3 Current Slicing Algorithms

This chapter describes current slicing algorithms together with their data structures ranging from the original approach where slicing is seen as a data flow problem to the state-of-the-art where slicing is seen as a graph-reachability problem.

Weiser used a control flow graph as an intermediate representation for his slicing algorithm. He computed slices by solving the data flow problem of relevant nodes. He gave algorithms for intraprocedural and interprocedural slicing. However, the interprocedural version did not account for the calling context and therefore produced imprecise slices. Ottenstein et al. [OtO84, FeOW87] recognized that intraprocedural backward slices could be efficiently computed using dependence graphs as intermediate representations by traversing the dependence edges backwards (from target to source). Horwitz et al. [HoRB90] introduced system dependence graphs for interprocedural slicing. They also developed a two-phase algorithm that computes precise interprocedural slices. With the help of summary edges they accounted for the transitive effects of procedure calls without descending into called procedures. They computed these summary edges by a variation on the technique to compute the subordinate characteristic graphs of an attribute grammar's nonterminals [Ka80]. Livadas et al. [LivC94, LivJ95] proposed a simpler method of computing the summary edges. In the following sections we will discuss these algorithms and their data structures in more detail.

3.1 Slicing as a Data Flow Problem

Weiser used a control flow graph as an intermediate representation for his slicing algorithm. Computing a slice from a control flow graph requires computation of the data flow information about the set of *relevant variables* at each node. The sets of relevant variables for the slice taken with respect to node *n* and variables *V* can be computed as follows:

1. Initialize the relevant sets of all nodes to the empty set.
2. Insert all variables of *V* into *relevant(n)*.
3. For *n*'s immediate predecessor *m*, compute *relevant(m)* as:

```
relevant(m) := relevant(n) - def(m)          (* exclude all variables that
                                                are defined at m          *)
if relevant(n) ∩ def(m) # {} then            (* if m defines a variable that
                                                is relevant at n          *)
    relevant(m) := relevant(m) ∪ ref(m)      (* include the variables that
                                                are referenced at m       *)
    include m into the slice
end
```

4. Work backwards in the control flow graph, repeating step 3 for *m*'s immediate predecessors until the entry node is reached or the relevant set is empty.

Table 3.1 shows an example for the computation of the relevant sets (taken from [BiG96]). The slice is computed for the last statement and the variable *a*. The nodes that are finally part of the slice are shown with bold node numbers.

n	Statement	ref(n)	def(n)	relevant(n)
1	b = 1		b	
2	c = 2		c	b
3	d = 3		d	b, c
4	a = d	d	a	b, c
5	d = b + d	b, d	d	b, c
6	b = b + 1	b	b	b, c
7	a = b + c	b, c	a	b, c
8	print a	a		a

Table 3.1 - Source code with relevant sets, slice for <8, {a}>

Step 2: relevant(8) = {a}
Step 3: relevant(7) = relevant(8) - def(7) = {a} - {a} = {}
relevant(7) = relevant(7) ∪ ref(7) = {} ∪ {b, c} = {b, c}
Since node 7 defines a variable relevant at node 8, it is included into the slice.
Step 3: relevant(6) = relevant(7) - def(6) = {b, c} - {b} = {c}
relevant(6) = relevant(6) ∪ ref(6) = {c} ∪ {b} = {b, c}
Since node 6 defines a variable relevant at node 7, it is included into the slice.
Step 3: relevant(5) = relevant(6) - def(5) = {b, c} - {d} = {b, c}
Step 3: relevant(4) = relevant(5) - def(4) = {b, c} - {a} = {b, c}
Step 3: relevant(3) = relevant(4) - def(3) = {b, c} - {d} = {b, c}
Step 3: relevant(2) = relevant(3) - def(2) = {b, c} - {c} = {b}
relevant(2) = relevant(2) ∪ ref(2) = {b} ∪ {} = {b}
Since node 2 defines a variable relevant at node 3, it is included into the slice.
Step 3: relevant(1) = relevant(2) - def(1) = {b} - {b} = {}
relevant(1) = relevant(1) ∪ ref(1) = {} ∪ {} = {}
Since node 1 defines a variable relevant at node 2, it is included into the slice.

For structured programs, a statement can have multiple predecessors. The algorithm outlined above must therefore be extended:

- to compute the control sets for each node,
- to combine the relevant sets at points where the control flow merges (by unioning the relevant sets), and
- to compute the relevant sets iteratively until there are no further changes.

The *control set* associates with each node the set of predicate statements that directly control its execution. Whenever a statement is added to the slice, the members of its control set are included into the slice. New slices are computed with respect to the nodes that have been included due to the control set and the variables referenced at these nodes. All statements of the new slices are considered to be part of the original slice.

The following example (taken from [BiG96]) shows the computation of the relevant sets. The slice is computed for node 11 with respect to variable *a*. Table 3.2 shows the source code of the example and for each statement the sets of referenced and defined variables as well as the control set and the relevant set.

n	Statement	ref(n)	def(n)	control(n)	relevant(n)
1	b = 1		b		
2	c = 2		c		b
3	d = 3		d		b, c
4	a = d	d	a		b, c, d
5	if a then	a			b, c, d
6	d = b + d	b, d	d	5	b, d
7	c = b + d	b, d	c	5	b, d
	else				
8	b = b + 1	b	b	5	b, c
9	d = b + 1	b	d	5	b, c
	endif				b, c
10	a = b + c	b, c	a		b, c
11	print a	a			a

Table 3.2 - Source code with relevant sets, slice for <11, {a}>

Step 2: relevant(11) = {a}
Step 3: relevant(10) = relevant(11) - def(10) = {a} - {a} = {}
relevant(10) = relevant(10) ∪ ref(10) = {} ∪ {b, c} = {b, c}
Since node 10 defines a variable relevant at node 11, it is included into the slice.
Step 3: relevant(9) = relevant(10) - def(9) = {b, c} - {d} = {b, c}
Step 3: relevant(8) = relevant(9) - def(8) = {b, c} - {b} = {c}
relevant(8) = relevant(8) ∪ ref(8) = {c} ∪ {b} = {b, c}
Since node 8 defines a variable relevant at node 9, it is included into the slice.
Since control(8) = 5, node 5 is included into the slice.
The slice for node 5 with respect to ref(5) is computed below.
Step 3: relevant(7) = relevant(10) - def(7) = {b, c} - {c} = {b}
relevant(7) = relevant(7) ∪ ref(7) = {b} ∪ {b, d} = {b, d}
Since node 7 defines a variable relevant at node 10, it is included into the slice.
Since control(7) = 5, node 5 is included into the slice.
The slice for node 5 with respect to ref(5) is computed below.
Step 3: relevant(6) = relevant(7) - def(6) = {b, d} - {d} = {b}
relevant(6) = relevant(6) ∪ ref(6) = {b} ∪ {b, d} = {b, d}
Since node 6 defines a variable relevant at node 7, it is included into the slice.
Step 3: relevant(5) = relevant(6) ∪ relevant(8) = {b, d} ∪ {b, c} = {b, c, d}
Step 3: relevant(4) = relevant(5) - def(4) = {b, c, d} - {a} = {b, c, d}
Step 3: relevant(3) = relevant(4) - def(3) = {b, c, d} - {d} = {b, c}
relevant(3) = relevant(3) ∪ ref(3) = {b, c} ∪ {} = {b, c}
Since node 3 defines a variable relevant at node 4, it is included into the slice.
Step 3: relevant(2) = relevant(3) - def(2) = {b, c} - {c} = {b}
relevant(2) = relevant(2) ∪ ref(2) = {b} ∪ {} = {b}
Since node 2 defines a variable relevant at node 3, it is included into the slice.
Step 3: relevant(1) = relevant(2) - def(1) = {b} - {b} = {}
relevant(1) = relevant(1) ∪ ref(1) = {} ∪ {} = {}
Since node 1 defines a variable relevant at node 2, it is included into the slice.

Table 3.3 shows how the slice is computed for node 5 with respect to variable *a*:

n	Statement	ref(n)	def(n)	control(n)	relevant(n)
1	b = 1		b		
2	c = 2		c		
3	d = 3		d		{}
4	a = d	d	a		{d}
5	if a then	a			{a}
6	d = b + d	b, d	d	5	
7	c = b + d	b, d	c	5	
	else				
8	b = b + 1	b	b	5	
9	d = b + 1	b	d	5	
	endif				
10	a = b + c	b, c	a		
11	print a	a			

Table 3.3 - Source code with relevant sets, slice for <5, {a}>

Step 2: relevant(5) = {a}
Step 3: relevant(4) = relevant(5) - def(4) = {a} - {a} = {}
Since node 4 defines a variable relevant at node 5, it is included into the slice.
relevant(4) = relevant(4) ∪ ref(4) = {} ∪ {d} = {d}
Step 3: relevant(3) = relevant(4) - def(3) = {d} - {d} = {}
Since node 3 defines a variable relevant at node 4, it is included into the slice.
relevant(3) = relevant(3) ∪ ref(3) = {} ∪ {} = {}
Since the relevant set is empty, no more nodes will be included into the slice.

The complete slice contains the nodes 10, 8, 7, 6, 5, 4, 3, 2, and 1.

When the program contains loops, iteration over parts of the control flow graph is necessary until the relevant sets and the slice stabilize. The maximum number of iterations is the same as the number of assignment statements in the loop. Weiser describes a method to derive interprocedural slices, essentially by in-line replacement of each procedure with appropriate substitutions for the parameters. However, his method does not account for the calling context and yields imprecise slices.

A big disadvantage of computing program slices this way is that the relevant sets have to be computed for each slice and that this information cannot be reused for other slices.

3.2 Slicing as a Graph-Reachability Problem

Another approach to compute program slices is to first derive an intermediate representation of the program that models the dependences among the program entities and to compute slices simply by traversing the dependences of this intermediate representation. The big advantage of this approach is that data flow analysis only has to be performed once and that the information can be reused for deriving all kinds of slices, such as forward and backward slices, as well as intraprocedural and interprocedural slices.

3.2.1 Program Dependence Graph

Ferrante et al. [FeOW87] introduced *program dependence graphs* which combine control dependences and data dependences into a common framework. The nodes of a program dependence graph represent statements and predicate expressions of the program. Each node of the graph has references to the nodes that it is control dependent on and to the nodes that define its operands. The set of all dependences induce a partial ordering on the statements and predicates in the program that must be followed in order to preserve the semantics of the original program.

They propose to use program dependence graphs for optimizations. Since both the essential control relationships and the essential data relationships are present in the program dependence graph, a single traversal of these dependences is sufficient to perform many optimizations.

3.2.2 System Dependence Graph

Horwitz et al. [HoRB90] enhance the program dependence graph to facilitate interprocedural slicing. They add vertices for the program entry, for the initial definition of variables and for the final use of variables to the graph. They label control dependences with either *true* or *false*, meaning that when the predicate at the origin of the dependence evaluates to *true* (or *false*), the target node of the dependence labeled *true* (or *false*) will eventually be executed. They classify data dependences as *loop carried* and *loop independent*. However, for slicing purposes the distinction between different kinds of data dependences is not necessary. Fig. 3.1 shows the program dependence graph for a small program.

```
MODULE PDG;

VAR i: INTEGER;

BEGIN
  i := 1;
  WHILE i < 11 DO
    i := i + 1
  END
END PDG.
```

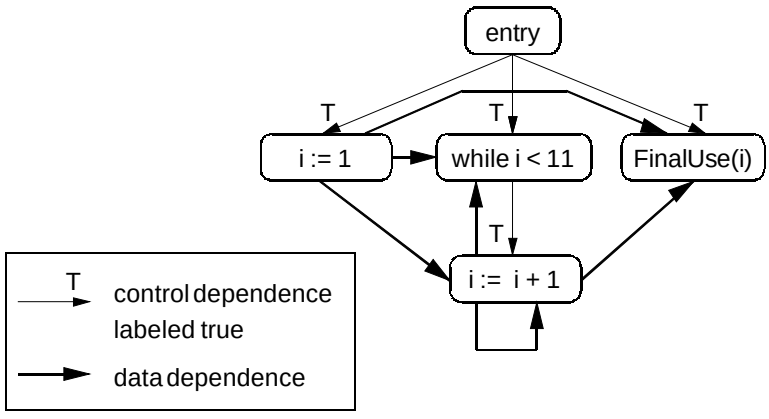

Fig. 3.1 - A small program with its program dependence graph

Using a program dependence graph, program slicing can be seen as a graph reachability problem. The slice of a dependence graph with respect to a node *n* of the graph, is the graph that contains all nodes that (directly or indirectly) reach *n* via a data dependence or control dependence. Horwitz et al. [HoRB90] give a simple worklist algorithm for deriving intraprocedural slices. Fig. 3.2 shows a recursive implementation.

```
PROCEDURE SliceNodeIntraproc (node: Node);
BEGIN
   IF node is not marked THEN
      mark node as visited
      FOR all nodes pred on which node depends DO
         SliceNodeIntraproc(pred)
      END
   END
END SliceNodeIntraproc;
```

Fig. 3.2 - Intraprocedural backward slicing algorithm

For programs that consist of several procedures, Horwitz et al. define the *system dependence graph* that contains one program dependence graph for each procedure of the program. They introduce several nodes to model procedure calls and parameter passing, where parameters are passed by value-result and accesses to global variables are modeled via additional parameters of the procedure:

- *Call-site nodes* represent the call sites.
- *Actual-in* and *actual-out nodes* represent the input and output parameters at the call sites. They are control dependent on the call-site node.
- *Formal-in* and *formal-out nodes* represent the input and output parameters at the called procedure. They are control dependent on the procedure's entry node.

They also introduce additional edges to link the program dependence graphs together:

- *Call edges* link the call-site nodes with the procedure entry nodes.
- *Parameter-in edges* link the actual-in nodes with the formal-in nodes.
- *Parameter-out edges* link the formal-out nodes with the actual-out nodes.

Finally, *summary edges* are used to represent the transitive dependences due to calls. A summary edge is added from an actual-in node *A* to an actual-out node *B*, if there exists a path of control, data and summary edges in the called procedure from the corresponding formal-in node *A'* to the formal-out node *B'*. Fig. 3.3 shows how the summary edges can be used to simulate the effects of a call without descending into the subgraph of the called procedure. The graph contains a summary edge from the actual-in node of *z* at the call site of *Add* to the actual-out node of *z* because there is a path (via data dependences) in the called procedure from the formal-in node of *a* to the formal-out node of *a*. Likewise there is a summary edge from the second actual-in node to the actual-out node of *z*. When computing the slice for the actual-out node of *z*, it suffices to follow the summary edges backwards in order to visit the actual-in nodes on which the value of *z* depends. It is not necessary to descend into the graph of procedure *Add*.

```
PROCEDURE Add (VAR a: INTEGER; b: INTEGER);
BEGIN
    a := a + b
END Add;

call of Add in Increment: Add(z, 1)
```

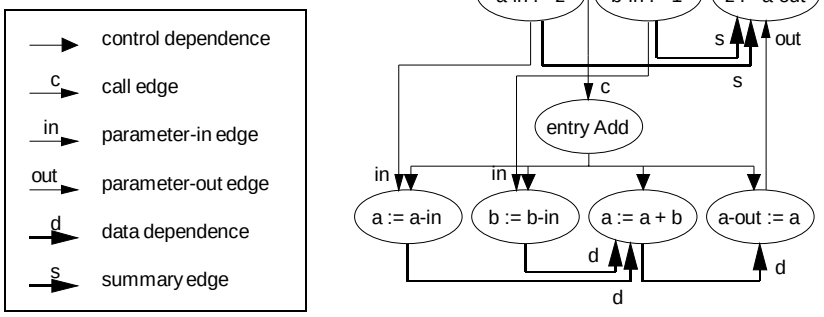

Fig. 3.3 - Summary edges

Summary edges permit movement across call sites without having to descend into the called procedures, while still regarding the effects of the called procedure. It is therefore not necessary to keep track of the calling context explicitly to ensure that only legal execution paths are traversed.

Horwitz et al. used a variation on the technique to compute the subordinate characteristic graphs of an attribute grammar's nonterminals [Ka80] in order to compute these summary edges. Livadas et al. [LivC94, LivJ95] proposed a simpler method of computing the summary edges.

Fig. 3.4 shows the system dependence graph of the program shown in Fig. 2.16. Control dependences are drawn with thin lines and broad arrows, call edges, par-in and par-out edges with thin lines and small arrows, and data dependences as well as summary edges are drawn with thick lines and big arrows.

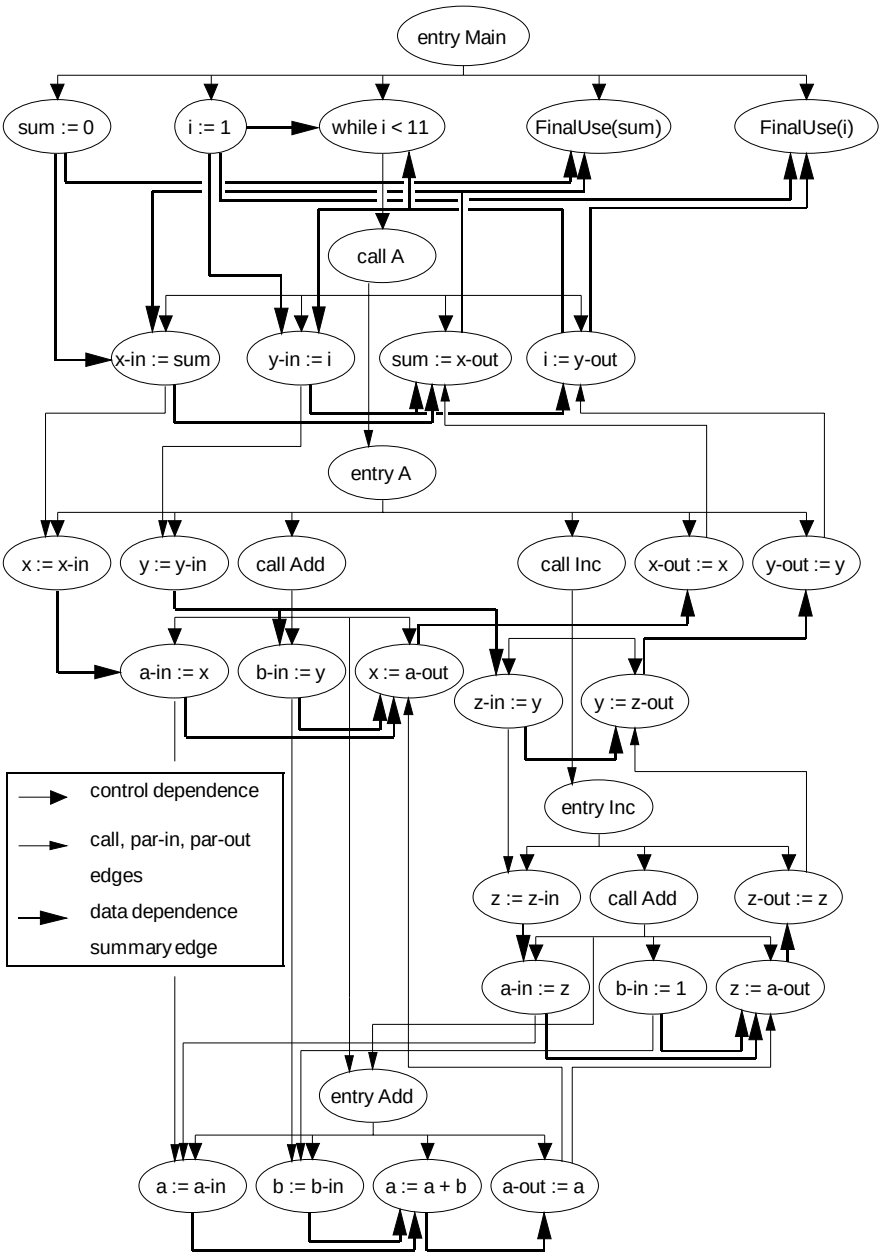

Fig. 3.4 - The system dependence graph for the program shown in Fig. 2.16

Interprocedural slicing can be implemented as a reachability problem over the system dependence graph. The transitive closure over all dependences yields a slice that does not regard the calling context and therefore contains irrelevant nodes. Horwitz et al. [HoRB90] developed a two-phase algorithm that computes precise interprocedural slices. In the following we give a brief outline of this algorithm (the slice shall be computed with respect to node *n* in procedure *P*):

- In the first phase, all edges except parameter-out edges (i.e., control and data dependences, summary, parameter-in and call edges) are followed backwards starting with node *n* in procedure *P*. All nodes are marked, that either reach *n* and are in *P* itself or in procedures that (transitively) call *P*, i.e. the traversal ascends from

procedure *P* upwards to the procedures that called *P*. Since parameter-out edges are not followed, phase 1 does not "descend" into procedures called by *P*. The effects of such procedures are not ignored, however; summary edges from actual-in nodes to actual-out nodes cause nodes to be included into the slice that would only be reached through the procedure call, although the graph traversal does not actually descend into the called procedure (see Fig. 3.3). The marked nodes represent all nodes that are part of the calling context of *P* and may influence *n*.

- In the second phase, all edges except parameter-in and call edges (i.e., control and data dependences, summary and parameter-out edges) are followed backwards starting from all nodes that have been marked during phase 1. Because parameter-in edges and call edges are not followed, the traversal does not "ascend" into calling procedures. Again, the summary edges simulate the effects of the calling procedures. The marked nodes represent all nodes in called procedures that induce summary edges.

Fig. 3.5 shows how the slice is computed for the formal-out parameter *z* in procedure *Inc*. Only the nodes and edges that are traversed during the first phase are shown.

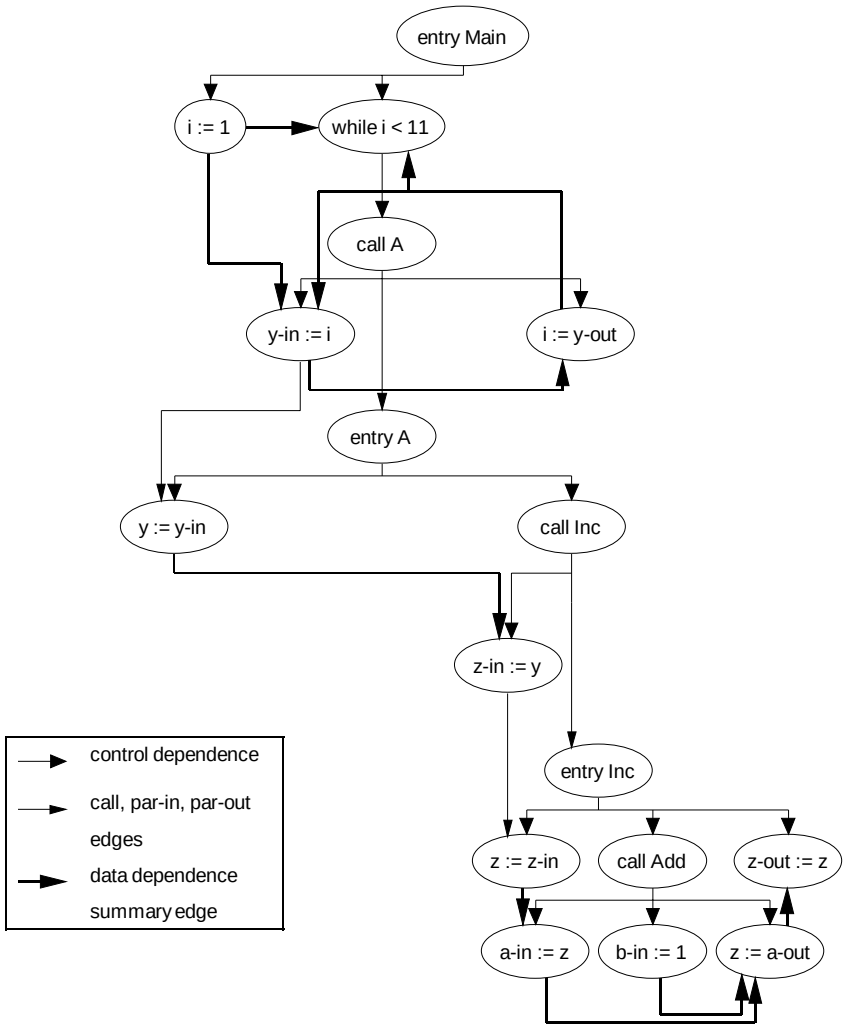

Fig. 3.5 - Nodes and edges visited during the first phase of computing the slice for the formal-out *z* in procedure *Inc*

Fig. 3.6 adds the nodes that are marked in the second phase (shown in bold). The complete slice consists of the nodes and edges visited during the two phases and the edges between them.

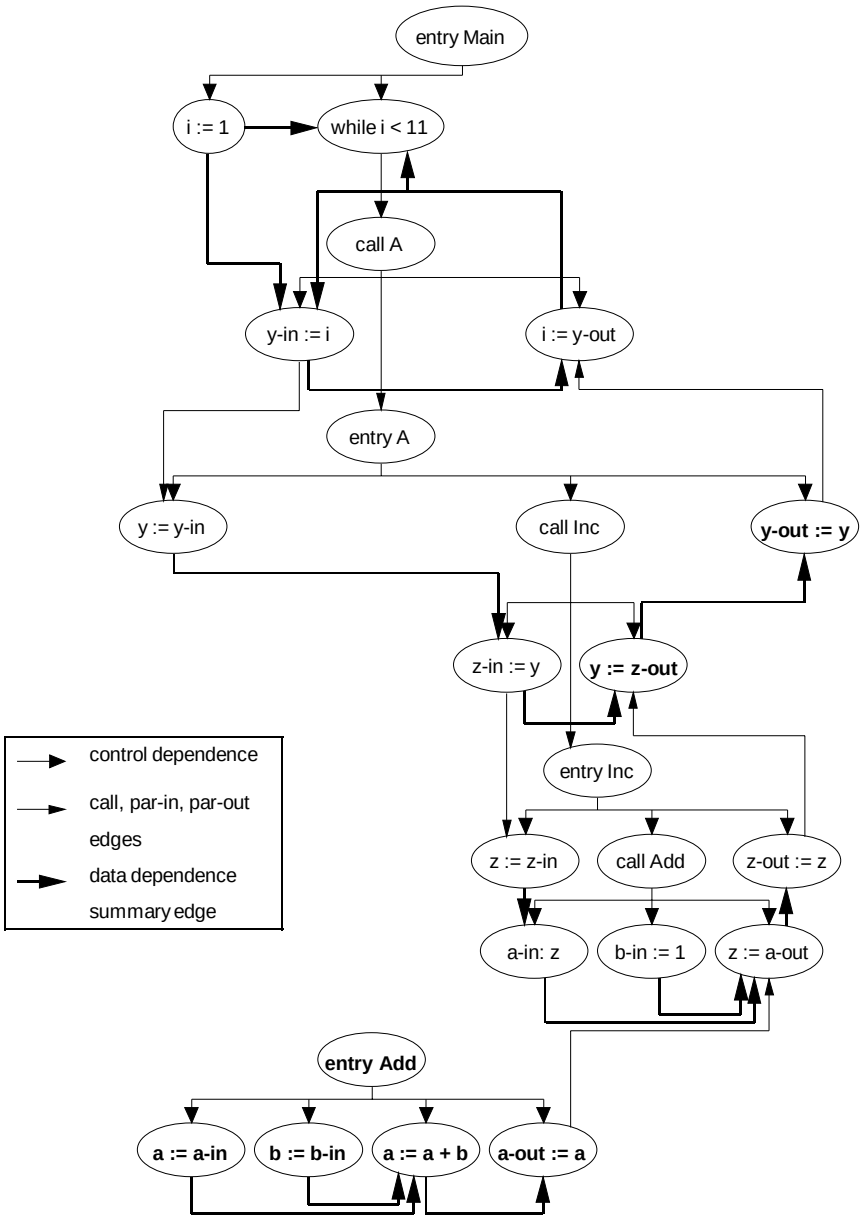

Fig. 3.6 - Nodes and edges visited during the second phase (shown in bold) of computing the slice for the formal-out *z* in procedure *Inc*

Fig. 3.7 shows a recursive implementation of the two-phase algorithm.

```
PROCEDURE SliceNodeInterproc (node: Node; excludeEdges: Set; visitedNodes: NodeSet);
BEGIN
  IF node is not marked THEN
    mark node as visited
    insert node into visitedNodes
    FOR all edges e leading from other nodes n to node DO
      IF kind of e is not in excludeEdges THEN
        SliceNodeInterproc(n,excludeEdges,visitedNodes)
      END
    END
  END
END SliceNodeInterproc;
```

```
PROCEDURE ComputeSlice (node: Node);
BEGIN
   (* phase 1: traverse control and data dependences, follow summary, par-in and call edges *)
   SliceNodeInterproc (node, {par-out}, visitedNodes);
   (* phase 2: traverse control and data dependences, follow summary, par-out edges *)
   FOR all nodes n in visitedNodes DO
      SliceNodeInterproc (n, {par-in, call}, visitedNodes)
   END
END ComputeSlice;
```

Fig. 3.7 - Interprocedural backward slicing algorithm

Forward Slicing

Horwitz et al. [HoRB90] showed that interprocedural forward slicing can be implemented in a very similar way to backward slicing where the edges are traversed from source to target. The first phase ignores parameter-in and call edges but follows parameter-out edges (thus ascends into calling procedures), whereas the second phase ignores parameter-out edges but follows parameter-in and call edges (thus descends into called procedures).

Dynamic Slicing

Agrawal and Horgan presented the first algorithm for finding dynamic slices using dependence graphs [AgH90]. The first approach to compute dynamic slices is to mark nodes and edges as the corresponding parts of the program are executed. After execution the slice is computed by applying the static slicing algorithm restricted to only marked nodes and the edges that connect them. Since multiple executions of a particular node are summarized by marking it once for all executions, these executions cannot be distinguished during analysis which makes the slices not as precise as possible. Another approach is to produce a *dynamic dependence graph* from the execution history that contains a node for each occurrence of a statement in the execution history along with only the executed edges. However, the dynamic dependence graph may be unbounded in length. Therefore Agrawal and Horgan also introduced the more economical version of a reduced dynamic dependence graph.

3.2.3 Computation of Summary Edges

Livadas et al. [LivC94, LivJ95] proposed a simpler method for the computation of summary edges. The basic idea is that for leaf procedures (procedures that do not call any other procedures) the summary edges can be computed via intraprocedural slicing, i.e. by following data dependences and control dependences backwards from the formal-out node. Summary edges are necessary from all formal-in nodes that are visited by this traversal to the formal-out node from which the traversal started. As long as there is no recursion, this idea can be applied to programs with procedure calls by analyzing the procedures recursively when they are encountered. The summary edges of a procedure are computed as soon as the

procedure is encountered:

- If a procedure *P* contains a call to another procedure *Q*, processing of *P* is suspended, and *Q* is processed. This process is continued until a procedure *R* is encountered that is either a leaf procedure or that has already been solved.
- If *R* is a leaf procedure, summary edges are computed directly via intraprocedural slicing. The processing of the calling procedure is then resumed.
- If *R* has already been solved, there is no reason to descend into the procedure again; subsequent calls to a solved procedure need only have the summary edges reflected (i.e. copied) to the call site.

We will illustrate this method by considering a program that consists of four procedures *M, A, B*, and *C* with calls as indicated in Fig. 3.8 (*M* calls *A* and *C, A* calls *B*, *B* calls *C, C* is a leaf procedure).

```
M  =  A  C.
A  =  B.
B  =  C.
C  =  .
```

Fig. 3.8 - Sample program abstraction

The computation of the summary edges starts with processing procedure *M*. The first call encountered is the call of procedure *A*. Since procedure *A* has not been solved, we descend into *A*. During processing *A*, the call to procedure *B* is encountered. Therefore we descend into the unsolved procedure *B*. Again, procedure *B* is not a leaf procedure and we descend into procedure *C*. But *C* is a leaf procedure and is solved immediately. The summary edges from *C* are reflected onto the call site of *C* in *B*. Similarly *B* can now be solved since it contains no more calls; its summary edges are reflected back to the call site of *B* in *A*. Now *A* can be solved and its summary edges are reflected back to the call site of *A* in *M*. Processing of *M* is resumed until the call to *C* is encountered. But *C* has already been solved; therefore its summary edges are simply reflected. Since *M* contains no more calls, *M* can be solved. Fig. 3.9 shows the call trace for this process.

```
Compute(M)
call A          =>  Compute(A)
                    call B          =>  Compute(B)
                                        call C          =>  Compute(C)
                                                            C is solved
                                                        <= reflect information of C
                                                           onto call in B
                                        B is solved
                                    <= reflect information of B onto call in A
                    A is solved
                <= reflect information of A onto call in M
call C
                <= reflect information of C onto call in M
M is solved
```

Fig. 3.9 - Call trace of the computation of summary edges
for a program without recursion (shown in Fig. 3.8)

The algorithm just described does not work well in the case of recursive procedures. The reason is that in the absence of recursion, it is guaranteed that a leaf procedure will be encountered that can be solved completely and its information can be reflected to its caller. In the case of recursion, the obtained information may be incomplete even if one processes a procedure in its entirety.

Therefore, recursion must be detected upon calls. Then only the partial information is reflected back to the call site and one does not descend into the (only partially solved) procedure. Finally if a procedure has been processed entirely, its information is reflected back on all its call sites. If the procedure has been part of a recursion chain, one has to iterate over the set of procedures that were part of the recursion. An iteration over a procedure *P* merely reflects the summary edges of all procedures *Q* that are called in *P* but does not descend into *Q*. Iteration is performed over the set of procedures until no more changes to the calculated information are found. At this point, all procedures of the iteration set are solved. Fig. 3.10 shows the a sample program with recursion.

```
M  =  A B.
A  =  B D.
B  =  C E.
C  =  C A.
D  =  .
E  =  F.
F  =  .
```

Fig. 3.10 - Sample program abstraction

Fig. 3.11 shows the call trace of the sample program of Fig. 3.10.

```
Compute(M)
call A          =>  Compute(A)
                    call B          =>  Compute(B)
                                        call C          =>  Compute(C)
                                                            call C          => recursion detected (C)
                                                                            <= partial information of C
                                                                               reflected
                                                            call A          => recursion detected
                                                                               (A, B, C)
                                                                            <= partial information of A
                                                                               reflected
                                                            C partially solved
                                                        <= (partial) solution of C reflected
                                        call E          =>  Compute(E)
                                                            call F          =>  Compute(F)
                                                                                F solved
                                                                            <= solution of F reflected
                                                            E solved
                                                        <= solution of E reflected
                                    <= solution of B reflected
                    call D          =>  Compute(D)
                                        D solved
                                    <= solution of D reflected
                    A partially solved
                    Iterate over all procedures of the recursion (A, B, C) until there are no changes
                                    => Iterate over A
                                                        <= reflect information of B and D
                                    => Iterate over B
                                                        <= reflect information of C and E
                                    => Iterate over C
                                                        <= reflect information of C and A
                                       A, B and C are solved completely
                                    <= complete information of A has been reflected to
                                       all call sites of A
                                    <= complete information of B has been reflected to
                                       all call sites of B
                                    <= complete information of C has been reflected to
                                       all call sites of C
                <= solution of A reflected
call B
                <= solution of B reflected
M solved
```

Fig. 3.11 - Call trace of the computation of summary edges
for a program with recursion (shown in Fig. 3.10)

3.2.4 Enhancing Slicing Accuracy

Horwitz et al. [HoRB90] noted that some imprecision is introduced if parameter nodes are generated for every parameter, regardless of whether it is changed by the called procedure or not. They use interprocedural data flow analysis to compute the sets of non-local variables and parameters that are used and modified by a procedure:

GMOD(P) is the set of non-local variables and parameters that might be modified by *P* itself or by a procedure (transitively) called from *P*.

GREF(P) is the set of non-local variables and parameters that might be referenced by *P* itself or by a procedure (transitively) called from *P*.

For each procedure *P*, there is one formal-in and one actual-in node for each variable or parameter in GMOD(P) ∪ GREF(P), and there is one formal-out and one actual-out node for each variable or parameter in GMOD(P).

Livadas et al. [LivC94, LivJ95] do not use interprocedural data flow analysis to derive this information but rather derive it during the construction of the system dependence graph. They further restrict the number of necessary parameter nodes depending on how the parameters are used within the called procedure:

- When a reference parameter is never modified (i.e., there is no path where the variable is defined), no formal-out and actual-out nodes are necessary.
- When a reference parameter is always modified (i.e., the variable is defined on every path), formal-out and actual-out nodes are necessary. At the call-site, a killing definition can be generated for the actual-out node.
- When a reference parameter is sometimes modified or when it is not known how it is used, formal-out and actual-out nodes are necessary. At the call-site a non-killing definition must be generated for the actual-out node.
- For value parameters no formal-out and actual-out nodes are necessary.