INTERPROCEDURAL SLICING USING DEPENDENCE GRAPHS

by

Susan Horwitz
Thomas Reps
David Binkley

Computer Sciences Technical Report #756

March 1988

Interprocedural Slicing Using Dependence Graphs

SUSAN HORWITZ, THOMAS REPS, and DAVID BINKLEY
University of Wisconsin-Madison

A slice of a program with respect to a program point p and variable x consists of all statements of the program that might affect the value of x at point p. This paper concerns the problem of interprocedural slicing – generating a slice of an entire program, where the slice crosses the boundaries of procedure calls. To solve this problem, we introduce a new kind of graph to represent programs, called a *system dependence graph*, which extends previous dependence representations to incorporate collections of procedures (with procedure calls) rather than just monolithic programs. Our main result is an algorithm for interprocedural slicing that uses the new representation.

The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. To handle this problem, system dependence graphs include some data-dependence edges that represent *transitive* dependencies due to the effects of procedure calls, in addition to the conventional direct-dependence edges. These edges are constructed with the aid of an auxiliary structure that represents calling and parameter-linkage relationships. This structure takes the form of an attribute grammar. The step of computing the required transitive-dependence edges is reduced to the construction of the subordinate characteristic graphs for the grammar's nonterminals.

It should be noted that our work concerns a somewhat restricted kind of slice: Rather than permitting a program to be sliced with respect to program point p and an *arbitrary* variable, a slice must be taken with respect to a variable that is *defined* at or *used* at p.

CR Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs – *control structures, procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors – *compilers, optimization*

General Terms: Algorithms, Design

Additional Key Words and Phrases: program dependence graph, control dependence, data dependence, program slicing, data-flow analysis, flow-insensitive summary information, attribute grammar, subordinate characteristic graph, program debugging, program integration

---

1. INTRODUCTION

The *slice* of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p. The value of x at program point p is *directly affected* by assignments to x that reach p and by the loops and conditionals that enclose p. An intraprocedural slice is determined from the closure of the directly-affects relation.

This work was supported in part by the National Science Foundation under grants DCR-8552602 and DCR-8603356 as well as by grants from IBM, DEC, and Xerox.

Authors' address: Computer Sciences Department, University of Wisconsin – Madison, 1210 W. Dayton St., Madison, WI 53706.

An abridged version of this paper will appear under the same title in: *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices*, June 1988.

Program slicing, originally defined in [Weiser84], can be used to isolate individual computation threads within a program, which can help a programmer understand complicated code. Program slicing is also used by the algorithm for automatically integrating program variants described in [Horwitz88]; slices are used to compute a safe approximation to the change in behavior between a program P and a modified version of P, and to help determine whether two different modifications to P interfere.

In Weiser's terminology, a *slicing criterion* is a pair $<p, V>$, where p is a program point and V is a subset of the program's variables. In his work, a slice consists of all statements and predicates of the program that might affect the value of variables in V at point p. This is a more general kind of slice than is often needed: Rather than a slice taken with respect to program point p and an *arbitrary* variable, one is often interested in a slice taken with respect to a variable that is *defined* at or *used* at p. Ottenstein and Ottenstein point out how well-suited *program dependence graphs* are for this kind of slicing and propose that program dependence graphs be used to represent procedures in software development environments [Ottenstein84].

This paper concerns the problem of interprocedural slicing – generating a slice of an entire program, where the slice crosses the boundaries of procedure calls. Our algorithm for interprocedural slicing produces a more precise interprocedural slice than the one given in [Weiser84]. Our work follows the example of [Ottenstein84] by defining the slicing algorithm in terms of operations on a dependence graph representation of programs; however, [Ottenstein84] only discusses programs consisting of a single monolithic procedure, and does not address the problem of slicing across procedure boundaries.

To solve the interprocedural-slicing problem, we introduce a new kind of graph to represent programs, called a *system dependence graph*, which extends previous dependence representations to incorporate collections of procedures (with procedure calls) rather than just monolithic programs. Our main result is an algorithm for interprocedural slicing that uses the new representation.

The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. To illustrate this problem, and the shortcomings of Weiser's algorithm, consider the following example program, which sums the integers from 1 to 10:

```
program Main
  sum := 0;
  i := 1;
  while i < 11 do
      call A(sum, i)
  od
end

procedure A(x, y)
  call Add(x, y);
  call Increment(y)
return

procedure Add(a, b)
  a := a + b
return

procedure Increment(z)
  call Add(z, 1)
return
```

Using Weiser's algorithm to slice this program with respect to variable z and the **return** statement of procedure *Increment*, we obtain everything from the original program. However, inspection reveals that computations involving the variable *sum* do not contribute to the value of z at the end of procedure *Increment*; in particular, neither the initialization of *sum* nor the call to *Add* from procedure A (which adds the current value of i to *sum*) should be included in the slice. The reason these statements are included in the slice computed by Weiser's algorithm is (roughly) the following: The statement "**call** *Add*$(z, 1)$" in procedure *Increment* causes the slice to "descend" into procedure *Add*; when the slice reaches the beginning of *Add* it "ascends" to *all* sites that call *Add*, both the site in *Increment* at which it "descended" and the (irrelevant) site in A.

A more precise slice consists of the following elements:

```
program Main
  i := 1;
  while i < 11 do
      call A(i)
  od
end

procedure A(y)
    call Increment(y)
return

procedure Add(a, b)
    a := a + b
return

procedure Increment(z)
    call Add(z, 1)
return
```

This set of program elements is computed by the slicing algorithm described in this paper.

To sidestep the calling-context problem, system dependence graphs include some data-dependence edges that represent *transitive* dependencies due to the effects of procedure calls, in addition to the conventional edges for direct dependencies. The cornerstone of our construction is the use of an attribute grammar to represent calling and parameter-linkage relationships among procedures. The step of computing the required transitive-dependence edges is reduced to the construction of the subordinate characteristic graphs for the grammar's nonterminals.

It is important to understand the distinction between two different but related "slicing problems:"

Version (1)

> The slice of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p.

Version (2)

> The slice of a program with respect to program point p and variable x consists of a reduced program that computes the same sequence of values for x at p. That is, at point p the behavior of the reduced program with respect to variable x is indistinguishable from that of the original program.

For in*tra*procedural slicing, a solution to Version (1) provides a solution to Version (2), since the "reduced program" required in Version (2) can be obtained by restricting the original program to just the statements and predicates found in the solution for Version (1) [Reps88].

For in*ter*procedural slicing, restricting the original program to just the statements and predicates found for Version (1) does not necessarily yield a program that is a satisfactory solution to Version (2). The reason has to do with multiple calls to the same procedure: It is possible that the program elements found by an algorithm for Version (1) will include more than one such call, each passing a different subset of the procedure's parameters. (It should be noted that, although it is imprecise, Weiser's algorithm produces a solution to Version (2).)

In this paper, we address Version (1) of the slicing problem. The interprocedural slicing algorithm presented in this paper identifies a subgraph of the system dependence graph whose components might affect the values of the variables defined at or used at a given program point p. A solution to Version (2) requires defining a mapping from this subgraph to a program whose behavior at p is indistinguishable from the original program. This mapping may involve duplicating code in order to specialize procedure bodies to particular parameter-usage patterns.

The remainder of the paper is organized as follows: Section 2 defines the program dependence graphs used to represent programs in a language without procedure calls; this definition is a slight refinement of the one given in [Horwitz88]. Section 2 also defines the operation of intraprocedural slicing on these program dependence graphs. Section 3 extends the definition of dependence graphs to handle a language that includes procedures and procedure calls. The new graphs are called *system dependence graphs*. Section 4 presents our slicing algorithm, which operates on system dependence graphs and correctly accounts for the calling context of a called procedure. It then describes how to improve the precision of interprocedural slices by using interprocedural summary information in the construction of system dependence graphs, how

to handle programs with aliasing, and how to slice incomplete programs. Section 5 discusses the complexity of the slicing algorithm. Section 6 discusses related work.

2. PROGRAM DEPENDENCE GRAPHS AND PROGRAM SLICES

Different definitions of program dependence representations have been given, depending on the intended application; nevertheless, they are all variations on a theme introduced in [Kuck72], and share the common feature of having explicit representations of both control dependencies and data dependencies. The definition of *program dependence graph* presented here is a slight refinement of the one given in [Horwitz88]; it is similar, but not identical, to the program dependence representations used by others, such as the "program dependence graphs" defined in [Ferrante87] and the "dependence graphs" defined in [Kuck81].

The definition of program dependence graph presented below is for a language with scalar variables, assignment statements, conditional statements, while loops, and a restricted kind of "output statement" called an *end statement*. An end statement, which can only appear at the end of a program, names one or more of the variables used in the program; when execution terminates, only those variables will have values in the final state. Intuitively, the variables named by the end statement are those whose final values are of interest to the programmer.

2.1. The Program Dependence Graph

The program dependence graph for program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges.[1] The vertices of G_P represent the assignment statements and control predicates that occur in program P. In addition, G_P includes three other categories of vertices:

(1) There is a distinguished vertex called the *entry vertex*.

(2) For each variable x for which there is a path in the standard control-flow graph for P [Aho86] on which x is used before being defined, there is a vertex called the *initial definition of* x. This vertex represents an assignment to x from the initial state. The vertex is labeled "$x := InitialState(x)$."

(3) For each variable x named in P's end statement, there is a vertex called the *final use of* x. It represents an access to the final value of x computed by P, and is labeled "FinalUse(x)".

The edges of G_P represent *dependencies* among program components. An edge represents either a *control dependency* or a *data dependency*. Control dependency edges are labeled either **true** or **false**, and the source of a control dependency edge is always the entry vertex or a predicate vertex. A control dependency edge from vertex v_1 to vertex v_2, denoted by $v_1 \rightarrow_c v_2$, means that during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will be executed (although perhaps not immediately). A method for determining control dependency edges for arbitrary programs is given in [Ferrante87]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependency edges of G_P can be determined in a much simpler fashion. For the language under consideration here, a program dependence graph contains a *control dependency edge* from vertex v_1 to vertex v_2 of G_P iff one of the following holds:

[1] 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; we say that b is the *source* and c the *target* of the edge.

i) v_1 is the entry vertex, and v_2 represents a component of P that is not subordinate to any control predicate; these edges are labeled **true**.

ii) v_1 represents a control predicate, and v_2 represents a component of P immediately subordinate to the control construct whose predicate is represented by v_1. If v_1 is the predicate of a while-loop, the edge $v_1 \longrightarrow_c v_2$ is labeled **true**; if v_1 is the predicate of a conditional statement, the edge $v_1 \longrightarrow_c v_2$ is labeled **true** or **false** according to whether v_2 occurs in the **then** branch or the **else** branch, respectively.[2]

A data dependency edge from vertex v_1 to vertex v_2 means that the program's computation might be changed if the relative order of the components represented by v_1 and v_2 were reversed. In this paper, program dependence graphs contain two kinds of data-dependency edges, representing *flow dependencies* and *def-order dependencies*.

A program dependence graph contains a flow dependency edge from vertex v_1 to vertex v_2 iff all of the following hold:

i) v_1 is a vertex that defines variable x.

ii) v_2 is a vertex that uses x.

iii) Control can reach v_2 after v_1 via an execution path along which there is no intervening definition of x. That is, there is a path in the standard control-flow graph for the program by which the definition of x at v_1 reaches the use of x at v_2. (Initial definitions of variables are considered to occur at the beginning of the control-flow graph.)

A flow dependency that exists from vertex v_1 to vertex v_2 will be denoted by $v_1 \longrightarrow_f v_2$.

Flow dependencies can be further classified as *loop carried* or *loop independent*. A flow dependency $v_1 \longrightarrow_f v_2$ is carried by loop L, denoted by $v_1 \longrightarrow_{lc(L)} v_2$, if in addition to i), ii), and iii) above, the following also hold:

iv) There is an execution path that both satisfies the conditions of iii) above and includes a backedge to the predicate of loop L.

v) Both v_1 and v_2 are enclosed in loop L.

A flow dependency $v_1 \longrightarrow_f v_2$ is loop independent, denoted by $v_1 \longrightarrow_{li} v_2$, if in addition to i), ii), and iii) above, there is an execution path that satisfies iii) above and includes *no* backedge to the predicate of a loop that encloses both v_1 and v_2. It is possible to have both $v_1 \longrightarrow_{lc(L)} v_2$ and $v_1 \longrightarrow_{li} v_2$.

A program dependence graph contains a def-order dependency edge from vertex v_1 to vertex v_2 iff all of the following hold:

i) v_1 and v_2 both define the same variable.

ii) v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.

iii) There exists a program component v_3 such that $v_1 \longrightarrow_f v_3$ and $v_2 \longrightarrow_f v_3$.

iv) v_1 occurs to the left of v_2 in the program's abstract syntax tree.

---

[2]In other definitions that have been given for control dependency edges, there is an additional edge from each predicate of a **while** statement to itself labeled **true**. By including the additional edge, the predicate's outgoing **true** edges consist of every program element that is guaranteed to be executed (eventually) when the predicate evaluates to **true**. This kind of edge is left out of our definition because it is not necessary for our purposes.

A def-order dependency from v_1 to v_2 is denoted by $v_1 \longrightarrow_{do(v_3)} v_2$.

Note that a program dependence graph is a multi-graph (*i.e.* it may have more than one edge of a given kind between two vertices). When there is more than one loop-carried flow dependency edge between two vertices, each is labeled by a different loop that carries the dependency. When there is more than one def-order edge between two vertices, each is labeled by a vertex that is flow-dependent on both the definition that occurs at the edge's source and the definition that occurs at the edge's target.

Example. Figure 1 shows an example program and its program dependence graph. The boldface arrows represent control dependency edges; dashed arrows represent def-order dependency edges; solid arrows represent loop-independent flow dependency edges; solid arrows with a hash mark represent loop-carried flow dependency edges.

The data-dependency edges of a program dependence graph are computed using data-flow analysis. For the restricted language considered in this section, the necessary computations can be defined in a syntax-

---

```
program Main
    sum := 0;
    i := 1;
    while i < 11 do
        sum := sum+i;
        i := i+1
    od
end(sum, i)
```

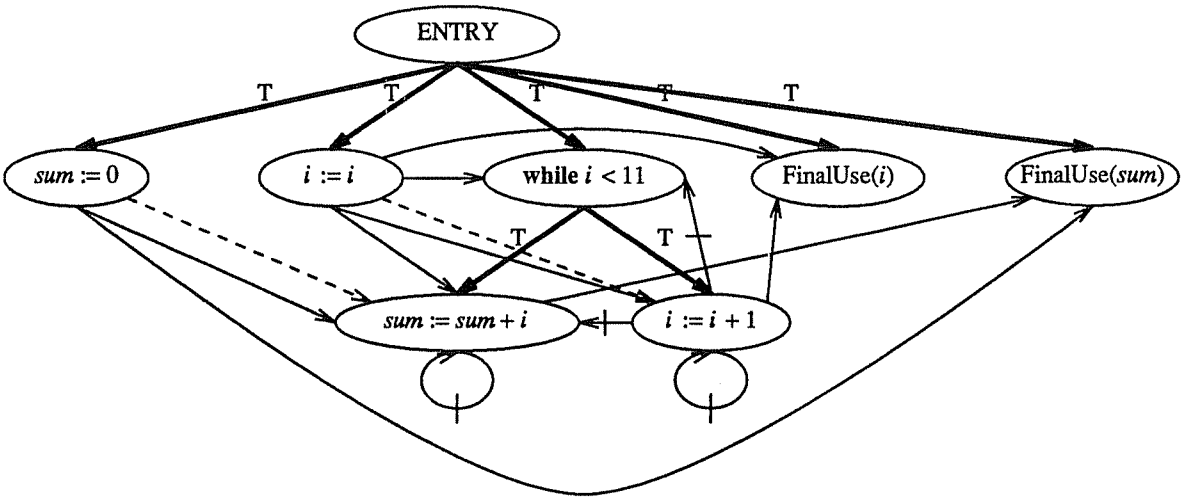


Figure 1. An example program, which sums the integers from 1 to 10 and leaves the result in the variable *sum*, and its program dependence graph. The boldface arrows represent control dependency edges, dashed arrows represent def-order dependency edges, solid arrows represent loop-independent flow dependency edges, and solid arrows with a hash mark represent loop-carried flow dependency edges.

directed manner.

2.2. Program Slices

For vertex s of program dependence graph G, the *slice* of G with respect to s, denoted by G/s, is a graph containing all vertices on which s has a transitive flow or control dependence (*i.e.* all vertices that can reach s via flow and/or control edges): $V(G/s) = \{ w \mid w \in V(G) \wedge w \longrightarrow^{*}_{c,f} s \}$. We extend the definition to a set of vertices $S = \bigcup_i s_i$ as follows: $V(G/S) = V(G/(\bigcup_i s_i)) = \bigcup_i V(G/s_i)$. Figure 2 gives a simple worklist algorithm for computing the vertices of a slice using a program dependence graph.

The edges in the graph G/S are essentially those in the subgraph of G induced by $V(G/S)$, with the exception that a def-order edge $v \longrightarrow_{do(u)} w$ is included only if G/S contains the vertex u that is directly flow dependent on the definitions at v and w. In terms of the three types of edges in a program dependence graph we define

$$\begin{aligned} E(G/S) = \quad & \{ (v \longrightarrow_f w) \mid (v \longrightarrow_f w) \in E(G) \wedge v,w \in V(G/S) \} \\ & \cup \{ (v \longrightarrow_c w) \mid (v \longrightarrow_c w) \in E(G) \wedge v,w \in V(G/S) \} \\ & \cup \{ (v \longrightarrow_{do(u)} w) \mid (v \longrightarrow_{do(u)} w) \in E(G) \wedge u,v,w \in V(G/S) \} \end{aligned}$$

Example. Figure 3 shows the graph resulting from taking a slice of the program dependence graph from Figure 1 with respect to the final-use vertex for i.

We say that G is a *feasible* program dependence graph iff G is the program dependence graph of some program P. For any $S \subseteq V(G)$, if G is a feasible program dependence graph, then the slice G/S is also a feasible program dependence graph; it corresponds to the program P' obtained by restricting the syntax tree of P to just the statements and predicates in $V(G/S)$ [Reps88].

```
procedure MarkVerticesOfSlice(G, S)
declare
  G: a program dependence graph
  S: a set of vertices in G
  WorkList: a set of vertices in G
  v, w: vertices in G
begin
  WorkList := S
  while WorkList ≠ ∅ do
    Select and remove vertex v from WorkList
    Mark v
    for each vertex w such that edge w →f v or edge w →c v is in G do
      if w is unmarked then Insert w into WorkList fi
    od
  od
end
```

Figure 2. A worklist algorithm that marks the vertices in G/S. Vertex v is in G/S if there is a path along flow and/or control edges from v to some vertex in S.

```
program Main
  i := 1;
  while i < 11 do
    i := i + 1
  od
end(i)
```

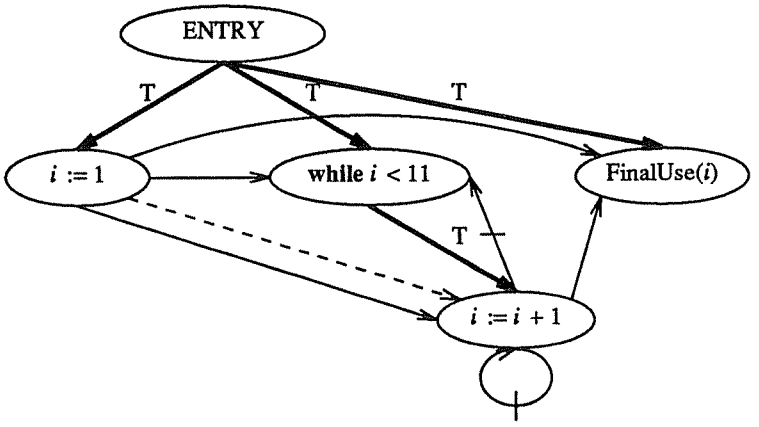


Figure 3. The graph and the corresponding program that result from slicing the program dependence graph from Figure 1 with respect to the final-use vertex for i.

The significance of an intraprocedural slice is that it captures a portion of a program's behavior. The programs P' and P, corresponding to the slice G/S and the graph G, respectively, compute the same final values for all variables x for which $FinalUse(x)$ is a vertex in S [Reps88].

3. THE SYSTEM DEPENDENCE GRAPH: AN INTERPROCEDURAL DEPENDENCE GRAPH REPRESENTATION

We now turn to the definition of the *system dependence graph.* The system dependence graph, an extension of the dependence graphs defined in Section 2.1, represents programs in a language that includes procedures and procedure calls.

Our definition of the system dependence graph models a language with the following properties:

(1) A complete system consists of a single (main) program and a collection of auxiliary procedures.

(2) Procedures end with **return** statements instead of **end** statements (as defined in Section 2). A **return** statement does not include a list of variables.

(3) Parameters are passed by value-result.

It should become clear that our approach is not tied to the particular language features enumerated above. Modeling different features will require some adaptation; however, the basic approach is applicable to languages that allow nested scopes and languages that use different parameter-passing mechanisms. The definition of system dependence graphs presented in this section relies on the absence of aliasing; Section

4.3 discusses how to convert a program with aliasing into one that is alias free. In the absence of aliasing, global variables can be treated simply as additional parameters to each procedure; thus, we do not discuss globals explicitly in this section.

A system dependence graph includes a *program dependence graph*, which represents the system's main program, *procedure dependence graphs*, which represent the system's auxiliary procedures, and some additional edges. These additional edges are of two sorts: (1) edges that represent direct dependencies between a call site and the called procedure, and (2) edges that represent transitive dependencies due to calls.

Section 3.1 discusses how procedure calls and procedure entry are represented in procedure dependence graphs and how edges representing dependencies between a call site and the called procedure are added to connect these graphs together. Section 3.2 defines the *linkage grammar*, an attribute grammar used to represent the call structure of a system. Transitive dependencies due to procedure calls are computed using the linkage grammar and are added as the final step of building a system dependence graph.

In the sections below, we use "procedure" as a generic term referring to both the main program and the auxiliary procedures when the distinction between the two is irrelevant.

3.1. Procedure Calls and Procedure Linkages

Extending the definition of dependence graphs to handle procedure calls requires representing *procedure linkages*: the passing of values between procedures. In designing the representation of procedure linkages we have three goals:

(1) It should be possible to build an individual procedure's procedure dependence graph (including the computation of data dependencies) with minimal knowledge of other system components.

(2) The system dependence graph should consist of a straightforward connection of the program dependence graph and procedure dependence graphs.

(3) It should be possible to extract efficiently a precise interprocedural slice by traversing the graph via a procedure analogous to the procedure MarkVerticesOfSlice given in Figure 2.

Goal (3) is the subject of Section 4.1, which presents our algorithm for slicing a system dependence graph.

To meet the goals outlined above our graphs model the following non-standard, two-stage mechanism for run-time procedure linkage: When procedure P calls procedure Q, values are transferred from P to Q by means of an intermediate *call linkage dictionary*, δ_Q. Values are transferred back from Q to P through a *return linkage dictionary*, δ'_Q. Before the call, P copies values into the call dictionary; Q then initializes local variables from this dictionary. Before returning, Q copies return values into the return dictionary, from which P retrieves them.

This model of procedure linkage is represented in procedure dependence graphs through the use of five new kinds of vertices. A call site is represented using a *call-site* vertex; information transfer is represented using four kinds of *linkage* vertices. On the calling side, information transfer is represented by a set of *pre-* and *post-processing* vertices. These vertices, which are control dependent on the call-site vertex, represent assignment statements that copy values into the call dictionary and out of the return dictionary, respectively. Similarly, information transfer in the called procedure is represented by a set of *initialization* and *finalization* vertices. These vertices, which are control dependent on the procedure's entry vertex, represent assignment statements that copy values out of the call dictionary and into the return dictionary, respectively.

Using this model, data dependencies between procedures are limited to dependencies from preprocessing vertices to initialization vertices and from finalization vertices to postprocessing vertices. Connecting procedure dependence graphs to form a system dependence graph is straightforward, involving the addition of three new kinds of edges: (1) a *call* edge is added from each call-site vertex to the corresponding procedure-entry vertex; (2) a *linkage-entry* edge is added from each preprocessing vertex at a call site to the corresponding initialization vertex in the called procedure; (3) a *linkage-exit* edge is added from each finalization vertex in the called procedure to the corresponding postprocessing vertex at the call site. (Call edges are a new kind of control dependency edge; linkage-entry and linkage-exit edges are new kinds of data dependency edges.)

Another advantage of this model is that flow dependencies within a procedure can be computed in the usual way, using data flow analysis on the procedure's control-flow graph in which each procedure call is replaced with the appropriate sequence of assignments to the call dictionary followed by the appropriate sequence of assignments from the return dictionary.

An important question is *which* values are transferred from a call site to the called procedure and back again. This point is discussed further in Section 4.2, which presents a strategy in which the results of interprocedural data flow analysis are used to omit some linkage vertices from procedure dependence graphs. For now, we will assume that all actual parameters are copied into the call dictionary and retrieved from the return dictionary. Thus, the linkage vertices associated with a call from procedure P to procedure Q are defined as follows (G_P denotes the procedure dependence graph for P):

> In G_P, subordinate to the call-site vertex that represents the call to Q, there is a pre-processing vertex for each actual parameter e of the call to Q. The pre-processing vertices are labeled $\delta_Q(r) := e$, where r is the formal parameter name.
>
> For each actual parameter a that is a variable rather than an expression, there is a postprocessing vertex. These are labeled $a := \delta'_Q(r)$ for actual parameter a and corresponding formal parameter r.

The linkage vertices associated with the entry to procedure Q and with the return from procedure Q are defined as follows (G_Q denotes the procedure dependence graph for Q):

> For each parameter of Q, G_Q contains an initialization vertex and a finalization vertex. These vertices are labeled $r := \delta_Q(r)$, and $\delta'_Q(r) := r$ respectively, where r is the formal parameter name.

Example. Figure 4 repeats the example system from the Introduction and shows the corresponding program and procedure dependence graphs connected with linkage-entry edges, linkage-exit edges, and call edges. (In this figure, as well as in the remaining figures of the paper, def-order edges are not shown.) Edges representing control dependencies are shown in boldface and are not labeled (all such edges would be labeled **true**); edges representing intraprocedural flow dependencies are shown using arcs; linkage-entry edges, linkage-exit edges, and call edges are shown using dashed lines.

3.2. The Linkage Grammar: An Attribute Grammar that Models Procedure-Call Structure

The linkage grammar is an attribute grammar that models the call structure of each procedure as well as the in*tra*procedural transitive flow dependencies among the procedures' linkage vertices. In*ter*procedural transitive flow dependencies among a system dependence graph's linkage vertices are determined from the linkage grammar using a standard attribute-grammar construction: the computation of the *subordinate*

```
program Main
  sum := 0;
  i := 1;
  while i < 11 do
      call A (sum, i)
  od
end(sum, i)

procedure A (x)
  call Add (x, y);
  call Increment (y)
return

procedure Add (a, b)
  a := a + b
return

procedure Increment (z)
  call Add (z, 1)
return
```

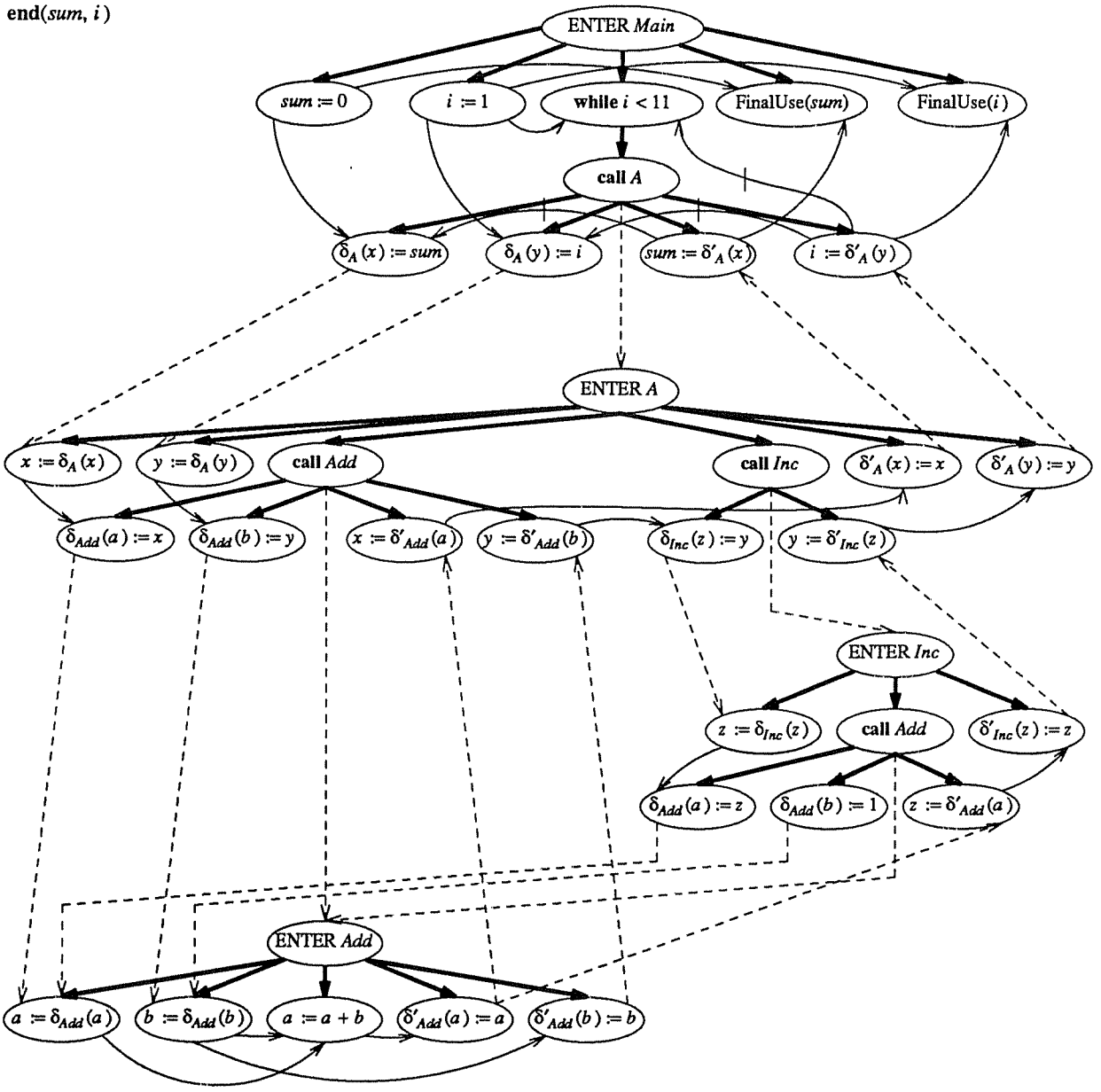


Figure 4. Example system and corresponding program and procedure dependence graphs connected with linkage-entry, linkage-exit, and call edges. Edges representing control dependencies are shown (unlabeled) in boldface; edges representing intraprocedural flow dependencies are shown using arcs; linkage-entry edges, linkage-exit edges, and call edges are shown using dashed lines.

characteristic graphs of the linkage grammar's nonterminals.[3] In this section, we describe the construction of the linkage grammar and the computation of its subordinate characteristic graphs.

The context-free part of the linkage grammar models the system's procedure-call structure. The grammar includes one nonterminal and one production for each procedure in the system. If procedure P contains no calls, the right-hand side of the production for P is ε; otherwise, there is one right-hand-side nonterminal for each call site in P.

Example. For the example system (shown in Figure 4) the productions of the linkage grammar are as follows:

$Main \rightarrow A$ $\quad A \rightarrow Add\ Increment$ $\quad Add \rightarrow \varepsilon$ $\quad Increment \rightarrow Add$

The attributes in the linkage grammar correspond to the parameters of the procedures. Procedure inputs are modeled as inherited attributes; procedure outputs are modeled as synthesized attributes (see Appendix A). For example, the productions shown above are repeated in Figure 5, this time in tree form. In Figure 5, each nonterminal is annotated with its attributes; a nonterminal's inherited attributes are placed to its left; its synthesized attributes are placed to its right.

More formally, the program's linkage grammar has the following elements:

- For each procedure P, the linkage grammar contains a nonterminal P.
- For each procedure P, there is a production $p : P \rightarrow \beta$, where for each site of a call on procedure Q in P there is a distinct occurrence of Q in β.
- For each initialization vertex of P, there is an inherited attribute of nonterminal P.

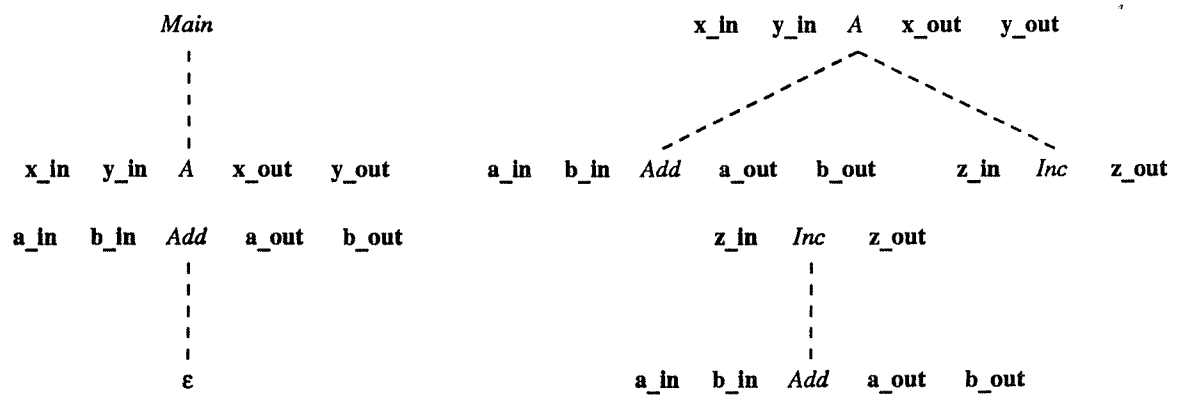


Figure 5. The productions of the example linkage grammar shown in tree form. Each nonterminal is annotated with its attributes; a nonterminal's inherited attributes are placed to its left; its synthesized attributes are placed to its right.

[3] A summary of attribute-grammar terminology can be found in Appendix A.

- For each finalization vertex of P, there is a synthesized attribute of nonterminal P.

Dependencies among the attributes of a linkage-grammar production are used to model the (possibly transitive) intraprocedural dependencies among the linkage vertices of the corresponding procedure. These dependencies are computed using (intraprocedural) slices of the procedure's procedure dependence graph. For each grammar production, attribute equations are introduced to represent the intraprocedural dependencies among the linkage vertices of the corresponding procedure dependence graph. For each attribute occurrence a, the procedure dependence graph is sliced with respect to the vertex that corresponds to a. An attribute equation is introduced for a so that a depends on the attribute occurrences that correspond to the linkage vertices identified by the slice. More formally:

> For each attribute occurrence $X.a$ of a production p, let v be the vertex of the procedure dependence graph G_P that corresponds to $X.a$. Associate with p an attribute equation of the form $X.a = f(..., Y.b, ...)$ where the arguments $Y.b$ to the equation consist of the attribute occurrences of p that correspond to the linkage vertices in G_P / v.

(The actual function on the right-hand side of the equation is unimportant because the attribute grammar is never used for evaluation; all we are concerned about is that the equation represent the dependence described above.) One property of the above definition is that the attribute dependency graph for each production is transitively closed.

It is entirely possible that a linkage grammar will be a circular attribute grammar (*i.e.* there may be attributes in some derivation tree of the grammar that depend on themselves). This does not create any difficulties as the linkage grammar is used only to compute transitive dependencies and not for attribute evaluation.

Example. Figure 6 shows the productions of the grammar from Figure 5, augmented with attribute dependencies. Note that there is an immediate cycle in the dependencies for the production $Main \rightarrow A$.

Transitive dependencies from a call site's preprocessing vertices to its postprocessing vertices are computed from the linkage grammar by constructing the subordinate characteristic graphs for the grammar's

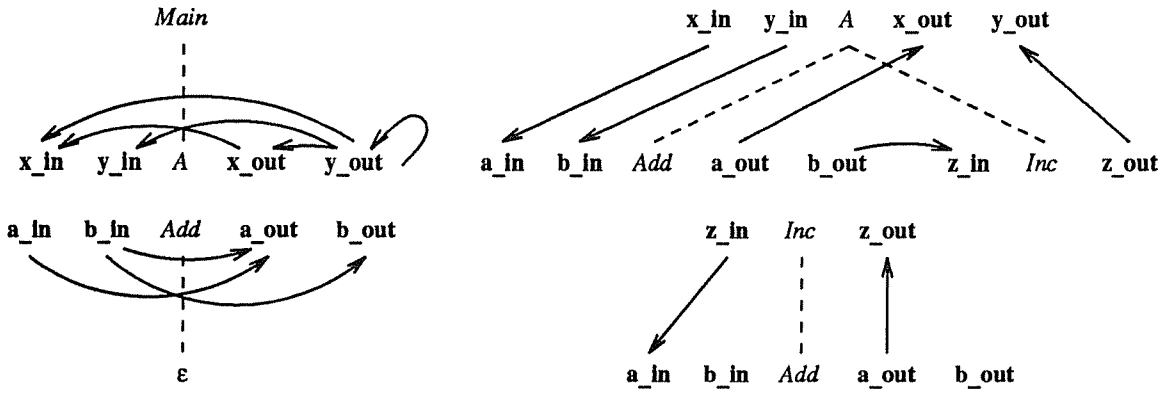


Figure 6. The productions of Figure 5, augmented with attribute dependencies.

nonterminals. The algorithm we give exploits the special structure of linkage grammars to compute these graphs more efficiently than can be done for attribute grammars in general. For general attribute grammars, computing the sets of possible subordinate characteristic graphs for the grammar's nonterminals may require time exponential in the number of attributes attached to some nonterminal. However, a linkage grammar is an attribute grammar of a restricted nature: For each nonterminal X in the linkage grammar, there is only one production with X on the left-hand side. Because linkage grammars are restricted in this fashion, for each nonterminal of a linkage grammar there is one subordinate characteristic graph that covers all of the nonterminal's other possible subordinate characteristic graphs. For such grammars, it is possible to give a polynomial-time algorithm for constructing the (covering) subordinate characteristic graphs.

The computation is performed by an algorithm that is a slight modification of an algorithm originally developed by Kastens to construct approximations to a grammar's transitive dependency relations [Kastens80]. The covering subordinate characteristic graph of a nonterminal X of the linkage grammar is captured in the graph $TDS(X)$ (standing for "Transitive Dependencies among a Symbol's attributes"). Initially, all the *TDS* graphs are empty. The construction that builds them up involves the auxiliary graph $TDP(p)$ (standing for "Transitive Dependencies in a Production"), which expresses dependencies among the attributes of a production's nonterminal occurrences.

The basic operation used in ConstructSubCGraphs is the procedure "AddEdgeAndInduce($TDP(p),(a,b)$)", whose first argument is the *TDP* graph of some production p and whose second argument is a pair of attribute occurrences in p. AddEdgeAndInduce carries out three actions:

(1) The edge (a,b) is inserted into the graph $TDP(p)$.

(2) Any additional edges needed to transitively close $TDP(p)$ are inserted into $TDP(p)$.

(3) In addition, for each edge added to $TDP(p)$ by (1) or (2), (*i.e.*, either the edge (a,b) itself or some other edge (c,d) added to reclose $TDP(p)$), AddEdgeAndInduce may add an edge to one of the *TDS* graphs. In particular, for each edge added to $TDP(p)$ of the form $(X_0.m, X_0.n)$, where X_0 is the left-hand-side occurrence of nonterminal X in production p and $(X.m, X.n) \notin TDS(X)$, an edge $(X.m, X.n)$ is added to $TDS(X)$.

An edge in one of the $TDS(X)$ graphs can be *marked* or *unmarked*; the edges that AddEdgeAndInduce adds to the $TDS(X)$ graphs are unmarked.

The *TDS* graphs are generated by the procedure ConstructSubCGraphs, given below in Figure 7, which is a slight modification of the first two steps of Kastens's algorithm for constructing a set of evaluation plans for an attribute grammar [Kastens80]. ConstructSubCGraphs performs a kind of closure operation on the *TDP* and *TDS* graphs. Step 1 of the algorithm – the first two for-loops of ConstructSubCGraphs – initializes the grammar's *TDP* and *TDS* graphs; when these loops terminate, the *TDP* graphs contain edges representing all direct dependencies that exist between the grammar's attribute occurrences. At the end of Step 1, $TDP(p)$ is a (transitively closed) graph whose edges represent the direct dependencies of production p. The *TDS* graphs contain unmarked edges corresponding to direct left-hand-side-to-left-hand-side dependencies in the linkage grammar's productions.

In Step 2 of ConstructSubCGraphs, the invariant for the **while**-loop is:

> If a graph $TDP(p)$ contains an edge e' that corresponds to a marked edge e in one of the *TDS* graphs, then e has been induced in all of the other graphs $TDP(q)$.

When all edges in all *TDS* graphs have received marks, the effects of all direct dependencies have been induced in the *TDP* and *TDS* graphs. Thus, the $TDS(X)$ graphs computed by ConstructSubCGraphs are guaranteed to cover the actual transitive dependencies among the attributes of X that exist at any

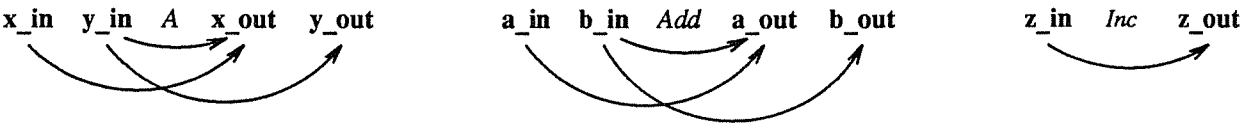
```
procedure ConstructSubCGraphs(L)
declare
    L: a linkage grammar
    p: a production in L
    X_i, X_j, X̂: nonterminal occurrences in L
    a, b: attributes of nonterminals in L
    X: a nonterminal in L
begin
  /* Step 1: Initialize the TDS and TDP graphs */
    for each nonterminal X in L do
      TDS(X) := the graph containing a vertex for each attribute X.b but no edges
    od
    for each production p in L do
      TDP(p) := the graph containing a vertex for each attribute occurrence X_j.b of p but no edges
      for each attribute occurrence X_j.b of p do
        for each argument X_i.a of X_j.b do
          Insert edge (X_i.a, X_j.b) into TDP(p)
          let X be the nonterminal corresponding to nonterminal occurrence X_j in
            if i = 0 and j = 0 and (X.a, X.b) ∉ TDS(X) then Insert an unmarked edge (X.a, X.b) into TDS(X) fi
          ni
        od
      od
    od
  /* Step 2: Determine the sets of induced transitive dependencies */
    while there is an unmarked edge (X.a, X.b) in one of the TDS graphs do
      Mark (X.a, X.b)
      for each occurrence X̂ of X in any production p do
        if (X̂.a, X̂.b) ∉ TDP(p) then AddEdgeAndInduce(TDP(p), (X̂.a, X̂.b)) fi
      od
    od
end
```

Figure 7. Computation of a linkage grammar's sets of *TDP* and *TDS* graphs.

occurrence of X in any derivation tree.

Put more simply, because for each nonterminal X in a linkage grammar there is only a single production that has X on the left-hand side, the grammar only derives one tree. (For a recursive grammar it will be an infinite tree.) All marked edges in *TDS* represent transitive dependencies in this tree, and thus the *TDS*(X) graph computed by ConstructSubCGraphs represents a subordinate characteristic graph of X that covers the subordinate characteristic graph of any partial derivation tree derived from X, as desired.

Example. The nonterminals of our example grammar are shown below annotated with their attributes and their subordinate characteristic graphs.



3.3. Recap of the Construction of the System Dependence Graph

The system dependence graph is constructed by the following steps:

(1) For each procedure of the system, construct its procedure dependence graph.

(2) For each call site, introduce a call edge from the call-site vertex to the corresponding procedure-entry vertex.

(3) For each preprocessing vertex v at a call site, introduce a linkage-entry edge from v to the corresponding initialization vertex in the called procedure.

(4) For each postprocessing vertex v at at call site, introduce a linkage-exit edge to v from the corresponding finalization vertex in the called procedure.

(5) Construct the linkage grammar corresponding to the system.

(6) Compute the subordinate characteristic graphs of the linkage grammar's nonterminals.

(7) At all call sites that call procedure P, introduce flow dependency edges corresponding to the edges in the subordinate characteristic graph for P.

Example. Figure 8 shows the complete system dependence graph for our example system. Control dependencies, which are shown unlabeled, are represented using medium-bold arrows; intraprocedural flow dependencies are represented using arcs; transitive interprocedural flow dependencies (corresponding to subordinate characteristic graph edges) are represented using heavy bold arcs; call edges, linkage-entry edges, and linkage-exit edges (the edges that connect program and procedure dependence graphs together) are represented using dashed arrows.

4. INTERPROCEDURAL SLICING

In this section we describe how to perform an interprocedural slice using the system dependence graph defined in Section 3. We then discuss modifications to the definition of the system dependence graph to permit more precise slicing and to extend the slicing algorithm's range of applicability.

4.1. An Algorithm for Interprocedural Slicing

As discussed in the Introduction, the algorithm presented in [Weiser84], while safe, is not as precise as possible. The difficult aspect of interprocedural slicing is keeping track of the calling context when a slice "descends" into a called procedure.

The key element of our approach is the use of the linkage grammar's characteristic graph edges in the system dependence graph. These edges represent transitive data dependencies from preprocessing vertices to postprocessing vertices due to procedure calls. The presence of such edges permits us to sidestep the "calling context" problem; the slicing operation can move "across" a call without having to descend into it.

Our algorithm for interprocedural slicing is given in Figure 9. In Figure 9, the computation of the slice of system dependence graph G with respect to vertex set S is performed in two phases. Both Phases 1 and 2 operate on the system dependence graph using essentially the method presented in Section 2.2 for per-

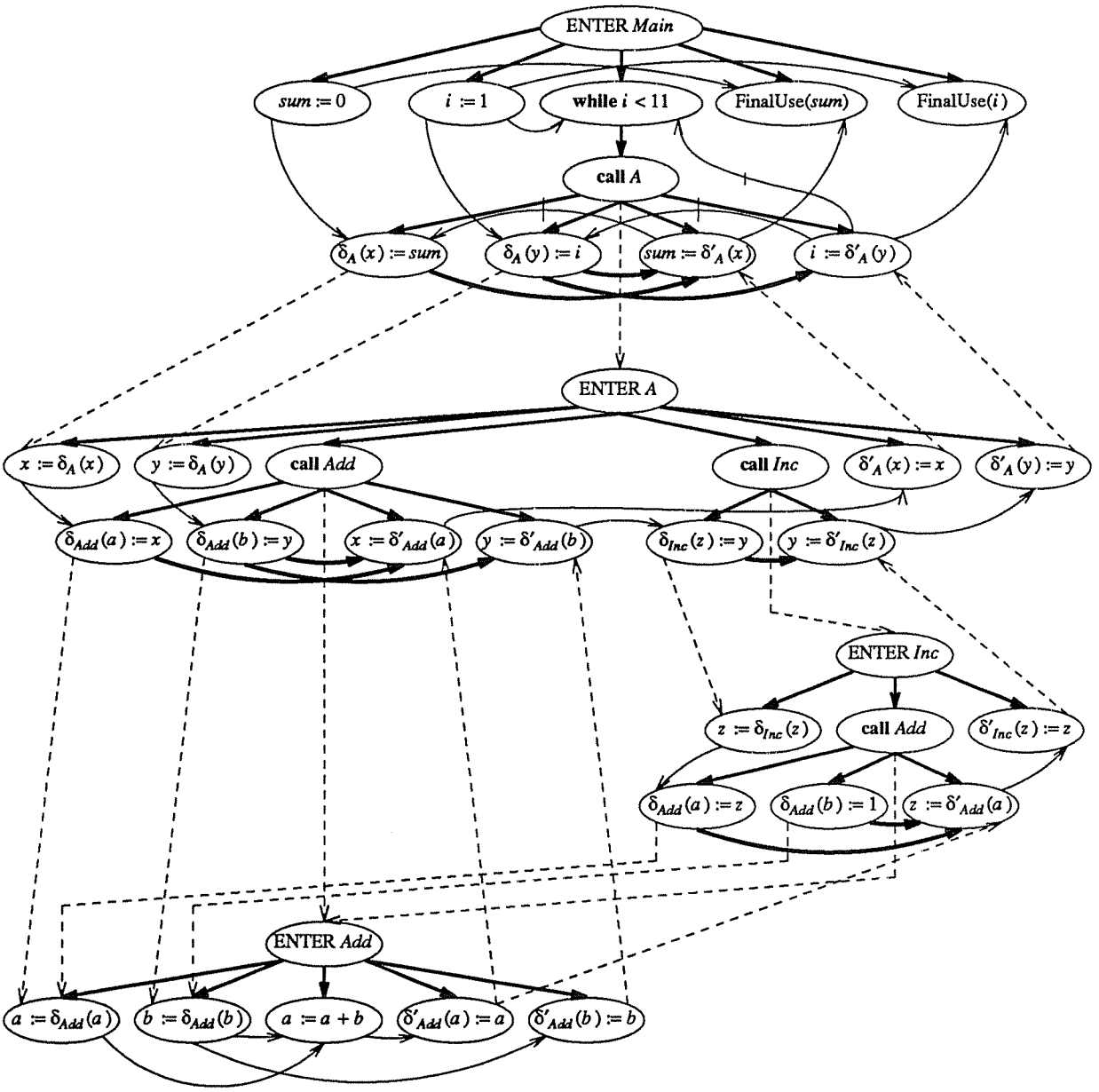


Figure 8. Example system's system dependence graph. Control dependencies, which are shown unlabeled, are represented using medium-bold arrows; intraprocedural flow dependencies are represented using arcs; transitive interprocedural flow dependencies (corresponding to subordinate characteristic graph edges) are represented using heavy bold arcs; call edges, linkage-entry edges, and linkage-exit edges (the edges that connect program and procedure dependence graphs together) are represented using dashed arrows.

```
procedure MarkVerticesOfSlice(G, S)
declare
  G: a system dependence graph
  S, S': sets of vertices in G
begin
  /* Phase 1: Slice without descending into called procedures */
    MarkReachingVertices(G, S, {def-order, linkage-exit})
  /* Phase 2: Slice called procedures without ascending to call sites */
    S' := all marked vertices in G
    MarkReachingVertices(G, S', {def-order, linkage-entry, call})
end

procedure MarkReachingVertices(G, V, Kinds)
declare
  G: a system dependence graph
  V: a set of vertices in G
  Kinds: a set of kinds of edges
  v, w: vertices in V
  WorkList: a set of vertices in G
begin
  WorkList := V
  while WorkList ≠ ∅ do
    Select and remove a vertex v from WorkList
    Mark v
    for each vertex w that is a predecessor of v in G such that there is an edge w →v whose kind is not in Kinds do
      Insert w into WorkList
    od
  od
end
```

Figure 9. The procedure MarkVerticesOfSlice marks the vertices of the interprocedural slice G / S. The auxiliary procedure MarkReachingVertices marks all vertices in G from which there is a path to a vertex in V along edges of kinds other than those in the set *Kinds*.

forming an in*tra*procedural slice – the graph is traversed to find the set of vertices that can reach a given set of vertices along certain kinds of edges. The traversal in Phase 1 follows flow edges, control edges, call edges, and linkage-entry edges, but does *not* follow def-order edges or linkage-exit edges. The traversal in Phase 2 follows flow edges, control edges, and linkage-exit edges, but does *not* follow call edges, def-order edges, or linkage-entry edges.

Figures 10 and 11 illustrate the two phases of the interprocedural slicing algorithm. Figure 10 shows the vertices of the example system dependence graph that are marked during Phase 1 of the interprocedural slicing algorithm when the system is sliced with respect to the finalization vertex for parameter z in procedure *Increment*. Edges "traversed" during Phase 1 are also included in Figure 10. Figure 11 adds (in boldface) the vertices that are marked and the edges that are traversed during Phase 2 of the slice.

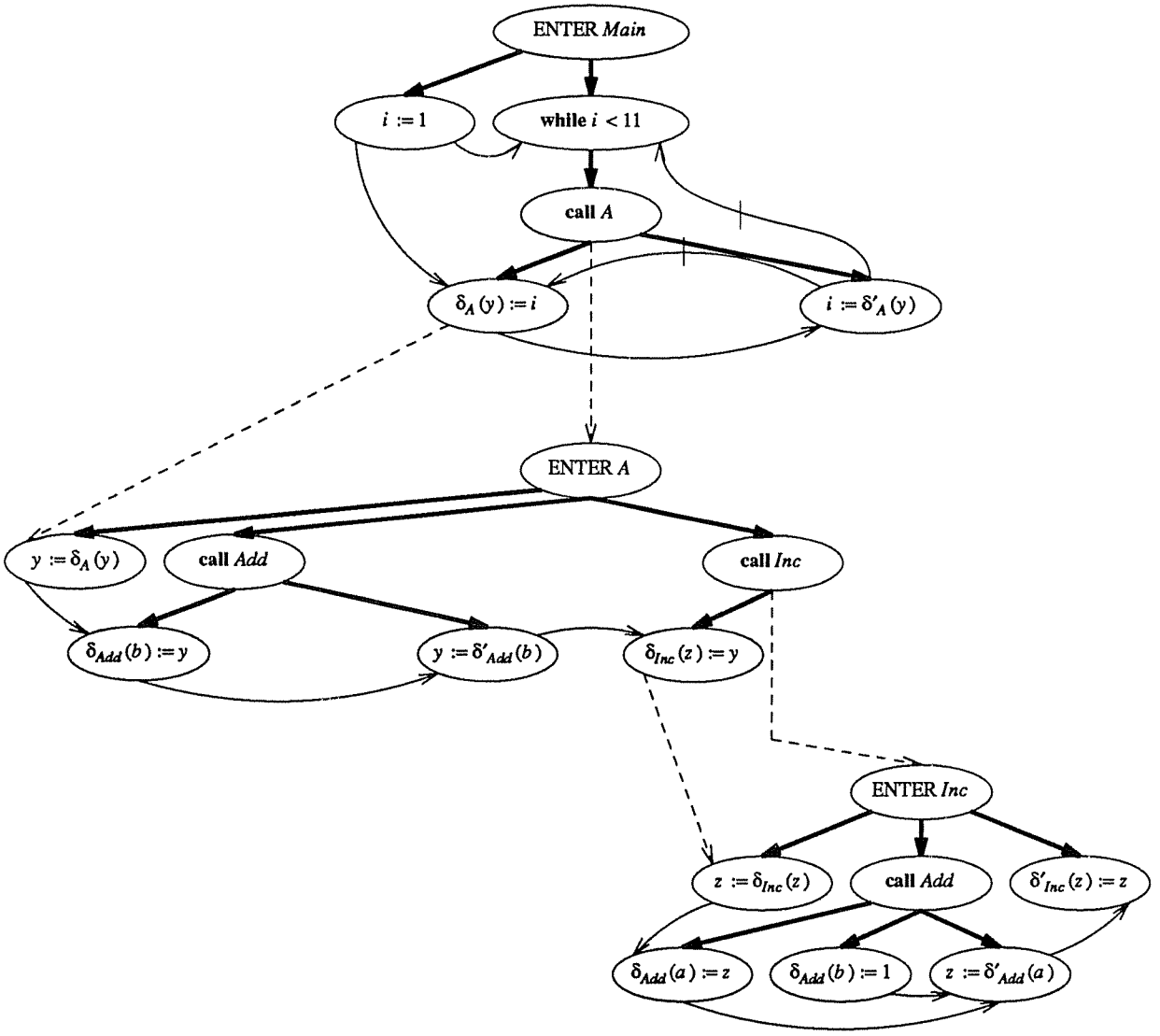


Figure 10. The example program's system dependence graph is sliced with respect to the finalization vertex for parameter z in procedure *Increment*. The vertices marked by Phase 1 of the slicing algorithm as well as the edges traversed during this phase are shown above.

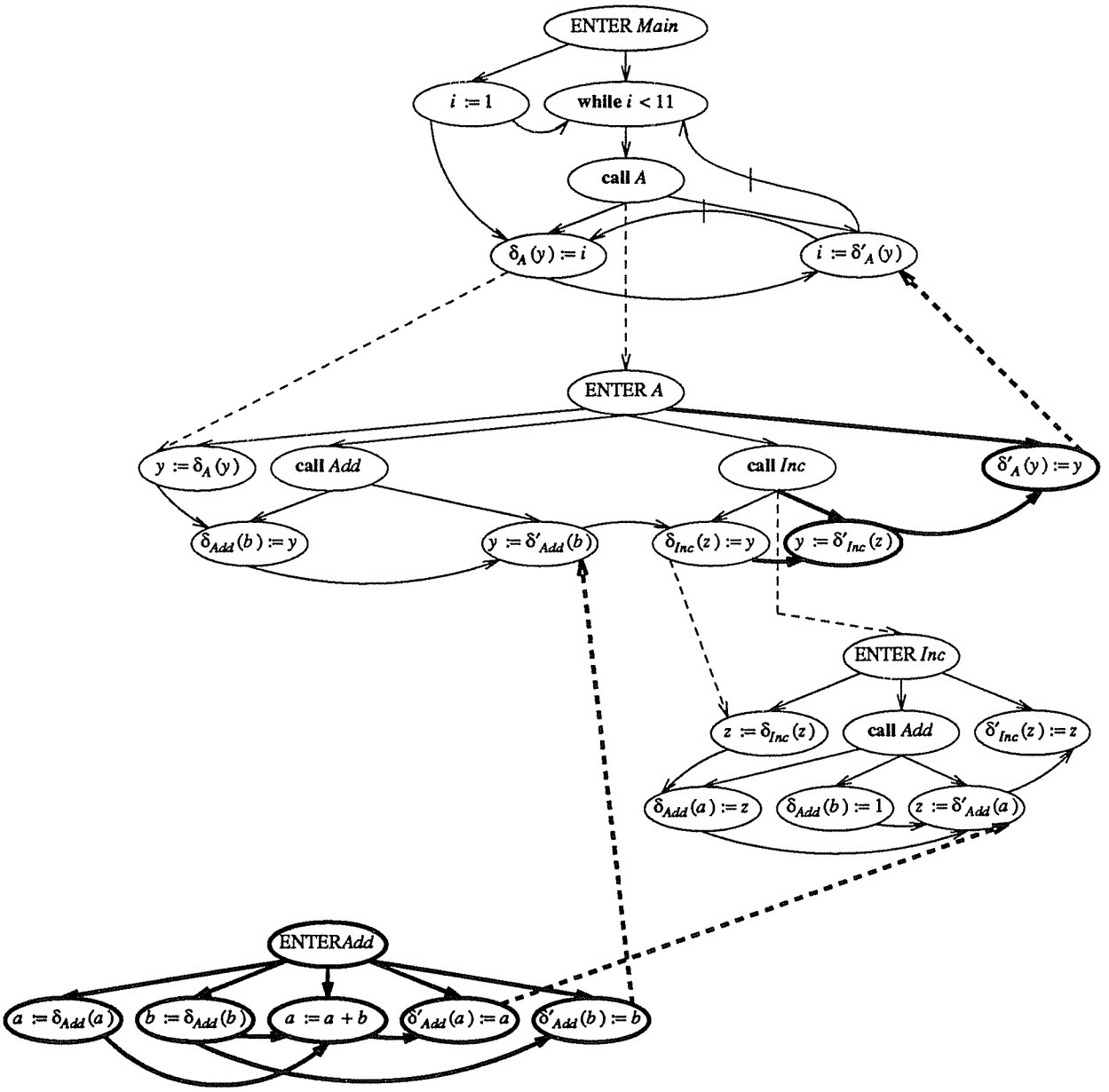


Figure 11. The example program's system dependence graph is sliced with respect to the finalization vertex for parameter z in procedure *Increment*. The vertices marked by Phase 2 of the slicing algorithm as well as the edges traversed during this phase are shown above in boldface.

The result of an interprocedural slice consists of the sets of vertices identified by Phase 1 and Phase 2, and the set of edges induced by this vertex set. Figure 12 shows the completed example slice.

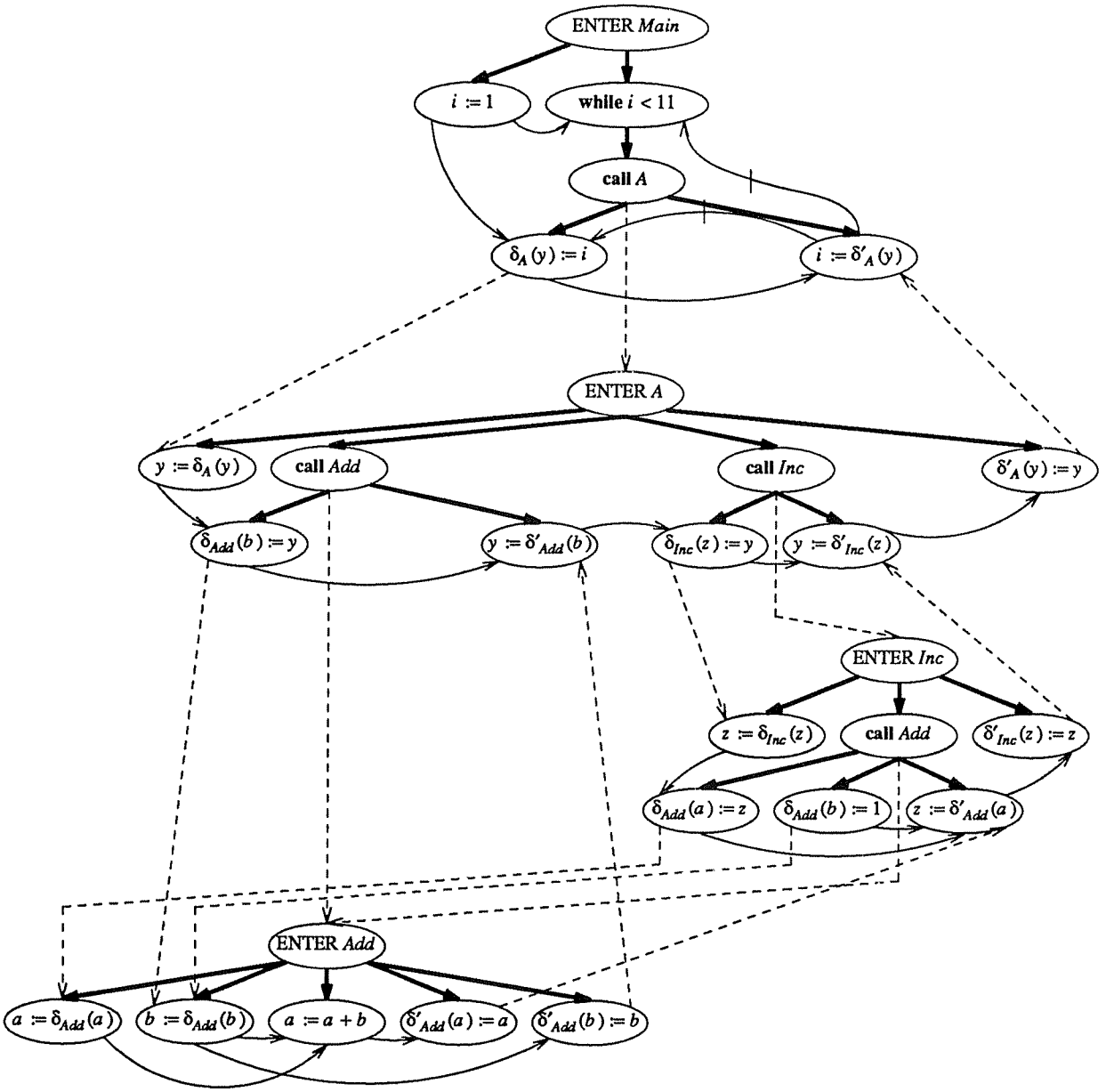


Figure 12. The complete slice of the example program's system dependence graph sliced with respect to the finalization vertex for parameter z in procedure *Increment*.

Given the goal of slicing system dependence graph G with respect to vertex s in procedure P, Phases 1 and 2 can be characterized as follows:

Phase 1

Phase 1 identifies vertices that can reach s, and that are either in P itself or in a procedure that calls

P. The effects of procedures called by P are not entirely ignored; the presence of flow dependence edges from preprocessing to postprocessing vertices (subordinate characteristic graph edges) permits the discovery of vertices that can reach s only through a procedure call, although the graph traversal does not actually "descend" into the called procedure.

Phase 2

Phase 2 identifies vertices that can reach s from procedures called by P or from procedures called by procedures that call P.

4.2. Using Interprocedural Summary Information to Build Procedure Dependence Graphs

The slice shown in Figure 12 illustrates a shortcoming of the method for constructing procedure dependence graphs described in Section 3. The problem is that including both a pre- and a post-processing vertex for *every* argument in a procedure call can affect the precision of an interprocedural slice. The slice shown in Figure 12 includes the call vertex that represents the call to *Add* from A; however, this call does not in fact affect the value of z in *Increment*. The problem is that a postprocessing vertex for argument y in the call to *Add* from A is included in A's procedure dependence graph even though *Add* does not change the value of y.

To achieve a more precise interprocedural slice we must use the results of interprocedural data flow analysis when constructing procedure dependence graphs in order to exclude vertices like the post-processing vertex for argument y.

The appropriate interprocedural summary information consists of the following sets, which are computed for each procedure P [Banning79]:

GMOD(P):

the set of variables that might be *modified* by P itself or by a procedure (transitively) called from P.

GREF(P):

the set of variables that might be *referenced* by P itself or by a procedure (transitively) called from P.

GMOD and GREF sets are used to determine which linkage vertices are included in procedure dependence graphs as follows: For each procedure P, the linkage vertices subordinate to P's entry vertex include one initialization vertex for each variable in GMOD(P) $\cup$ GREF(P), and one finalization vertex for each variable in GMOD(P). Similarly, for each site at which P is called, the linkage vertices subordinate to the call-site vertex include one preprocessing vertex for each variable in GMOD(P) $\cup$ GREF(P), and one postprocessing vertex for each variable in GMOD(P). (It is necessary to include a preprocessing and an initialization vertex for a variable x that is in GMOD(P) and is not in GREF(P) because there may be an execution path through P on which x is *not* modified. In this case, a slice of P with respect to the final value of x must include the initial value of x; thus, there must be an initialization vertex for x in P, and a corresponding preprocessing vertex at the call to P.)

Example. The GMOD and GREF sets for our example system are:

procedure	GMOD	GREF
A	x, y	x, y
Add	a	a, b
Inc	z	z

Because parameter b is not in GMOD(*Add*), *Add*'s procedure dependence graph should not include a

finalization vertex for b, and the call to *Add* from A should not include the corresponding postprocessing vertex.

Figure 13 shows A's procedure dependence graph as it would be built using GMOD and GREF information. The postprocessing vertex for argument y of the call to *Add* is omitted, and the flow edge from that vertex to the preprocessing vertex "$\delta_{Inc}(z) := y$" is replaced by an edge from the initialization vertex "$y := \delta_A(y)$" to "$\delta_{Inc}(z) := y$". The new edge is traversed during Phase 1 of the interprocedural slice instead of the (now omitted) flow edge from "$y := \delta'_{Add}(a)$" to "$\delta_{Inc}(z) := y$", thus (correctly) bypassing the call to *Add* in procedure A.

4.3. Procedure and System Dependence Graphs in the Presence of Aliasing

The problem of interprocedural slicing in the presence of aliasing can be reduced to the problem of interprocedural slicing in the *absence* of aliasing at the expense of the time and space needed to convert the original program into one that is alias free. (These costs may, in the worst case, be exponential in the maximum number of non-local variables – globals and parameters – visible to a procedure.)

The conversion is performed by simulating the calling behavior of the program (using the usual activation-tree model of procedure calls [Banning79]) to discover, for each instance of a procedure call, exactly how variables are aliased at that instance. (Although a recursive program's activation tree is infinite, the number of different alias configurations is finite; thus, only a finite portion of the activation tree is needed to compute aliasing information.) A new copy of the procedure (with a new procedure name) is created for each different alias configuration; the procedure names used at call sites are similarly adjusted. Within each procedure, variables are renamed so that each set of aliased variables is replaced by a single variable.

This process may generate multiple copies of a vertex v with respect to which we are to perform a slice. If this happens, it is necessary to slice the transformed program with respect to *all* occurrences of v.

Example. Consider the following program in which aliasing occurs:

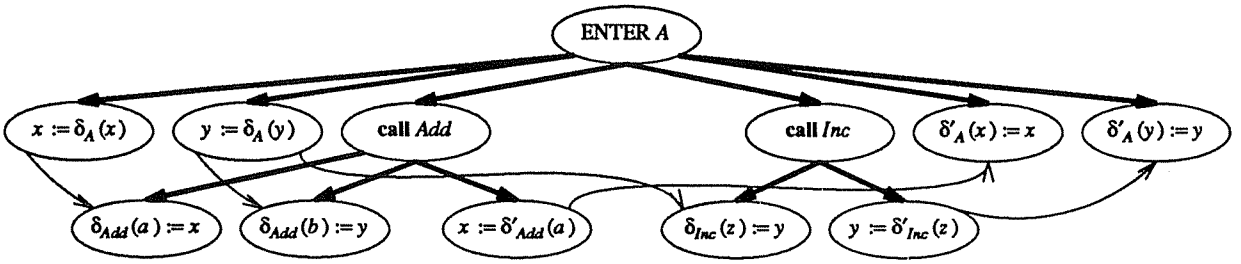


Figure 13. Procedure A's procedure dependence graph built using interprocedural summary information. The postprocessing vertex for argument y of the call to *Add* has been omitted, and the flow edge from that vertex to the vertex "$\delta_{Inc}(z) := y$" has been replaced by an edge from the vertex "$y := \delta_A(y)$" to the vertex "$\delta_{Inc}(z) := y$".

```
program Main                    procedure P(x)
global var a, b, c                x := 0;
  b := 0;                         a := x + b;
  call P(b);                      call P(c)
  call P(a)                     return
end
```

Figure 14 shows the portion of this program's activation tree that is used to compute alias information for each call instance. We use the notation of [Banning79], in which each node of the activation tree is labeled with the mapping from variable names to storage locations. The transformed, alias-free version of the program is shown below:

```
program Main                    procedure P(x)
global var a, b, c                x := 0;
  b := 0;                         a := x + b;
  call P(b);                      call P(c)
  call P(a)                     return
end
```

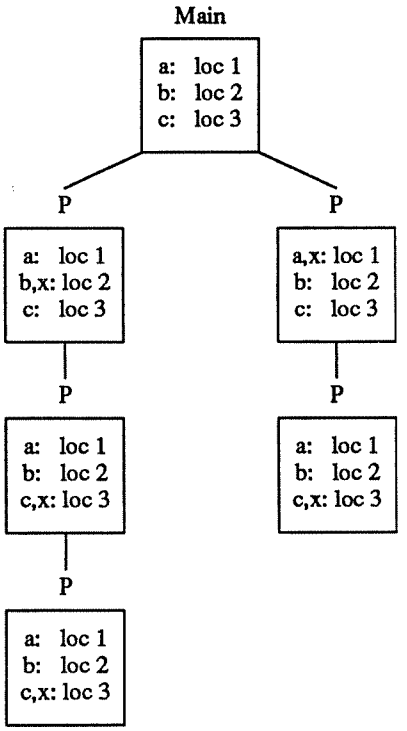


Figure 14. A program with aliasing and the portion of its activation tree needed to compute all alias configurations.

```
program Main          procedure P1(bx)      procedure P2(cx)      procedure P3(ax)
global var a,b,c        bx := 0;              cx := 0;              ax := 0;
  b := 0;               a := bx + bx;         a := cx + b;          ax := ax + b;
  call P1(b);           call P2(c)            call P2(cx)           call P2(c)
  call P3(a)          return                return                return
end
```

If our original goal had been to slice with respect to the statement "$a := x + b$" in procedure P, we must now slice with respect to the set of statements:
{ "$a := bx + bx$", "$a := cx + b$", "$ax := ax + b$" }.

4.4. Slicing Partial System Dependence Graphs

The interprocedural slicing algorithm presented above is designed to be applied to a complete system dependence graph. In this section we discuss how to slice *incomplete* system dependence graphs.

The need to handle incomplete systems arises, for example, when slicing a program that calls a library procedure that is not itself available, or when slicing programs under development. In the first case, the missing components are procedures that are called by the incomplete system; in the second case, the missing components can either be not-yet-written procedures called by the incomplete system (when the program is developed top-down), or possible calling contexts (when the program is developed bottom-up).

In either case, information about the possible effects of missing calls and missing calling contexts is needed to permit slicing. This information takes the form of (safe approximations to) the subordinate characteristic graphs for missing called procedures and the superior characteristic graphs for missing calling contexts.

When no information about missing program components is available, complete bipartite subordinate and superior characteristic graphs must be used. This is because the slice of the incomplete system should include all vertices that could be included in the slice of some "completed" system, and it is always possible to provide a call or a calling context that corresponds to a complete bipartite subordinate or superior characteristic graph.

For library procedures, it is possible to provide precise subordinate characteristic graphs even when the procedures themselves are not provided. For programs under development, it might be possible to compute characteristic graphs, or at least better approximations to them than complete bipartite graphs, given specifications for the missing program components.

5. THE COMPLEXITY OF THE SLICING ALGORITHM

This section discusses the complexity of the interprocedural slicing algorithm presented in Section 4.1. In the absence of aliasing, the cost is polynomial in (various) parameters of the system. In the presence of aliasing, the costs increase by an exponential factor that reflects the number of aliasing patterns in the program. The increased cost is due to the blow-up in program size that can occur when a program with aliasing is converted to one that is alias free. Below we assume that such conversion has already been accomplished; the measures of system size used below are those associated with the alias-free system.

5.1. Cost of Constructing the System Dependence Graph

The cost of constructing the system dependence graph can be expressed in terms of the parameters given in the following tables:

Parameters that measure the size of an individual procedure	
V	the largest number of predicates and assignments in a single procedure
E	the largest number of edges in a single procedure dependence graph
Params	the largest number of formal parameters in any procedure
Sites	the largest number of call sites in any procedure

Parameters that measure the size of the entire system	
P	the number of procedures in the system (= the number of productions in the linkage grammar)
Globals	the number of global variables in the system
$TotalSites \leq P \cdot Sites$	the total number of call sites in the system

Interprocedural data flow analysis is used to compute summary information about side effects. Flow-insensitive interprocedural summary information (*e.g.* GMOD and GREF) can be determined particularly efficiently. In particular, in the absence of nested scopes, GMOD and GREF can be determined in time $O(P^2 + P \cdot TotalSites)$ steps by the algorithm described in [Cooper88].

Intraprocedural data flow analysis is used to determine the data dependencies of procedure dependence graphs. For the structured language under consideration here, this analysis can be performed in a syntax-directed fashion (for example, using an attribute grammar) [Horwitz87]. This involves propagating sets of program points, where each set consists of program points in a single procedure. This computation has total cost $O(V^2)$.

The cost of constructing the linkage grammar and computing its subordinate characteristic graphs can be expressed in terms of the following parameters:

Parameters that measure the size of the linkage grammar	
$R = Sites + 1$	the largest number of nonterminal occurrences in a single production
$G = P + TotalSites$ $\leq P \cdot R$ $= P \cdot (Sites + 1)$	the number of nonterminal occurrences in the linkage grammar
$X = Globals + Params$	the largest number of attributes of a single nonterminal
$D \leq R \cdot X$ $= (Sites + 1)$ $\cdot (Globals + Params)$	the largest number of attribute occurrences in a single production

To determine the dependencies among the attribute occurrences in each production, its corresponding procedure is sliced with respect to the linkage vertices that correspond to the attribute occurrences of the production. The cost of each slice is linear in the size of the procedure dependence graph; that is, the cost is bounded by $O(V+E)$. Consequently, the total cost of constructing the linkage grammar is bounded by $O(G \cdot X \cdot (V+E))$.

It remains for us to analyze the cost of computing the linkage grammar's subordinate characteristic graphs. Because there are at most D^2 edges in each $TDP(p)$ relation, the cost of AddEdgeAndInduce, which re-closes a single $TDP(p)$ relation, is $O(D^2)$. The cost of initializing the *TDP* relations with all direct dependencies in ConstructSubCGraphs is bounded $O(P \cdot D^2)$.

In the inner loop of procedure ConstructSubCGraphs, the AddEdgeAndInduce step is executed once for each occurrence of nonterminal N. There are at most X^2 edges in each graph $TDS(N)$ and G nonterminal occurrences where an edge may be induced. No edge is induced more than once because of the marks on *TDS* edges; thus, the total cost of procedure ConstructSubCGraphs is bounded by $O(G \cdot X^2 \cdot D^2)$ [Kastens80].

5.2. Slicing Costs

An interprocedural slice is performed by two traversals of the system dependence graph, starting from some initial set of vertices. The cost of each traversal is linear in the size of the system dependence graph, which is bounded by $O(P \cdot (V + E) + TotalSites \cdot X)$.

6. RELATED WORK

In recasting the interprocedural slicing problem as a reachability problem in a graph, we are following the example of [Ottenstein84], which does the same for intraprocedural slicing. The reachability approach is conceptually simpler than the data-flow equation approach used in [Weiser84] and is also much more efficient when more than one slice is desired.

The recasting of the problem as a reachability problem does involve some loss of generality; rather than permitting a program to be sliced with respect to program point p and an *arbitrary* variable, a slice can only be taken with respect to a variable that is defined at or used at p. For such slicing problems the interprocedural slicing algorithm presented in this paper is an improvement over Weiser's algorithm because our algorithm is able to produce a more precise slice than the one produced by Weiser's algorithm. However, the extra generality is not the source of the imprecision of Weiser's method; instead, the imprecision is due to the lack of a mechanism to keep track of the calling context of a called procedure.

Weiser's method for interprocedural slicing is described as follows (recall that, in Weiser's terminology, a *slicing criterion* is a pair $<p, V>$, where p is a program point and V is a subset of the program's variables):

> For each criterion C for a procedure P, there is a set of criteria $\mathrm{UP}_0(C)$ which are those needed to slice callers of P, and a set of criteria $\mathrm{DOWN}_0(C)$ which are those needed to slice procedures called by P. . . . $\mathrm{UP}_0(C)$ and $\mathrm{DOWN}_0(C)$ can be extended to functions UP and DOWN which map sets of criteria into sets of criteria. Let CC be any set of criteria. Then
>
> $$\mathrm{UP}(CC) = \bigcup_{C \in CC} \mathrm{UP}_0(C)$$
>
> $$\mathrm{DOWN}(CC) = \bigcup_{C \in CC} \mathrm{DOWN}_0(C)$$
>
> The union and transitive closure of UP and DOWN are defined in the usual way for relations. $(\mathrm{UP} \cup \mathrm{DOWN})^*$ will map any set of criteria into all those criteria necessary to complete the corresponding slices through all calling and called routines. The complete interprocedural slice for a criterion C is then just the union of the intraprocedural slices for each criterion in $(\mathrm{UP} \cup \mathrm{DOWN})^*(C)$.
>
> [Weiser84]

This method does not produce as precise a slice as possible; the use of the transitive closure operation fails to account for the calling context of a called procedure.

[Myers81] presents algorithms for a specific set of interprocedural data flow problems, all of which require keeping track of calling context; however, Myers's approach to handling this problem differs from ours. Myers performs data flow analysis on a graph representation of the program, called a *super graph*, which is a collection of control-flow graphs (one for each procedure in the program), connected together by call and return edges. The information maintained at each vertex of the super graph includes a *memory*

component, which keeps track of calling context (essentially by using the name of the call site). Our use of the system dependence graph permits keeping track of calling context while propagating simple marks rather than requiring the propagation of sets of names. In particular, in Phase 1 of our interprocedural slicing algorithm, the presence of the linkage grammar's subordinate-characteristic-graph edges (representing transitive dependencies due to procedure calls) permits the entire effect of a call to be accounted for by a single backward step over the call site's subordinate-characteristic-graph edges.

It is no doubt possible to formulate interprocedural slicing as a data flow analysis problem on a super graph, and to solve the problem using an algorithm akin to those described by Myers to account correctly for the calling context of a called procedure. One advantage of the approach described in this paper over the one postulated above arises when one wishes to compute multiple slices of the same system. The system dependence graph (with its subordinate characteristic graph edges) can be computed once and for all and then used for each slicing operation. By contrast, the alternative approach would involve solving a new data flow analysis problem from scratch for each slice.

The vertex-reachability approach we have used here has some similarities to a technique used in [Kou77], [Callahan88], and [Cooper88] to transform data flow analysis problems to vertex-reachability problems. In each case a data flow analysis problem is solved by first building a graph representation of the program, and then performing a reachability analysis on the graph, propagating simple marks rather than, for example, sets of variable names. One difference between our work and that cited above, is that our work concerns a "demand problem" [Babich78] whose goal is to determine information concerning a specific set of program points rather than an "exhaustive problem" in which the goal is to determine information for all program points.

APPENDIX A: ATTRIBUTE GRAMMARS AND ATTRIBUTE DEPENDENCIES

An attribute grammar is a context-free grammar extended by attaching *attributes* to the terminal and nonterminal symbols of the grammar, and by supplying *attribute equations* to define attribute values [Knuth68]. In every production $p: X_0 \rightarrow X_1, ..., X_k$, each X_i denotes an *occurrence* of one of the grammar symbols; associated with each such symbol occurrence is a set of *attribute occurrences* corresponding to the symbol's attributes.

Each production has a set of attribute equations; each equation defines one of the production's attribute occurrences as the value of an *attribute-definition function* applied to other attribute occurrences in the production. The attributes of a symbol X, denoted by $A(X)$, are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes. Each attribute equation defines a value for a synthesized attribute occurrence of the left-hand-side nonterminal or an inherited attribute occurrence of a right-hand-side symbol.

An attribute grammar is *well formed* when the terminal symbols of the grammar have no synthesized attributes, the root nonterminal of the grammar has no inherited attributes, and each production has exactly one attribute equation for each of the left-hand-side nonterminal's synthesized attribute occurrences and for each of the right-hand-side symbols' inherited attribute occurrences.

A derivation tree node that is an instance of symbol X has an associated set of *attribute instances* corresponding to the attributes of X. (We shall sometimes shorten "attribute instances" and "attribute occurrences" to "attributes;" however, the intended meaning should be clear from the context). An *attributed tree* is a derivation tree together with an assignment of either a value or the special token **null** to each attribute instance of the tree.

Ordinarily, although not in this paper, one is interested in analyzing a string according to its attribute-grammar specification. To do this, one first constructs the string's derivation tree with an assignment of **null** to each attribute instance, and then evaluates as many attribute instances as possible, using the appropriate attribute equation as an assignment statement. The latter process is termed *attribute evaluation.*

Functional dependencies among attribute occurrences in a production p (or attribute instances in a tree T) can be represented by a directed graph, called a *dependency graph*, denoted by $D(p)$ (respectively, $D(T)$) and defined as follows:

- For each attribute occurrence (instance) b, the graph contains a vertex b'.
- If attribute occurrence (instance) b appears on the right-hand side of the attribute equation that defines attribute occurrence (instance) c, the graph contains an edge (b', c'), directed from b' to c'.

An attribute grammar that has a derivation tree whose dependency graph contains a cycles is called a *circular* attribute grammar. The grammars that arise in this paper are potentially circular grammars.

A node's *subordinate* and *superior characteristic graphs* provide a convenient representation of transitive dependencies among the node's attributes. (A *transitive dependency* exists between attributes that are related in the transitive closure of the tree's attribute dependency relation, or, equivalently, that are connected by a directed path in the tree's dependency graph.) The vertices of the characteristic graphs at node r correspond to the attributes of r; the edges of the characteristic graphs at r correspond to transitive dependencies among r's attributes.

The subordinate characteristic graph at r is the projection of the dependencies of the subtree rooted at r onto the attributes of r. To form the superior characteristic graph at node r, we imagine that the subtree rooted at r has been pruned from the derivation tree, and project the dependency graph of the remaining tree onto the attributes of r. To define the characteristic graphs precisely, we make the following definitions:

- Given a directed graph $G = (V, E)$, a *path* from vertex a to vertex b is a sequence of vertices, $[v_1, v_2, ..., v_k]$, such that: $a = v_1$, $b = v_k$, and $\{ (v_i, v_{i+1}) \mid i = 1, ..., k-1 \} \subseteq E$.
- Given a directed graph $G = (V, E)$ and a set of vertices $V' \subseteq V$, the *projection* of G onto V' is defined as:

  $G/V' = (V', E')$

  where $E' = \{(v, w) \mid v, w \in V'$ and there exists a path $[v = v_1, v_2, ..., v_k = w]$ such that $v_2, ..., v_{k-1} \notin V'$. (That is, G/V' has an edge from $v \in V'$ to $w \in V'$ when there exists a path from v to w in G that does not pass through any other elements of V'.)

The subordinate and superior characteristic graphs of a node r, denoted $r.C$ and $r.\overline{C}$, respectively, are defined formally as follows: Let r be a node in tree T, let the subtree rooted at r be denoted T_r, and let the attribute instances at r be denoted $A(r)$, then the subordinate and superior characteristic graphs at r satisfy:

$$r.C = D(T_r)/A(r)$$

$$r.\overline{C} = (D(T) - D(T_r))/A(r)$$

A characteristic graph represents the projection of attribute dependencies onto the attributes of a single tree node; consequently, for a given grammar, each graph is bounded in size by some constant.

REFERENCES

Aho86.
Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

Babich78.
Babich, W.A. and Jazayeri, M., "The method of attributes for data flow analysis: Part II. Demand analysis," *Acta Informatica* 10(3) pp. 265-272 (October 1978).

Banning79.
Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, (San Antonio, TX, Jan. 29-31, 1979), ACM, New York (1979).

Callahan88.
Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices*, (June 1988).

Cooper88.
Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices*, (June 1988).

Ferrante87.
Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* 9(3) pp. 319-349 (July 1987).

Horwitz87.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," TR-690, Computer Sciences Department, University of Wisconsin, Madison, WI (March 1987).

Horwitz88.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," pp. 133-145 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York (1988).

Kastens80.
Kastens, U., "Ordered attribute grammars," *Acta Inf.* 13(3) pp. 229-256 (1980).

Knuth68.
Knuth, D.E., "Semantics of context-free languages," *Math. Syst. Theory* 2(2) pp. 127-145 (June 1968).

Kou77.
Kou, L.T., "On live-dead analysis for global data flow problems," *Journal of the ACM* 24(3) pp. 473-483 (July 1977).

Kuck72.
Kuck, D.J., Muraoka, Y., and Chen, S.C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Trans. on Computers* C-21(12) pp. 1293-1310 (December 1972).

Kuck81.
Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York (1981).

Myers81.
Myers, E., "A precise inter-procedural data flow algorithm," pp. 219-230 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York (1981).

Ottenstein84.
Ottenstein, K.J. and Ottenstein, L.M., "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, Apr. 23-25, 1984), *ACM SIGPLAN Notices* 19(5) pp. 177-184 (May 1984).

Reps88.
Reps, T. and Yang, W., "The semantics of program slicing," Tech. Rep. in preparation, Computer Sciences Department, University of Wisconsin, Madison, WI (Spring 1988).

Weiser84.
Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* SE-10(4) pp. 352-357 (July 1984).