Slicing Programs with Arbitrary Control Flow

THOMAS BALL* SUSAN HORWITZ*
tom@cs.wisc.edu horwitz@cs.wisc.edu
Computer Sciences Department
University of Wisconsin − Madison
1210 West Dayton Street
Madison, Wisconsin 53706 USA

December 21, 1992

Abstract

Program slicing is a program transformation that is useful in program debugging, program maintenance, and other applications that involve understanding program behavior. Given a program point *p* and a set of variables *V*, the goal of slicing is to create a *projection* of the program (by eliminating some statements), such that the projection and the original program compute the same values for all variables in *V* at point *p*.

This paper addresses the problem of slicing programs with arbitrary control flow. Previous slicing algorithms do not always form semantically correct program projections when applied to such programs. We present an algorithm for slicing programs with complex control flow and a proof of its correctness. Our algorithm works for programs with completely arbitrary control flow, including irreducible control flow.

1. INTRODUCTION

Program slicing, a program transformation originally defined by Mark Weiser [13], is useful in program debugging [9], program maintenance [6], and other applications that involve understanding program behavior [7]. Given a program point *p* and a set of variables *V*, the goal of slicing is to create a *projection* of the program (by eliminating some statements), such that the projection and the original program compute the same values for all variables in *V* at point *p*.

Example. The program shown in Figure 1(a) computes the sum and product of the numbers from 1 to *N*[1]. Figure 1(b) shows the result of slicing the example program with respect to the statement **output**(*prod*) and the set of variables { *prod* }. For any value of *N*, the example program and the program projection compute the same value for variable *prod* in the **output** statement. □

This paper addresses a problem that has not been discussed in the literature on program slicing [8, 10, 12, 13]. The problem is how to slice programs with unstructured control flow, *i.e.*, programs that include constructs such as **break** and **goto**. Previous algorithms do not slice such programs correctly. We give a program-slicing algorithm that correctly handles such programs, and we prove that the program projections produced by our algorithm meet the semantic goal of program slicing: Both the original program and the projection compute the same values at the point of the slice. Our algorithm works for programs with completely arbitrary control flow, including irreducible graphs [1]. (As given here, our algorithm has a slight restriction: A program is sliced with respect to a point *p* and the set of variables used or defined at *p* rather than an arbitrary set of variables. Extending the algorithm to permit a

* This work was supported in part by the National Science Foundation under grant CCR-8958530, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from Xerox and 3M.

[1] In the example program, variable *N* is used without being explicitly initialized. It is assumed that such variables get their values from an initial state on which the program is executed.

(a) Example Program	(b) Result of slicing with respect to **output**(*prod*)
begin	**begin**
sum := 0	
prod := 1	*prod* := 1
i := 0	*i* := 0
while *i* < *N* **do**	**while** *i* < *N* **do**
i := *i*+1	*i* := *i*+1
sum := *sum*+*i*	
prod := *prod***i*	*prod* := *prod***i*
od	**od**
output(*sum*)	
output(*prod*)	**output**(*prod*)
end	**end**

Figure 1. An example program, and the result of slicing with respect to **output**(*prod*).

slice to be taken with respect to an arbitrary set of variables V at point p is straightforward: Add artificial uses of all variables in V at point p.)

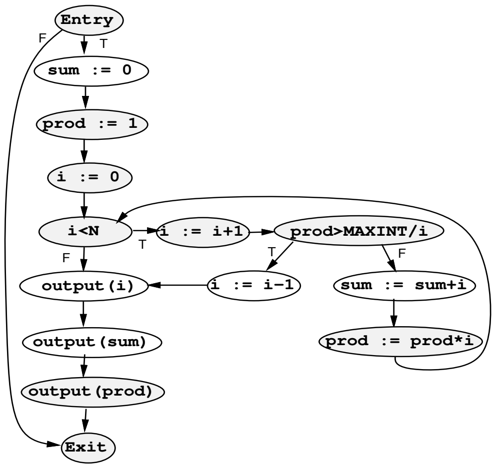
Algorithms for slicing programs with *structured* control flow have been defined by Weiser [13] and by the Ottensteins [10]. Let us consider the problems that arise if one tries to apply either of those algorithms to programs with unstructured control flow. Both algorithms make use of a program's control flow graph; Weiser's algorithm operates directly on the control flow graph, and the Ottensteins' algorithm operates on the program dependence graph [4], which includes edges that are defined in terms of relationships between vertices in the control flow graph. Both algorithms have two steps. In Weiser's algorithm, the output of Step 1 is a subset S of the vertices of the control flow graph. In the Ottensteins' algorithm, the output of Step 1 is a subset S' of the vertices of the program dependence graph. The sets S and S' identify the same set of program components. In both algorithms, Step 2 produces the projection by eliminating from the original program all components that do not correspond to a vertex identified in Step 1. If one uses the standard control flow graph for programs with unstructured control flow (*i.e.*, a graph in which a jump such as **break** is represented as a single edge—see Figure 2(b)), the program projections computed by both Weiser's and the Ottensteins' algorithms may fail to meet the semantic goal of program slicing; that is, the projections may compute different values than the original program at the point of the slice.

Example. Consider the program shown in Figure 2(a). Figure 2(b) shows the standard control flow graph for this program, and Figure 2(c) shows the program dependence graph that corresponds to this flow graph. In the two graphs, shading is used to indicate the vertices that would be identified by Weiser's and the Ottensteins' algorithms when slicing with respect to the statement "**output**(*prod*)". Figure 2(d) shows the program projection obtained by eliminating all components not identified by the slicing algorithms. It is clear that this projection does not satisfy the semantic goal of program slicing because for some values of N and *MAXINT*, different final values of *prod* will be output by the original program and by the projection. A projection that does satisfy the semantic goal (and that would be produced by the slicing algorithm defined in this paper) is shown in Figure 2(e). □

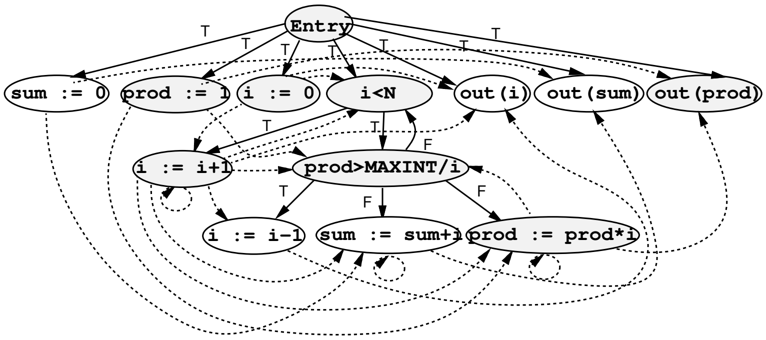
The problem with both algorithms is that they do not correctly detect when unconditional jumps in the program (such as **break**) are required in the program projection. Simply including a vertex for the **break** in the control flow graph, as shown in Figure 3(a), does not solve the problem; both algorithms will still omit the **break**. (The Ottensteins' algorithm, which involves following edges backwards in the program dependence graph, will omit the **break** from the slice because in the program dependence graph—shown in Figure 3(b)—there is no path from the **break** vertex to the vertex that represents the statement "**output**(*prod*)"; in fact, the **break** vertex has *no* outgoing edges, so it will not be included in any slice other than the slice with respect to the **break** itself.)

```
begin
  sum := 0
  prod := 1
  i := 0
  while i < N do
     i := i+1
     if prod > MAXINT/i then
        i := i−1
        break
     fi
     sum := sum+i
     prod := prod*i
  od
  output(i)
  output(sum)
  output(prod)
end
```

(a) Example program



(b) Control flow graph



(c) Program dependence graph

```
begin
  prod := 1
  i := 0
  while i < N do
     i := i+1
     if prod > MAXINT/i then
     fi
     prod := prod*i
  od
  output(prod)
end
```

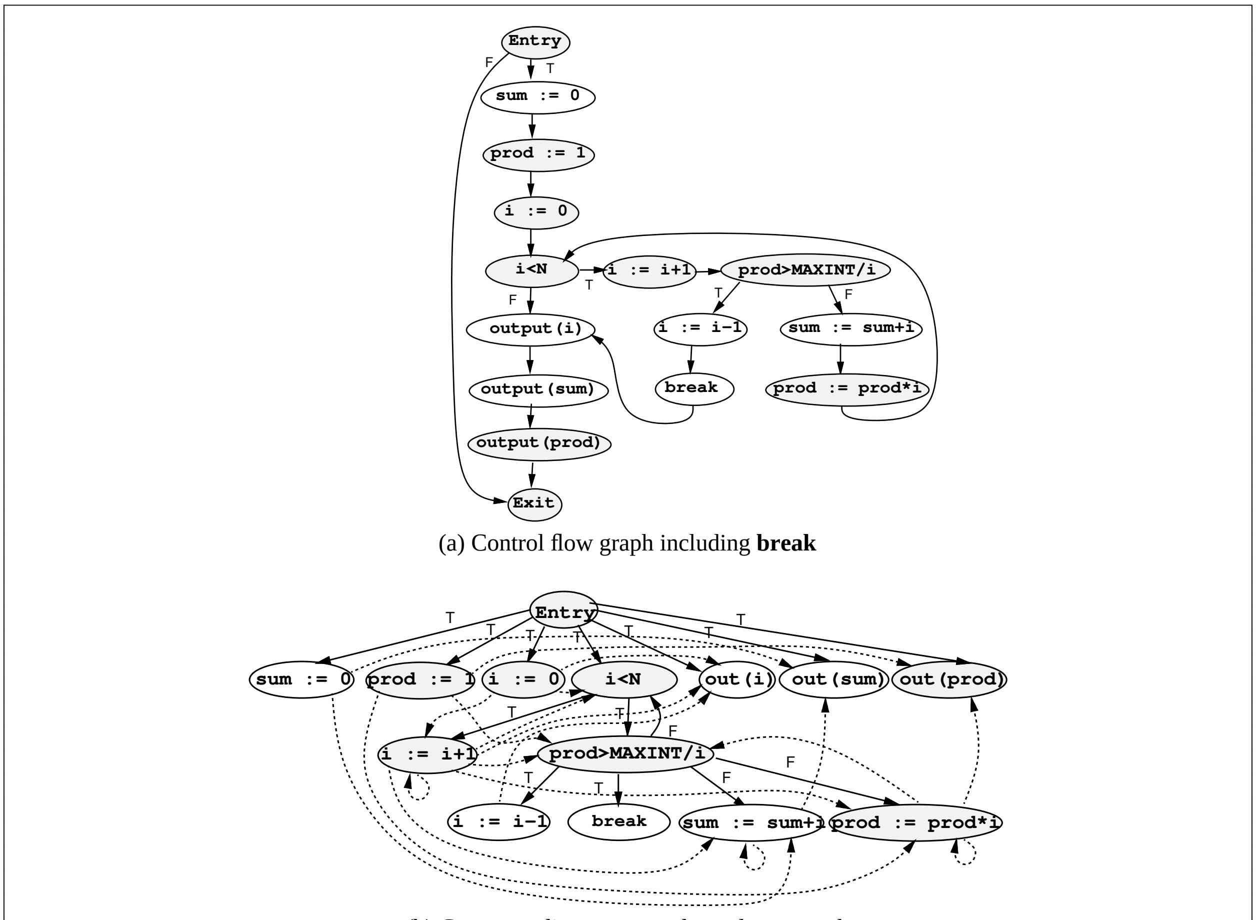
(d) Incorrect projection

```
begin
  prod := 1
  i := 0
  while i < N do
     i := i+1
     if prod > MAXINT/i then
        break
     fi
     prod := prod*i
  od
  output(prod)
end
```

(e) Correct projection

Figure 2. An example program, its control flow graph, its program dependence graph, the (incorrect) projection that would be computed using Weiser's or the Ottensteins' algorithms to slice with respect to **output**(*prod*), and the correct projection.

The main result of this paper is a slicing algorithm for programs with unstructured control flow, and a proof of the correctness of this algorithm; that is, we show that the program projections produced by the algorithm have the desired semantic property: Both the original program and the projection compute the same values at the point of the slice. In fact, we prove a stronger result (as done by Reps and Yang for structured programs [12]): the original program and the projection compute the same values at *every* shared component. The algorithm is in the style of the Ottensteins' algorithm in that it operates on a program dependence graph representation of a program; however, the



(a) Control flow graph including **break**

(b) Corresponding program dependence graph

Figure 3. The same sets of vertices are identified by Weiser's and the Ottensteins' algorithms whether or not a **break** vertex is included in the control flow graph.

program dependence graph is based on an *augmented* control flow graph in which a jump is represented as a pseudo-predicate vertex (that always evaluates to *true*). The jump vertex's *true*-successor is the target of the jump, and its *false*-successor is the vertex that represents the jump statement's continuation (that is, the vertex that would be the jump vertex's successor if it were a "no-op" rather than a jump). We are able to prove that by using this augmented control flow graph, a projection of the program that has the desired semantic property can be formed.

The remainder of the paper is organized as follows. Section 2 provides background material, including a discussion of the language under consideration, and the definitions of the control flow graph and program dependence graph. Section 3 presents our slicing algorithm and gives an outline of its proof of correctness. Section 4 fills in the details of the proof. Section 5 discusses the issues of minimal slices, the extensibility of our slicing algorithm, and alternative methods for constructing slices for programs with arbitrary control flow. Section 6 discusses related work and Section 7 summarizes our results.

2. BACKGROUND

2.1. The Language Under Consideration

To simplify our presentation and focus on the problem of slicing with arbitrary control flow, we consider a simplified language with the following characteristics: Expressions contain only scalar variables and constants; statements are either assignment statements, jump statements (**break, halt, goto**), output statements, conditional statements (**if-then** or **if-then-else**), or loops (**while** or **repeat**). It is easy to generalize our techniques to handle languages with *N*-way branch constructs, such as **case** statements, and other looping constructs, such as **for** loops. The problems of slicing in the presence of multiple procedures, non-scalar variables, and dynamic control flow are orthogonal to the problem discussed here.

2.2. The Control Flow Graph and Its Semantics

In this section we define the control flow graph and its execution semantics. We also give the standard translation from a program written in the language described above to the corresponding control flow graph. In Section 3, we discuss the *augmented* translation that we use as the basis for our slicing algorithm.

The Control Flow Graph

A *control flow graph* (CFG) is any directed, rooted graph[2] that satisfies the following conditions. The CFG has three types of vertices: Fall-through vertices (either assignment statements or output statements), which have one successor, predicate vertices, which have one *true*-successor and one *false*-successor, and an *EXIT* vertex, which has no successors. The root of the CFG is the *ENTRY* vertex, which is a predicate that has the *EXIT* vertex as its *false*-successor. Every vertex is reachable from the *ENTRY* vertex, and the *EXIT* vertex is reachable from every vertex. Edges in the CFG are labeled; the outgoing edges of a predicate vertex are labeled *true* or *false* (as appropriate) and the outgoing edge of a fall-through vertex is labeled *null*.

Standard Control Flow Translation

In the standard translation from a program to a CFG, the CFG includes a vertex for every assignment statement, output statement, and predicate in the program. The edges of the CFG represent the flow of control (the *ENTRY* vertex's *true*-successor is the first statement in the program). Jump statements are not represented directly as vertices in the CFG; instead, they are represented indirectly in that they affect the flow of control, and therefore the targets of some CFG edges.

Example. Figure 2(b) shows the CFG of the program in Figure 2(a). □

Figure 5 presents an attribute grammar for the example language, in which the attributes are used to define the translation from a program to its control flow graph. The grammar is given in the style used in [11], in which the underlying context free grammar defines a program's *abstract* (rather than concrete) syntax. Operator names are used to identify productions uniquely. Each production in the grammar is of the form "x_0: $op(x_1\ x_2\ \cdots\ x_k)$", where *op* is an operator name and each x_i is a nonterminal. Every nonterminal has a synthesized attribute *entry* and an inherited attribute *cont*, both of which represent vertices in the CFG. The constructor Pred(t,v,w) creates a predicate vertex with text t, *true*-successor v, and *false*-successor w, while the constructor FallThrough(t,v) creates a fall-through vertex with text t and successor v.

---

[2]A *directed graph G* consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$, where $E(G) \subseteq V(G) \times V(G)$. Each edge $(b, c) \in E(G)$ is directed from b to c ($b \rightarrow c$); we say that b is the *source* and c the *target* of the edge.

The notation “*attr* = { · · · }” is an upward remote attribute reference; its value is the first instance of a set element encountered on the path to the root in the abstract syntax tree. For example, a **break** statement passes control to the continuation of the innermost enclosing loop. A global symbol table (with operations *insert* and *lookup*) is used to manage the control flow between **Goto** and **Label**. In a pure attribute grammar the symbol table would be threaded through the abstract syntax tree.

The grammar presented in Figure 5 makes precise the relationship between a program’s abstract syntax tree and its CFG. Every vertex in the CFG is associated with the *stmt* production that created the vertex. In the remainder of the paper, we use the term *program component* to refer to an instance of a *stmt* production in a program (*i.e.*, a node in the abstract syntax tree) that creates a CFG vertex (some productions, such as **Label** do not create a vertex). Given a *stmt* subtree T, $vert(T)$ denotes the CFG vertex associated with T’s root production.

We note that there are programs whose standard control flow translation does not yield a CFG. For example, it is possible to create code that is not reachable from *ENTRY* or from which the *EXIT* vertex is unreachable, or to define a program whose control flow translation is not well-defined (for example, a **goto** to an undefined label or a **break** that is not enclosed in a loop). In the remainder of the paper, we consider only programs that yield a CFG under the standard translation.

```
prog:
  Program ( seq ) {
    prog.entry = Pred( "ENTRY", seq.entry, seq.cont)
    prog.cont = seq.cont = FallThrough( "EXIT", null)
  };

seq:
  NullSeq () {
    seq.entry = seq.cont
  }
| Sequence ( stmt seq ) {
    seq1.entry = stmt.entry
    stmt.cont = seq2.entry
    seq2.cont = seq1.cont
  };

stmt:
  NullStmt () {
    stmt.entry = stmt.cont
  }
| While ( expr seq ) {
    stmt.entry = Pred( "expr", seq.entry, stmt.cont)
    seq.cont = stmt.entry
  }
| Repeat ( seq expr ) {
    stmt.entry = seq.entry
    seq.cont = Pred( "expr", stmt.cont, seq.entry)
  }
| IfThen ( expr seq ) {
    stmt.entry = Pred( "expr", seq.entry, stmt.cont)
    seq.cont = stmt.cont
  }
| IfThenElse ( expr seq seq ) {
    stmt.entry = Pred( "expr", seq1.entry, seq2.entry)
    seq1.cont = seq2.cont = stmt.cont
  }
| Assign ( ID expr ) {
    stmt.entry = FallThrough( "ID := expr", stmt.cont)
  }
| Label ( ID ) {
    stmt.entry = stmt.cont
    insert(ID, stmt.entry)
  }
| Break () {
    stmt.entry = { Repeat.cont, While.cont }
  }
| Halt () {
    stmt.entry = { Program.cont }
  }
| Goto ( ID ) {
    stmt.entry = lookup(ID)
  };
```

Figure 5. Abstract syntax for example language with attributes that defines the control flow graph.

Control Flow Graph Semantics

The standard operational semantics for the CFG is defined as follows: Execution starts at the *ENTRY* vertex (which always evaluates to *true*), with an initial state σ; at any moment there is a single point of control together with a state mapping variables to values; the execution of each fall-through or predicate vertex passes control to a single successor (for a predicate vertex, that successor is determined by evaluating the predicate in the current state). The execution of an assignment statement changes the state. Execution terminates normally if *EXIT* is reached (execution can fail to terminate normally because of an infinite loop or an exception such as division by zero). An execution of CFG G on initial state σ is denoted by $G(\sigma)$.

For an execution $G(\sigma)$, we characterize the behavior at a vertex by the sequence of values that arise at that vertex[3]. This is defined as follows: For an assignment statement, the sequence of values assigned to the left-hand-side variable; for an output statement, the sequence of values output; and for a predicate vertex, the sequence of boolean values to which the predicate evaluates. $G(\sigma)(v)$ denotes the sequence of values that arise at vertex v in execution $G(\sigma)$.

The following definition defines what we mean for two vertices in different CFGs to have equivalent behavior. Because a CFG includes a vertex for every program component, this definition also makes precise the semantic goal of slicing a program with respect to component c: To create a projection whose behavior at c is equivalent to that of the original program at c.

DEFINITION. Vertices v_G and v_H of CFGs G and H, respectively, have *equivalent behavior* iff all the following hold:

- For all σ such that both $G(\sigma)$ and $H(\sigma)$ terminate normally, $G(\sigma)(v_G) = H(\sigma)(v_H)$.
- For all σ such that neither $G(\sigma)$ nor $H(\sigma)$ terminates normally, either $G(\sigma)(v_G)$ is a prefix of $H(\sigma)(v_H)$, or vice versa.
- For all σ such that $G(\sigma)$ terminates normally but $H(\sigma)$ does not, $H(\sigma)(v_H)$ is a prefix of $G(\sigma)(v_G)$.
- For all σ such that $H(\sigma)$ terminates normally but $G(\sigma)$ does not, $G(\sigma)(v_G)$ is a prefix of $H(\sigma)(v_H)$.

2.3. The Program Dependence Graph

The program dependence graph (PDG) is defined in terms of a program's control flow graph. The PDG includes the same set of vertices as the CFG, excluding the *EXIT* vertex. The edges of the PDG represent the *control* and *flow* dependences induced by the CFG as follows[4]:

DEFINITION (postdominance). Let v and w be vertices in a CFG. Vertex w *postdominates* vertex v iff $w \neq v$ and w is on every path from v to the *EXIT* vertex. Vertex w *postdominates* the L-branch of predicate vertex v (where L is either *true* or *false*) iff w is the L-successor of v or w postdominates the L-successor of v. While no vertex can postdominate itself, a vertex can postdominate its own L-branch.

---

[3]Note that our definition of a vertex's behavior differs from the more standard definition, which characterizes a vertex's behavior as the sequence of *states* that arise at that vertex during an execution (where a state associates a value with *every* variable in the program). The standard definition is not suitable for program slicing, since the semantic goal of slicing is *not* to preserve the values of all variables, but only to preserve the values of the variables used or defined at the point with respect to which the slice is taken.

[4] In addition to control and flow dependences, program dependence graphs usually include either def-order dependences [5] or output and anti-dependences [4]. These additional edges are not needed for program slicing, and so are omitted from the definition given here. We also do not need to distinguish between loop-independent and loop-carried dependences (which are ill-defined for irreducible control flow anyway).

DEFINITION (control dependence). Let v and w be vertices in a CFG. Vertex w is directly L-control dependent on v (written $v \longrightarrow_c^L w$) iff w postdominates the L-branch of v and w does not postdominate v. A vertex can be directly control dependent on itself. Intuitively, if $v \longrightarrow_c^L w$, then whenever v executes and evaluates to L, w will eventually execute, barring abnormal termination. Furthermore, if v executes and does not evaluate to L, w might not execute. In this way, v directly controls whether or not w executes. Note that a vertex with only one successor in the CFG can never be the source of a control-dependence edge.

Under the standard definition, there is a *flow dependence* from vertex v to vertex w iff vertex v assigns to variable x, vertex w uses x, and there is a path from v to w that does not include an assignment to x (excluding v and w). However, the augmented translation from programs to CFGs that will be introduced in Section 3 causes the CFG to include "non-executable" edges; that is, edges that are never traversed in any execution (namely, the *false* edges out of vertices that represent jump statements). Under the standard definition of flow dependence, there would be dependences that could never be realized in any execution. To eliminate these false dependences (and thus increase the precision of our slicing algorithm), we use the following slightly non-standard definition of flow dependence:

DEFINITION (flow dependence). Let v and w be vertices in a CFG. There is a *flow dependence* from vertex v to vertex w (written $v \longrightarrow_f w$) iff vertex v assigns to variable x, vertex w uses x, and there is a path from v to w that does not include an assignment to x (excluding v and w) and that does not include any non-executable edge.

Example. Figure 2(c) shows the PDG of the program in Figure 2(a). Control-dependence edges are shown using solid arrows; flow dependence edges are shown using dashed arrows. □

3. SLICING PROGRAMS WITH ARBITRARY CONTROL FLOW

In this section we present our slicing algorithm and we sketch a proof that it produces program projections with the desired semantic property: Given program P and component c, our algorithm produces a projection of P such that both P and its projection have equivalent behavior at every shared component, including component c.

3.1. The Slicing Algorithm

As discussed in the Introduction, our slicing algorithm (given in Figure 7) is similar to the Ottensteins' algorithm in that it uses a program dependence graph (PDG) to identify the program components in the slice. Given a PDG and a vertex v from which to slice, Step 1 of the Ottensteins' and our algorithm identifies the subset of the PDG's vertices from which there is a path along control and/or flow dependence edges to vertex v (*i.e.*, Step 1 computes the backwards reflexive transitive closure with respect to v). Step 2 creates a program projection by eliminating *stmt* subtrees that do not correspond to the vertices identified in Step 1. For each vertex w that is not identified by Step 1 of the algorithm, Step 2 eliminates the *stmt* subtree T such that $vert(T) = w$. We discuss the exact details of subtree elimination later in this section.

The important difference between our algorithm and the Ottensteins' is that we use an *augmented* translation from the program to the control flow graph (CFG) from which the PDG is built. In particular, jump statements are explicitly represented in the CFG as pseudo-predicate vertices that always evaluate to *true*. Our augmented control flow translation uses the same translation for structured constructs and **Label** given in Figure 5, but replaces the translations for the unconditional jumps (*i.e.*, **Break, Halt, Goto**) with those in Figure 6. A jump vertex's *true*-successor is the target of the jump; its *false*-successor is the vertex that represents the jump statement's continuation (that is, the vertex that would be the jump vertex's successor if it were a "no-op" rather than a jump).

Representing a jump statement this way causes it to be the source of control dependence edges in the PDG. This

```
stmt:
   Break () {
      stmt.entry = Pred( "break", { Repeat.cont, While.cont }, stmt.cont)
   }
|  Halt () {
      stmt.entry = Pred( "halt", { Program.cont }, stmt.cont)
   }
|  Goto ( ID ) {
      stmt.entry = Pred( "goto ID", lookup(ID), stmt.cont)
   };
```

Figure 6. Augmented control flow translations for jump statements.

function Slice(P: program, c: component of P): program
declare
  v: vertex
  G: control flow graph
  D: program dependence graph
  S: subset of D's vertices
  Q: projection of program P
begin
 /* STEP 0: BUILD THE (AUGMENTED) PDG */
   G := the augmented control flow graph for P
   D := the program dependence graph that corresponds to G
 /* STEP 1: IDENTIFY VERTICES */
   v := vertex in G corresponding to component c
   S := $\{ w \mid v$ is reachable from vertex w in $D \} \cup \{ EXIT \}$
 /* STEP 2: CREATE THE PROGRAM PROJECTION */
   $Q := P$
   eliminate from Q all *stmt* subtrees T such that $vert(T) \notin S$ /* see Figure 9 */
 return(Q)
end

Figure 7. The main result of this paper: A slicing algorithm that correctly handles programs with arbitrary control flow.

in turn allows the jump vertex to be included in the set identified by Step 1 of our algorithm[5].

Example. Figure 8(a) repeats the program of Figure 2(a) and shows the program's augmented CFG. Figure 8(b) shows the vertices, control edges, and some of the flow edges of the corresponding PDG (flow edges that are not relevant to the slice with respect to "**output**(*prod*)" are omitted).

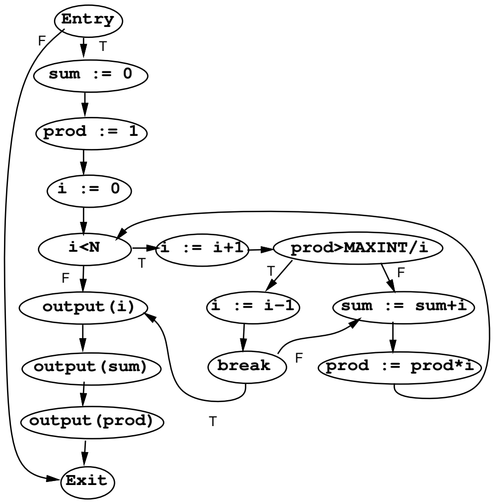
[5] It is important to note that representing jump statements this way in the CFG does not change the semantics of the CFG as defined in Section 2.2. In particular, since a jump is treated as a predicate that always evaluates to *true*, and since the jump vertex's *true*-successor is the target of the jump, it is clear that for every vertex v in the standard CFG G and every initial state σ, the behavior at v when G is executed on σ is the same as the behavior at the corresponding vertex when the augmented CFG is executed on σ.

Note that in this PDG, the **break** vertex has three outgoing control dependence edges (which are not in the PDG of Figure 2(c)). These edges are consistent with the intuition behind control dependence: Removing the **break** might change the number of times the assignments to *sum* and *prod* as well as the evaluation of the loop predicate were performed (and therefore there are control dependence edges from the **break** vertex to the vertices that represent these three components). However, the presence or absence of the **break** has *no* effect on the execution of any statement outside the loop (and therefore there are no control dependence edges from the **break** vertex to a vertex that represents a statement outside the loop).

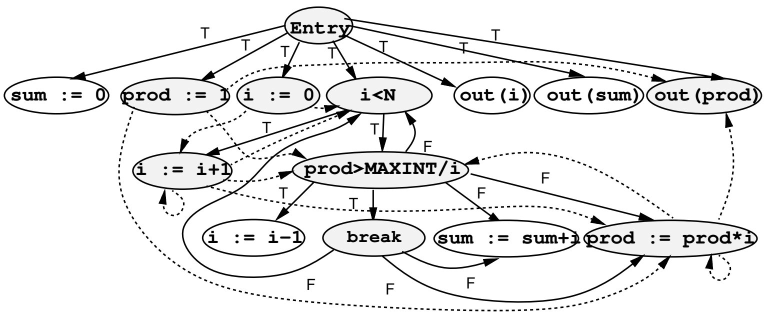
In Figure 8(b), shading is used to indicate the PDG vertices that are identified by Step 1 of our slicing algorithm when slicing with respect to "**output**(*prod*)". Note that the shaded vertices correspond to the program components that are included in the correct program projection shown in Figure 2(e). □

Eliminating a *stmt* subtree *T* that contains no **Label** subtrees is accomplished simply by replacing that subtree by the **NullStmt** subtree (Figure 9(a)). If *T* contains **Label** subtrees, they are sequenced together to replace *T* (Figure 9(b)). The order of the labels in the sequence is not important. If there is no **Goto** to a **Label** subtree in the program, then the **Label** subtree can be eliminated by replacing it with **NullStmt**.

```
begin
  sum := 0
  prod := 1
  i := 0
  while i < N do
     i := i+1
     if prod > MAXINT/i then
        i := i−1
        break
     fi
     sum := sum+i
     prod := prod*i
  od
  output(i)
  output(sum)
  output(prod)
end
```



(a) Example program and its augmented CFG



(b) Corresponding PDG

Figure 8. The example program from Figure 2(a), its augmented CFG, and the corresponding PDG. Shading is used to indicate the PDG vertices identified by the slicing algorithm of Figure 7 when slicing with respect to "**output**(*prod*)".

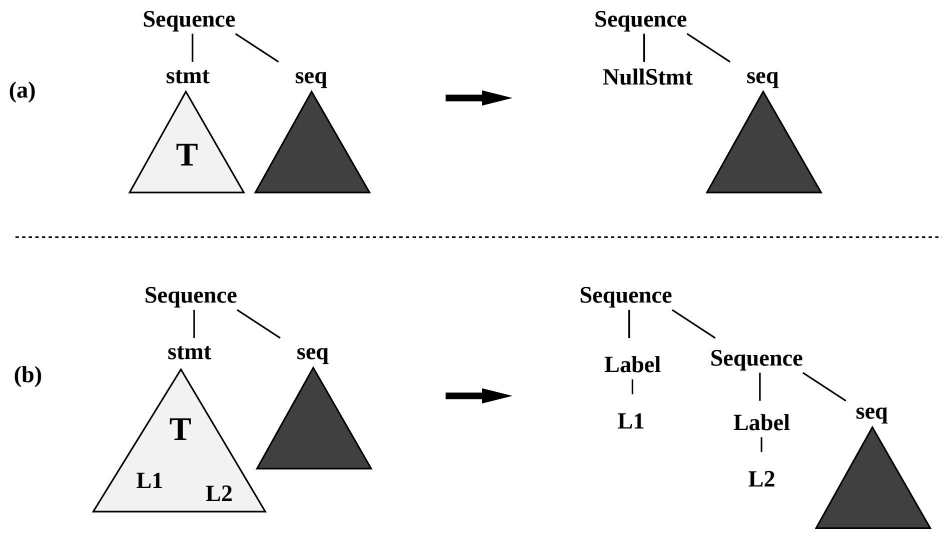


Figure 9. Eliminating a *stmt* subtree without labels (a) and with labels (b).

3.2. Proof of Correctness

In this section we sketch a proof that the slicing algorithm of Figure 7 produces a program projection with the desired semantic property. The details of the proof can be found in Section 4. The proof has two main parts.

A semantics-preserving transformation on CFGs

The first step of the proof is to show that eliminating the vertices not identified by Step 1 of our slicing algorithm is a semantics-preserving transformation on CFGs; that is, the behaviors of all the vertices in the resulting CFG are equivalent to the behaviors of the corresponding vertices in the original CFG[6]. This part of the proof does not rely at all on the augmented translation. That is, the results here are for arbitrary control flow graphs, irrespective of the program from which they were derived.

Example. Figure 10 repeats the (augmented) CFG of Figure 8(b) and shows the CFG that results from eliminating the vertices not identified by Step 1 of the algorithm of Figure 7 when slicing with respect to "**output**(*prod*)". It is clear that the two CFGs have equivalent behavior at all of the shaded vertices. □

The proof that eliminating the CFG vertices not identified by Step 1 of the algorithm is a semantics-preserving transformation relies on the following definitions and lemmas.

[6] To eliminate a vertex x from CFG G: For every vertex a such that there is an edge $a \rightarrow^{L1} x$ and for every vertex b such that there is an edge $x \rightarrow^{L2} b$, remove edges $a \rightarrow^{L1} x$ and $x \rightarrow^{L2} b$; add edge $a \rightarrow^{L1} b$. Remove vertex x. If elimination results in two (or more) edges of the form $v \rightarrow^{L} w$ then eliminate all but one of the edges.

DEFINITION (path-projection). Graph H is a *path-projection* of graph G iff all of the following hold:

(1) The vertices of H are a subset of the vertices of G.

(2) For every path in G (a sequence of vertex, edge-label pairs), if the vertices that are not in H are eliminated along with their outgoing edge labels, then the resulting sequence is a path in H.

(3) For every path PTH in H, there is a path in G whose projection is PTH.

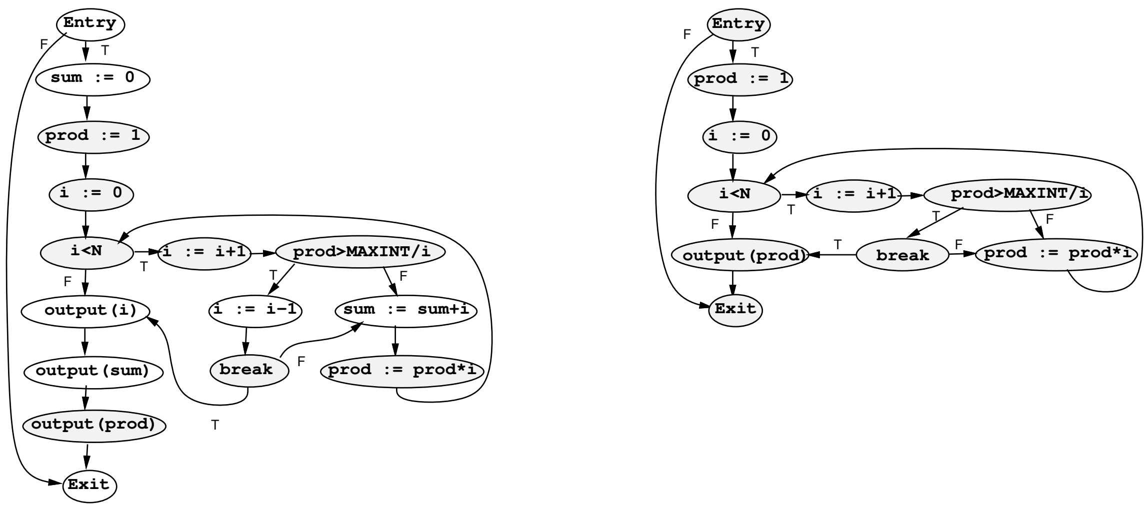
DEFINITION (flow/path-projection). CFG H is a *flow/path-projection* of CFG G iff:

(1) H is a path-projection of G, and

(2) for every vertex $w \in H$, if G induces the flow dependence $v \longrightarrow_f w$, then vertex v is also in H.

Example. Figure 11 shows four CFGs. Both H and J are path-projections of G; however, K is not. This is because G includes the path ((Entry, T)(x >0, F)(y :=0, *null*)(**output**(y), *null*), (**output**(x), *null*)(Exit)), but the path ((Entry, T)(Exit)) is not in K. H is also a flow/path-projection of G, but J is not. This is because vertex "**output**(y)" is in J, graph G induces a flow dependence from "y :=1" to "**output**(y)", but vertex "y :=1" is not in J. □

THEOREM 3.1. (flow/path-projections preserve CFG semantics). If CFG H is a flow/path-projection of CFG G, then the behavior of every vertex in H is equivalent to the behavior of the corresponding vertex in G.
PROOF. See Section 4.1.



(a) Original (augmented) CFG of the example program. Shading indicates the vertices identified by Step 1 of the algorithm of Figure 7 when slicing from "**output**(*prod*)".

(b) After eliminating unshaded vertices.

Figure 10. Eliminating vertices not identified by Step 1 of the slicing algorithm of Figure 7 preserves CFG semantics.

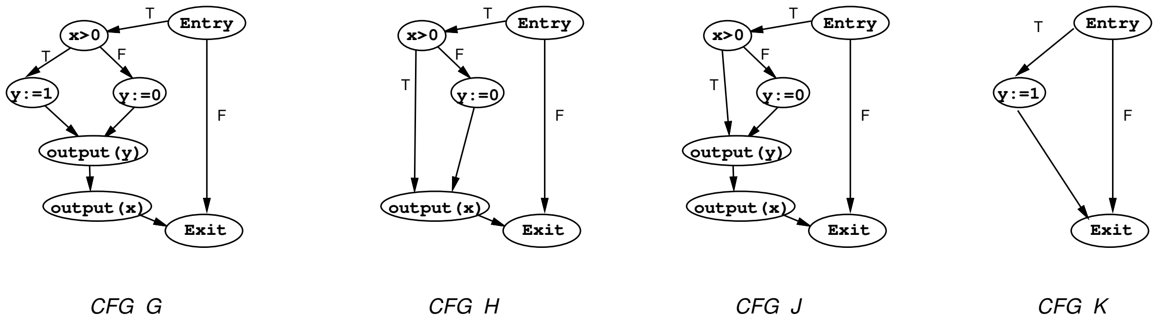


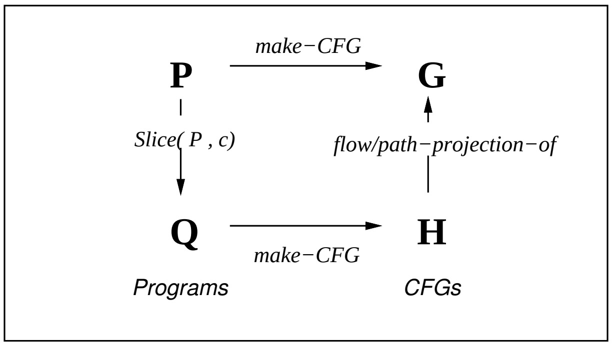
Figure 11. *H* and *J* are path-projections of *G*; *K* is not. *H* is also a flow/path-projection of *G*; *J* is not.

THEOREM 3.2. (Step 1 of our slicing algorithm can be used to produce flow/path-projections). Given CFG *G*, its PDG *D*, and program component *c*, eliminating from *G* the vertices not identified by Step 1 of the algorithm of Figure 7 (applied to *D* and *c*) produces a CFG that is a flow/path-projection of *G*.
PROOF. See Section 4.2.

A semantics-preserving transformation on programs

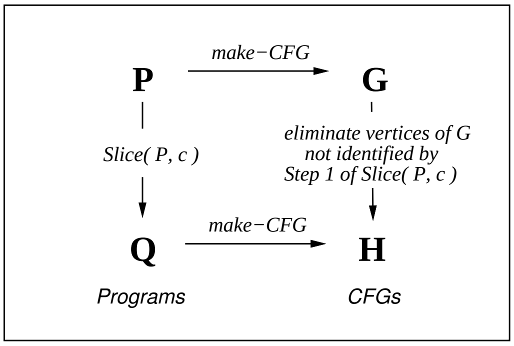
Recall that the goal of program slicing is to produce a projection of a given *program*, not to produce a projection of a given CFG. As illustrated by the example of Figure 2, creating a program projection by eliminating components that do not correspond to the vertices identified by Step 1 of Weiser's or the Ottensteins' algorithms does *not* result in a projection with the desired semantic property (*i.e.*, that elimination operation does not define a semantics-preserving transformation on programs).

The second part of the proof of correctness of our algorithm involves showing that eliminating program components that do not correspond to the vertices identified by Step 1 of our algorithm *is* a semantics-preserving transformation on programs. To prove this, we show that the relationships pictured below hold.



That is, given a program *P* and a component *c*, we show that the program *Q* that results from applying our slicing algorithm to *P* has a CFG *H* that is a flow/path-projection of *P*'s CFG *G*. By the results of the previous section, this means that the vertices of *H* (in particular, the vertex that corresponds to *c*) have behaviors that are equivalent to the behaviors of the corresponding vertices of *G*.

Rather than arguing directly that H is a flow/path-projection of G, we show that H is identical to the CFG obtained from G by eliminating all vertices not identified by Step 1 of our slicing algorithm (*i.e.*, we show that the diagram shown below commutes). It follows from the results of the previous section that H is a flow/path-projection of G.



See Section 4.3 for the proof.

4. PROOF OF CORRECTNESS

This section contains proofs of the results stated in Section 3.2.

4.1. The Behavior of flow/path-projections

This section proves Theorem 3.1. First, we precisely define path-projection. We denote a directed path in a graph by a sequence of (vertex, label) pairs $[(v_1, l_1), ..., (v_n, l_n)]$ such that for any v_i and v_{i+1}, there is an edge $v_i \to^{l_i} v_{i+1}$ in the CFG.[7] Given a path *PTH* and a set of vertices *V*, project(*PTH*,*V*) is defined to be the sequence resulting from deleting from *PTH* each pair (v_i, l_i) such that $v_i \notin V$. We refer to *PTH* as a *generating path* for project(*PTH*, *V*).

Restating the definition of path-projection in these terms, a graph H is a path-projection of graph G iff (1) $V(H) \subseteq V(G)$, and (2) for each path *PTH* in G project(*PTH*, $V(H)$) is a path in H, and (3) for each path in H there is a generating path in G for that path.

The *execution path* of execution $G(\sigma)$ is the (possibly infinite) path executed by $G(\sigma)$. If the execution terminates normally, the path ends with the *EXIT* vertex. Otherwise, the path is infinite or ends at the first point of failure (*i.e.*, a vertex at which an exception occurs). The i^{th} *instance* of a vertex v in $G(\sigma)$, denoted by v^i, is the i^{th} occurrence of vertex v in the execution path of $G(\sigma)$.

The proof of Theorem 3.1 relies on the following lemma. This lemma shows that if H is a flow/path-projection of G, then, for any initial state σ, the execution path of $H(\sigma)$ is a projection of the execution path of $G(\sigma)$ and, furthermore, that intermediate states at corresponding instances in the two executions agree on certain variables. This result directly implies that each corresponding vertex of H and G has equivalent behavior, as discussed later. Since H is a path-projection of G, it may omit some assignment statements that appear in G. Therefore, we cannot expect the intermediate states of $H(\sigma)$ and $G(\sigma)$ to agree on all variables. Instead, we show that the intermediate states before an instance v^i in $H(\sigma)$ and its corresponding instance in $G(\sigma)$ are guaranteed to agree on all variables that are live before vertex v in CFG H.

---

[7]The value of label l_n is unconstrained.

DEFINITION. state($G(\sigma)$, v^i) denotes the state immediately before the execution of instance v^i in $G(\sigma)$.

DEFINITION. live_before(G,v) = { x | there is a path in G from v to a vertex w that uses variable x such that no vertex in the path (excluding w but including v) contains an assignment to x }.[8]

LEMMA 4.1. If CFG H is a flow/path-projection of CFG G then for any initial state σ, if *PTH* is a prefix of $G(\sigma)$'s execution path then

(1) project(*PTH*, $V(H)$) is a prefix of $H(\sigma)$'s execution path, and

(2) for every instance v^i in *PTH* such that $v \in V(H)$, $\forall x \in$ live_before(H,v):

$$\text{state}(G(\sigma), v^i)(x) = \text{state}(H(\sigma), v^i)(x).$$

PROOF. By induction on n, the number of vertices in *PTH* that are in $V(H)$.

Base Case: $n = 1$. In this case, *PTH* = (*ENTRY*). Trivial.

Induction Step: Assume that the lemma holds when *PTH* includes n vertices that are in $V(H)$. Show that the lemma holds when *PTH* includes $n+1$ vertices that are in $V(H)$. Let v^i be the n^{th} vertex in *PTH* that is in $V(H)$; let w^j be the $n+1^{st}$ vertex in *PTH* that is in $V(H)$. *PTH* can be considered as the concatenation of three paths (see Figure 12(a)):

PTH1 is the prefix of *PTH* that includes every (vertex, edge) pair up to and including (v^i, l).

PTH2 is the middle part of *PTH* that includes every (vertex, edge) pair in *PTH* from the end of *PTH1* up to and including (w^j, m).

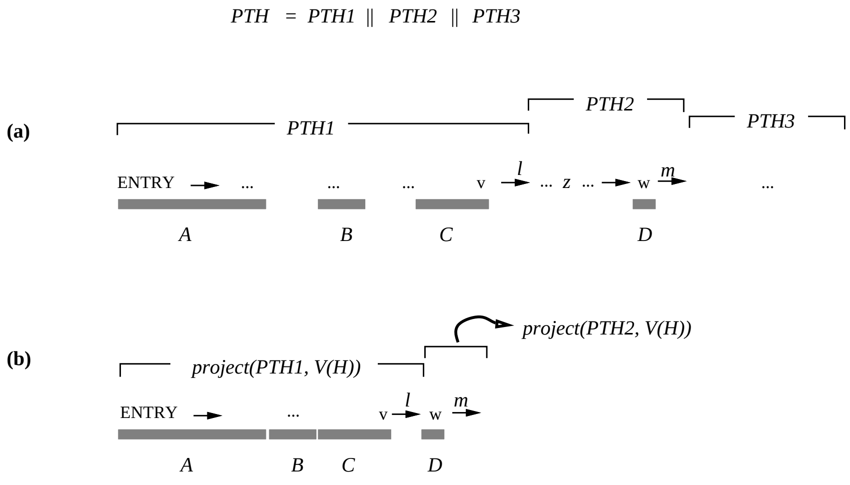


Figure 12. An execution path in CFG G (a) and its projection in CFG H (b). The grey bars denote those vertices in the path in CFG G that are in $V(H)$.

[8]We can also add the restriction that no edge in the path is a non-executable edge (as done for flow dependences). This does not affect the correctness of the proof.

PTH3 (possibly empty) is the suffix of *PTH* that includes every (vertex, edge) pair from the end of *PTH2* to the end of *PTH*. Note that *PTH3* does not include any vertex in $V(H)$.

By point (1) of the Induction Hypothesis, project(*PTH1*, $V(H)$) is a prefix of $H(\sigma)$'s execution path. By point (2), $\forall x \in$ live_before(H,v), state($G(\sigma),v^i$)(x) = state($H(\sigma),v^i$)(x). Every variable used at v^i is in live_before(H,v), so the value computed at v^i is the same in the two executions. Since H is a path-projection of G, if v is a fall-through vertex, then w will be the next vertex to execute after v^i in $H(\sigma)$. If v is a predicate vertex, then since the value computed at v^i is the same in $H(\sigma)$ as in $G(\sigma)$, w will be the next vertex to execute after v^i in $H(\sigma)$. *PTH3* does not include any vertices in $V(H)$. Therefore, project(*PTH*, $V(H)$) is a prefix of $H(\sigma)$'s execution path (see Figure 12(b)).

We now show that $\forall x \in$ live_before(H,w): state($G(\sigma),w^j$)(x) = state($H(\sigma),w^j$)(x). Suppose that $\exists x \in$ live_before(H,w) such that state($G(\sigma),w^j$)(x) $\neq$ state($H(\sigma),w^j$)(x). Since $x \in$ live_before(H,w), either $x \in$ live_before(H,v) and v does not assign to x, or v assigns to x. By the Induction Hypothesis, if $x \in$ live_before(H,v), state($G(\sigma),v^i$)(x) = state($H(\sigma),v^i$)(x). Furthermore, the difference in the value of x at w^j cannot have been caused by an assignment at v, since the same value is computed at v^i in both executions. Therefore, there must be a vertex z that occurs in *PTH2* between v^i and w^j such that: (a) z assigns to x, and (b) there is no other vertex that assigns to x between z and w^j. Let PTH_H be a path in H from w to vertex y that induces x's membership in live_before(H,w) (y uses variable x and no vertex in PTH_H, except y, assigns to x). Let PTH_G be a generating path for PTH_H. Without loss of generality, assume that PTH_G begins with w and ends with y. Since H is a flow/path-projection of G, no vertex in PTH_G, except y, can assign to x (otherwise PTH_H would contain an assignment to x that occurs before y). This implies that G induces the flow dependence $z \longrightarrow_f y$. Since $y \in V(H)$ and H is a flow/path-projection of G, z must be a member of $V(H)$. Contradiction. □

Let CFG H be a flow/path-projection of CFG G. We now show that lemma 4.1 implies that for all $v \in V(H)$, vertex v has equivalent behavior in CFGs G and H, as defined in Section 2.1. Let *PTH* be a prefix of $G(\sigma)$'s execution path. By point (1) of lemma 4.1, project(*PTH*,$V(H)$) is a prefix of $H(\sigma)$'s execution path. Point (2) implies that the value of the expressions in corresponding instances v^i in *PTH* and project(*PTH*,$V(H)$) are the same. We make the following observations with respect to terminating and nonterminating:

- If $G(\sigma)$ terminates normally then its execution path is finite and ends with *EXIT*, so $H(\sigma)$ must terminate normally. In this case, for all $v \in V(H)$, $H(\sigma)(v) = G(\sigma)(v)$.
- The previous point implies that it is not possible for $G(\sigma)$ to terminate and $H(\sigma)$ not to terminate.
- If $G(\sigma)$ does not terminate normally, then $H(\sigma)$ may or may not terminate normally, depending on whether or not the non-terminating or exception-producing computation in G is present in H. In either case, for all $v \in V(H)$, $G(\sigma)(v)$ is a prefix of $H(\sigma)(v)$.

4.2. A Semantics-preserving Operation on CFGs

In this section we argue that eliminating from CFG G the vertices not identified by Step 1 of the algorithm of Figure 7 produces a CFG H that is a flow/path-projection of G. In fact, we only need to argue that H is a path-projection of CFG G. The slicing algorithm guarantees that if a vertex w is included in H then all of w's flow dependence predecessors in G's PDG are also in H. Therefore, if H is a path-projection of G, it is guaranteed to be a flow/path-projection too. We also show that CFG H is the minimal flow/path-projection that contains the vertex to which the slicing algorithm was applied.

The set of vertices S identified by Step 1 of the algorithm has the property that if w is in S and $v \longrightarrow_c w$ in CFG G's PDG, then v is in S. The following lemma shows that for any set of vertices that satisfies this property, eliminating the vertices not in S from G yields a graph H that is a CFG and a path-projection of G.

LEMMA 4.2. Let G be a CFG and let S be a set of vertices in G such that if $w \in S$ and $v \longrightarrow_c w$ is in G's PDG, then $v \in S$. Eliminating the vertices not in S from CFG G yields a graph H that is a CFG and a path-projection of G.

PROOF. The vertex elimination operation has two properties that are trivial to prove: first, the order in which the vertex elimination operations are applied does not affect the resulting graph; second, if G is a CFG and vertex elimination is applied to some vertex (other than *ENTRY* and *EXIT*) then the resulting graph is a path-projection of G and meets the reachability requirements of a CFG. The problem is that a single application of the vertex elimination operation is not guaranteed to produce a graph that is a CFG. This happens because the operation may create a graph that contains a vertex with two distinct L-successors. For example, consider the graph that results from eliminating the vertex $(x>0)$ from CFG G in Figure 11: the *ENTRY* vertex in this graph has two T-successors, $(y:=1)$ and $(y:=0)$.

To complete the proof we must show that there is no vertex with two distinct L-successors in H. The proof is by contradiction. Suppose H contains a vertex v with distinct L-successors y and z. Since H is formed from G by eliminating vertices of G, H is a path-projection of G, as discussed before. Let P_1 be a generating path in G for $v \rightarrow^L y$ and let P_2 be a generating path in G for $v \rightarrow^L z$. Let P'_1 be the tail of P_1 and let P'_2 be the tail of P_2. The first vertex in both P'_1 and P'_2 is the L-successor of v in CFG G. The only vertex in P'_1 that is in $V(H)$ (or equivalently, S) is y. Similarly, the only vertex in P'_2 that is in $V(H)$ is z.

It is impossible for both y and z to postdominate each other in G. Without loss of generality, assume that z does not postdominate y in G. Since z cannot occur in P'_1, it follows that z cannot postdominate any vertex in P'_1. Let a be the last vertex in P'_2 such that $z \neq a$ and z does not postdominate a in G (a vertex with these two properties must exist: for example, the first vertex in P'_2). Let b be the successor of a in P'_2. It is clear that either $z = b$ or z **pd** b. Therefore, G induces $a \longrightarrow_c z$, implying that $a \in S$. However, the only vertex in P'_2 that is in S is z and we already have that $z \neq a$. □

The next lemma shows that backwards closure over control dependence is necessary for creating CFG path-projections. That is, if CFG H is a path-projection of CFG G, $w \in V(H)$ and G induces $v \longrightarrow_c w$, then $v \in V(H)$. This implies that our algorithm creates the minimal flow/path-projection that includes the slicing vertex.

LEMMA 4.3. If CFG H is a path-projection of CFG G, G induces $v \longrightarrow_c^L w$ and $w \in V(H)$, then $v \in V(H)$.

PROOF. Suppose that G induces $v \longrightarrow_c^L w$, $w \in V(H)$, and $v \notin V(H)$. Let P_1 be a path in G from *ENTRY* to v. Let z be the last vertex in P_1 that is in $V(H)$ ($z \neq v$ since $v \notin V(H)$). Let L' be the label on the outgoing edge from z in P_1. Because $v \longrightarrow_c^L w$, the following two paths exist in G: P_2, a w-free path from a successor of v to *EXIT*; P_3, an acyclic path from v's L-successor to w. Since w postdominates the L-branch of v, w postdominates every vertex in P_3 (except itself). It is clear that w cannot postdominate any vertex in P_2. Therefore, P_2 and P_3 have no vertices in common.

Since H is a path-projection of G and z is the last vertex in P_1 that is in $V(H)$, z must have an L'-successor in project$(P_2, V(H))$ and an L'-successor in project$(P_3, V(H))$. As shown above, these two vertices must be distinct, which means H is not a CFG. Contradiction. □

4.3. A Semantics-preserving Operation on Programs

This section shows that eliminating *stmt* subtrees that do not correspond to the vertices identified by Step 1 of our algorithm is a semantics-preserving transformation on *programs*. To prove this, we show that given a program P and a component c, the program Q that results from applying our slicing algorithm to P has a CFG H that is a flow/path-projection of P's CFG G. Rather than arguing directly that H is a flow/path-projection of G, we show that H is identical to the CFG obtained from G by eliminating all vertices not identified by Step 1 of our slicing algorithm. The results of the previous section imply that this CFG is a flow/path-projection of G.

The proof of this result focuses on the relationship between transitive control dependence and a program's abstract syntax tree. Given a CFG G and vertex v, transCD(G,v) denotes the set of vertices that are reflexively and transitively control dependent on v (*i.e.*, $\{ w \mid v \longrightarrow_c^* w \}$). This set can be equivalently defined as the set of vertices reachable from v in G via a path that does not include the immediate postdominator of v.[9] The proof has three main parts:

(1) We first show that (under the augmented control flow translation) the CFG vertices generated by productions in *stmt* subtree T are a subset of transCD(G,*vert*(T)). This implies that when the slicing algorithm eliminates a *stmt* subtree (because *vert*(T) is not in the set S identified by Step 1) it does not eliminate any vertices that are in S. (Section 4.3.1).

(2) We next show that eliminating the subtrees (see Figure 9 in Section 3.1) from program P that correspond to vertices in transCD(G,v) yields a program Q with CFG H that is identical to the CFG resulting from eliminating the vertices in transCD(G,v) from G (Section 4.3.2). Lemma 4.2 guarantees that eliminating the vertices in transCD(G,v) from CFG G produces a CFG. The edge set of H is:

$$\{\, y \rightarrow^L z \mid \text{neither } y \text{ nor } z \text{ is in transCD}(G,v) \text{ and } y \rightarrow^L z \text{ is in CFG } G \,\}$$
$$\cup\ \{\, y \rightarrow^L \mathrm{ipd}(G,v) \mid y \notin \text{transCD}(G,v)\text{, and } \exists z \in \text{transCD}(G,v) \text{ such that } y \rightarrow^L z \text{ is in CFG } G \,\}$$

Figure 13 illustrates the effect of eliminating the vertices in transCD(G,v) from CFG G.

(3) Let S be the set of vertices identified by the slicing algorithm. Eliminating the vertices in $V(G)$–S corresponds to eliminating multiple transCD sets rather than just one transCD set (as in point (2)). Using the above two results, we show the main result of this section: eliminating the subtrees from program P that correspond to vertices in $V(G)$–S yields a program whose CFG is identical to the CFG obtained by

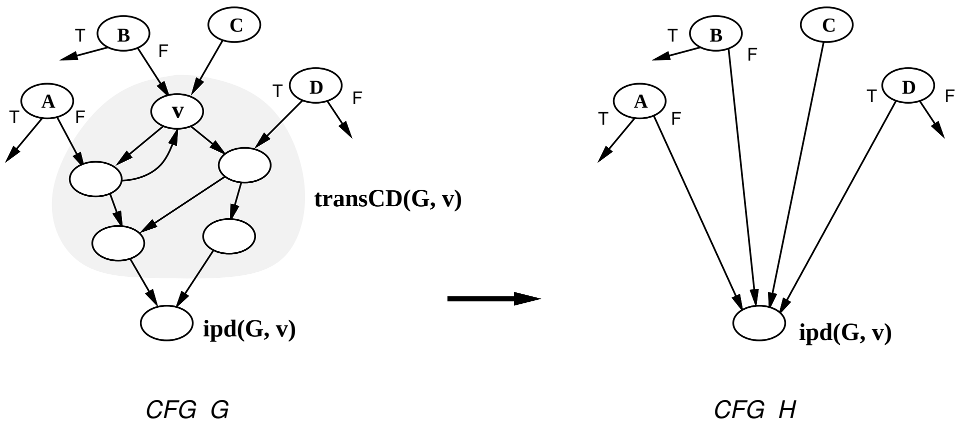


Figure 13. Eliminating the vertices in transCD(G,v).

[9]The immediate postdominator of a vertex v, denoted by ipd(G,v), is the postdominator of v such that there is no vertex w such that ipd(G,v) **pd** w **pd** v.

eliminating the vertices in $V(G)–S$ from G (Section 4.3.3).

4.3.1. The relationship between transCD(G,v) and *stmt* subtrees

Consider any *stmt* subtree T in a program and the program's augmented CFG. Let $Verts(T)$ be the set of CFG vertices defined by the productions in *stmt* subtree T. In the augmented translation, every vertex in $Verts(T)$ is reachable from $vert(T)$ and no vertex in $Verts(T)$ postdominates $vert(T)$ (this is clearly true for any *stmt* subtree that does not contain a jump statement, even under the standard translation; because jump statements generate an edge to their continuation as well as to their target in the augmented translation, it is also true for subtrees containing jump statements). Therefore, $Verts(T) \subseteq$ transCD($G, vert(T)$). However, since unconditional jumps may transfer control to any place in the program, transCD($G, vert(T)$) may contain other vertices. It is clear that if $w \in$ transCD(G,v) then transCD(G,w) $\subseteq$ transCD(G,v). Therefore, if S is a *stmt* subtree and $vert(S) \in$ transCD($G, vert(T)$) then

$$Verts(S) \subseteq \text{transCD}(G, vert(S)) \subseteq \text{transCD}(G, vert(T)).$$

This implies that any transCD(G,v) can be expressed as the union of the *Verts* sets of a set of subtrees. We say that a *stmt* subtree T is contained in a set of vertices V iff $Verts(T) \neq \emptyset$ and $Verts(T) \subseteq V$.

Example. In Figure 8(b), transCD(G, **break**) corresponds to the subtree for the **while** loop. Figure 14 presents four example programs that further illustrate this correspondence. In each program, the boxes outline the subtrees that are contained in the transCD set for the jump statement identified with a bullet. In (a), the **halt** transfers control to the end of the program, so transCD(G, **halt**) = { **halt**, Q, C, R, D, E, F }. In (b), there is an edge from **goto L** to Q and an edge from **goto L** to D; transCD(G, **goto L**) = { **goto L**, D, E }. In (c), the presence of the **break** causes the entire **while** loop to be contained in transCD(G, **goto L**). Case (d) presents an example of irreducible control flow. □

```
begin
  A
  if P then
     B
       while Q do
          C
          if R then
•            halt
          fi
          D
       od
       E
  fi
  F
end
```

(a)

```
begin
  A
L:while Q do
     B
     if R then
        C
•          goto L
     fi
        D
        E
  od
  G
end
```

(b)

```
begin
  A
 L:while Q do
      B
      if R then
         C
•         goto L
      fi
      D
      if S then
         break
      fi
   od
  E
end
```

(c)

```
begin
   A
   if P then
      B
     •  goto L
   fi
     C
   repeat
        D
        E
L:      F
        G
   until Q
   H
end
```

(d)

Figure 14. The subtrees contained in the transCD sets for the statement marked with •.

4.3.2. Vertex elimination and subtree elimination commute for transCD(G,v)

Let P be a program with CFG G. Under the augmented translation, eliminating a subtree from program P yields a program that defines a CFG (this is not necessarily true under the standard translation). Let P_i be the program with CFG G_i resulting from eliminating i subtrees in transCD(G,v) from program P. We show (by induction) that the following invariant holds for all i: the edge set of G_i minus the set $\{\, y \to z \mid$ both y and z are in transCD(G,v) $\}$ is

$$
\begin{aligned}
&\{\, y \to^L z \mid \text{neither } y \text{ nor } z \text{ is in transCD}(G,v) \text{ and } y \to^L z \text{ is in CFG } G \,\} \\
\cup\ &\{\, y \to^L z \mid y \notin \text{transCD}(G,v),\ z \in \text{transCD}(G,v) \cup \text{ipd}(G,v), \\
&\qquad \text{and } \exists z' \in \text{transCD}(G,v) \text{ such that } y \to^L z' \text{ is in CFG } G) \,\}
\end{aligned}
$$

Let Q be the program resulting from eliminating all subtrees in transCD(G,v) from program P. Since all the subtrees contained in transCD(G,v) have been eliminated, the CFG of program Q does not contain any of the vertices in transCD(G,v). By the invariant, the CFG of program Q is:

$$
\begin{aligned}
&\{\, y \to^L z \mid \text{neither } y \text{ nor } z \text{ is in transCD}(G,v) \text{ and } y \to^L z \text{ is in CFG } G \,\} \\
\cup\ &\{\, y \to^L \text{ipd}(G,v) \mid y \notin \text{transCD}(G,v), \text{ and } \exists z \in \text{transCD}(G,v) \text{ such that } y \to^L z \text{ is in CFG } G \,\}
\end{aligned}
$$

which is exactly the graph that results from eliminating all vertices in transCD(G,v) from CFG G. The proof of the above invariant relies on the following lemma, which characterizes how the elimination of a single *stmt* subtree from a program affects control flow. Let $T_P.attr$ denote the value of the attribute *attr* in the root production of subtree T in program P.

LEMMA 4.4. Let P be a program with CFG G. Let Q be the program resulting from eliminating *stmt* subtree T from program P. The edge set of program Q's CFG is:

$$
\begin{aligned}
&\{\, y \to^L z \mid \text{neither } y \text{ nor } z \text{ is in } \mathit{Verts}(T) \text{ and } y \to^L z \text{ is in CFG } G \,\} \\
\cup\ &\{\, y \to^L T_P.\mathit{cont} \mid y \notin \mathit{Verts}(T) \text{ and } \exists z \in \mathit{Verts}(T) \text{ such that } y \to^L z \text{ is in CFG } G \,\}
\end{aligned}
$$

PROOF. Subtree T is replaced by the **NullStmt** subtree (or a sequence of **Label** subtrees). Note that $T_Q.entry = T_Q.cont = T_P.cont$. The proof of this lemma follows from the following observations: any attribute in program P whose value is a vertex in *Verts*(T) has the value $T_P.cont$ in program Q; any attribute in program P whose value is a vertex not in *Verts*(T) has the same value in program Q. □

We are now in a position to prove the invariant. Let P be the original program with CFG G. The proof is by induction on the number of subtrees in transCD(G,v) that have been eliminated from program P.

Base Case: No subtrees eliminated. The invariant clearly holds.

Induction Step: Suppose that the invariant is true after the elimination of n subtrees contained in transCD(G,v) from program P. Let P' be the resulting program with CFG G'. Let T be a *stmt* subtree in transCD(G,v) that is in program P' and let Q be the program (with CFG H) resulting from eliminating T from P'. By lemma 4.4, the edge set of H is:

$$
\begin{aligned}
&\{\, y \to^L z \mid \text{neither } y \text{ nor } z \text{ is in } \mathit{Verts}(T) \text{ and } y \to^L z \text{ is in CFG } G' \,\} \\
\cup\ &\{\, y \to^L T_{P'}.\mathit{cont} \mid y \notin \mathit{Verts}(T) \text{ and } \exists z \in \mathit{Verts}(T) \text{ such that } y \to^L z \text{ is in CFG } G' \,\}
\end{aligned}
$$

If we can show that $T_{P'}.cont \in$ transCD(G,v) $\cup$ ipd(G,v) then CFG H satisfies the invariant (since *Verts*(T) $\subseteq$ transCD(G,v) and CFG G' satisfies the invariant). Suppose that $T_{P'}.cont \notin$ transCD(G,v) $\cup$ ipd(G,v). Because of the augmented translation, there must be a vertex $y \in$ *Verts*(T) such that $y \to T_{P'}.cont$ is in CFG G' (this is clearly true if T does not contain a jump statement, even under the standard translation; because jump statements generate an edge to their continuation as well as to their target in the augmented translation, it is also true if T contains jump statements). The existence of this edge violates the invariant, which was assumed to have held for G'. □

4.3.3. Putting it all together

Let S be the set of CFG vertices identified by the slicing algorithm. Let $V = V(G) - S$. For every vertex $v \in V$, transCD(G,v) $\subseteq V$ (equivalently, transCD(G,v) $\cap$ $S = \emptyset$). Eliminating the vertices in V corresponds to eliminating a set of transCD sets, which corresponds to a set of subtrees in program P. Let Q be the program (with CFG H) resulting from eliminating the subtrees associated with V. We show (by induction on the size of V) that the CFG H is identical to the CFG obtained by eliminating the vertices in V from CFG G.

Base Case: $|V| = 1$. That is, $V = \{v\}$, so transCD(G,v) = $\{v\}$. The result of the previous section implies that CFG H is the CFG resulting from eliminating v from CFG G.

Induction Step: Assume that the result is true for V of size less than n. Suppose that V is of size n. Let v be a vertex in V. Let P' be the program resulting from eliminating the subtrees in transCD(G,v) from program P, and let G' be the CFG of P'. By the results of the previous section, G' is the CFG resulting from eliminating the vertices in transCD(G,v) from CFG G.

Let $V' = V -$ transCD(G,v). It can be shown (see lemma below) that for any vertex w in G', transCD(G',w) = transCD(G,w) − transCD(G,v). This fact implies the following: (1) the subtrees of P' contained in V' are the subtrees of P contained in V minus the subtrees of P contained in transCD(G,v); (2) for every vertex $w \in V'$, transCD(G',w) $\subseteq V'$. The size of V' is clearly less than n. By the Induction Hypothesis, eliminating the subtrees in V' from P' yields a program Q whose CFG H is identical to the CFG resulting from eliminating the vertices in V' from G'. Program Q is the program that results from eliminating all subtrees in V from program P and CFG H is the CFG that results from eliminating the vertices in V from G. This proves our main result. □

LEMMA. Let G be a CFG and let G' be the CFG resulting from eliminating the vertices in transCD(G,v) from G. For any vertex w in G', transCD(G',w) = transCD(G,w) − transCD(G,v).
PROOF. Since G' is a path-projection of G, for any distinct pair of vertices (y,z) in G', y **pd** z in G' iff y **pd** z in G.

Any path $PTH_{G'}$ in G' that starts with w and contains no postdominators of w contributes all its vertices to transCD(G',w). Any generating path in G for PTH_G must contribute the same vertices to transCD(G,w). None of these vertices are in transCD(G,v). Therefore, transCD(G',w) $\subseteq$ transCD(G,w) − transCD(G,v).

Any path PTH_G in G that starts with w and contains no postdominators of w contributes all its vertices to transCD(G,w). Any vertices in the projection of PTH_G in G' must be in transCD(G',w). This projected path does not include vertices from transCD(G,v). Therefore, transCD(G,w) − transCD(G,v) $\subseteq$ transCD(G',w). □

5. MINIMALITY AND EXTENSIONS

This section addresses some interesting issues regarding slicing programs with arbitrary control flow.

5.1. Issues of minimality

A slicing algorithm identifies a program projection that behaves similarly to the original program at some point of interest. As has been noted before, the usefulness of a slicing algorithm is inversely proportional to the size of the slices it produces. While it is an undecidable problem to find slices of minimal size, it would be possible to employ common compiler optimizations to further reduce the size of slices. For example, copy propagation could be used to prune away copy chains from a slice, as shown below (of course, some renaming may need to be done also):

```
x := x+1        x := x+1
y := x
z := y          z := x
```

We believe that smaller slices are useful, up to a point. In this paper, we have shown that our slicing algorithm based on the program dependence graph produces programs that are flow/path-projections of the original program's control flow graph. That is, they preserve paths of the original program (modulo projection) and the flow of values between components. While these properties are useful for proving the semantic results in Section 4.1, they also are intuitively appealing. A slice that does not preserve the paths in a program or the flow of values amongst its components may compute the same result as the original program, but does so in a different way than the programmer originally intended. Flow/path-projections retain the *structure* of the computation as well as its result.

We also have shown that our algorithm identifies the minimal flow/path-projection of a program that includes a particular component from the original program (see Section 4.2). In particular, transitive control dependence identifies the vertices that must necessarily be included in a slice in order to form a path-projection. Of course, the flow/path-projections are minimal with respect to the *augmented* control flow translation, rather than the standard translation. There are cases where a flow/path-projection that is minimal under the augmented translation is not minimal under the standard translation. In the example below, program *Y* is the projection that results from slicing program *X* with respect to *A* (using the augmented translation and assuming no flow dependences). However, under the standard translation, the control flow graphs of programs *Y* and *Z* are identical, but program *Z* is clearly smaller than program *Y*.

X	*Y*	*Z*
if P **then**	**if** P **then**	**if** P **then**
goto L	**goto** L	
M: A	M: A	A
fi	**fi**	**fi**
goto N	**goto** N	
L: B	L:	
goto M	**goto** M	
N: C	N:	

However, it is possible to show that for certain languages with limited unstructured control flow, our algorithm also produces programs that are minimal flow/path-projections with respect to the standard control flow translation. One example of such a language is a structured language with multi-level **break** statements that pass control to the continuation of a specified enclosing control construct, be it a loop or conditional. Furthermore, as we show in Section 5.3, if we drop the requirement that the resulting program be a projection of the original program then it is easy to construct programs that are minimal flow/path-projections with respect to the standard control flow translation.

5.2. Other control constructs

The language considered in this paper has arbitrary control flow, due to the inclusion of the **goto** statement. It also has looping and conditional constructs found in many languages. However, the question naturally arises: do the results of this paper extend to other control constructs, such as **for** loops and **switch** statements? As we have shown, the program dependence graph can be used to form flow/path-projections of completely arbitrary control flow graphs. However, to ensure that the program projection operation works (see the commutative square on page 13) control constructs must satisfy a few simple properties.

A looping construct must generate a vertex *v* in the control flow graph such that: (1) *v* passes control to the continuation of the loop construct (*i.e.*, there is a loop exit); (2) every vertex generated by the (abstract syntax) subtrees enclosed by the looping construct is reachable from *v*. A **for** loop meets these requirements. However, a construct such as **loop-forever** does not. Fortunately, it is usually possible to translate a construct so that by the addition of dummy vertices and non-executable edges, it meets the requirements. For example, a **loop-forever** construct can be treated as a **while** loop where the predicate is *true*. This results in a dummy vertex with an outgoing *false* edge that is not executable.

A selection construct must generate a vertex *v* such that: (1) every vertex generated by subtrees enclosed by the selection construct is reachable from *v*; (2) every subtree immediately enclosed by the selection construct passes control to the continuation of the selection construct; The translations of selection constructs such as **if-then**, **if-then-else**, and **switch** meet these requirements.

5.3. Alternative methods for slicing

We have defined the slice of a program to be a projection of that program. That is, the program slice must be formed by eliminating statements from the original program. Because one of the major applications of slicing is debugging, this is a natural restriction. Presenting the programmer with a slice that does not resemble the original program is clearly unsatisfactory.

It is certainly possible to construct programs that meet the semantic goal of slicing but are not program projections. For example, given a program *P* with standard CFG *G*, one could construct the minimal flow/path-projection of *G* with respect to some vertex and then synthesize a program from that CFG using a structuring algorithm such as Baker's [2]. However, in a language with unstructured control flow, there can be many programs with the same CFG. The program that results from such an approach may not be a projection of the original program, even though it meets the semantic goal (because its CFG is a flow/path-projection of the original program's CFG).

6. RELATED WORK

As mentioned previously, Weiser defined the first program slicing algorithm [13]. The Ottensteins defined a more efficient program slicing algorithm using the program dependence graph [10]. Neither algorithm handles unstructured control flow correctly.

Reps and Yang gave the first formal proof that the program slices formed by using the program dependence graph have the desired semantic property [12]. Furthermore, they showed that slicing using the program dependence graph guarantees equivalent behavior at every point in the slice (not just at the slicing vertex). However, they proved this only for programs with structured control flow. We have shown that the program dependence graph can be used to slice programs with arbitrary control flow with the guarantee of equivalent behavior at every point (see Section 3.2). We note that Reps and Yang defined a program slice to allow the possible reordering of program statements (including conditionals and loops). Under their definition, the second program shown below would be a slice of the first (and vice versa):

a := 1	b := 2
b := 2	a := 1
c := a+b	c := a+b

Although the ordering of statements in the two programs differs, the same sequence of values is computed at each point. Under our framework, neither program's CFG is a path-projection of the other's CFG, so our semantic result about flow/path-projections cannot be applied to compare the programs' behaviors. However, we believe it is possible to extend slicing with reordering even in the presence of complex control flow.

Choi and Ferrante independently discovered the same problem of slicing programs with complex control flow [3]. They proposed two solutions to the problem, both based on the program dependence graph. The first uses the idea of an augmented control flow graph, much the same as ours. The second solution uses the PDG of the program's standard control flow graph to decide which statements to eliminate. In addition to deleting statements from the original program, their second approach inserts additional **goto**s to ensure correct control flow. Thus, the resultant program may not be a projection of the original program. As discussed in Section 5.3, if it is not necessary to form a program projection, then the PDG of the standard control flow graph can be used to form a minimal CFG flow/path-projection (with respect to the standard translation). A structuring algorithm can then be used to form a program from the CFG. Structuring algorithms attempt to minimize the number of **goto**s needed and will probably produce more readable code than the second approach of Choi and Ferrante.

The major difference between our work and the first solution proposed by Choi and Ferrante is the generality of the results. Our algorithm is defined for a language that includes (arbitrarily nested) conditional statements and loops as well as **break**s and **goto**s. In addition, in Section 5.2 we state exactly what is needed to permit our results to be extended to new control constructs. In contrast, Choi and Ferrante's first algorithm is defined for a much more limited language in which the only constructs that affect control flow are conditional and unconditional **goto**s. As Choi and Ferrante note, any structured control construct (such as an **if-then-else** or a **while** loop) can be synthesized in this simple language. However, as the following example shows, synthesizing control constructs in their simple language can lead to unnecessarily larger slices when the augmented control flow graph is used. Consider the following structured code and its translation into Choi and Ferrante's language:

if P **then**	**if** not(P) **then goto** 1;
A	A;
if Q **then**	**if** Q **then goto** 3;
halt	**goto** 2;
fi	1: B;
else	2: C;
B	3:
fi	
C	

Under the augmented control flow translation of the first program, there is no path from predicate *Q* to statement *B*, so *B* cannot be control dependent on *Q*. However, in the second program, statement *B* is control dependent on *Q* because of the edge from "**goto** 2" to *B*. Thus, a slice with respect to *B* in this program picks up predicate *Q*. One could argue that the statement "**goto** 2" should not be treated the same as other **goto**s that are explicitly written by the programmer. However, then one must define some other procedure for determining when these implicit **goto**s are needed in a slice.

Another difference is that we have characterized the structure of the control flow graphs produced using the program dependence graph (flow/path-projections) and have shown that control dependence is necessary and sufficient for forming minimal flow/path-projections.

7. CONCLUSIONS

This paper has addressed the problem of slicing programs with arbitrary control flow. Previous slicing algorithms do not always form semantically correct program projections when applied to such programs. This is due to the fact that the algorithms do not detect when a jump statement such as a **break** is required in a projection. Our work solves this problem by using a program dependence graph defined using an augmented control flow graph that represents jumps as pseudo-predicates.

REFERENCES

1. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA (1986).

2. B. Baker, “An Algorithm for Structuring Flow Graphs,” *J. ACM* **24**(1) pp. 98-120 New York, NY, (January 1977).

3. J. D. Choi and J. Ferrante, “What is in a slice,” Unpublished draft, IBM T.J. Watson Research Center (December 1992).

4. J. Ferrante, K. Ottenstein, and J. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems* **9**(5) pp. 319-349 (July 1987).

5. S. Horwitz, J. Prins, and T. Reps, “On the adequacy of program dependence graphs for representing programs,” pp. 146-157 in *Conference Record of the 15th ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York (1988).

6. S. Horwitz, J. Prins, and T. Reps, “Integrating non-interfering versions of programs,” *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

7. S. Horwitz, “Identifying the semantic and textual differences between two versions of a program,” *Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation (published as SIGPLAN Notices)* **25**(6) pp. 234-245 ACM, (June 20-22, 1990).

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

9. B. Korel, “PELAS—Program Error-Locating Assistant System,” *IEEE Transactions on Software Engineering* **SE-14**(9) pp. 1253-1260 (September 1988).

10. K.J. Ottenstein and L.M. Ottenstein, “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (Pittsburgh, PA, April 23-25, 1984), ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May, 1984).

11. T. Reps and T. Teitelbaum, *The Synthesizer Generator: A system for constructing language-based editors,* Springer-Verlag, New York, NY (1988).

12. T. Reps and W. Yang, “The semantics of program slicing and program integration,” in *Proceedings of the Colloquium on Current Issues in Programming Languages,* (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science*, Springer-Verlag, New York, NY (March 1989).

13. M. Weiser, “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July, 1984).