. This results in adding lines 1, 2, 8, and 11 to the slice. The same process is performed for the nodes that represent the added lines of code until we reach nodes with no incoming edges. As a result, the backward slice calculated for variable *i* at line 5 contains the C function lines of code numbered 1,2, 5, 8, 9, and 11.

3 Computing All Static Slices Algorithm

The algorithm for computing all intra-procedural static slices of a module is given in Figure 3 and named *ComputeAllSlices* algorithm. Each node in the PDG is associated with an empty set before applying the algorithm. After the algorithm is applied, the set associated with a node *n* consists of the lines of code included in the slice computed at node *n*. The algorithm builds the set associated with each node in the PDG incrementally as the function called *ComputeASlice* is applied recursively. The *ComputeASlice* function takes a node *n* as an argument. If the node is not already visited, the node is marked visited, the node identifier is added to the set associated with node *n*, and all incoming edges to node *n* are traversed backwards. If an incoming edge is attached to a visited node *v*, the node identifiers included in the set associated with node *v* are added to the set associated with node *n*. Otherwise, if the incoming edge is attached to a not visited node *m*, node *m* is passed as an argument to the *ComputeASlice* function. The function finds the set of nodes included in the slice computed at node *m*. After that, the node identifiers included in the set associated with node *m* are added to the set associated with node *n*.

The algorithm requires performing four necessary preparations before applying the *ComputeASlice* function as follows.

1. Adding an exit node to the PDG. The exit node is a node that has no outgoing edges. Since a node in a PDG represents a program code, an exit node represents a program code on which no other code depends. When the *ComputeAllSlices* algorithm is applied, the exit node is passed as an argument to the *ComputeASlice* function as shown in the second step of the algorithm given in Figure 3. Since a PDG can have several exit nodes, we are in need for a unique exit node to start with. As a result, a special exit node is added to represents the end of the program, or module, and a control flow edge is added from each of the exit nodes in the PDG to the added exit node. In the PDG given in Figure 2, nodes 7 and 10 have no outgoing edges and, therefore, they are exit nodes. As shown in Figure 4, a new node, node 12, is added to be the special exit node and two control dependence edges are added from nodes 7 and 10 to node 12. In this case, node 12 is first passed as an argument to the *ComputeASlice* function to compute all static slices of the C function given in Figure 1.

Input: A PDG that has a single exit node, an empty set of node identifiers associated with each node, and all nodes contained in a cycle combined in one node.
Output: The PDG that each of its nodes is associated with a set of identifiers of certain nodes. These certain nodes represent the lines of code contained in the computed backward slice.
Algorithm:

1. Mark all PDG nodes as not visited
2. *ComputeASlice*(exit node)

```
ComputeASlice(node n) {
   if node n is not visited
        Mark node n as visited
        Add the identifier of node n to the set
          associated with node n
        for each node m in which node n
          depends
             ComputeASlice(m)
             Add the contents of the set
             associated with node m to the set
             associated with node n
}
```

Figure 3: The *ComputeAllSlices* algorithm

2. Combining all nodes contained in each cycle in the PDG in one node. Having a cycle between two or more nodes in the PDG implies that each of the nodes depends directly or indirectly on the other nodes in the cycle. This results in having same slice contents for each of the nodes in the cycle. Therefore, combining the nodes in a graph cycle in one node does not change the slicing results. However, having cycles in the graph leads to an infinite recursion when *ComputeASlice* function is applied. Combining nodes in a cycle is performed by replacing the nodes by a new node. All incoming edges to

each of the combined nodes are redirected to be incoming edges to the new node. Similarly, all outgoing edges from each of the combined nodes are redirected to be outgoing edges from the new node. Finally, any resulting self-loop edge is removed because such an edge is not considered when computing program slices. In the PDG given in Figure 2, the two nodes that represent lines 5, 8, and 9 are contained in a cycle. Therefore, as shown in Figure 4, the two nodes are replaced by the node labeled 5,8,9. All incoming edges to the nodes that represent lines 5, 8, and 9 are redirected to be incoming edges to the new node. All outgoing edges from the nodes that represent lines 5, 8, and 9 are redirected to be outgoing edges from the new node. This results in having two self-loop edges linked to the new node, and these edges are removed.

3. Associating an empty set with each node in the PDG. When the algorithm is applied, the set associated with each node contains the identifiers of the nodes that represent the program slice at the program point represented by the node.
4. Marking all nodes in the PDG as not visited. After applying the *ComputeAllSlices* algorithm and computing all slices, all nodes are marked visited.

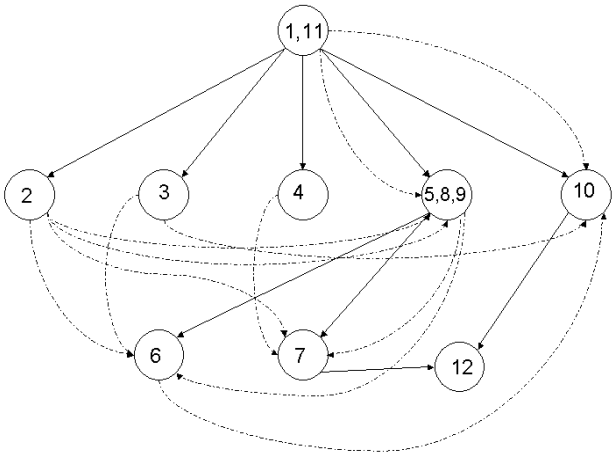

Figure 4: The PDG prepared for applying the *ComputeAllSlices* algorithm. The PDG is derived from the PDG given in Figure 2.

ComputeAllSlices algorithm ensures that each edge is not traversed more than once by marking a traversed node is visited. Nodes initially are marked as not visited. Whenever a node is passed as an argument to the *ComputeASlice* function, it is checked to be either marked visited or not. If the node is not marked as visited, the node is marked as visited and all its incoming edges are traversed. If the node is marked as visited, the *ComputeASlice* function is terminated without traversing the incoming edges. As a result, the incoming edges of any node are traversed once when the node is first passed as an argument to the *ComputeASlice* function. Therefore, when the *ComputeAllSlices* algorithm is applied, no edges are traversed more than once.

4 Example

As an example, a slice is to be computed at each line in the C function given in Figure 1. Figure 4 shows the updated PDG as discussed in Section 3. To compute the backward static slices, the *ComputeAllSlices* algorithm is applied and node 12 is passed as an argument to the *CompueASlice* function. Since node 12 is initially marked as not visited, its marked now as visited and the node identifier "12" is added to the slice set of node 12. Nodes 7 and 10 are linked by direct edges to node 12 and, therefore, the *ComputeASlice* function is applied for both of them. After computing their slices by recursively applying the *ComputeASlice* function, the set of identifiers associated with each of the two nodes is added to the set of identifiers associated with node 12. The resulting contents of sets of identifiers associated with each of the PDG nodes are listed in Table 1. These contents are computed using the *ComputeAllSlices* function.

Line of code	Slice contents
1	1,11
2	1,2,11
3	1,3,11
4	1,4,11
5	1,2,5,8,9,11
6	1,2,3,5,6,8,9,11
7	1,2,4,5,7,8,9,11
8	1,2,5,8,9,11
9	1,2,5,8,9,11
10	1,2,3,5,8,9,10,11
11	1,11
12	1,2,3,4,5,6,7,8,9,10,11,12

Table 1: The slice contents computed for each line of code of the function given in Figure1. The contents of the slices are computed using the *ComputeAllSlices* algorithm.

5 Conclusions and Future Work

In this paper, an algorithm is introduced to compute all static backward slices of a program by traversing

the PDG that represents the program once. The algorithm uses a recursive function to incrementally compute the slices as the PDG is traversed. The algorithm is useful for software engineering applications that require computing slices at different program points. In this case, the PDG is traversed once to find all slices instead of traversing the graph several times using other algorithms. This introduced algorithm is limited to compute backward slices for intra-procedural programs only.

In future, we plan to extend the algorithm to compute all forward and backward slices for both intra- and inter-procedural programs. In addition, we plan to extend the algorithm to compute all slices for object-oriented programs. Finally, we plan to develop a prototype tool and use it to compare the efficiency of our algorithm with the efficiency of applying the single-point-based slicing algorithms.

References:

[1] M. Weiser, Program slicing, *IEEE Transactions on Software Engineering*, 1984, 10(4), pp. 352-357.

[2] B. Korel and J. Laski, Dynamic slicing of computer programs, *The Journal of Systems and Software*, 1990, 13(3), pp. 187-195.

[3] S. Horwitz, T. Reps, and D. Binkley, Interprocedural slicing using dependence graphs, *ACM Transactions on Programming Languages and Systems*, 1990, 12(1), pp. 26-60.

[4] P. Hausler, Denotational program slicing, *In Proceedings of the 22nd Hawaii International Conference on System Sciences*, Hawaii, 1989, pp. 486-494.

[5] J. Bergstar and B. Carre, Information-flow and data flow analysis of while-programs, *ACM Transactions on Programming Languages and Systems*, 7(1), 1985, pp. 37-61.

[6] K. Ottenstein and L. Ottenstein, The program dependence graph in software development environment, *In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SIGPLAN Notices 19(6), 1984, pp. 177-184.

[7] F. Tip, A survey of program slicing techniques, *Technical Report: CS-R9438, CWI (Centre for Mathematics and Computer Science)*, Amsterdam, The Netherlands, 1994.

[8] M. Weiser, Programmers use slices when debugging, *Communications of the ACM*, 1982, 25, pp. 446-452.

[9] R. Gupta, M. Harrold, and M. Soffa, An approach to regression testing using slicing, *Proceedings of the International Conference on Software Maintenance*, 1992, pp. 299-308.

[10] K. Gallagher and J. Lyle, Using program slicing in software maintenance, *IEEE Transactions on Software Engineering*, 1991, 17(8), pp. 751 – 761.

[11] S. Horwitz, J. Prins, and T. Reps, Integrating non-interfering versions of programs, *ACM Transactions on Programming Languages and Systems*, 1989, 11(3), pp. 345-387.

[12] L. Ott and J. Thuss, Slice based metrics for estimating cohesion, *Proceedings of the IEEE-CS International Metrics Symposium*, 1993, pp. 78-81.

[13] H. Longworth, Slice based program metrics, *Master's thesis, Michigan Technological University*, 1985.