On Slicing Programs with Jump Statements

Hiralal Agrawal
Bellcore
445 South Street
Morristown, NJ 07960
hira@bellcore.com

Abstract

Program slices have potential uses in many software engineering applications. Traditional slicing algorithms, however, do not work correctly on programs that contain explicit jump statements. Two similar algorithms were proposed recently to alleviate this problem. Both require the flowgraph and the program dependence graph of the program to be modified. In this paper, we propose an alternative algorithm that leaves these graphs intact and uses a separate graph to store the additional required information. We also show that this algorithm permits an extremely efficient, conservative adaptation for use with programs that contain only "structured" jump statements.

1 Introduction

A *program slice* of a program, P, with respect to a variable, *var*, and a location, *loc*, is a subprogram, P', of P such that P' computes the same value(s) of *var* at *loc* as that computed by P [29]. Program slices have applications in many areas including program understanding, testing, debugging, maintenance, optimization, parallelization, and integration of program versions (see, e.g., [1, 2, 12, 16, 19, 29]).

This paper was published in the *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 20–24, 1994, pages 302–312.

A preliminary position paper on the work reported here was presented at the *Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Slicing, Merging and Integration*, held at Naval Postgraduate School, Monterey, California, October 13–15, 1993.

Algorithms to compute program slices are based on finding the transitive closure of the data and control dependences of the appropriate statement(s) in the program [17, 24]. Although the algorithms to compute control dependences in the presence of jump statements have been available for some time [9, 10, 14], the algorithms to decide which jump statements themselves to include in a slice have obtained relatively little attention (see Section 5, Related Work). Even the "structured" derivatives of the `goto` statement such as the `break`, `continue`, and `return` statements, as in C, have not been adequately considered in this context[1]. Not surprisingly, use of program slices without the relevant jump statements included in them may produce misleading results in many applications mentioned above.

Two similar algorithms to determine which jump statements to include in a slice were proposed recently by Ball and Horwitz [5] and Choi and Ferrante [8]. Both algorithms require that the control dependence graph of a program be constructed from an "augmented" control flowgraph of the program while the data dependence graph be constructed from the the standard control flowgraph. In this paper we present an alternative algorithm that does not require the construction of the augmented flowgraph. Instead, it requires that a separate "lexical successor" tree of the program be maintained. The algorithm works by first finding the statements included in the slice by the conventional slicing algorithm and then determining the appropriate jump statements to be included in the slice. A jump statement is added to the slice if its nearest postdominator in the slice is different from its nearest lexical successor in the slice.

Our algorithm has the same precision as that of Ball and Horwitz and Choi and Ferrante. It is, however, appealing in that it leaves the flowgraph and the program dependence graph of the program intact and uses a separate graph to store the additional required infor-

[1] We use the term "jump statement" here to refer to both the `goto` statement as well as its structured derivatives such as the `break`, `continue` and `return` statements.

```
 1: sum = 0;
 2: positives = 0;
 3: while (!eof()) {
 4:     read(x);
 5:     if (x <= 0)
 6:          sum = sum + f1(x);
        else {
 7:          positives = positives + 1;
 8:          if (x%2 == 0)
 9:               sum = sum + f2(x);
             else
10:               sum = sum + f3(x);
        }
    }
11: write(sum);
12: write(positives);
```

(a) an example program

```
 2: positives = 0;
 3: while (!eof()) {
 4:     read(x);
 5:     if (x <= 0)
        else {
 7:          positives = positives + 1;
        }
    }
12: write(positives);
```

(b) the slice obtained using the conventional algorithm

Figure 1: An example program without any jump statements and its program slice with respect to positives on line 12.

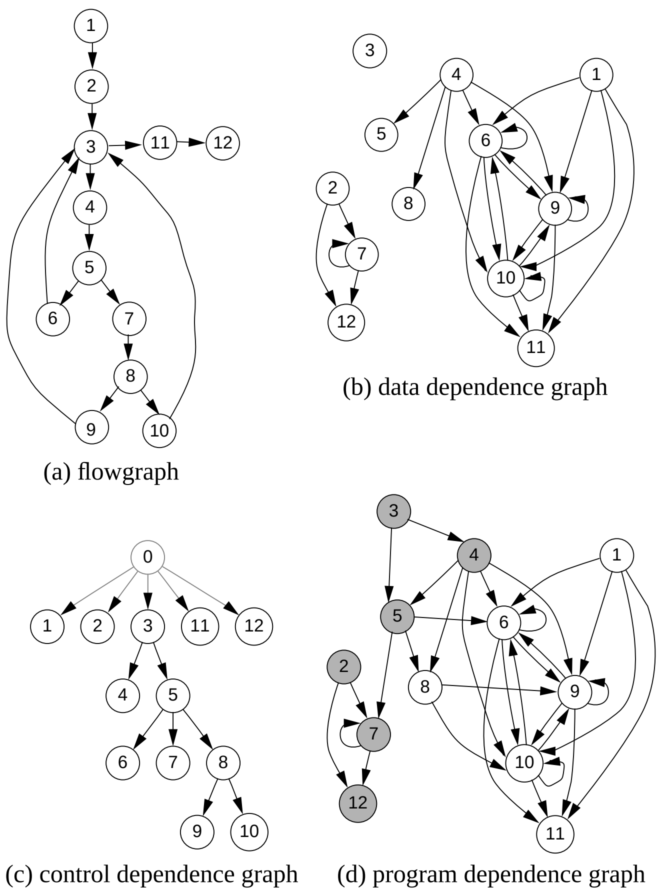
(a) flowgraph
(b) data dependence graph
(c) control dependence graph
(d) program dependence graph

Figure 2: Various graphs of the program in Figure 1-a. The shaded nodes represent the statements included in the slice by the conventional slicing algorithm.

mation. More importantly, it lends itself to substantial simplification when all jump statements in a program are "structured" jump statements, such as break, continue, and return statements, or goto statements that transfer control in the "forward" direction. In this case, only those jump statements that are directly control dependent on a predicate in the slice need to be considered for possible inclusion in the slice. The simplified algorithm, in turn, directly leads to a conservative approximation algorithm that permits on-the-fly detection of the relevant jump statements to be included in the slice while applying the conventional slicing algorithm. This algorithm is extremely efficient and should suffice for use with most programs written in modern procedural languages.

In the next section, we review the conventional slicing algorithm used to find slices of programs that do not contain any jump statements. In Section 3, we propose a general algorithm to determine which jump statements to include in a slice when the program under consideration contains jump statements. Then, in Section 4, we discuss how this algorithm may be simplified for use with programs that contain only structured jump statements. Finally, we discuss related work in Section 5 and summarize our results in Section 6.

2 The Conventional Slicing Algorithm

The conventional slicing algorithm involves finding the transitive closure of the data and control dependences of the appropriate node(s) in the program dependence graph of the program [17, 24]. The program dependence graph of a program is obtained by merging its data and control dependence graphs, which depict the inter-statement data and control dependences, respectively, in the program.

For example, consider the program in Figure 1-a and its slice with respect to positives on line 12. Figure 2 shows the flowgraph and the data-, control-, and program dependence graphs of this program. Node 12 is data dependent on nodes 2 and 7 as the assignments on lines 2 and 7 assign a value to positives that may be used by the write statement on line 12. Node 7 is control dependent on node 5 as the *if* statement on line 5 determines whether or not the assignment on line 7 is executed. The shaded nodes in the program dependence graph in Figure 2-d depict the nodes in the transitive closure of the data- and control dependences of node 12. Figure 1-b shows the corresponding program slice.

```
 1:        sum = 0;
 2:        positives = 0;
 3:   L3:  if (eof()) goto L14;
 4:        read(x);
 5:        if (x > 0) goto L8;
 6:        sum = sum + f1(x);
 7:        goto L13;
 8:   L8:  positives = positives + 1;
 9:        if (x%2 != 0) goto L12;
10:        sum = sum + f2(x);
11:        goto L13;
12:  L12:  sum = sum + f3(x);
13:  L13:  goto L3;
14:  L14:  write(sum);
15:        write(positives);
```

(a) a "goto" version of the example program

```
 2:        positives = 0;
 3:   L3:  if (eof()) goto L14;
 4:        read(x);
 5:        if (x > 0) goto L8;
 8:   L8:  positives = positives + 1;
     L14:
15:        write(positives);
```

(b) the slice obtained using the conventinal algorithm

```
 2:        positives = 0;
 3:   L3:  if (eof()) goto L14;
 4:        read(x);
 5:        if (x > 0) goto L8;
 7:        goto L13;
 8:   L8:  positives = positives + 1;
13:  L13:  goto L3;
     L14:
15:        write(positives);
```

(c) the slice obtained using the new algorithm

Figure 3: A "goto" version of the program in Figure 1-a, and its program slice with respect to positives on line 15.

3 Finding the Relevant Jump Statements

A jump statement does not assign a value to any variable. Thus no statement may be data dependent on it. Also, a jump statement is not a predicate. So no statement may be control dependent on it. Hence, the conventional slicing algorithm may not cause any jump statement to be included in a slice.

It is easy to adapt the conventional slicing algorithm to determine which *conditional* jump statements, such as those on lines 3 and 5 in Figure 3-a, to include in a slice: If the predicate in a conditional jump statement is included in a slice because some other statement is control dependent on it, then the associated jump must also be included, for the predicate will not serve any purpose in the slice without the accompanying jump. Hereafter, we will use the phrase conventional slicing algorithm to refer to this adaptation of the conventional slicing algorithm.

Figure 3-a shows a program with goto statements that is equivalent in functionality to that in Figure 1-a. Figure 3-b shows its slice with respect to positives on line 15 obtained using the conventional slicing algorithm. Unfortunately, unlike the original program, the slice fails to ensure that the assignment on line 8 is executed iff $x > 0$ because it does not include the relevant unconditional jump statements. Figure 3-c shows the correct slice. Note that although the unconditional jumps on lines 7 and 13 are included in it, that on line 11 is not.

The same situation occurs in the presence of break,

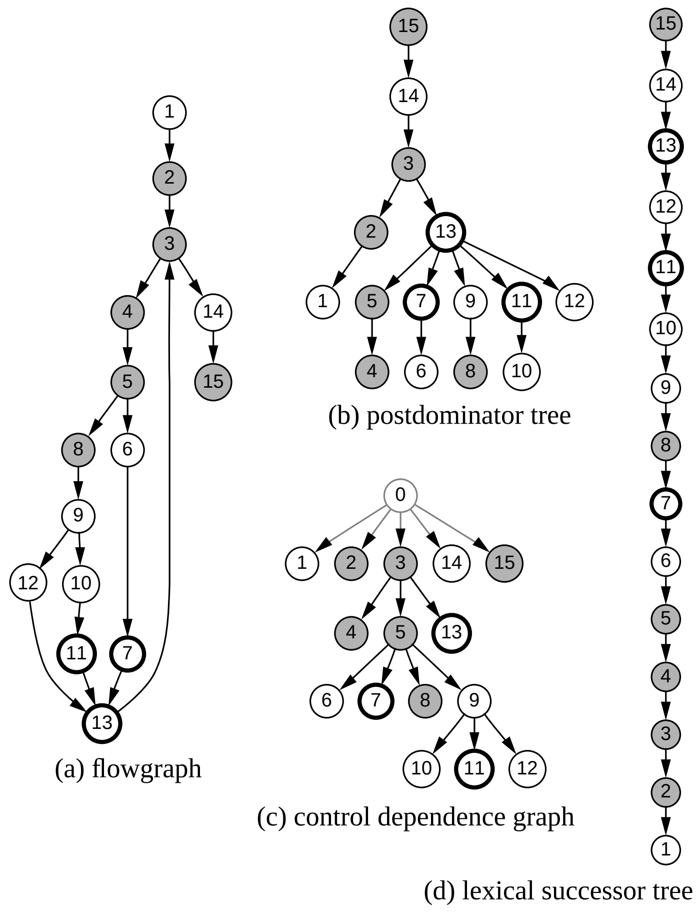

Figure 4: Various graphs of the program in Figure 3-a. The shaded nodes represent the statements included in the slice by the conventional slicing algorithm.

continue, and return statements, which may be thought of as special cases of the unconditional goto statement with their targets implicitly defined. Figure 5-a shows a program with continue statements that is equivalent in functionality to the program in Figure 3-a. Figure 5-b shows the corresponding slice with respect to positives on line 14 obtained using the conventional slicing algorithm. Note that, without the relevant continue statement(s) in the slice, the assignment on line 8 is incorrectly executed during each loop iteration irrespective of the value of x. Figure 5-c shows the correct slice. Note that although the continue statement on line 7 is included in the slice, that on line 11 is not.

The question, then, is: how does one determine which unconditional jump statements to include in a slice? Consider a sequence, $S_1; S_2; S_3$, of three statements in a program. Suppose S_1 and S_3 do not have any explicit jump statements embedded in them. Also suppose S_1 and S_3 belong to a slice obtained using the conventional slicing algorithm whereas S_2 does not. If S_2 is an assignment statement, then control always passes from S_1 to S_2 to S_3 in the original program.

```
 1:  sum = 0;
 2:  positives = 0;
 3:  while (!eof()) {
 4:     read(x);
 5:     if (x <= 0) {
 6:         sum = sum + f1(x);
 7:         continue;   /* goto line 3 */
        }
 8:     positives = positives + 1;
 9:     if (x%2 == 0) {
10:         sum = sum + f2(x);
11:         continue;   /* goto line 3 */
        }
12:     sum = sum + f3(x);
    }
13:  write(sum);
14:  write(positives);
```

(a) a "continue" version of the example program

```
 2:  positives = 0;
 3:  while (!eof()) {
 4:     read(x);
 5:     if (x <= 0) {
        }
 8:     positives = positives + 1;
    }
14:  write(positives);
```

(b) the slice obtained using the conventional algorithm

```
 2:  positives = 0;
 3:  while (!eof()) {
 4:     read(x);
 5:     if (x <= 0) {
 7:         continue;   /* goto line 3 */
        }
 8:     positives = positives + 1;
    }
14:  write(positives);
```

(c) the slice obtained using the new algorithm

Figure 5: A "continue" version of the program in Figure 3-a and its slice with respect to positives on line 14.

Deleting S_2 from the program will cause it to always transfer from S_1 to S_3. The same holds true if S_2 is a compound statement, e.g., an *if* or a *while* statement, provided its body does not contain any explicit jump statement.

If the body of S_2 does contain one or more explicit jump statements, then control need not always pass from S_2 to S_3 in the original program as the explicit jump(s) in S_2 may cause the control to transfer elsewhere. In this case we may not omit S_2 from the slice, because otherwise, unlike in the original program, the control will always pass unconditionally from S_1 to S_3 in the slice. We need not, however, include all statements in S_2 in the slice. We only need to include certain jump statements in it along with the statements they transitively depend on. This is required because in the presence of jump statements, the statement that lexically follows a statement in a program need not also be its immediate postdominator, as discussed below.

A statement, S', is said to *postdominate* a statement, S, in a program if S' dominates [3] S in the reverse flowgraph of the program. In other words, S' postdominates S if every path from S to the exit node in the flowgraph contains S'. S' is said to be the *immediate postdominator* of S if every other postdominator of S also postdominates S'. The postdominator relationship among statements may be represented in the form of a postdominator tree. S' postdominates S iff S' is an ancestor of S in the postdominator tree. Algorithms used to construct dominator trees [3, 13, 20, 25] may also be used to construct postdominator trees. Postdominator trees are also re-

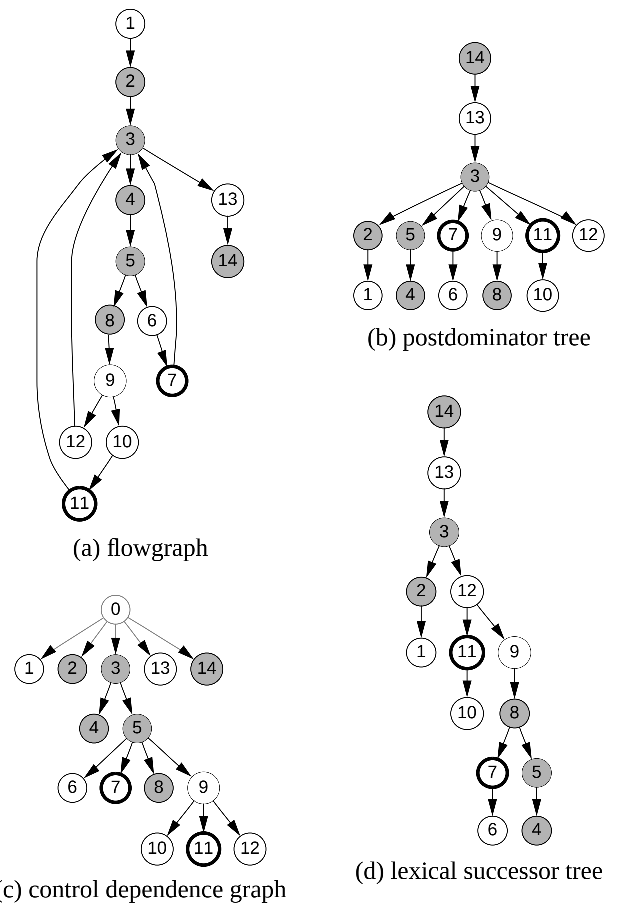

Figure 6: Various graphs of the program in Figure 5-a. The shaded nodes represent the statements included in the slice by the conventional slicing algorithm.

quired by the algorithms used to construct the control dependence graphs of programs that contain jump statements [9, 10]. Figures 4-b and 6-b show the postdominator trees of the programs in Figures 3-a and 5-a, respectively.

A statement, S', is said to be the *immediate lexical successor* of a statement, S, in a program if deleting S from the program will cause the control to pass to S' whenever it reaches the corresponding location in the new program[2]. This notion of the immediate lexical successor is the same as that of the "continuation statement" in [5] and the "fall-through statement" in [8]. Like the postdominator relationship, the lexical successor relationship may be represented graphically in the form of a *lexical successor tree*. It may be constructed in a purely syntax directed manner. A statement, S', is said to be a *lexical successor* of a statement, S, in a program if S' is an ancestor of S in the lexical successor tree of the program. Figures 4-d and 6-d show the

[2]If S is a compound statement, such as an *if* or a *while* statement, deleting it means deleting it along with the statements that constitute its body.

```
Slice = the slice obtained using the conventional slicing algorithm;
do {
    Traverse the postdominator tree using the preorder traversal, and for each jump
    statement, J, encountered that is (i) not in Slice and (ii) whose nearest postdominator
    in Slice is different from the nearest lexical successor in Slice) do {
        Add J to Slice;
        Add the transitive closure of the dependences of J to Slice;
    }
} until no new jump statements may be added to Slice;
For each goto statement, Goto L, in Slice, if the statement labeled L is not in Slice then
associate the label L with its nearest postdominator in Slice;
return (Slice);
```

Figure 7: An algorithm to find slices of programs with jump statements.

lexical successor trees of the programs in Figures 3-a and 5-a, respectively.

In a program that does not contain any jump statements, the immediate lexical successor of a statement is always the same as its immediate postdominator. In other words, the lexical successor and the postdominator trees of such programs are identical. Consequently, a slice of such a program may be constructed by deleting all those statements from the program that are not selected for inclusion in the slice by the conventional slicing algorithm. For a program that contains jump statements, however, the lexical successor of a statement need not also be its immediate postdominator. Hence, we may not obtain a slice of such programs by simply deleting the statements not selected by the conventional slicing algorithm. We must also include certain jump statements, as well as the statements they transitively depend on, to ensure that the statements included in the slice by the conventional slicing algorithm get executed in the same relative order as in the original program.

The decision about whether or not to include a jump statement, J, in a slice depends on whether or not its nearest postdominator in the slice is the same as its nearest lexical successor in the slice. If the two are different, we must include J as well as the closure of its dependences in the slice, which, in turn, may cause other jump statements to be included in the slice. If, however, the nearest postdominator of J in the slice is the same as its nearest lexical successor in the slice, then omitting J from the slice will not adversely affect the flow of control among the statements included in the slice.

As the inclusion or exclusion of one jump statement in a slice may affect the inclusion or exclusion of another jump statement, care must be taken in the order in which various jump statements are considered for possible inclusion in the slice. *Preorder traversal* of the postdominator tree, where a node is visited before any of its children are visited, ensures that all jump statements are examined in the desired order. Alternatively, the preorder traversal of the lexical successor tree may be used. Figure 7 shows an algorithm to determine which statements to include in a slice when the program under consideration contains jump statements. It employs the preorder traversal of the postdominator tree.

Figure 4 shows the flowgraph, the postdominator tree, the control dependence graph, and the lexical successor tree of the program in Figure 3-a. Nodes with thick outlines in these graphs denote unconditional jump statements[3].

The shaded nodes in the graphs in Figure 4 represent statements included in the slice by the conventional slicing algorithm. During the preorder traversal of the postdominator tree, Node 13 is the first jump statement encountered. Figures 4-b and 4-d show that nodes 3 and 15 are the nearest postdominator and the nearest lexical successor nodes, respectively, of node 13 in the slice. As the two are different, node 13 is included in the slice. Its inclusion does not cause any other nodes to be included in the slice as the nodes on which it is control dependent are already in the slice. Node 7 is the next jump node encountered during the preorder traversal of the postdominator tree. As the nearest postdominator and lexical successor of node 7 in the slice are also different, it too is included in the slice. Node 11 is the only remaining jump node not yet examined. Inclusion of node 13 in the slice above makes it both the nearest postdominator as well as the nearest lexical successor of node 11 in the slice. Therefore, node 11 is not included in the slice. Figure 3-c

[3]The control dependence graph also contains a dummy predicate node, viz., node 0. All top-level nodes—nodes that are not control dependent on any predicate in the program—are made control dependent on this node.

```
 1:       sum = 0;
 2:       positives = 0;
 3:  L3:  if (eof()) goto L14;
 4:       read(x);
 5:       if (x > 0) goto L8;
 6:       sum = sum + f1(x);
 7:       goto L3;
 8:  L8:  positives = positives + 1;
 9:       if (x %2 != 0) goto L12;
10:       sum = sum + f2(x);
11:       goto L3;
12: L12:  sum = sum + f3(x);
13:       goto L3;
14: L14:  write(sum);
15:       write(positives);
```

(a) another "goto" version of the example program

```
 2:       positives = 0;
 3:  L3:  if (eof()) goto L14;
 4:       read(x);
 5:       if (x > 0) goto L8;
 8:  L8:  positives = positives + 1;
     L14:
15:       write(positives);
```

(b) the slice obtained using the conventional algorithm

```
 2:       positives = 0;
 3:  L3:  if (eof()) goto L14;
 4:       read(x);
 5:       if (x > 0) goto L8;
 7:       goto L3;
 8:  L8:  positives = positives + 1;
 9:       if (x %2 != 0) goto L12;
11:       goto L3;
     L12:
13:       goto L3;
     L14:
15:       write(positives);
```

(c) the slice obtained using the new algorithm

Figure 8: Another "goto" version of the program in Figure 3-a and its slice with respect to positives on line 15.

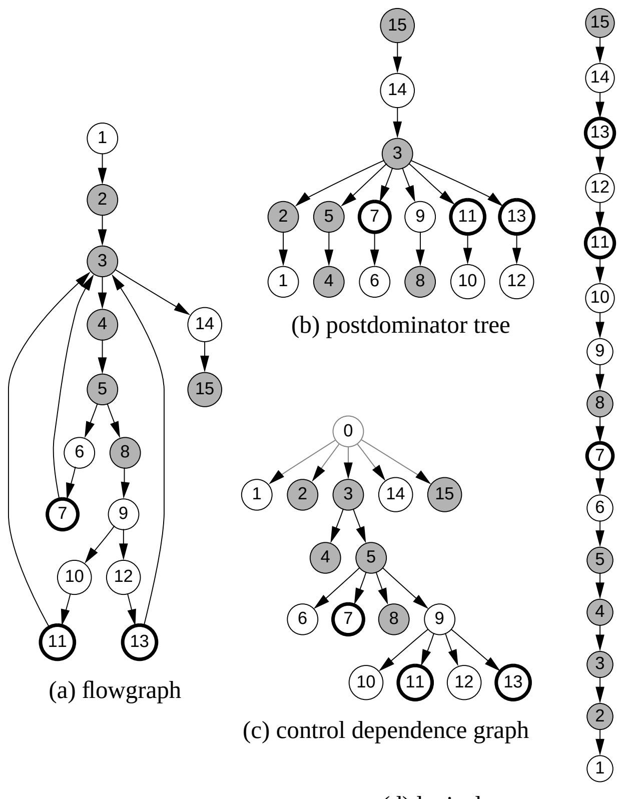

Figure 9: Various graphs of the program in Figure 8-a. The shaded nodes represent the statements included in the slice by the conventional slicing algorithm.

shows the final program slice.

Figure 8-a shows another version of the program in Figure 3-a where the indirect jumps from lines 7 and 11 via line 13 to line 3 have been replaced with direct jumps to line 3. Figure 9 shows the corresponding graphs for this program. Node 7 is the first jump statement visited during the preorder traversal of the postdominator tree. As its nearest postdominator and lexical successor nodes in the slice are different, it is included in the slice. Further traversal of the postdominator tree causes nodes 11 and 13 to also be included in the slice. Their inclusion, in turn, causes node 9 to be included in the slice, as both nodes 11 and 13 are control dependent on it, as shown in Figure 9-c. Figure 8-c shows the final program slice.

For both examples above, a single traversal of the postdominator tree was sufficient, i.e., no new nodes could be added to the slice after the first traversal. For some uncommon programs, though, multiple traversals of the postdominator tree may be required. Consider, for example, the program in Figure 10-a, adapted from a similar example in [5], and its slice with respect to y on line 9. Figure 11 shows the corresponding graphs. During the first preorder traversal of the postdominator tree, node 4 is not added to the slice as its nearest postdominator and the nearest lexical successor are the same, viz., node 9. Nodes 7 and 2, however, are added to the slice as their nearest postdominator and lexical successors in the slice are different. The addition of node 2, in turn, causes node 1 to be added to the slice as the former is control dependent on the latter. Also, the addition of node 7 to the slice during the first traversal causes it to become the nearest lexical successor of node 4 in the slice. Node 9, on the other hand, continues to be its nearest postdominator in the slice. Thus node 4 is added to the slice during the second preorder traversal of the postdominator tree. Figure 10-b shows the final program slice.

The algorithm in Figure 7 uses the preorder traversal of the postdominator tree to decide the order in which jump statements are examined for possible inclusion in the slice. As mentioned earlier, we could also have used the preorder traversal of the lexical successor tree. Although the choice of the tree used to drive the search may cause a difference in the number of traversals required, the same final slice is obtained in each case. While one method may require less traversals than the other in the case of one slice, the opposite may be true in the case of another slice.

Multiple traversals are required, in general, when a program contains a pair of nodes, N_1 and N_2, such that N_1 postdominates N_2 and N_2 lexically succeeds N_1. For example, consider nodes 4 and 7 in Figures 11-b and 11-d. Whereas node 4 postdominates node 7, node 7 lexically succeeds node 4. When a program contains no such pairs, a single traversal is sufficient to identify all jump statements to be included in

```
 1:       if (C1) {
 2:             goto L6;
 3:  L3:        y = ...;
 4:             goto L8;
          }
 5:       z = ...;
 6:  L6:  x = ...;
 7:       goto L3;
 8:  L8:  write(x);
 9:       write(y);
10:       write(z);
```

(a) the original program

```
1:       if (C1) {
2:             goto L6;
3:  L3:        y = ...;
4:             goto L8;
         }
    L6:
7:       goto L3;
    L8:
9:       write(y);
```

(b) the slice obtained using the new algorithm

Figure 10: An unstructured program and its slice with respect to y on line 9.

a slice. As the programs in Figures 3 and 8 contain no such pairs, a single traversal of the postdominator tree was sufficient to obtain slices of those programs[4].

It can be shown that the algorithm in Figure 7 always produces correct program slices. Alternatively, it can be shown that a statement is included in a slice by this algorithm iff it is included in the corresponding slice obtained using Ball and Horwitz's algorithm [5]. Ball and Horwitz have shown that their algorithm always yields correct slices. Hence the equivalence of the two algorithms may also be used to infer that the algorithm in Figure 7 produces correct slices.

4 Slicing Programs with Structured Jumps

A jump statement is said to be a *structured jump statement* if its target statement is also its lexical successor. Break, continue, and return statements, as in C, are examples of structured jumps as they all pass control to their lexical successors, though not necessarily to their immediate lexical successors. A program is said to be a *structured program* if all jump statements in it are structured jump statements. Otherwise, it is said to be an *unstructured* program. It can be shown that a structured program always satisfies the following two properties:

1. It does not contain any pair (N_i, N_j) of nodes such that N_i is a postdominator of N_j and N_j is a lexical successor of N_i.

2. If a jump statement, J, is directly control dependent on a predicate, P, then J will not be included in a slice if P is not included in it by the conventional slicing algorithm.

The first conclusion implies that for structured programs, a single traversal of the postdominator tree is sufficient to identify all jump statements to be included in a slice. The second conclusion implies that a jump statement may be included in a slice only if a predicate on which it is directly control dependent is also included in the slice. It also implies that when a jump statement *is* included in a slice, there is no need to explicitly include the closure of its dependences in the slice as they have already been included in it. Consequently, we may simplify the algorithm in Figure 7 for use with structured programs to that shown in Figure 12. The program in Figure 5-a is a structured program. The simplified algorithm in Figure 12 will yield the same slice as that obtained using the algorithm in Figure 7 for this program for all slicing criteria. For example, both algorithms will find the same slice with respect to positives on line 14—the slice shown in Figure 5-c.

Figure 13 shows a further simplification of the algorithm in Figure 12. It does not require any preorder traversals of the postdominator tree nor does it require that the lexical successor tree of the program be available. It is, however, a conservative simplification of the algorithm in Figure 12. A slice obtained using the former may include more jump statements than that

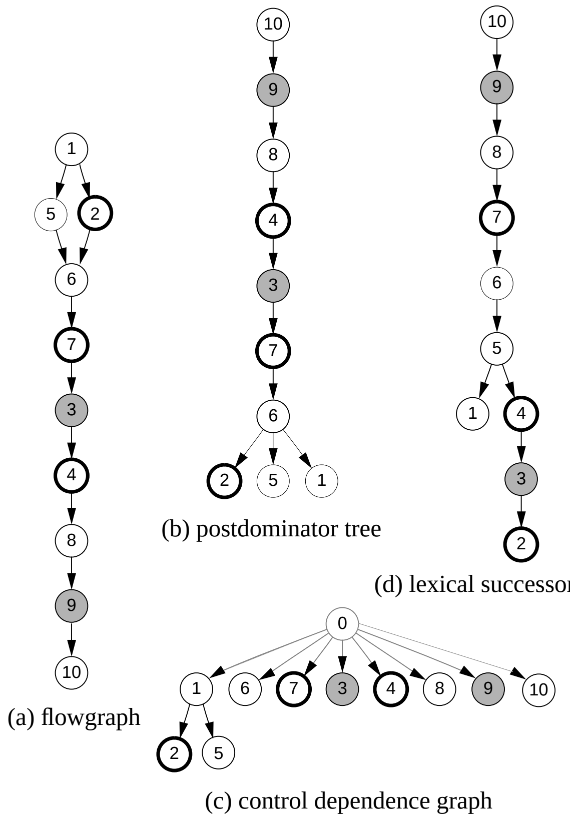

Figure 11: Various graphs of the program in Figure 10-a. The shaded nodes represent the statements included in the slice by the conventional slicing algorithm.

[4] This is not to say that multiple traversals are always required whenever a program contains such pairs.

```
Slice = the slice obtained using the conventional slicing algorithm;

Traverse the postdominator tree using the preorder traversal, and add each jump statement
encountered that is (i) directly control dependent on a predicate in Slice and (ii) whose nearest
postdominator in Slice is different from its nearest lexical successor in Slice, to Slice;

For each goto statement, Goto L, in Slice, if the statement labeled L is not in Slice then
associate the label L with its nearest postdominator in Slice;

return (Slice);
```

Figure 12: An algorithm to find slices of programs with *structured* jump statements.

```
Slice = the slice obtained using the conventional slicing algorithm;

Add all jump statements that are directly control dependent on a predicate in Slice, to Slice;

For each goto statement, Goto L, in Slice, if the statement labeled L is not in Slice, then
associate the label L with its nearest postdominator in Slice;

return (Slice);
```

Figure 13: A *conservative* algorithm to find slices of programs with structured jump statements.

obtained using the latter. This is so because, unlike the algorithm in Figure 12, the algorithm in Figure 13 includes any jump statement in the slice that is directly control dependent on a predicate in the slice, even if its nearest postdominator and lexical successor nodes in the slice are the same.

For the example shown in Figure 5-a, this algorithm will give the same slice as that given by the algorithm in Figure 12. Figure 14-a shows an example of a structured program for which the algorithm in Figure 13 may yield a bigger slice than that given by the algorithm in Figure 12. Figures 14-b and 14-c show its two slices with respect to y on line 9 obtained using the two algorithms. Note that the latter slice also includes the jump statements on lines 5 and 7 that are not included in the former.

5 Related Work

The concept of program slicing was first proposed by Weiser [29]. He also presented an algorithm to compute program slices based on an iterative solution of dataflow equations. His algorithm was able to determine which predicates to include in the slice even when the program contained jump statements. It did not, however, make any attempt to determine the relevant jump statements themselves to be included in the slice.

Use of the program dependence graph to compute program slices was first proposed by Ottenstein and Ottenstein [24]. They proposed a slicing algorithm based on graph reachability in the program dependence graph but they only considered the intraprocedural case. Horwitz, Reps, and Binkley extended the program dependence graph representation to what they call the "system dependence graph" to find interprocedural program slices [17]. Several studies investigating the semantic basis of program slicing have also been conducted [7, 15, 26, 27, 28]. Uses of program slices have been suggested in many applications, e.g., program debugging, verification, maintenance, regression testing, automatic parallelization of program execution, automatic integration of program versions, and software metrics (see, e.g., [1, 2, 6, 12, 16, 19, 21, 23, 29]). None of the references cited above, however, addresses the question of how to determine the appropriate jump statements to include in a slice.

Lyle proposed an extremely conservative algorithm to determine which jump statements to include in a slice [22]. Suppose a statement, S, is included in a slice with respect to a variable, var, and a location, loc, in a program. Then, except in certain degenerate cases, Lyle's algorithm will include all jump statements that lie between S and loc in the control flowgraph of the program, in the slice. For example, consider the example in Figure 5. Unlike any of the algorithms presented in this paper, Lyle's algorithm will also include the continue statement on line 11, and therefore the predicate on line 9, in the slice. Similarly, it will include all goto statements and all predicates in the example in Figure 3, although some of them could be omitted.

Gallagher proposed a modification of the above algorithm where a jump statement, *Goto L*, is included in a slice only if a statement in the block labeled L and the predicates on which the jump statement is control dependent are included in the slice [11]. Although this algorithm only considered the goto statement, we assume it can be extended to handle other jump state-

```
1:   switch (C) {
2:       case 1: x = ...;
3:               break;
4:       case 2: y = ...;
5:               break;
6:       case 3: z = ...;
7:               break;
     }
8:   write(x);
9:   write(y);
10:  write(z);
```

(a) the original program

```
1:   switch (C) {
         case 1:
3:               break;
4:       case 2: y = ...;
     }
9:   write(y);
```

(b) the slice obtained using the simplified algorithm

```
1:   switch (C) {
         case 1:
3:               break;
4:       case 2: y = ...;
5:               break;
         case 3:
7:               break;
     }
9:   write(y);
```

(c) the slice obtained using the conservative algorithm

Figure 14: A structured program and its two slices with respect to y on line 9 obtained using the algorithms in Figures 12 and 13.

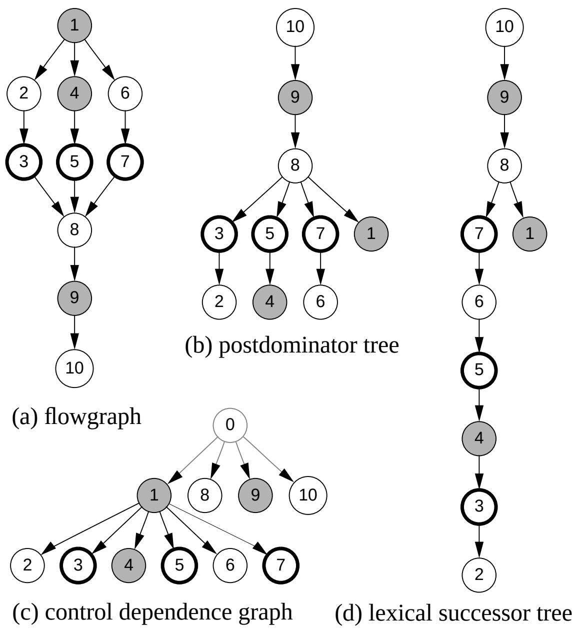

Figure 15: Various graphs of the program in Figure 14-a. The shaded nodes represent the statements included in the slice by the conventional slicing algorithm.

ments such as break and continue if we think of them as having dummy labels associated with them. Assuming this extension, the above algorithm will correctly omit the continue statement on line 11, and thus the predicate on line 9, from being included in the slice in Figure 5. Unfortunately, this algorithm may fail to identify all relevant jump statements for inclusion in a slice in some other programs. Consider, for example, the program in Figure 16-a and its slice with respect to y on line 10. Figure 16-b shows the incorrect slice obtained using the above algorithm. It fails to include the jump statement on line 4 because no statement in the block labeled L6 is included in the slice. Without this jump statement the assignment to y on line 5 will always be executed in the slice irrespective of the value of x, unlike in the correct slice, shown in Figure 16-c, where it is only executed when x is nonnegative.

Jiang, Zhou, and Robson have also proposed a set of rules to determine which jump statements to include in a slice [18]. Unfortunately their rules also fail to identify all relevant jump statements. For example, they will fail to include both jump statements on lines 11 and 13 in the slice in Figure 8.

As mentioned in Section 1, recently two similar algorithms to determine the relevant jump statements to include in a slice were independently proposed by Ball and Horwitz [5] and Choi and Ferrante [8]. Both require that the control dependence graph of a program be constructed from an augmented flowgraph of the program. The augmented flowgraph is obtained by adding new edges to the flowgraph from the nodes representing jump statements to the nodes that immediately lexically succeed them in the program. The data dependence graph, on the other hand, is constructed from the *unaugmented* flowgraph. Both algorithms, then, involve applying the conventional slicing algorithm on the augmented program dependence graph obtained by merging the control- and the data dependence graphs.

As mentioned earlier, our algorithm in Figure 7 has the same precision as that of the above two algorithms. It is, however, appealing in that it leaves the flowgraph and the program dependence graph of the program intact and uses a separate graph to store the additional required information. More importantly, it lends itself to substantial simplification, as discussed in the previous section, when the program under consideration is a structured program. Also, the simplified algorithm directly leads to a conservative approximation algorithm that permits on-the-fly detection of the relevant jump statements while applying the conventional slicing algorithm. It is unclear if the algorithms mentioned in the previous paragraph permit similar adaptations.

Choi and Ferrante have also proposed another algorithm to construct an executable slice in the presence of jump statements when a "slice" is not constrained to be a subprogram of the original program [8]. This algorithm, like our algorithm, proceeds by first finding the statements included in the slice by the conventional slicing algorithm. Then, unlike our algorithm which finds the relevant jump statements in the original program to include in the slice, it constructs *new* jump statements to add to the slice to ensure that other statements in it are executed in the correct order. Also, when a jump statement is added to a slice, it does not require the transitive closure of its control and data

```
 1:        read(x);
 2:        if (x < 0) {
 3:              y = f1(x);
 4:              goto L6;
           }
 5:        y = f2(x);
 6:  L6:   if (y < 0) {
 7:              z = g1(y);
 8:              goto L10;
           }
 9:        z = g2(y);
10:  L10:  write(y);
11:        write(z);
```

(a) an example program

```
 1:        read(x);
 2:        if (x < 0) {
 3:              y = f1(x);
           }
 5:        y = f2(x);
10:  L10:  write(y);
```

(b) incorrect slice

```
 1:        read(x);
 2:        if (x < 0) {
 3:              y = f1(x);
 4:              goto L6;
           }
 5:        y = f2(x);
     L6:
10:  L10:  write(y);
```

(c) the correct slice

Figure 16: An example program and its slice with respect to y on line 10.

dependences to be added to the slice. This may lead to construction of smaller "slices" compared to those produced by algorithms that require a slice to be a subprogram of the original program. It may, however, cause the relative nesting structure of statements included in the slice to be different from that in the original program. An alternative approach to find a "slice" that need not be a projection of the original program would be, as noted by Ball and Horwitz [5], to apply a flowgraph structuring algorithm [4] on the flowgraph induced by the statements included in the slice by the conventional slicing algorithm.

6 Summary

In a program with jump statements the lexical successor of a statement need not be the same as its immediate postdominator. Hence, a slice obtained using the conventional slicing algorithm for such a program may not preserve the behavior of the original program with respect to the slicing criterion. In this paper, we have proposed an algorithm that uses the lexical successor and the postdominator trees of the program to identify the appropriate jump statements to include in a slice. We have also presented a simpler version of this algorithm for use with programs that contain structured jump statements, and an efficient, conservative approximation algorithm that does not require the construction of the lexical successor tree of the program at all.

Acknowledgements

We would like to thank Thomas Ball, Jong-Deok Choi, Jeanne Ferrante, and the anonymous referees for their useful comments on an earlier draft of this paper.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. L. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 348–357. The IEEE Computer Society Press, Sept. 1993.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] B. S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, Jan. 1977.

[5] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (Lecture Notes in Computer Science 749)*, pages 206–222. Springer-Verlag, Nov. 1993.

[6] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, Jan. 1985.

[7] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation.* ACM Press, June 1989. SIGPLAN Notices, 24(7):13–27, July 1989.

[8] J.-D. Choi and J. Ferrante. Static slicing in the presence of GOTO statements. *ACM Letters on Programming Languages and Systems*, 1994. To appear.

[9] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[11] K. B. Gallagher. *Using Program Slicing for Program Maintenance.* PhD thesis, University of Maryland, College Park, MaryLand, 1990.

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

[13] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194, May 1985.

[14] M. J. Harrold, B. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 160–170. ACM Press, June 1993.

[15] P. Hausler. Denotational program slicing. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, volume II, pages 486–494, 1989.

[16] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

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

[18] J. Jiang, X. Zhou, and D. J. Robson. Program slicing for C—the problems in implementation. In *Proceedings of the IEEE Conference on Software Maintenance.* IEEE Computer Society Press, Oct. 1991.

[19] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, Nov. 1990.

[20] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[21] H. Longworth, L. Ott, and M. Smith. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of COMPSAC.* IEEE Computer Society Press, 1986.

[22] J. R. Lyle. *Evaluating Variations on Program Slicing for Debugging.* PhD thesis, University of Maryland, College Park, MaryLand, 1984.

[23] L. Ott and J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the Eleventh International Conference on Software Engineering.* IEEE Computer Society Press, May 1989.

[24] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments.* ACM Press, Apr. 1984. SIGPLAN Notices, 19(5):177–184, May 1984.

[25] P. W. Purdom, Jr. and E. F. Moore. Immediate predominators in directed graphs. *Communications of the ACM*, 15(8):777–778, Aug. 1972.

[26] T. Reps and W. Yang. The semantics of program slicing. Technical Report TR-777, Computer Science Department, University of Wisconsin, Madison, Wisconsin, June 1988.

[27] R. P. Selke. A rewriting semantics for program dependence graphs. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 12–24. ACM Press, Jan. 1989.

[28] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation.* ACM Press, June 1991. SIGPLAN Notices, 26(6):107–119, June 1991.

[29] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.