Interprocedural Slicing Using Dependence Graphs

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

The notion of a *program slice*, originally introduced by Mark Weiser, is useful in program debugging, automatic parallelization, and program integration. A slice of a program is taken with respect to a program point p and a variable x; the slice 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. (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* or *used* at p.)

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* dependences 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.

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: Attribute grammar, control dependence, data dependence, data-flow analysis, flow-insensitive summary information, program debugging, program dependence graph, program integration, program slicing, subordinate characteristic graph

An earlier version of this paper appeared in abridged form in the *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, Ga., June 22–24, 1988), *ACM SIGPLAN Not. 23*, 7 (July 1988) [10].

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602, DCR-8603356, and 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 IBM, DEC, and Xerox.

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 0164-0925/90/0100-0026 $01.50

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. This concept, originally discussed by Mark Weiser in [22], can be used to isolate individual computation threads within a program. Slicing can help a programmer understand complicated code, can aid in debugging [17], and can be used for automatic parallelization [3, 21]. Program slicing is also used by the algorithm for automatically integrating program variants described in [11]; 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 $\langle p, V \rangle$, 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 values 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 x that is *defined* or *used* at p. The value of a variable x defined at p is directly affected by the values of the variables used at p and by the loops and conditionals that enclose p. The value of a variable y *used* at p is directly affected by assignments to y that reach p and by the loops and conditionals that enclose p. When slicing a program that consists of a single monolithic procedure (which we will call in*tra*procedural slicing), a slice can be determined from the closure of the directly-affects relation. Ottenstein and Ottenstein pointed out how well suited *program dependence graphs* are for this kind of slicing [19]; once a program is represented by its program dependence graph, the slicing problem is simply a vertex-reachability problem, and thus slices may be computed in linear time.

This paper concerns the problem of in*ter*procedural 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 answer than that produced by the algorithm given by Weiser in [22]. Our work follows the example of Ottenstein and Ottenstein by defining the slicing algorithm in terms of operations on a dependence graph representation of programs [19]; however, in [19] Ottenstein and Ottenstein only discuss the case of programs that consist of a single monolithic procedure, and do not discuss the more general case where slices cross 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.

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 [20].

For in*ter*procedural slicing, restricting the original program to just the statements and predicates found for Version 1 may yield a program that is syntactically incorrect (and thus certainly not a solution to Version 2). The reason behind this phenomenon 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 interprocedural slicing problem (with the further restriction, mentioned earlier, that a slice can only be taken with respect to program point p and variable x if x is defined or used at p). The algorithm given in the paper identifies a subgraph of the system dependence graph whose components might affect the sequence of values for x at p. A solution to Version 2 requires either that the slice be extended or that it be transformed by duplicating code to specialize procedure bodies for particular parameter-usage patterns.

Weiser's method for interprocedural slicing is described in [22] as follows:

> 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 $(\mathrm{UP} \cup \mathrm{DOWN})^*(C)$.

However, this method does not produce as precise a slice as possible because the transitive-closure operation fails to account for the calling context of a called procedure.[1]

[1] For example, the relation $(\mathrm{UP} \cup \mathrm{DOWN})^*(\langle p, V\rangle)$ includes the relation $\mathrm{UP}(\mathrm{DOWN}(\langle p, V\rangle))$. $\mathrm{UP}(\mathrm{DOWN}(\langle p, V\rangle))$ includes all call sites that call procedures containing the program points in $\mathrm{DOWN}(\langle p, V\rangle)$, not just the procedure that contains p. This fails to account for the calling context, namely the procedure that contains p.

Example. To illustrate this problem, and the shortcomings of Weiser's algorithm, consider the following example program, which sums the integers from 1 to 10. (Except in Section 4.3, where call-by-reference parameter passing is discussed, parameters are passed by value-result.)

```
program Main         procedure A(x, y)       procedure Add(a, b)   procedure Increment(z)
  sum := 0;            call Add(x, y);         a := a + b            call Add(z, 1)
  i := 1;              call Increment(y)     return                return
  while i < 11 do    return
    call A(sum, i)
  od
end
```

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, a closer 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 first actual parameter of the call on procedure A in *Main*, nor the call on *Add* in A (which adds the current value of i to *sum*) should be included in the slice. The reason these components are included in the slice computed by Weiser's algorithm is as follows: the initial slicing criterion "⟨end of procedure *Increment*, z⟩", is mapped by the DOWN relation to a slicing criterion "⟨end of procedure *Add*, a⟩". The latter criterion is then mapped by the UP relation to *two* slicing criteria—corresponding to *all* sites that call *Add*—the criterion "⟨call on *Add* in *Increment*, z⟩" and the (irrelevant) criterion "⟨call on *Add* in A, x⟩". Weiser's algorithm does not produce as precise a slice as possible because transitive closure fails to account for the calling context (*Increment*) of a called procedure (*Add*), and thus generates a spurious criterion (⟨call on *Add* in A, x⟩).

A more precise slice consists of the following elements:

```
program Main         procedure A(y)          procedure Add(a, b)   procedure Increment(z)
  i := 1;              call Increment(y)       a := a + b            call Add(z, 1)
  while i < 11 do    return                  return                return
    call A(i)
  od
end
```

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

The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. To address the calling-context problem, system dependence graphs include some data dependence edges that represent *transitive* dependences due to the effects of procedure calls, in addition to the conventional edges for direct dependences. The presence of transitive-dependence edges permits interprocedural slices to be computed in two passes, each of which is cast as a reachability problem.

The cornerstone of the construction of the system dependence graph 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. The need to express this step in this fashion (rather than, for example, with transitive closure) is discussed further in Section 3.2.

The remainder of the paper is organized as follows: Section 2 defines the dependence graphs used to represent programs in a language without procedure calls. Section 2 also defines the operation of intraprocedural slicing on these 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 slicing by using interprocedural summary information in the construction of system dependence graphs, how to handle programs with aliasing, how to slice incomplete programs, and how to compute *forward slices* (i.e., the program elements potentially *affected by* a given variable at a given point). Section 5 discusses the complexity of the slicing algorithm. We have not yet implemented this algorithm in its entirety; thus, Section 5 provides an analysis of the costs of building system dependence graphs and of taking interprocedural slices rather than presenting empirical results. Section 6 discusses related work.

With the exception of the material on interprocedural data-flow analysis employed in Section 4.2, the paper is self-contained; an introduction to the terminology and concepts from attribute-grammar theory that are used in Section 3.2 may be found in the Appendix.

2. PROGRAM-DEPENDENCE GRAPHS AND PROGRAM SLICES

Different definitions of program dependence representations have been given, depending on the intended application; they are all variations on a theme introduced in [16], and share the common feature of having an explicit representation of data dependences (see below). The "program dependence graphs" defined in [7] introduced the additional feature of an explicit representation for control dependences (see below). The definition of program dependence graph given below differs from [7] in two ways. First, our definition covers only a restricted language with scalar variables, assignment statements, conditional statements, while loops, and a restricted kind of "output statement" called an *end statement*,[2] and hence is less general than the one given in [7]. Second, we omit certain classes of data dependence edges and make use of a class introduced in [8, 11]. Despite these differences, the structures we define and those defined in [7] share the feature of explicitly representing both control and data dependences; therefore, we refer to our graphs as "program dependence graphs," borrowing the term from [7].

[2] 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; 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.[3] 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 on which x is used before being defined (see [1]), 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 *dependences* among program components. An edge represents either a *control dependence* or a *data dependence.* Control dependence edges are labeled either **true** or **false**, and the source of a control dependence edge is always the entry vertex or a predicate vertex. A control dependence 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 eventually be executed if the program terminates. A method for determining control dependence edges for arbitrary programs is given in [7]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependence edges of G_P can be determined in a much simpler fashion. For the language under construction here, the control dependences reflect a program's nesting structure; program dependence graph G_P contains a *control dependence edge* from vertex v_1 to vertex v_2 of G_P iff one of the following holds:

(1) v_1 is the entry vertex and v_2 represents a component of P that is not nested within any loop or conditional; these edges are labeled **true.**
(2) v_1 represents a control predicate and v_2 represents a component of P immediately nested within the loop or conditional whose predicate is represented by v_1. If v_1 is the predicate of a while-loop, the edge $v_1 \rightarrow_c v_2$ is labeled **true**; if v_1 is the predicate of a conditional statement, the edge $v_1 \rightarrow_c v_2$ is labeled **true** or **false** according to whether v_2 occurs in the **then** branch or the **else** branch, respectively.[4]

[3] 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.

[4] In other definitions that have been given for control dependence edges, there is an additional edge from each predicate of a **while** statement to itself, labeled **true**. This kind of edge is left out of our definition because it is not necessary for our purposes.

A data dependence 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 dependence edges, representing *flow dependences* and *def-order dependences*.[5] The data dependence 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-directed manner.

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

(1) v_1 is a vertex that defines variable x.

(2) v_2 is a vertex that uses x.

(3) 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; final uses of variables are considered to occur at the end of the control-flow graph.)

A flow dependence that exists from vertex v_1 to vertex v_2 is denoted by $v_1 \longrightarrow_f v_2$.

Flow dependences can be further classified as *loop carried* or *loop independent.* A flow dependence $v_1 \longrightarrow_f v_2$ is carried by loop L, denoted by $v_1 \longrightarrow_{lc(L)} v_2$, if in addition to (1), (2), and (3) above, the following also hold:

(4) There is an execution path that both satisfies the conditions of (3) above and includes a backedge to the predicate of loop L.

(5) Both v_1 and v_2 are enclosed in loop L.

A flow dependence $v_1 \longrightarrow_f v_2$ is loop-independent, denoted by $v_1 \longrightarrow_{li} v_2$, if in addition to (1), (2), and (3) above, there is an execution path that satisfies (3) 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 dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

(1) v_1 and v_2 both define the same variable.

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

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

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

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

Note that a program dependence graph is a multigraph (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 dependence edge between two vertices, each is labeled by a

[5] For a complete discussion of the need for these edges and a comparison of def-order dependences with anti- and output dependences see [9].

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

Edge Key

- control dependence
- loop-independent flow dependence
- loop-carried flow dependence
- def-order dependence

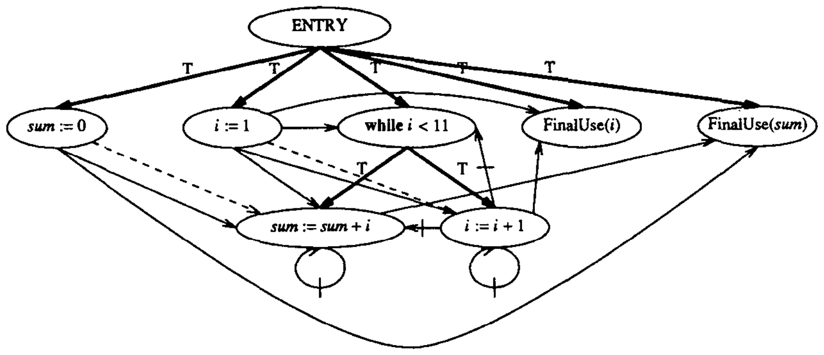


Fig. 1. An example program, which sums the integers from 1 to 10 and leaves the result in the variable *sum*, and its program dependence graph. The boldface arrows represent control dependence edges, solid arrows represent loop-independent flow dependence edges, solid arrows with a hash mark represent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

different loop that carries the dependence. 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 dependence edges; solid arrows represent loop-independent flow dependence edges; solid arrows with a hash mark represent loop-carried flow dependence edges; dashed arrows represent def-order dependence edges.

2.2 Program Slices (of Single-Procedure Programs)

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

```
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 $WorkList \neq \emptyset$ do
    Select and remove vertex $v$ from $WorkList$
    Mark $v$
    for each unmarked vertex $w$ such that edge $w \longrightarrow_f v$ or edge $w \longrightarrow_c v$ is in $E(G)$ do
      Insert $w$ into $WorkList$
    od
  od
end
```

Fig. 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.

v and w. In terms of the three types of edges in a program dependence graph, we define

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

The relationship between a program's dependence graph and a slice of the graph has been addressed in [20]. 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, 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 $V(G/S)$ [20].

Example. Figure 3 shows the graph that results from taking a slice of the program dependence graph from Figure 1 with respect to the final-use vertex for i, together with the one program to which it corresponds.

The significance of an intraprocedural slice is that it captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice [20]. In our case, a program point may be (1) an assignment statement, (2) a control predicate, or (3) a final use of a variable in an end statement. Because a statement or control predicate may be reached repeatedly in a program by "computing the same sequence of values for each element of the slice," we mean: (1) for any assignment statement the same *sequence* of values are assigned to the target variable; (2) for the predicate the same *sequence* of Boolean values are produced; and (3) for each final use the same value for the variable is produced.

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

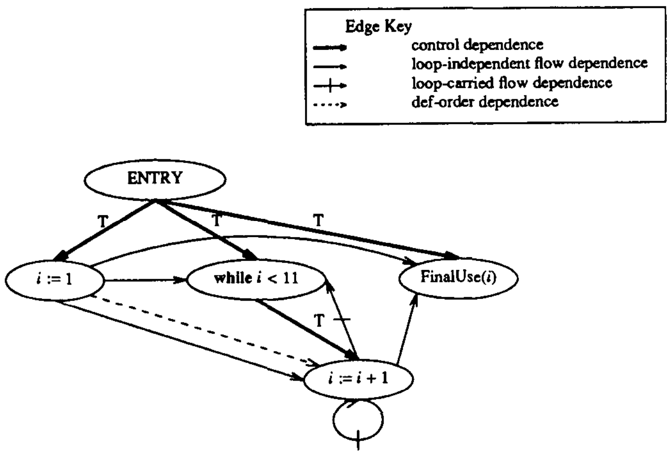


Fig. 3. The graph and the corresponding program that result from slicing the program dependence graph from Figure 1 with respect to the final-use vertex for *i*.

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.

We make the further assumption that there are no call sites of the form $P(x, x)$ or of the form $P(g)$, where g is a global variable. The former restriction sidesteps potential copy-back conflicts. The latter restriction permits global variables to be treated as additional parameters to each procedure; thus, we do not discuss global variables explicitly in this paper.

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. Section 4.3 discusses how to deal with systems that use call-by-reference parameter passing and contain aliasing.

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 dependences between a call site and the called procedure, and (2) edges that represent transitive dependences due to calls.

Section 3.1 discusses how procedure calls and procedure entry are represented in procedure dependence graphs and how edges representing dependences 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 dependences 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 Parameter Passing

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

(1) It should be possible to build an individual procedure's procedure dependence graph (including the computation of data dependences) 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 a precise interprocedural slice efficiently 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 slightly nonstandard, two-stage mechanism for runtime parameter passing: when procedure P calls procedure Q, values are transferred from P to Q by means of intermediate temporary variables, one for each parameter. A different set of temporary variables is used when Q returns to transfer values back to P. Before the call, P copies the values of the actual parameters into the call temporaries; Q then initializes local variables from these temporaries. Before returning, Q copies return values into the return temporaries, from which P retrieves them.

This model of parameter passing 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 *parameter* vertices. On the calling side, information transfer is represented by a set of vertices called *actual-in* and *actual-out* vertices. These vertices, which are control dependent on the call-site vertex, represent assignment statements that copy the values of the actual parameters to the call temporaries and from the return temporaries, respectively. Similarly, information transfer in the called procedure is represented by a set of vertices called *formal-in* and *formal-out* vertices. These vertices, which are control dependent on the procedure's entry vertex, represent assignment statements that copy the values of the formal parameters from the call temporaries and to the return temporaries, respectively.

Using this model, data dependences between procedures are limited to dependences from actual-in vertices to formal-in vertices and from formal-out vertices to actual-out 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 *parameter-in* edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; (3) a *parameter-out* edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site. (Call edges are a new kind of control dependence edge; parameter-in and parameter-out edges are new kinds of data dependence edges.)

Another advantage of this model is that flow dependences can be computed in the usual way, using data-flow analysis on the procedure's control-flow graph. The control-flow graph for a procedure includes nodes analogous to the actual-in, actual-out, formal-in and formal-out vertices of the procedure dependence graph. A procedure's control-flow graph starts with a sequence of assignments that copy values from call temporaries to formal parameters and ends with a sequence of assignments that copy values from formal parameters to return temporaries. Each call statement within the procedure is represented in the procedure's control-flow graph by a sequence of assignments that copy values from actual parameters to call temporaries, followed by a sequence of assignments that copy values from return temporaries to actual parameters.

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 parameter vertices from procedure dependence graphs. For now, we assume that all actual parameters are copied into the call temporaries and retrieved from the return temporaries. Thus, the parameter 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 an actual-in vertex for each actual parameter e of the call to Q. The actual-in vertices are labeled $r\_in := e$, where r is the formal parameter name.
>
> For each actual parameter a that is a variable (rather than an expression), there is an actual-out vertex. These are labeled $a := r\_out$ for actual parameter a and corresponding formal parameter r.

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

> For each formal parameter r of Q, G_Q contains a formal-in vertex and a formal-out vertex. These vertices are labeled $r := r\_in$ and $r\_out := r$, respectively.

Example. Figure 4 repeats the example system from the Introduction and shows the corresponding program and procedure dependence graphs connected with parameter-in edges, parameter-out edges, and call edges.

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

```
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
```

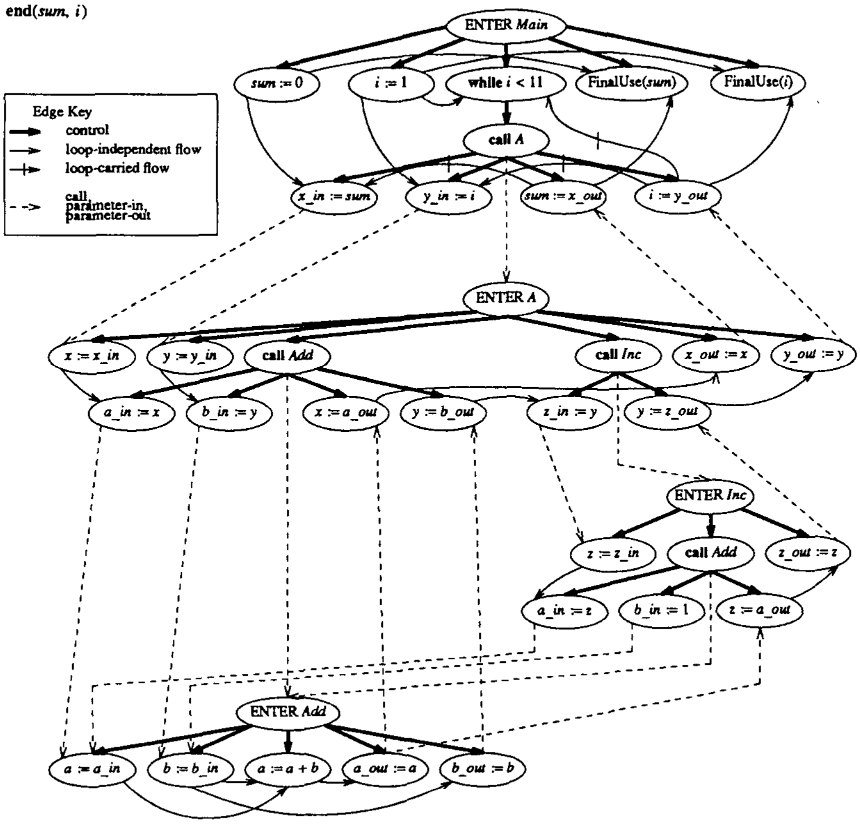


Fig. 4. Example system and corresponding program and procedure dependence graphs connected with parameter-in, parameter-out, and call edges. Edges representing control dependences are shown (unlabeled) in boldface; edges representing intraprocedural flow dependences are shown using arcs; parameter-in edges, parameter-out edges, and call edges are shown using dashed lines.

(In Figure 4, as well as in the remaining figures of the paper, def-order edges are not shown. Edges representing control dependences are shown unlabeled; all such edges in this example would be labeled **true**.)

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

Using the graph structure defined in the previous section, interprocedural slicing could be defined as a graph-reachability problem, and the slices obtained would be the same as those obtained using Weiser's slicing method. As explained in the Introduction, Weiser's method does not produce as precise a slice as possible because it fails to account for the calling context of a called procedure.

Example. The problem with Weiser's method can be illustrated using the graph shown in Figure 4. In the graph-reachability vocabulary, the problem is that there is a path from the vertex of procedure *Main* labeled "*x_in := sum*" to the vertex of *Main* labeled "*i := y_out*", even though the value of *i* after the call to procedure *A* is independent of the value of *sum* before the call. The path is as follows:

$$\begin{array}{llll}
Main\text{: “}x\_in := sum\text{”} & \rightarrow A\text{: “}x := x\_in\text{”} & \rightarrow A\text{: “}a\_in := x\text{”} & \rightarrow Add\text{: “}a := a\_in\text{”} \\
 & \rightarrow Add\text{: “}a := a + b\text{”} & \rightarrow Add\text{: “}a\_out := a\text{”} & \rightarrow Inc\text{: “}z := a\_out\text{”} \\
 & \rightarrow Inc\text{: “}z\_out := z\text{”} & \rightarrow A\text{: “}y := z\_out\text{”} & \rightarrow A\text{: “}y\_out := y\text{”} \\
 & \rightarrow Main\text{: “}i := y\_out\text{”} & &
\end{array}$$

The source of this problem is that not all paths in the graph correspond to possible execution paths (e.g., the path from vertex "*x_in := sum*" of *Main* to vertex "*i := y_out*" of *Main* corresponds to procedure *Add* being called by procedure *A*, but returning to procedure *Increment*).

To overcome this problem, we add an additional kind of edge to the system dependence graph to represent transitive dependences due to the effects of procedure calls. The presence of transitive-dependence edges permits interprocedural slices to be computed in two passes, each of which is cast as a reachability problem. Thus, the next step in the construction of the system dependence graph is to determine such transitive dependences. For example, for the graph shown in Figure 4, we need an algorithm that can discover the transitive dependence from vertex "*x_in := sum*" of *Main* to vertex "*sum := x_out*" of *Main*. This dependence exists because the value of *sum* after the call to *A* depends on the value of *sum* before the call to *A*.

One's first impulse might be to compute transitive dependences due to calls by taking the transitive closure of the graph's control, flow, parameter, and call edges. However, this technique is imprecise for the same reason that transitive closure (or, equivalently, reachability) is imprecise for interprocedural slicing, namely that not all paths in the system dependence graph correspond to possible execution paths. Using transitive closure to compute the dependence edges that represent the effects of procedure calls would put in a (spurious) edge from vertex "*x_in := sum*" of *Main* to vertex "*i := y_out*" of *Main*.

For a language without recursion, this problem could be eliminated by using a separate copy of a procedure dependence graph for each call site; however, to handle a language *with* recursion, a more powerful technique is required. The technique we use involves defining an attribute grammar, called the *linkage grammar*, to model the call structure of each procedure as well as the in*tra*procedural transitive flow dependences among the procedure's parameter vertices. In*ter*procedural transitive flow dependences among a system dependence graph's parameter vertices are determined from the linkage grammar using a standard attribute-grammar construction: the computation of the *subordinate characteristic graphs* of the linkage grammar's nonterminals.[6]

In this section we describe the construction of the linkage grammar and the computation of its subordinate characteristic graphs. It should be understood that the linkage grammar is used *only* to compute transitive dependences due to

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

calls; we are not interested in the language defined by the grammar, nor in actual attribute values.

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:

$$Main \rightarrow A \qquad A \rightarrow Add\ Increment \quad Add \rightarrow \epsilon \qquad Increment \rightarrow Add$$

The attributes in the linkage grammar correspond to the parameters of the procedures. Procedure inputs are modeled as inherited attributes, procedure outputs as synthesized attributes. 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:

(1) For each procedure P, the linkage grammar contains a nonterminal P.
(2) 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.
(3) For each actual-in vertex of P, there is an inherited attribute of nonterminal P.
(4) For each actual-out vertex of P, there is a synthesized attribute of nonterminal P.

Attribute a of nonterminal X is denoted by "$X.a$".

Dependences among the attributes of a linkage-grammar production are used to model the (possibly transitive) intraprocedural dependences among the parameter vertices of the corresponding procedure. These dependences are computed using (intraprocedural) slices of the procedure's procedure dependence graph as described in Section 2.2. For each grammar production, attribute equations are introduced to represent the intraprocedural dependences among the parameter 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 parameter vertices identified by the slice. More formally:

For each attribute occurrence of $X.a$ 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 $X.a = f(\ldots, Y.b, \ldots)$ where the arguments $Y.b$ to the equation consist of the attribute occurrences of p that correspond to the parameter vertices in G_P/v.

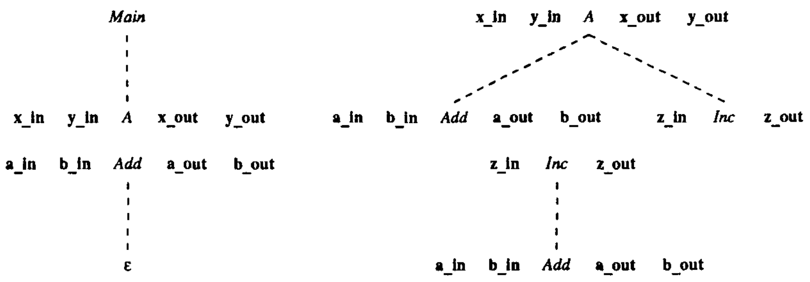

Main
x_in y_in A x_out y_out
a_in b_in Add a_out b_out
ε
x_in y_in A x_out y_out
a_in b_in Add a_out b_out z_in Inc z_out
z_in Inc z_out
a_in b_in Add a_out b_out

Fig. 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.

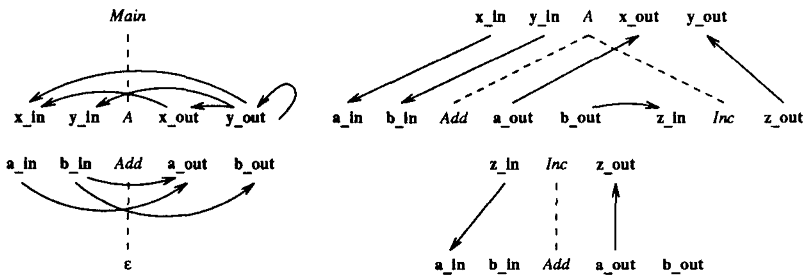

Main
x_in y_in A x_out y_out
a_in b_in Add a_out b_out
ε
x_in y_in A x_out y_out
a_in b_in Add a_out b_out z_in Inc z_out
z_in Inc z_out
a_in b_in Add a_out b_out

Fig. 6. The productions of Figure 5, augmented with attribute dependences.

Note that the actual function f on the right-hand side of the equation is completely irrelevant because the attribute grammar is *never* used for evaluation; all we need is that the equation induce the dependences described above.

Example. Figure 6 shows the productions of the grammar from Figure 5, augmented with attribute dependences.

The dependences for production $Main \rightarrow A$, for instance, correspond to the attribute-definition equations

$$
\begin{aligned}
&A.x\_in = f1(A.x\_out, A.y\_out)\\
&A.y\_in = f2(A.y\_out)\\
&A.x\_out = f3(A.y\_out)\\
&A.y\_out = f4(A.y\_out)
\end{aligned}
$$

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); additionally, the grammar may not be well formed (e.g., a production may have equations for synthesized attribute occurrences of right-hand side symbols). This does not create any difficulties as the linkage grammar is used *only* to compute transitive dependences and *not* for attribute evaluation.

Example. The equation $A.y\_out = f4(A.y\_out)$ makes the example attribute grammar both circular and not well formed. This equation is added to the attribute grammar because of the following (cyclic) path in the graph shown in Figure 4:

$$\textit{Main}\text{: "}i := y\_out\text{"} \longrightarrow \textit{Main}\text{: "}\textbf{while } i < 11\text{"} \longrightarrow \textit{Main}\text{: "}\textbf{call } A\text{"} \longrightarrow \textit{Main}\text{: "}i := y\_out\text{"}$$

Transitive dependences from a call site's actual-in vertices to its actual-out vertices are computed from the linkage grammar by constructing the subordinate characteristic graphs for the grammar's 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, called ConstructSubCGraphs, which is a slight modification of an algorithm originally developed by Kastens to construct approximations to a grammar's transitive dependence relations [13]. The covering subordinate characteristic graph of a nonterminal X of the linkage grammar is captured in the graph TDS(X) (standing for "Transitive Dependences 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 Dependences in a Production"), which expresses dependences 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 graphs can be *marked* or *unmarked*; the edges that AddEdgeAndInduce adds to the TDS graphs are unmarked.

```
procedure ConstructSubCGraphs($L$)
declare
    $L$: a linkage grammar
    $p$: a production in $L$
    $X_i$, $X_j$, $\hat{X}$: 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 the equation that defines $X_j.b$ 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 $(X.a, X.b) \notin TDS(X)$ 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 dependences */
        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 $\hat{X}$ of $X$ in any production $p$ do
                if $(\hat{X}.a, \hat{X}.b) \notin TDP(p)$ then AddEdgeAndInduce($TDP(p)$, $(\hat{X}.a, \hat{X}.b)$) fi
            od
        od
end
```

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

The TDS graphs are generated by the procedure ConstructSubCGraphs, given in Figure 7, which is a slight modification of the first two steps of Kasten's algorithm for constructing a set of evaluation plans for an attribute grammar [13].

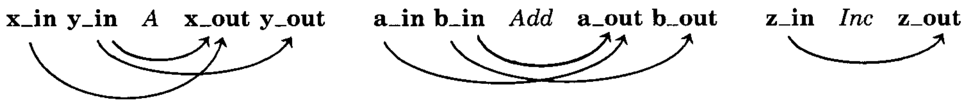
ConstructSubCGraphs performs a kind of closure operation on the TDP and TDS graphs. Step 1 of the algorithm—the first two for-loops of Construct-SubCGraphs—initializes the grammar's TDP and TDS graphs; when these loops terminate, the TDP graphs contain edges representing all direct dependences that exist between the grammar's attribute occurrences, and the TDS graphs contain unmarked edges corresponding to direct left-hand-side-to-left-hand-side dependences in the linkage grammar's productions. Our construction of attribute equations for the linkage grammar ensures that the graph of direct attribute dependences is transitively closed; thus, at the end of Step 1, TDP(p) is a transitively closed graph. 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 dependences have been induced in the TDP and TDS graphs. Thus, the TDS(X) graphs computed by ConstructSubCGraphs are guaranteed to cover the transitive dependences among the attributes of X that exist at any 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 dependences 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 actual-in vertex v at a call site, introduce a parameter-in edge from v to the corresponding formal-in vertex in the called procedure.
(4) For each actual-out vertex v at a call site, introduce a parameter-out edge to v from the corresponding formal-out 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 dependence 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.

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 [22], 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 dependences from actual-in vertices to actual-out vertices due to

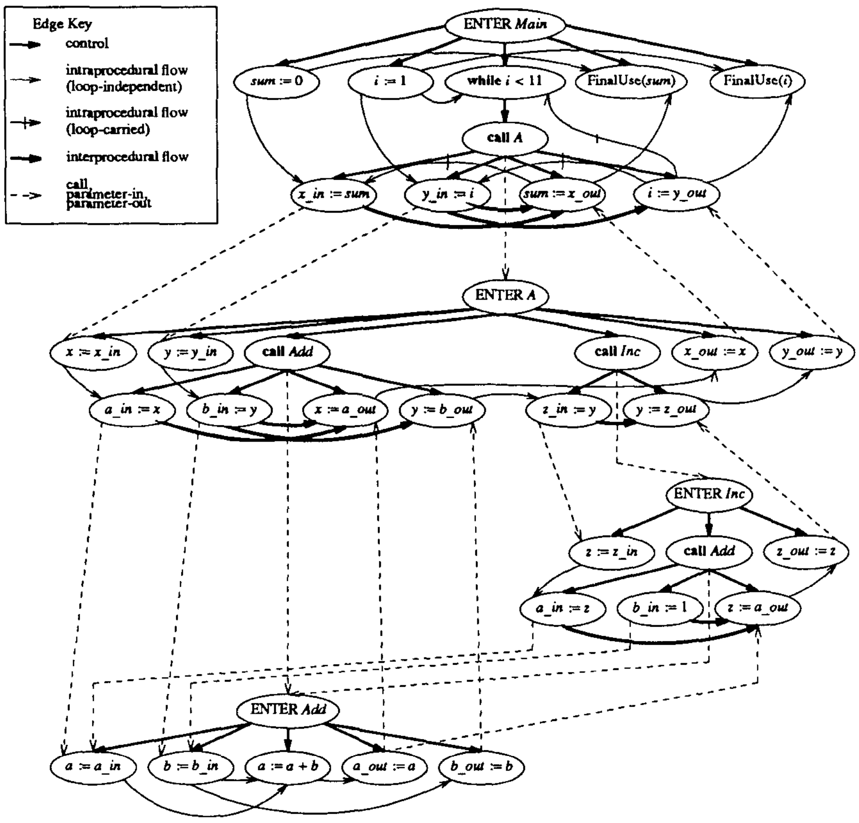


Fig. 8. Example system's system dependence graph. Control dependences, shown unlabeled, are represented using medium-bold arrows; intraprocedural flow dependences are represented using arcs; transitive interprocedural flow dependences (corresponding to subordinate characteristic graph edges) are represented using heavy, bold arcs; call edges, parameter-in edges, and parameter-out edges (which connect program and procedure dependence graphs together) are represented using dashed arrows.

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 performing 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 parameter-in edges, but does *not* follow def-order edges or parameter-out edges. The traversal in Phase 2 that follows flow edges, control edges, and parameter-out edges, but does *not* follow def-order edges, call edges, or parameter-in edges.

```
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, parameter-out})
  /* Phase 2: Slice called procedures without ascending to call sites */
    S′ := all marked vertices in G
    MarkReachingVertices(G, S′, {def-order, parameter-in, 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 G
  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 unmarked vertex w such that there is an edge w → v whose kind is not in Kinds do
      Insert w into WorkList
    od
  od
end
```

Fig. 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*.

Suppose the goal is to slice system dependence graph G with respect to some 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 are either in P itself or in a procedure that calls P (either directly or transitively). Because parameter-out edges are not followed, the traversal in Phase 1 does not "descend" into procedures called by P. The effects of such procedures are not ignored, however; the presence of *transitive flow dependence edges* from actual-in to actual-out 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 (transitively) called by P or from procedures called by procedures that (transitively) call P. Because call edges and parameter-in edges are not followed, the traversal in Phase 2 does not "ascend" into calling procedures; the transitive flow dependence edges from actual-in to actual-out vertices make such "ascents" unnecessary.

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 formal-out vertex for parameter z in

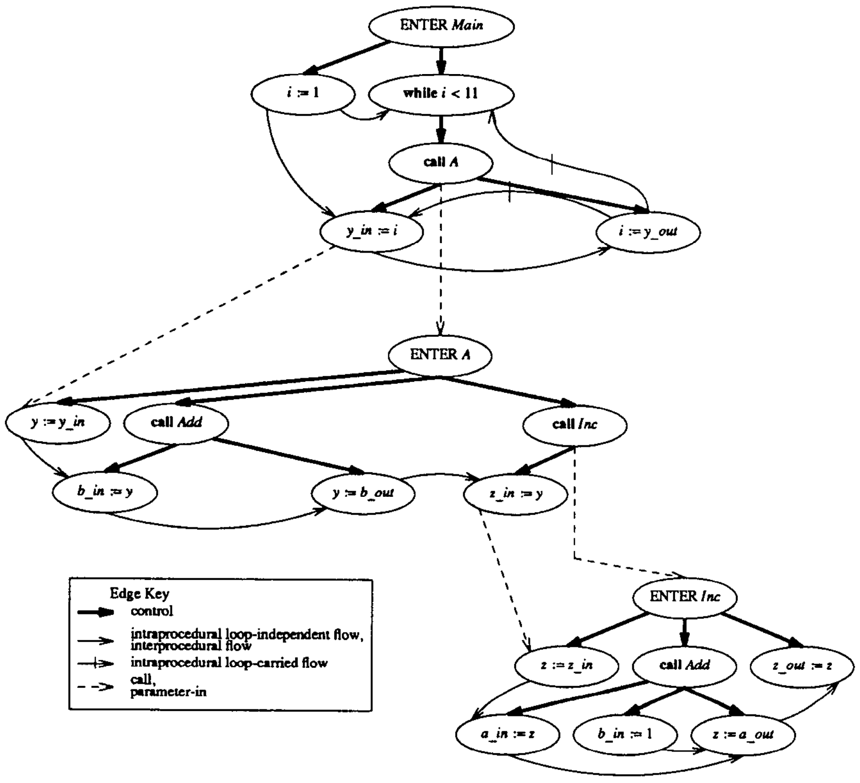


Fig. 10. The example program's system dependence graph is sliced with respect to the formal-out vertex for parameter z in procedure *Increment*. The vertices marked by Phase 1 of the slicing algorithm as well as the edges traversed during this phase are shown above.

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.

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 (excluding def-order edges.)

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 an actual-in and an actual-out 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 an actual-out vertex for argument y in the call to *Add* from A is

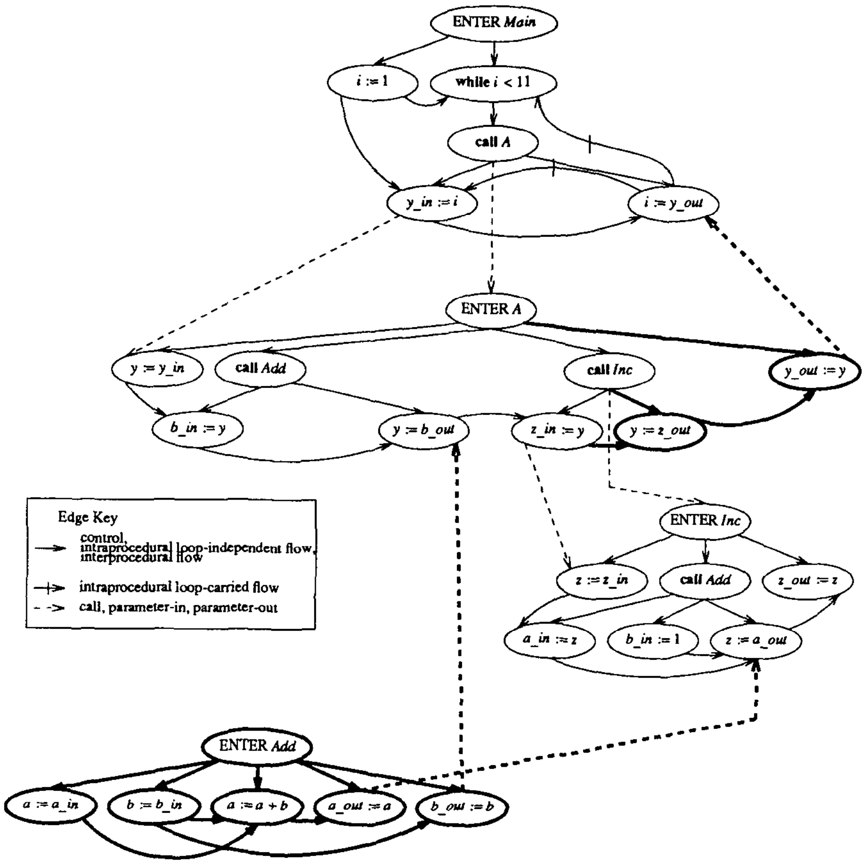


Fig. 11. The example program's system dependence graph is sliced with respect to the formal-out vertex for parameter z in procedure *Increment*. The vertices marked by Phase 2 of the slicing algorithm as well as the edges traversed during this phase are shown above in boldface.

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 use the results of interprocedural data-flow analysis when constructing procedure dependence graphs, in order to exclude vertices like the actual-out vertex for argument y.

The appropriate interprocedural summary information consists of the following sets, which are computed for each procedure P [4]:

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 parameter vertices are included in procedure dependence graphs as follows: for each procedure P, the parameter vertices subordinate to P's entry vertex include one formal-in vertex

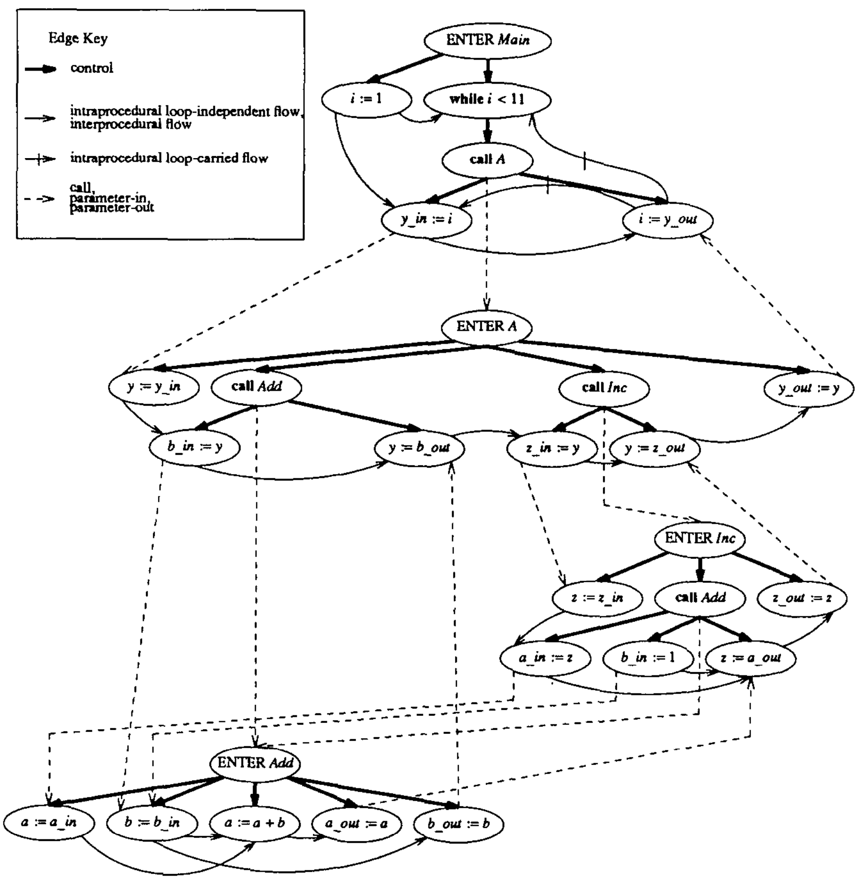


Fig. 12. The complete slice (excluding def-order edges) of the example program's system dependence graph sliced with respect to the formal-out vertex for parameter z in procedure *Increment*.

for each variable in GMOD(P) ∪ GREF(P) and one formal-out vertex for each variable in GMOD(P). Similarly, for each site at which P is called, the parameter vertices subordinate to the call-site vertex include one actual-in vertex for each variable in GMOD(P) ∪ GREF(P) and one actual-out vertex for each variable in GMOD(P). (It is necessary to include an actual-in and a formal-in 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 a formal-in vertex for x in P and a corresponding actual-in 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

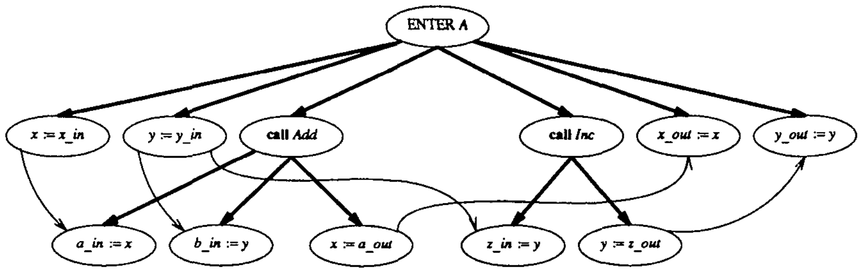


Fig. 13. Procedure *A*'s procedure dependence graph built using interprocedural summary information. The actual-out vertex for argument *y* of the call to *Add* has been omitted, and the flow edge from that vertex to the vertex "*z_in* := *y*" has been replaced by an edge from the vertex "*y* := *y_in*" to the vertex "*z_in* := *y*".

Because parameter *b* is not in GMOD(*Add*), *Add*'s procedure dependence graph should not include a formal-out vertex for *b*, and the call to *Add* from *A* should not include the corresponding actual-out vertex.

Figure 13 shows *A*'s procedure dependence graph as it would be built using GMOD and GREF information.

The actual-out vertex for argument *y* of the call to *Add* is omitted, and the flow edge from that vertex to the actual-in vertex "*z_in* := *y*" is replaced by an edge from the formal-in vertex "*y* := *y_in*" to the actual-in vertex "*z_in* := *y*". The new edge is traversed during Phase 1 of the interprocedural slice instead of the (now omitted) flow edge from "*y* := *a_out*" to "*z_in* := *y*", thus (correctly) bypassing the call to *Add* in procedure *A*.

4.3 Interprocedural Slicing in the Presence of Call-By-Reference Parameter Passing and Aliasing

Our definitions of system dependence graphs and interprocedural slicing have assumed that parameters are passed by value-result. The same definitions hold for call-by-reference parameter passing in the absence of aliasing; however, in the presence of aliasing, some modifications are required. This section presents two approaches for dealing with systems that use call-by-reference parameter passing and contain aliasing. The first approach provides a more precise slice than the second, at the expense of the time and space needed to convert the original system into one that is alias-free. (These costs may, in the worst case, be exponential in the maximum number of parameters passed to a procedure.) The second approach avoids this expense by making use of a generalized notion of flow dependence that includes flow dependences that exist under the possible aliasing patterns.

Our first approach to the problem of interprocedural slicing in the presence of aliasing is to reduce the problem to that of interprocedural slicing in the *absence* of aliasing. The conversion is performed by simulating the calling behavior of the system (using the usual activation-tree model of procedure calls [4]) to discover, for each instance of a procedure call, exactly how variables are aliased at that instance. (Although a recursive system'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 the vertex v, with respect to which we are to perform a slice. If this happens, it is necessary to slice the transformed system with respect to *all* occurrences of v. The slice of the original system is obtained from the slice of the transformed system by projecting elements in the slice of the transformed system back into the original system; a vertex is in the slice of the original system if any of its copies are in the slice of the transformed system.

Example. Figure 14 shows a system with aliasing, and the portion of the system's activation tree that is used to compute alias information for each call instance.

We use the notation of [4], in which each node of the activation tree is labeled with the mapping from variable names to memory locations. The transformed, alias-free version of the system is shown below.

```
program Main         procedure P1(x, y)     procedure P2(xy)
  a := 1;              if y = 0 then          if xy = 0 then
  b := 0;                call P2(x)             call P2(xy)
  call P1(a, b);       fi;                    fi;
  z := b               y := y + 1             xy := xy + 1
end                  return                 return
```

If our original goal had been to slice with respect to the statement "$y := y + 1$" in procedure P, we must now slice with respect to the set of statements {"$y := y + 1$", "$xy := xy + 1$"}.

Our second approach to the problem of interprocedural slicing in the presence of aliasing is to generalize the definition of a flow dependence to include dependences that arise under the possible aliasing patterns. A procedure dependence graph has a flow dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

(1) v_1 is a vertex that defines variable x.
(2) v_2 is a vertex that uses variable y.
(3) x and y are potential aliases.
(4) Control can reach v_2 after v_1 via a path in the control-flow graph along which there is no intervening definition of x or y.

Note that clause (4) does not exclude there being definitions of other variables that are potential aliases of x or y along the path from v_1 to v_2. An assignment to a variable z along the path from v_1 to v_2 only overwrites the contents of the memory location written by v_1 if x and z refer to the same memory location. If z is a potential alias of x, then there is only a *possibility* that x and z refer to the same memory location; thus, an assignment to z does not necessarily overwrite the memory location written by v_1, and it may be possible for v_2 to read a value written by v_1.

```
program Main            procedure P(x, y)
  a := 1;                 if y = 0 then
  b := 0;                   call P (x, x)
  call P (a, b);          fi;
  z := b                  y := y + 1
end                     return
```

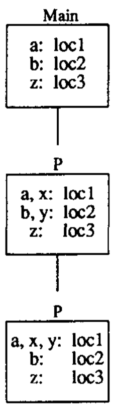


Fig. 14. A program with aliasing and the portion of its activation tree needed to compute all alias configurations.

The notion of a def-order edge must also be generalized in the presence of aliasing. A procedure dependence graph has a def-order dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

(1) v_1 and v_2 define variables x_1 and x_2, respectively.
(2) x_1 and x_2 are potential aliases.
(3) v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.
(4) There exists a program component v_3 such that $v_1 \longrightarrow_f v_3$ and $v_2 \longrightarrow_f v_3$.
(5) v_1 occurs to the left of v_2 in the procedure's abstract syntax tree.

The interprocedural slice of a system dependence graph containing dependence edges as defined above is computed by the same two-phase algorithm used to compute the interprocedural slice of a system in the absence of aliasing. The data dependences in a procedure provide a safe approximation to the true dependences required for each alias configuration. Because these edges cover all possible alias configurations, the resulting slice may contain unnecessary program elements.

Example. Consider again the system shown in Figure 14. The possibility of aliasing between formal parameters x and y of procedure P gives rise to flow dependences from the actual-out vertices "$x := x\_out$" and "$x := y\_out$" of the call $P(x, x)$ to the vertex "$y := y + 1$". Because of these dependences, the slice with respect to the statement "$z := b$" in the main program yields the entire system, even though the statement "$a := 1$" in *Main* and the conditional statement in P have no effect on the value computed for z. The approach based on replicating procedures determines a more precise slice that does not include the statement "$a := 1$" or the conditional statement, as shown below:

```
program Main      procedure P1(y)
  b := 0;           y := y + 1
  call P1(b);     return
  z := b
end
```

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, subordinate characteristic graphs in which there is an edge from each inherited attribute to each synthesized attribute, and superior characteristic graphs in which there is an edge from each synthesized attribute to each other attribute (including the other synthesized attributes), 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 the graphs described above.

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 the worst-case graphs, given specifications for the missing program components.

4.5 Forward Slicing

Whereas the *slice* of a program with respect to a 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 *forward slice* of a program with respect to a program point p and variable x consists of all statements and predicates of the program that might be affected by the value of x at point p. An algorithm for *forward* interprocedural slicing can be defined on system dependence graphs, using the same concepts employed for (backward) interprocedural slicing. As before, the key element is the use of the linkage grammar's characteristic graph edges in the system dependence graph to represent transitive dependences from actual-in vertices to actual-out vertices due to the effects of procedure calls.

An algorithm for forward interprocedural slicing is given as procedure MarkVerticesOfForwardSlice of Figure 15.

In Figure 15, the computation of the forward slice of system dependence graph G with respect to vertex set S is performed in two phases. The traversal in Phase 1 follows flow edges, control edges, and parameter-out edges, but does *not* follow call edges, def-order edges, or parameter-in edges. Because call edges

```
procedure MarkVerticesOfForwardSlice(G, S)
declare
  G: a system dependence graph
  S, S′: sets of vertices in G
begin
  /* Phase 1: Slice forward without descending into called procedures */
    MarkVerticesReached(G, S, {def-order, parameter-in, call})
  /* Phase 2: Slice forward into called procedures without ascending to call sites */
    S′ := all marked vertices in G
    MarkVerticesReached(G, S′, {def-order, parameter-out})
end

procedure MarkVerticesReached(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 G
  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 unmarked vertex w such that there is an edge v ⟶ w whose kind is not in Kinds do
      Insert w into WorkList
    od
  od
end
```

Fig. 15. The procedure MarkVerticesOfForwardSlice marks the vertices of the forward interprocedural slice G/S. The auxiliary procedure MarkVerticesReached marks all vertices in G to which there is a path from a vertex in V along edges of kinds other than those in the set *Kinds*.

and parameter-in edges are not followed, the traversal in Phase 1 does not descend into called procedures. The traversal in Phase 2 follows flow edges, control edges, call edges, and parameter-in edges, but does *not* follow def-order edges or parameter-out edges. Because parameter-out edges are not followed, the traversal in Phase 2 does not ascend into calling procedures.

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 cost remains polynomial if we use the generalized definitions of data dependences given in Section 4.3 (at the price of somewhat less precision in taking slices). Alternatively, if we follow the approach of transforming the system to one that is alias-free, more precise slices can be obtained, but the cost can increase by an exponential factor that reflects the blow-up in size that can occur due to the number of aliasing patterns in the program. The measures of system size used below are those associated with the system dependence graph created according to one or the other of these approaches. In particular, if the approach of transforming to an alias-free system is used, 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 larest 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 [6].

Intraprocedural data-flow analysis is used to determine the data dependences 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) [8]. 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
$D \leq R \cdot X = (Sites + 1) \cdot (Global + Params)$	The largest number of attribute occurrences in a single production

To determine the dependences 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 recloses a single TDP(p) relation, is $O(D^2)$. The cost of initializing the TDP relations with all direct dependences in ConstructSubCGraphs is bounded by $O(P \cdot D^2)$.

In the inner loop of Step 2 of procedure ConstructSubCGraphs, AddEdge-AndInduce is called 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 $O(G \cdot X^2 \cdot D^2)$ [13].

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 $O(P \cdot (V + E) + \mathit{TotalSites} \cdot X)$.

6. RELATED WORK

In recasting the interprocedural slicing problem as a reachability problem in a graph, we are following the example of [19], which does the same for intraprocedural slicing. The reachability approach is conceptually simpler than the dataflow equation approach used in [22], 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 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; as explained in the Introduction and in Section 3.2, the imprecision of Weiser's method is due to the lack of a mechanism to keep track of the calling context of a called procedure.

After the initial publication of our interprocedural-slicing algorithm [10], a different technique for computing interprocedural slices was presented by Hwang et al. [12]. The slicing algorithm presented in [12] computes an answer that is as precise as our algorithm, but differs significantly in how it handles the calling-context problem. The algorithm from [12] constructs a *sequence* of slices of the system—where each slice of the sequence essentially permits there to be one additional level of recursion—until a fixed-point is reached (i.e., until no further elements appear in a slice that uses one additional level of recursion). Thus, each slice of the sequence represents an approximation to the final answer. During each of these slice approximations, the algorithm uses a stack to keep track of the calling context of a called procedure. In contrast, our algorithm for interprocedural slicing is based on a two-phase process for propagating marks on the system dependence graph. In Phase 1 of the algorithm, the presence of the linkage grammar's subordinate-characteristic-graph edges (representing transitive dependences due to the effects of 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.

Hwang et al. do not include an analysis of their algorithm's complexity in [12], which makes a direct comparison with our algorithm difficult; however, there are several reasons why our algorithm may be more efficient. First, the algorithm from [12] computes a sequence of slices, each of which may involve reslicing a procedure multiple times; in contrast, through its use of marks on system-dependence-graph vertices, our algorithm processes no vertex more than once during the computation of a slice. Second, if one wishes to compute multiple slices of the same system, our approach has a significant advantage. The system dependence graph (with its subordinate-characteristic-graph edges) need be computed only once; each slicing operation can use this graph, and the cost of each such slice is linear in the size of the system dependence graph. In contrast, the approach of [12] would involve finding a new fixed point (a problem that appears to have complexity comparable to the computation of the subordinate characteristic graphs) for each new slice.

In [18], Myers 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 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.

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. As in the comparison with [12], our algorithm has a significant advantage when one wishes to compute multiple slices of the same system. Whereas the system dependence graph can be computed once and then used for each slicing operation, the approach postulated above 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 [5], [6], and [15] 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 the interprocedural slicing problem and the problems addressed by the work cited above, is that interprocedural slicing is a "demand problem" [2] 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: ATTRIBUTE GRAMMARS AND ATTRIBUTE DEPENDENCES

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 [14]. In every production p: $X_0 \rightarrow X_1, \ldots, X_k$, 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 are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes.

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. (The grammars that arise in this paper are potentially *not* well formed, in that a production may have equations for synthesized attribute occurrences of right-hand side symbols. The reason that this does not cause problems is that the "linkage grammar" of the interprocedural slicing algorithm is used *only* to compute transitive dependences due to calls; we are not interested in the language defined by the grammar, nor in actual attribute values.)

A derivation tree node that is an instance of symbol X has an associated set of *attribute instances* corresponding to the attributes of X. 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 dependences among attribute occurrences in a production p (or attribute instances in a tree T) can be represented by a directed graph, called a *dependence graph*, denoted by $D(p)$ (respectively, $D(T)$), and defined as follows:

(1) For each attribute occurrence (instance) b, the graph contains a vertex b'.
(2) If attribute occurrence (instance) b appears on the right-hand side of the attribute equation that defines attribute occurrence (instance) c, the graph contains the edge $b' \rightarrow c'$.

An attribute grammar that has a derivation tree whose dependence graph contains a cycle is called a *circular* attribute grammar. (The grammars that arise in this paper can be circular grammars.)

A node's *subordinate* and *superior characteristic graphs* provide a convenient representation of transitive dependences among the node's attributes. (A *transitive dependence* exists between attributes that are related in the transitive closure of the tree's attribute dependence relation, or, equivalently, that are connected by a direct path in the tree's dependence 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 dependences among r's attributes.

The subordinate characteristic graph at r is the projection of the dependences 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 dependence graph of the remaining tree onto the attributes of r. To define the characteristic graphs precisely, we make the following definitions:

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

(2) 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 $[v = v_1, v_2, \ldots, v_k = w]$ in G such that $v_2, \ldots, v_{k-1} \notin V'\}$. (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 $r.\bar{C}$, 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:

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

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

REFERENCES

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1986.
2. BABICH, W. A., AND JAZAYERI, M. The method of attributes for data flow analysis: Part II. Demand analysis. *Acta Inf. 10*, 3 (Oct. 1978), 265–272.
3. BADGER, L., AND WEISER, M. Minimizing communication for synchronizing parallel dataflow programs. In *Proceedings of the 1988 International Conference on Parallel Processing* (St. Charles, IL, Aug. 15–19, 1988). Pennsylvania State University Press, University Park, PA, 1988.
4. BANNING, J. P. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages* (San Antonio, Tex., Jan. 29–31, 1979). ACM, New York, 1979, pp. 29–41.
5. CALLAHAN, D. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation* (Atlanta, Ga., June 22–24, 1988). *ACM SIGPLAN Not. 23*, 7 (July 1988), 47–56.

6. COOPER, K. D., AND KENNEDY, K. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation* (Atlanta, Ga., June 22–24, 1988). *ACM SIGPLAN Not. 23*, 7 (July 1988), 57–66.
7. FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.
8. HORWITZ, S., PRINS, J., AND REPS, T. Integrating non-interfering versions of programs. TR-690, Computer Sciences Dept., Univ. of Wisconsin, Madison, March 1987.
9. HORWITZ, S., PRINS, J., AND REPS, T. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan. 13–15, 1988). ACM, New York, 1988, pp. 146–157.
10. HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation* (Atlanta, Ga., June 22–24, 1988). *ACM SIGPLAN Not. 23*, 7 (July 1988), 35–46.
11. HORWITZ, S., PRINS, J., AND REPS, T. Integrating non-interfering versions of programs. *ACM Trans. Program. Lang. Syst. 11*, 3 (July 1989), 345–387.
12. HWANG, J. C., DU, M. W., AND CHOU, C. R. Finding program slices for recursive procedures. In *Proceedings of the IEEE COMPSAC 88* (Chicago, Oct. 3–7, 1988). IEEE Computer Society, Washington, D.C., 1988.
13. KASTENS, U. Ordered attribute grammars. *Acta Inf. 13*, 3 (1980), 229–256.
14. KNUTH, D. E. Semantics of context-free languages. *Math. Syst. Theor. 2*, 2 (June 1968), 127–145.
15. KOU, L. T. On live-dead analysis for global data flow problems. *J. ACM 24*, 3 (July 1977), 473–483.
16. 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. Comput. C-21*, 12 (Dec. 1972), 1293–1310.
17. LYLE, J., AND WEISER, M. Experiments on slicing-based debugging tools. In *Proceedings of the First Conference on Empirical Studies of Programming* (June 1986).
18. MYERS, E. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 26–28, 1981). ACM, New York, 1981, pp. 219–230.
19. OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pa., April 23–25, 1984). *ACM SIGPLAN Not. 19*, 5 (May 1984), 177–184.
20. REPS, T., AND YANG, W. The semantics of program slicing. TR-777, Computer Sciences Dept., Univ. of Wisconsin, Madison, June 1988.
21. WEISER, M. Reconstructing sequential behavior from parallel behavior projections. *Inf. Process. Lett. 17* (Oct. 1983), 129–135.
22. WEISER, M. Program slicing. *IEEE Trans. Softw. Eng. SE-10*, 4 (July 1984), 352–357.

Received April 1988; revised August 1989; accepted August 1989