Program Slicing

Panos E. Livadas
Stephen Croll

Computer and Information Sciences Department
University of Florida
Gainesville, FL 32611

ABSTRACT

The concept of static program slicing was first introduced by Weiser. Ottenstein et al. indicated that an intraprocedural slice can be found in linear time by traversing a suitable graph representation of the program referred to as the program dependence graph (PDG). Horwitz et al. introduced algorithms to construct interprocedural slices by extending the program dependence graph to a supergraph of the PDG referred to as the system dependence graph (SDG). This extension captures the calling context of procedures.

In a previous paper, we demonstrated that a parse-tree-based SDG provides us with ‘‘smaller’’ and therefore more precise slices than a statement-based SDG. Furthermore, we described extensions to the SDG in order to handle particular constructs in a language that is a subset of ANSI C. In this paper, we will describe a new method for the calculation of transitive dependences and therefore build a SDG that does require neither the calculation of the GMOD and GREF sets nor the construction of a linkage grammar and the corresponding subordinate characteristic graphs of the linkage grammar's nonterminals. Additionally, we illustrate the versatility of the SDG as an internal program representation by briefly presenting a tool that we have developed that permits slicing, dicing, and ripple analyzing in addition to other software engineering activities to be performed on programs written in a subset of ANSI C. This report is an extension of our previous report published in December 1991.

1. Introduction

Software maintenance is an expensive, demanding, and ongoing process. Lientz and Swanson [Lie80] have reported that large organizations devoted 50% of their total programming effort to the maintenance of existing systems. Boehm [Boe75] has estimated that one US Air Force system cost $30 per instruction to develop and $4,000 per instruction to maintain over its lifetime. These figures are perhaps exceptional; but, on the average, they seem to be between two and four times higher than development costs for large embedded systems. We therefore aim to reduce maintenance costs by increasing the productivity of the maintainer. This can be accomplished with an integrated software environment that provides one with an assortment of tools.

Program slicing provides one such useful tool for the maintenance programmer. Formally, a slice of a program *P* at program point *p* relative to a variable *v* that is either defined or used at *p*, is defined to be the set of all statements and predicates of *P* that *might* affect *v*[1]. Program slices could be used in a variety of ways to aid in a number of software engineering activities such as the ones that are briefly discussed below.

[1] Weiser [Wei82] gave the following definition of a slice. Let *P* be a program, let *p* be a point in *P*, and let *V* be a subset of the variables of *P*. A static slice or simply a slice of *P* relative to the *slicing criterion* <*p*,*V*> is defined as the set of all statements and predicates of *P* that *might* affect the values of variable *V* at the point *p*. This definition is less general; but, it is all that is needed for our purpose.

Weiser [Wei82] has shown through his empirical studies that programmers use slices when debugging. Assume that during testing we find that the value of a certain variable, *v*, is incorrectly computed at some statement, *n*, of our program. By obtaining a slice of *v* at *n*, we may extract a significantly smaller piece of code to examine that allows location of the bug more easily. In addition, program slicing provides a meaningful way to decompose a large program into smaller components and can therefore aid in program understanding. Moreover, Horwitz [Hor88] has used the concepts of slicing in integrating program variants and Badger [Bad88] has demonstrated how slicing can be used for automatic parallelization. Furthermore, slicing also aids in code reusability. If a large program computes the value of a variable *x* and the code associated with its computation is needed in another application, then one may slice on this variable and use the extracted program in the application at hand. Furthermore, a number of metrics based on program slicing have been proposed [Wei82] which include coverage, component overlap, functional clustering, parallelism, and tightness.

Weiser's slicer was based on a flow-graph representation of Simple_D programs. Ottenstein et al [OTT84], showed that an *intraprocedural slice* could be found in linear time by traversing a suitable graph representation of the program which they referred to as the *program dependence graph (PDG)*. Horwitz et al [Hor90], have introduced algorithms to construct interprocedural slices by extending the program dependence graph to a supergraph of the PDG which is referred to as the *system dependence graph (SDG)*. This extension also captures the calling context of the procedures that was lacking in the method proposed by Weiser; and, it also permits slicing to be performed even if a program contains calls to *unknown* procedures, provided that *transitive dependences* are known.

Informally, the SDG is a labeled, directed, multigraph where each vertex represents a program construct such as declarations, assignment statements, and control predicates. Edges represent several kinds of dependences among the vertices which can be distinguished by the labels attached to them. The SDG is a suitable form of internal representation which supports the integrated maintenance environment that was discussed earlier since slicing, ripple analysis, and dicing can be reduced to a graph reachability problem.

Realizing the versatility of this IPR, we have developed a prototype that accepts programs written in a subset of ANSI C and which generates a SDG. We have also implemented tools such as a slicer, a dicer, and a ripple analyzer that can utilize this SDG. In addition, we are exploring the use of this SDG in some of the software engineering activities discussed above.

To accomplish this task, we had to ''expand'' the language described in [Hor90], modify the SDG, and modify the algorithms presented. In particular, the SDG described in [Hor90] models a language that consists of a single (main) program and which supports scalar variables, assignment statements, conditional statements, and while loops. Furthermore, the language consists of a collection of procedures where parameters are passed by value-result, there exist global variables, and procedures end with `return` statements. These `return` statements can not be arbitrarily located in the procedure nor do they actually return values to their calling procedure(s). Additionally, recursive procedures are supported.

We extend this grammar and consequently modify the SDG as follows. First, declarations of local, global, and local variables are supported. Second, the distinction is made between the two methods of parameter passing: pass-by-value and pass-by-reference. The same notation is employed as in C, so that the type of parameter passing can be determined. However, pointer operations are restricted to those that constitute ''pass-by-reference parameters''; i.e., if `x` and `y` are pointer variables, we permit assignments of these variables such as `*x = 4`, and `*y = *x` (where `*` denotes a dereferencing of the contents of the variables); but, general pointer assignments such as `x = y` are not allowed. Third, any number of `return` statements are permitted to appear anywhere in a procedure. These `return` statements can contain expressions that may

include variables and are modeled after the `return` statements in the language C. Fourth, we distinguish among functions that return values as opposed to those that do not. Fifth, additional C constructs are ''handled'' except `goto`, `break`, and `continue`. Finally, a *parse tree* is used as the basis of our SDG. This allows slicing to be more precise than if the "resolution" of the SDG was only at the statement level.

The method presented in this paper for calculating dependencies for procedures (including recursive procedures) eliminates the need to construct a linkage grammar and the corresponding subordinate characteristic graphs of the linkage grammar's nonterminals. As described by [Hor90], a linkage grammar consisting of one nonterminal and one production must be constructed for each procedure. The attributes in the linkage grammar correspond to the parameters in the procedures. This attribute grammar is the input to an algorithm that is a slight modification of the algorithm described in [Kas80]. Horwitz's algorithm requires the construction of an auxiliary graph which expresses the dependencies among the attributes of a production's nonterminal occurrences.

Additionally, we describe our reaching definitions calculator tool that uses the SDG to show reaching definitions. These definitions can span procedure boundaries.

Presently, we have implemented the algorithms that are presented here as part of our Ghinsu tool. This tool accepts a source program written in a subset of ANSI C as input and produces an internal program representation that is based on the (SDG) and which can subsequently be used by any of our four implemented tools: slicer, dicer, ripple analyzer, and reaching definitions calculator.

The definition of a static slice that will be employed in the sequence is the same as the one given in [Hor90] which is less general than the one proposed by Weiser but which does capture the ''spirit'' of the slice. In particular, a static slice of a program P at a program point p relative to a variable v, that is either defined or used at p, is the set of all statements and predicates of P that *might* affect v.

2. The Program Dependence Graph and the System Dependence Graph

2.1. Program Dependence Graph

The *program dependence graph* (PDG) for a program P, with no procedures, denoted by G_P, is a labeled, directed, multigraph.

Each node represents a program construct such as declarations, assignment statements, and control predicates; there is also a special node called the *entry* node. Generally speaking an entry node is the root of the tree that represents the ''body'' of a function.

Edges represent several kinds of dependences among the nodes which can be distinguished by the label attached to them. Specifically, three dependences are distinguished[2]: *control*, *data flow*, and *declaration*, each of which will be briefly discussed below. A more complete discussion can be found in [Liv91]. In particular, let v_1 and v_2 be two nodes of G_P.

If the execution of v_2 is determined by the predicate represented by v_1 at the time of execution, then v_2 is *control dependent* on v_1; and, a control dependence edge from v_1 to v_2 is defined. In other words, control dependence exists incident from a node v_1 to a node v_2 if v_1 represents a control predicate and v_2 represents those components of the program P *immediately* nested within the loop or conditional whose predicate is represented by v_1. The above relationship is denoted via the following notation:

[2] In reality there is a further dependence edge due to the return statement. We will defer discussion of this edge to the next section.

$$v_1 \xrightarrow{cd} v_2$$

We note that every component of P that is not subordinate to any control predicate is control dependent on the entry node. Given the grammar's constructs[3] under consideration, control dependences reflect the program's *nesting structure*. The source v_1 of a control dependence is always either the entry vertex (in which case v_2 is not nested within any loop or conditional) or a predicate vertex (in which case v_2 is immediately nested within the loop or conditional whose predicate is represented by v_1) . Consider the program in Table 1. Statements [6], [7] and [8], [9] are control dependent on [5].

If data propagate from v_1 to v_2, then we say that v_2 is *data flow dependent* on v_1; and, a data dependence edge exists[4]. Stating the concept differently, if v_2 is data flow dependent on v_1, then the value computed by program P will be different if the positions of statements v_1 and v_2 are reversed. The above relationship can be denoted by using the following notation:

$$v_1 \xrightarrow{dd} v_2$$

Dependences that exist between the declaration and the definition of variables in a program are represented by *declaration edges*. Such an edge exists from a node v_1 corresponding to the declaration of a variable to each of the nodes v_2 corresponding to that variable's subsequent definitions. Declaration edges can be considered to be a special kind of data flow edge. The above relationship can be identified by using the following notation:

$$v_1 \xrightarrow{de} v_2$$

We give one more definition: Let s_0 and w be two nodes in G_P. We define an *intraslice-path* from w to s_0 and denote by $S_w^{s_0}$ a path on G_P with initial vertex w and terminal vertex s_0 having the property that the edges of $S_w^{s_0}$ represent data, control, or declaration dependences. In symbols

$$\forall v_i, v_j, w, s \in G_P \ni: \vec{e}_{i,j} = (\overline{v_i, v_j}) \in S_w^{s_0} \rightarrow \left[(v_i \xrightarrow{dd} v_j) \vee (v_i \xrightarrow{cd} v_j) \vee (v_i \xrightarrow{de} v_j) \right]$$

We will say that s_0 is an intraslice-path reachable from w, if and only if, there exists an intraslice path from w to s_0.

Having defined the program dependence graph, we are now in a position to define the slice of a program with respect to a variable in a statement for a single-procedure program such as defined in [Ott84]. Specifically, if G_P is a program dependence graph and s_0 is a node in G_P, then the slice of G_P with respect to s_0, denoted by G_P/s_0, is that subgraph of G_P which consists of those nodes w from which s_0 is intraslice-path reachable. Hence,

$$V(G_P/s) = \left\{ w \varepsilon V(G_P) : \exists\ S_w^s \right\}$$

Equivalently, the nodes of G_P/s are those of G_P that are encountered when one traverses the graph G_P starting from the node s and following all of the edges *backwards.*

---

[3] The handling of a `return` statement requires the introduction of additional edges that are discussed in the following section.

[4] In [Hor90], a distinction is made between loop-carried and loop-independent dependence edges. This distinction is not made here. Furthermore, another type of data dependence, the *def-order* dependence, is not employed.

2.2. System Dependence Graph

Our discussion now moves to slicing on a program which consists of a collection of one or more procedures and their associated parameters. To address this problem, the program dependence graph is extended to what is called a system dependence graph (SDG)[Hor90]. An SDG for a program P consists of a PDG that models the main program M and a collection of L *procedure dependence graphs* that model the program's K procedures F^k for each non-negative integer k such that[5] $0 \le k \le K = L$. The extension of the PDG to SDG that captures the calling context also requires the introduction of an additional set of nodes and an additional set of edges. Each of these sets is discussed in turn below.

2.3. The Nodes

First, whenever a call to a function F^k is encountered, a node (referred to as the *call-site node*, denoted by $cs_j^{F^k}$ where j is a nonnegative integer that is employed to enumerate *static* calls to F^k) is created. Then a number of nodes, referred to as the *actual-in* nodes, denoted by $a\_in_{i,j}^{F^k}$ where i is equal to the actual number of parameters of the function F^k, are created. At the same time, another set of nodes, referred to as the *actual-out* nodes and denoted by $a\_out_{l,j}^{F^k}$ where l is equal to the number of parameters of the function F^k that are passed-by-reference, are built[6]. It is worth noting that the set $a\_out_{l,j}^{F^k}$ could be empty; moreover, for a fixed k, the cardinality of $a\_out_{l,j}^{F^k}$ remains the same for each call j. By definition, all such nodes are control dependent to the call-site node. In symbols,

$$\left[\forall j \quad \forall k \quad \forall v \in (a\_in_{i,j}^{F^k} \bigcup a\_out^{F^k{}_{l,j}})\right] \longrightarrow (cs_j^{F^k} \xrightarrow{cd} v)$$

Secondly, at the time that the *first* static call to a function F^k was encountered, an additional node, called the *entry node* and denoted by en^{F^k}, was created. Moreover, a number of additional nodes, referred to as the *formal-in* nodes ($f\_in_i^{F^k}$) and equal to the number of formal parameters are created. To complete the scenario, a number of nodes, referred to as the *formal-out* nodes ($f\_out_l^{F^k}$) and equal to the number of parameters that are passed-by-reference, are built. It is worth noting that for all j and fixed k, the set $a\_in_{i,j}^{F^k}$ is isomorphic to $f\_in_i^{F^k}$, whereas the set $a\_out_{l,j}^{F^k}$ is isomorphic to $f\_out_l^{F^k}$. Finally, by definition all such nodes are control dependent on the entry node. In symbols:

$$\left[\forall j \quad \forall k \quad \forall v \in (f\_in_i^{F^k} \bigcup f\_out_l^{F^k})\right] \longrightarrow (en^{F^k} \xrightarrow{cd} v)$$

2.4. The Edges

At this point, we present additional types of edges that will enable us to build the system dependence graph.

By definition, for each k and each j, the vertex en^{F^k} is adjacent to $cs_j^{F^k}$. Each such edge that is incident from a call-site node and incident to an entry node is referred to as a *call edge*. Notice that the $indegree(en^{F^k}) = j_k$ where j_k is the number of call-sites corresponding to the function F^k. Hence, for each k

$$\forall j \quad (cs_j^{F^k} \xrightarrow{ce} en^{F^k})$$

---

[5] We will see shortly that in the presence of aliasing $K \le L$.

[6] Note that in [Hor90], the number of actual-out nodes is equal to the number of actual-in parameters to facilitate the calling mechanism.

A *parameter-in edge* is an edge from an actual-in node to its corresponding formal-in node. Similarly, a *parameter-out edge* is an edge from a formal-out node to its corresponding actual-out node. In symbols we have,

$$\forall j \quad \left[\left[\forall i \quad a\_in_{i,j}^{F^k} \xrightarrow{pi} f\_in_i^{F^k}\right] \wedge \left[\forall l \quad f\_out_l^{F^k} \xrightarrow{po} a\_out_{l,j}^{F^k}\right]\right]$$

The system dependence graph must represent direct dependences between a call-site and the called procedure and the *transitive dependences* due to procedure calls. In this case, a transitive dependence edge exists from an actual-in node to an actual-out node if the formal-out node corresponding to the latter node is intraslice-path reachable from the formal-in node corresponding to the former node. Notice that this is equivalent to saying that the intraprocedural slice of the procedure dependence graph at the formal-out node *contains* the formal-in node. In other words, for each fixed k

$$(\exists i \quad \exists l \ni: a\_in_{i,j}^{F^k} \xrightarrow{td} a\_out_{l,j}^{F^k}) \iff \left[(\exists i \quad \exists l \ni: f\_in_i^{F^k} \xrightarrow{td} f\_out_l^{F^k}) \iff (f\_in_i^{F^k} \in V(G_{F^k}/f\_out_l^{F^k}))\right]$$

where by G_{F^k} we denote the procedure dependence graph of the function F^k. The collection of these transitive dependence edges at each call-site F_j^k is defined as the procedure's *summary* information ([Hor90] and denoted by$\sigma^{F_j^k}$. Hence,

$$\sigma^{F_j^k} = \bigcup_{i=1}^{i_k} \left\{\vec{e}_{i,l} = (\overline{a\_in_{i,j}^{F^k}, a\_out_{l,j}^{F^k}}) : \exists i \ \exists l \ni: (f\_in_i^{F^k} \xrightarrow{td} f\_out_l^{F^k})\right\}$$

Now, we will define three new types of edges that have been designed so that they permit us to ''properly'' handle functions that do return values (as opposed to those that do not) as well as handling C-like `return` statements. Properly, in the sense that when the slicing algorithm is eventually applied, the contributed slice will be as ''minimal'' as possible.

C functions may or may not return a value to the call-site. In the former case, the returned value may be data dependent on one or more of the actual parameters. If that is the case, then these parameters should be included in the slice. Therefore, we define a new edge, the *affect-param edge*, that indicates this parameter-returned value dependence. Such an edge, if it exists, is by definition incident from the actual-in node corresponding to the actual parameter that influences the returned value and incident to the function's call-site node. In symbols,

$$\forall i \quad \ni: \textit{the value returned by } F^k \textit{ is data dependent on } a\_in_i^{F^k} \rightarrow \left[a\_in_i^{F^k} \xrightarrow{ap} cs_j^{F^k} \ \forall j\right]$$

As we indicated in [Liv91] two new types of edges, an intraprocedural edge the *return-control edge* and an interprocedural edge the *return-link edge* are needed to properly handle the `return` statements.

The *return-control edge* indicates the dependence between the return statement of a procedure and other statements following the return statement which will not be executed when the program exits on a return statement. In other words, a return-control dependence exists between a return node v_{rn} and another node of v_s of G_{F^k}, if and only if, execution of the return statement corresponding to the former node excludes execution of the statement corresponding to the latter node. The above relationship can be defined as follows:

$$\left[\forall v_s, v_{rn} \in G_{F^k} \rightarrow (v_{rn} \xrightarrow{rc} v_s)\right] \iff (v_s \xrightarrow{cd} v_{rn})$$

A *return-link edge* connects a return node to the corresponding function call-site. Specifically,

$$\forall v_{rn} \in G_{F^k} \ \forall j \quad (v_{rn} \xrightarrow{rl} cs_j^{F^k})$$

Given now that our grammar allows call-by-reference parameters, `return` statements, as well as functions that may return values, we can define as the *summary information*, $\overline{\sigma}^{F_j^k}$, at a call-site $cs_j^{F^k}$ to be the union of three types of dependences: transitive dependences, affect-param dependences, and return-link dependences. Therefore,

$$\overline{\sigma}^{F_j^k} = \sigma^{F_j^k} \cup \left[\bigcup_{i=1}^{i_k} \left\{ \vec{e}_{i,j} = (\overline{a\_in_{i,j}^{F^k}, cs_j^{F^k}}) \; : \; \exists i \; \ni : \; (a\_in_{i,j}^{F^k} \xrightarrow{ap} cs_j^{F^k}) \right\} \right]$$

$$\cup \left[\bigcup_{rn} \left\{ \vec{e}_{rn,j} = (\overline{v_{rn}, cs_j^{F^k}}) \; : \; \exists rn \; \ni : (v_{rn} \in G_{F^k}) \wedge (v_{rn} \xrightarrow{rl} cs_j^{F^k}) \right\} \right]$$

Figure 1 presents the system dependence graph of the program that is shown in Table 1. Declaration edges are not shown.[7]

At this time, we should note that the nodes of the SDG (in the figures) are shown ‘‘resolved’’ at the statement level. In actuality, the SDG is ‘‘resolved’’ at the token level. Using a parse tree representation as the basis for our SDG allows more precise slices to be calculated. This is further detailed in Section 5. In the sequel, by the term SDG we will denote a parse-tree-based SDG unless otherwise noted. For the purposes of simplicity, the figures in this paper are shown resolved at the statement level.

```
void main()                    int CalcSum(int s)   void Inc(int *x)
{                              {                    {
    int sum;                       Inc(&s);             *x = *x + 1;
    int i;                         s = s + 9;       }
    i = 0;                         return s;
    while (i < 10) {           }
        i = i + 1;
        sum = CalcSum(sum);
    }
    i = i;
    sum = sum;
}
```

Table 1. A sample program.

The edges representing the dependences of this set permit one to descend into a procedure during slicing and calculate the statements of the function that should be included in the slice. Since the effects of the parameters and of the return statements have been ‘‘summarized’’, it is necessary to descend into the procedure only once to calculate dependences. As additional function call-sites are encountered, the procedure’s summary can be reflected onto the appropriate nodes in the graph. An additional benefit is that there is no requirement that the contents of a procedure be known in order to perform slicing; we only need to know its effects.

As we indicated earlier, determination of the transitive dependences of a procedure F^k is accomplished by determining for each $f\_out_l^{F^k}$ all formal-in nodes $f\_in_j^{F^k}$ from which the former node is intraslice-path reachable. But, the structure of the procedure dependence graph G_{F^k} is different from the one presented in Section 2 because it may contain a number of call-site nodes $cs_{j_k}^{F^{m_k}}$ together with their associated summary information where j_k is the number of static calls that are made from F^k to functions F^{m_k}, respectively. Hence, the definition of an intraslice-path $S_w^{s_0}$ given in Section 2 is extended to a path that is denoted by $\overline{S}_w^{s_0}$ so that it is also including edges that represent the transitive dependences and affect-param edges associated with the call-sites of $cs_{j_k}^{F^{m_k}}$

[7] In order to keep the graph from becoming ‘‘busier’’ than it already is.

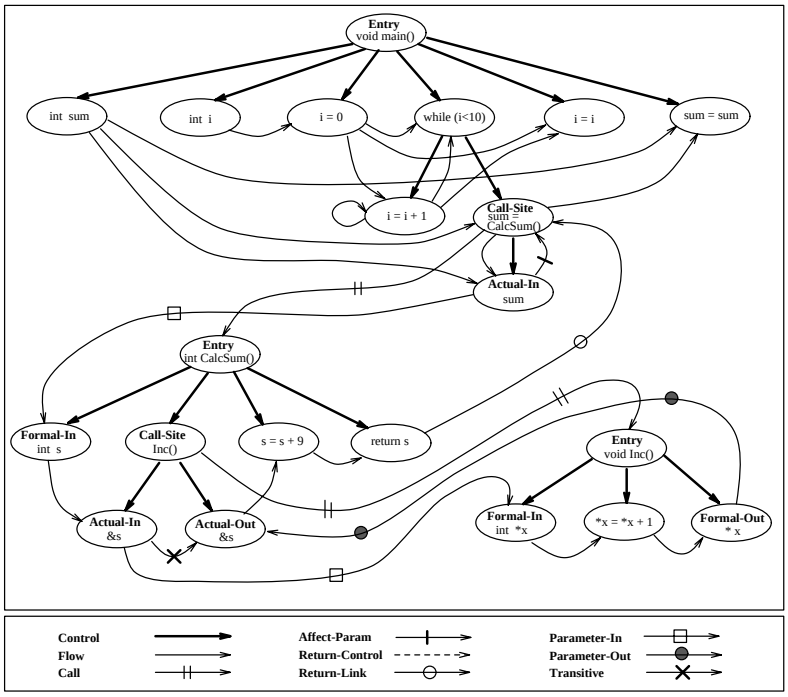

Figure 1. The program dependence graph corresponding to the program in Table 1.

as well as the return-control edges. Hence,

$$\forall v_i, v_j, w, s_0 \in G_{F^k} \ \ni: \vec{e}_{i,j} = (\overline{v_i, v_j}) \in \overline{S}_w^{s_0} \ \rightarrow \ \left[\vec{e}_{i,j} \in \overline{S}_w^{s_0} \ \vee \ (v_i \xrightarrow{ap} v_j) \ \vee \ (v_i \xrightarrow{rc} v_j) \right]$$

We will say that s_0 is intraslice-path reachable from w, if and only if, there exists an intraslice-path from w to s_0. Hence, an intraprocedural slice is defined via

$$V(G_{F^k}/s_0) \ = \ \left\{ w \varepsilon V(G_{F^k}) \ : \exists \ \overline{S}_w^{s_0} \right\}$$

The algorithm that performs this task is similar to the one described in the previous section with the additional requirement that the affect-param, and return-control edges also be traversed ‘‘backwards’’.

2.5. Global Variables, Static Variables, and Aliasing

2.5.1. Global Variables

Source programs which contain global or static variables, or in which aliasing is present, need to be transformed before they can be represented by a SDG. The following sections describe these conversions.

The handling of global variables is based on the method suggested by [Hor90]. Specifically, globals are solved by introducing them as additional pass-by-reference parameters to the procedures that use or define them. All procedures that call a procedure directly or a procedure which indirectly uses or defines global variables are modified to include the global variables as pass-by-reference parameters. The call-sites are also modified to include the new parameters.

This however is an incomplete solution. Because of possible naming conflicts, the global variables may need to be renamed. Consider the program in Table 2. In procedure `Inc`, there is a naming conflict. Although `Inc` does not directly use the global variable `g`, `Inc` calls function `IncGlobal` which does. Adding an additional parameter `g` to procedure `Inc` would create an obvious naming conflict. Naming conflicts can arise when a formal parameter or a local variable share the same name with a global variable.

```
int g;

void main(void)      int Inc(int g)      void IncGlobal(void)
{                    {                   {
    int i = 4;           IncGlobal();        g = g + 1;
    i = Inc(i);          return g+1;     }
}                    }
```

Table 2. Illustration of global naming conflicts.

The solution is to *rename* the global variables to avoid this conflict. A simple approach to choosing unique names would be to simply append an "illegal" character to the end of a global variable. For example, the global variable `g` can be renamed `g+`. Note that this renaming can be done on the SDG; the source program need not be altered.

2.5.2. Static Variables

Static variables in C are essentially global variables with limited visibility. These variables *exist* across invocations of the procedure in which they are declared. They can be handled in the same manner as "regular" global variables except special attention must be paid to avoid naming conflicts; there may be several static variables with the same name among modules, procedures, or even within the same procedure. As in the renaming of global variables, a simple approach to choosing unique names would be to append an "illegal" character to the end of the static variable. Additionally, the name of the procedure in which it is declared is also appended. This will remove naming conflicts between procedures. To avoid naming conflicts *within* the same procedure, the scoping level of the static variable is also appended. For example, the static variable `s` in the procedure `Add`, that is declared in the first scope of the procedure, would be renamed `s+Add1`.

2.5.3. Aliasing

Our previous discussions have not included the problem of aliasing. The reason is that when aliasing phenomena occur during a call to a procedure, they are then resolved (during the SDG construction) via a transformation to an alias free procedure. We first note that when a global variable `g` is encountered in the body of a function `alias` that is invoked via the call[8] `alias(`$\&x_1,\&x_2,\ldots,\&x_m$`)` and function's header `alias(`$*y_1,*y_2,\ldots,*y_m$`)` then *internally* it is assumed that the call was `alias(`$\&x_1,\&x_2,\ldots,\&x_m,\&g+$`)` and the function's header was `alias(`$*y_1,*y_2,\ldots,*y_m,*g+$`)`; actual as well as formal nodes are adjusted accordingly. Hence, from that point on we may assume the existence of neither global nor static variables and that each function call is of the form `alias(`$\&x_1,\&x_2,\ldots,\&x_n$`)` and the function's header `alias(`$*y_1,*y_2,\ldots,*y_n$`)`. Now given our grammar aliasing can occur, if and only if, a call of the form `alias(`$\&x_1,\&x_2,\ldots,\&x_n$`)` is made with $x_i = x_j$ where $i \neq j$ and $1 \leq i,j \leq n$. In that case $*y_i$ and $*y_j$ are aliases.

[8] The types of the formal parameters are omitted.

The transformation to an alias-free procedure dependence graph is simple. When a procedure must be solved, a *tag* is attached to it. A tag of a procedure with n parameters is a mapping from $\underset{i=1}{\overset{n}{X}}\{i\}$ to N^n that indicates the aliasing pattern for that particular call. The mapping is straight forward; if $y_{i_1}, y_{i_2}, \ldots, y_{i_k}$ are aliases, the positions $i_1 \le i_2 \le \ldots \le i_k$ of the image vector are set to the same value i_1. For example, if no aliasing is present, the mapping is the identity on $\underset{i=1}{\overset{n}{X}}\{i\}$; whereas if y_2 and y_4 are aliases, the mapping is given by

$$tag(1,2,3,4,5,....,n-1,n) = (1,2,3,2,5,...,n-1,n)$$

which indicates that the second and fourth parameters are aliases. When aliasing is detected at the call-site to a function F^k, the call-site is tagged; a new entry node is created and tagged as described[9]; the (alias-free) abstract syntax tree representing `alias` is copied so it is rooted at this new entry node; and, data dependence analysis is performed by ''identifying'' the sets of variables that are aliased. We note here that the possible number of alias configurations for a procedure with n passed-by-reference parameters is $2^n - n$.

3. The Interprocedural Slicing Algorithm

The interprocedural slicing algorithm is based on the algorithm suggested in [Hor90]. Modifications are necessary given the additional constructs introduced in the grammar. The algorithm finds the slice relative to a node s_0 of a program G_P in two phases. During the first phase, a set of nodes U_1 of G_P is captured with the property that $u \in U_1$, if and only if, s_0 is *phase 1 reachable* from w. Phase 1 reachability is equivalent to the property that there is a path from u to v consisting of any of the following types of edges: control dependence, data dependence, declaration dependence, return-control, parameter-in, transitive dependence, affect-param, and/or call. In the second phase, we capture an additional set of nodes U_2 of G_P with the property that $w \in U_2$, if and only if, there exists a node $u \in U_1$ such that u is *Phase 2 reachable* from w. Phase 2 reachability is equivalent to the property that there is a path from w to u consisting of any of the following types of edges: control dependence, data dependence, declaration dependence, return-control, parameter-out, transitive dependence, affect-param, and/or return-link edges. Finally, the vertices of the interprocedural slice are defined as the union of the nodes visited in both phases. In symbols

$$V(G_P/s_0) = U_1 \cup U_2$$

In general, at each phase all indicated edges are followed recursively backwards as they were when intraprocedural slicing was performed.

Finally, the slicing algorithm is modified to handle the return-control edges. This modification is based on the observation that the slicer must recognize when a return-control edge is being traversed. The node at the end of the return-control edge (a return statement) is marked as being in the slice. The slicer now ''short circuits'' to the control predicate of the return statement and slicing continues as normal.

We should note that when a call to a procedure F yields aliasing and a slice at a statement s_0 that is *internal* to the body of procedure F, special care must be taken. Assuming that the total number of aliasing patterns is $m > 0$, let s_0^m represent the instance of s_0 in each procedure dependence graph associated with F. Then the slice is given by

$$\bigcup_{i=1}^{m} \left\{ slice\ at\ \ s_0^m \right\}$$

[9] In our implementation both call-site and entry nodes are identified via $F^k.tag$.

4. Enhancing Slicing Accuracy

There are a number of instances in which an actual-out node should not exist as when a passed-by-reference parameter is not modified. In this case, the presence of its actual-out node could adversely affect the precision of an interprocedural slice. A method is described in [Hor90] to detect such a phenomena that is based on the calculation of the GMOD and GREF sets (via the method proposed in [Ban79]) for *each* procedure F^k. We have determined that calculating these sets is not necessary under our method since all information required for that determination is contained in the procedure's dependence graph. Furthermore, as we will show in the next section, we are deriving this information *during* construction of the SDG.

In particular, we consider three cases for a pass-by-reference variable. The first case is where the variable is passed to the procedure and is *never* modified. The second is where the variable is passed and may *sometimes* be modified. And, in the third case, the variable is passed and is *always* modified. For the purposes of slicing, the second and third cases can be combined. However, by differentiating between the second and third case, we are able to use that information for other related applications such as calculating reaching definitions.

To illustrate how the *never*, *sometimes*, and *always* cases affect the architecture of the SDG, consider the subgraph of SDG in Figure 2. If the variable `v` of procedure `Func()` is classified as *never* modified, the actual-out node is *not* considered a definition of variable `v`. Therefore only flow edge #1 exists. If the variable is classified as *always* modified, the actual-out node *is* considered a definition of variable `v`. In this case, the definition *kills* all the reaching definitions of variable `v`. Correspondingly, the SDG will contain only flow edge #2. In the final case where the variable is classified as *sometimes* modified, the actual-out node *is* also considered a definition of variable `v`. However, this definition does *not kill* the reaching definitions of `v`. Therefore both flow edge #1 and flow edge #2 exist.

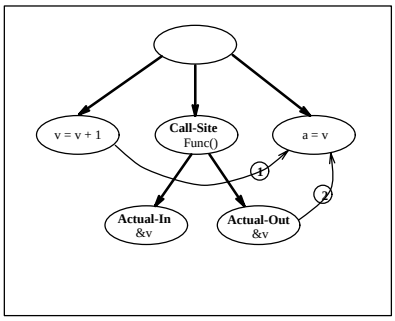

Figure 2. Subgraph of SDG.

Once a procedure has been summarized (solved), we can examine the indegree of the formal-out (relative to the flow edges) node, $f\_out_i^{F^k}$, to determine whether a pass-by-reference variable `i` corresponding to that node is always, sometimes, or never modified by the procedure. Consider Figures 3 through 5 which illustrate a partial procedure dependence graph and where only the flow edges are shown for the formal-out node corresponding to pass-by-reference variable `*i`. A variable is *never modified* as in Figure 3 since there is only one flow edge incident to $f\_out_i^{F^k}$; and, it is also incident from the formal-in node $f\_in_{i_i}^{F^k}$ which suggests that there is no intervening definition of `i`.

Figure 4 illustrates the case where the variable i is *sometimes modified* depending on whether or not the `if` condition evaluates to true. In this case there exists more than one flow edge incident to the formal-out node; and, one of these edges connects the formal-out node with its formal-in node.

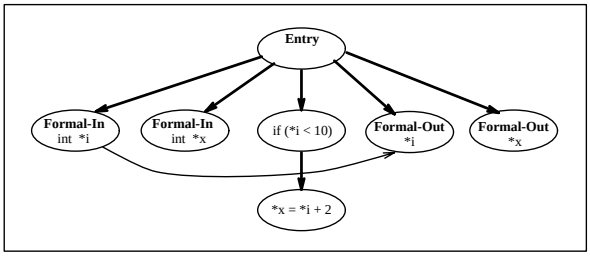

Figure 3. The formal-out parameter `*i` is never modified.

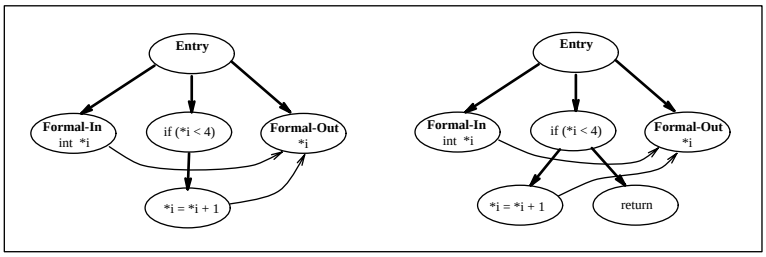

Figure 4. The formal-out parameter `*i` is sometimes modified.

Notice that in the last case (Figure 5), the variable is *always* modified since the absence of the flow edge connecting the formal-out node to its formal-in node suggests that *every* possible execution path contains a definition of *i*.

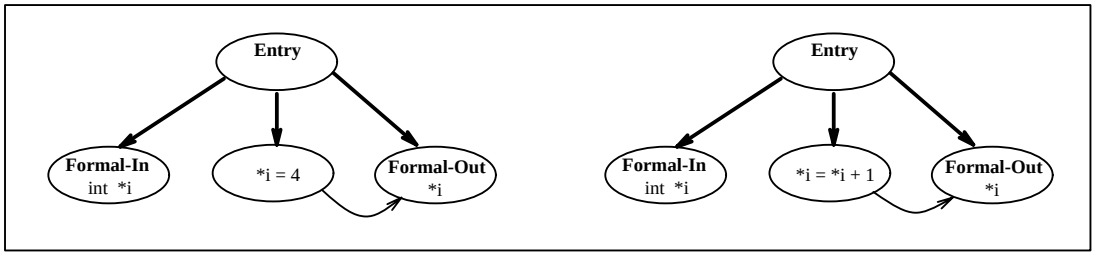

Figure 5. The formal-out parameter `*i` is always modified.

We can summarize as follows: Let *A*, *S*, and *N* denote the set of formal-out nodes of F^k that are always, sometimes, and never modified, respectively. Then a node $f\_out_l^{F^k}$ is classified as follows:

1. $(f\_out_l^{F^k} \in A) \leftrightarrow (\forall i \neg (f\_in_i^{F^k} \xrightarrow{dd} f\_out_l^{F^k}))$,
2. $(f\_out_l^{F^k} \in S) \leftrightarrow ((f\_in_{i_l}^{F^k} \xrightarrow{dd} f\_out_l^{F^k}) \wedge (indegree_{dd}(f\_in_i^{F^k}) > 1))$,
3. $(f\_out_l^{F^k} \in N) \leftrightarrow ((f\_in_{i_l}^{F^k} \xrightarrow{dd} f\_out_l^{F^k}) \wedge (indegree_{dd}(f\_in_i^{F^k}) = 1))$.

where $indegree_{dd}(v)$ denotes the indegree of the node v relative to the data dependence edges.

Additionally, a fourth case exists where the indegree of the formal-out node (relative to the data dependence edges) is equal to zero. This case is referred to as the *unknown* case. Initially, all actual-out nodes are classified as *unknown*.

5. Slicing Inaccuracies Due to System Dependency Graph Representation

As described in [Liv91], instead of the nodes of the SDG being ‘‘resolved’’ at the statement level, we map this SDG into a parse tree so that it ‘‘resolves’’ at the token level. The marked nodes of the SDG are then *interpreted* to calculate the correct slice. This method yields smaller and therefore more precise slices than those contributed by the statement level SDG.

Consider the code fragment presented in Table 3(i) and its corresponding partial system dependence graph illustrated in Figure 6.

```
[1]. i = 0;                          [1].  i = 0;
[2]. sum = 0;
[3]. sum = sum + Alpha(&i);          [3].  Alpha(&i);
[4]. a = i;                          [4].  a = i;
[5]. b = sum;

       (i)                                  (ii)
```

Table 3. The code fragment of a program (i) and its slice (ii) at statement [4].

The function of the procedure `Alpha` (whose code is not shown) is to increment the pass-by-reference parameter and to return the new value of the parameter (as the return value of the procedure).

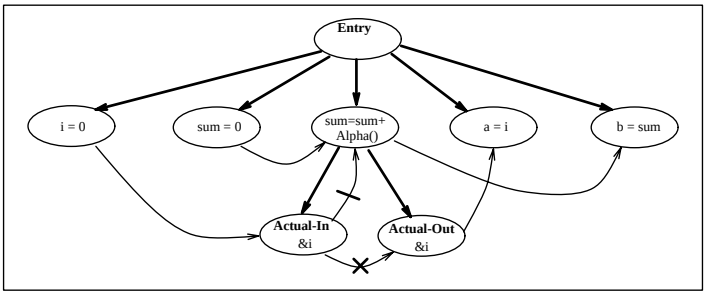

Figure 6. The partial system dependence graph corresponding to the code fragment of Table 3(i).

If we slice this program via the algorithm presented earlier on the statement-based system dependence graph of Figure 6 at the statement `a = i`, we will obtain the set of statements {[1], [2] , [3], [4]}. Clearly, the variable `a` at line [4] does not depend on the variable `sum` in any way. Nevertheless, the *entire* statement [3] will be captured. Consequently, *all* other statements of the program that belong in the slice of statement [3] (relative to variable `sum`) *will* unnecessarily be included in the slice of `a`[10]! Notice that this inaccuracy would have been propagated further upwards if statement [2] was of the form `sum=u+v;`. This shortcoming is a direct result of the fact that when the node corresponding to statement [3] is encountered during slicing, *all* flow edges are followed backwards.

We can avoid this inaccuracy and consequently obtain more precise (‘‘smaller’’ slices) by modifying the system dependence graph. So far, the nodes of the SDG are ‘‘resolved’’ at the statement level. We map this SDG into a parse tree so that it does ‘‘resolve’’ at the token level; and, we *modify* accordingly the slicing algorithm in such a way so that it operates on the parse-tree-based SDG and calculates more precise slices. Consider the parse-tree-based SDG of the Figure 7 counterpart of the statement-based SDG of Figure 6 which corresponds to the code fragment

[10] Notwithstanding the fact that all reaching definitions of the variable `sum` are killed at statement [2] of this example.

of Table 3(i).

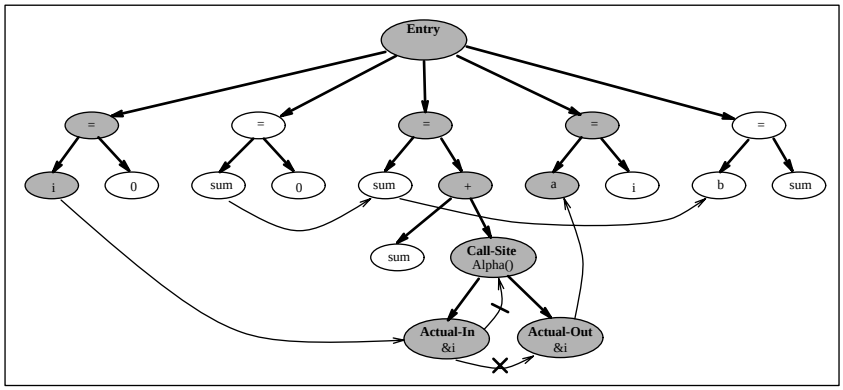

Figure 7. The parse-tree-based SDG.

Instead of slicing at *statement* `a = i`, we now slice at the *assignment* of `a` at that statement. The shaded nodes show the effect of slicing at the node containing the variable `a`. The marked nodes of the SDG are then *interpreted* to calculate the correct slice. Specifically, if the node containing the assignment variable is marked, then the entire statement is captured as being in the slice (except stripped parameters). Otherwise, only the marked call-sites are considered as being in the slice. Notice that no node of this new SDG corresponding to the variable `sum` has been marked; therefore, the slice so obtained is more precise than the slices supplied by any other method[11].

6. Building the System Dependence Graph

Let F^k be a procedure of a program P. We will say that F^k is *solved*, if and only if, all data dependences and control dependences have been computed. We will say that the procedure has been *summarized*, if and only if, all summary dependences have been calculated. On the other hand, determination of the summary information of F^k requires that the procedure be solved. The method that is proposed in [Hor90] for the calculation of the transitive dependences distinguishes between grammars that do not support recursion and those that do. In the former case, the solution proposed is via the use of a *separate* copy of a procedure dependence graph for each call-site. In the latter case, the solution requires the construction of an attribute grammar and the calculation of the corresponding subordinate characteristic graphs of the linkage grammar's nonterminals to determine the transitive dependences. Furthermore, in either case the GMOD and GREF sets must be calculated *before* solution of the dependences is initiated.

In this section we will describe a method that permits one to solve *all* procedures including the *construction* of the SDG in a bottom-up fashion and so that *only one* copy of a procedure dependence graph is required for all sites[12]. Hence, the advantages our algorithm are that it is conceptually simpler; there is no need to build an attribute grammar or calculate the corresponding subordinate graphs for the determination of the transitive dependences; actual-out nodes that are deemed *N*-nodes are identified as such *during* the SDG construction and dependence

[11] We should note that the mapping from the source file to our parse-tree-based SDG described in Section 10 permits us to display the *entire* statement [3] to the user as a member of the slice.

[12] Notwithstanding the fact that in the case of aliasing phenomena each aliasing pattern gives rise to a distinct procedure dependence graph.

calculation that makes calculation of the GMOD and GREF sets of all procedures unnecessary. Finally, to repeat, the algorithm operates on a parse-tree-based SDG that yields smaller slices.

The Algorithm

Our method is an extension of the method that we discussed in [Liv92] which handled the case where no recursive procedures were present. Specifically, we noted that with the use of our algorithm, each procedure is solved as soon as it is encountered. In other words, our method does not need to ''know'' whether a procedure F^k is terminal (i.e., does not contain static calls to any procedure) or not. If F^k is terminal, then it can be solved with no interruptions. If it is not, then upon encountering a call to a procedure F^l, calculation of the dependences of procedure F^k is suspended; the *partial solution* of F^k (denoted by $\partial\sigma F^k$) obtained up to this point is preserved; and, dependence calculation is initiated at the called procedure F^l. This process is continued until a procedure F^τ is encountered that is either terminal or has already been solved. In the former case, the terminal procedure is solved. At any rate, F^τ's summary information σF^τ is ''reflected'' back onto its corresponding calling site in the form of edges and we write $\rho\sigma F^\tau$ to indicate this operation of reflection. Calculation of the dependences of the *calling* procedure is then resumed. It should be noted that once a procedure has been summarized, there is no reason to descend into the procedure again; subsequent calls to a summarized procedure need only have the summary information edges reflected (i.e., copied) to the call-site.

The method just previously discussed will be illustrated by considering a program *abstraction* consisting of four procedures *M, A, B,* and *C* with calls as indicated in Table 4. For example, the procedure *M* calls procedures *A* and *C*; procedure *A* calls procedure *B*; procedure *B* calls procedure *C*; whereas *C* is a terminal procedure.

$M \rightarrow A\ C$
$A \rightarrow B$
$B \rightarrow C$
$C \rightarrow \wedge$

Table 4. Program Abstraction.

We begin by descending[13] into the main procedure (procedure *M*). The first procedure encountered is procedure *A*; the partial solution of M $\partial\sigma M$ is saved. Since procedure *A* has not been solved, we descend into it and begin its solution. During this process the call to procedure *B* is encountered; a partial solution of *B* is retained and we descend into the unsolved procedure *B*. Again, procedure *B* is not terminal and its partial solution is saved before we descend into procedure *C*. But, *C* is terminal, and therefore its solution σC can be obtained. The steps outlined so far can be encapsulated in equation (6.a)

$$\partial\sigma M \rightarrow \quad \partial\sigma A \rightarrow \quad \partial\sigma B \rightarrow \quad \sigma C \tag{6.a}$$

The summary of *C* is reflected onto its call site (in *B*); and, the partial solution of *B* is updated to $\partial^2\sigma B = \partial\sigma B \bigcup \rho\sigma C$ and (6.a) becomes

$$\partial\sigma M \rightarrow \quad \partial\sigma A \rightarrow \quad \partial^2\sigma B \tag{6.b}$$

Similarly, *B* can now be solved (since it contains no more static calls); its summary is reflected back to its caller and (6.b) becomes

$$\partial\sigma M \rightarrow \quad \partial^2\sigma A \tag{6.c}$$

FCA now can be solved and by letting $\partial^2\sigma M = \partial\sigma M \bigcup \rho\sigma A$ (6.c) becomes

[13] By *descend*, we mean to begin processing the procedure at the entry node.

$$\partial^2 \sigma M \tag{6.d}$$

Solution of *M* is resumed until the call to *C* is encountered. But *C* has been solved; therefore its summary is reflected and we obtain

$$\partial^3 \sigma M = \partial^2 \sigma M \bigcup \rho \sigma C$$

Now *M* can be solved since it contains no additional calls; σM is obtained as was desired. We should note that (in absence of recursion), we keep track of the call sites within the partially solved procedures via a singly linked list that we refer to as the *call sequence graph (CSG)*. The CSG is a dynamic structure where a procedure is pushed whenever it is invoked and popped when ever it is solved. Naturally its summary information is reflected to the call site of the procedure that is represented by the top of the CSG. For example, Figure 8 represents the CSG at the point which corresponds to equation (6.a). Whenever solution of procedure *C* is completed, the CSG is popped and *C*'s summary information is reflected in the appropriate call-site in procedure *B* (the new top of the stack). The process is repeated until the CSG becomes empty and the SDG has thereby been built.

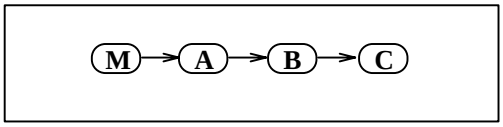

Figure 8. Call Sequence Graph.

The algorithm just described does not work well when recursive procedures are present. The reason is that in the absence of recursion, it is guaranteed that a terminal procedure will be encountered that can be completely solved, and its summary information can be reflected to its caller. In the case of recursion even if we process a procedure in its entirety, the summary information that will be obtained may be incomplete; therefore a number of dependences may not be found. To counter this problem, the algorithm described above is modified as follows.

The *extended call sequence graph (ECSG)* is employed to detect (*on the fly*) when a recursive procedure has been encountered and to also keep track of the set of procedures (as in the case of mutual recursion) that must be iterated over. An ECSG, Ω, is a dynamic multilist based on the CSG of the form $\Omega = \{\omega_{i,j} : 0 \le i \le n_j, 0 \le j \le m\}$ just previously discussed. The *backbone* of Ω is nothing more than the CSG itself defined by $\{\omega_{0,j} : 0 \le j \le m\}$. Associated with each node in the backbone, $\omega_{0,l}$, there is a list of procedures $\{\omega_{k,l} : 1 \le k \le n_l\}$ referred to as the *iterate* list *rooted* at $h=\omega_{0,l}$. By definition, if an iterate list is not empty, then no procedure in the list appears in that list more than once; and, as we will see, these are the procedures over which iteration must take place. As an example, in Figure 9 one can see three possible ECSG's; the backbone in each case is the ''horizontal'' list (consisting of procedures *M*, *A*, *B*, and *C*). Furthermore, in (i), all iterate lists are empty; whereas in (ii) and (iii), there is a non-empty iterate list with root nodes *C* and *A*, respectively.

The insert operation on the ECSG differs from that of the CSG. Specifically, whenever a call to an unsolved procedure *U* is encountered during the solution of a procedure *V*, a search in ''column-major'' order is performed on the ECSG, Ω, starting from $\omega_{0,0}$ to determine if a node ω_{i_k,j_l} labeled with that procedure's name (*U*) is encountered in Ω.

If no such node is found, the new procedure is inserted at the tail of the backbone; the iterate list of this procedure is set to *empty*; and, calculation will proceed as normal by preserving $\partial \sigma V$ and descending into *U*.

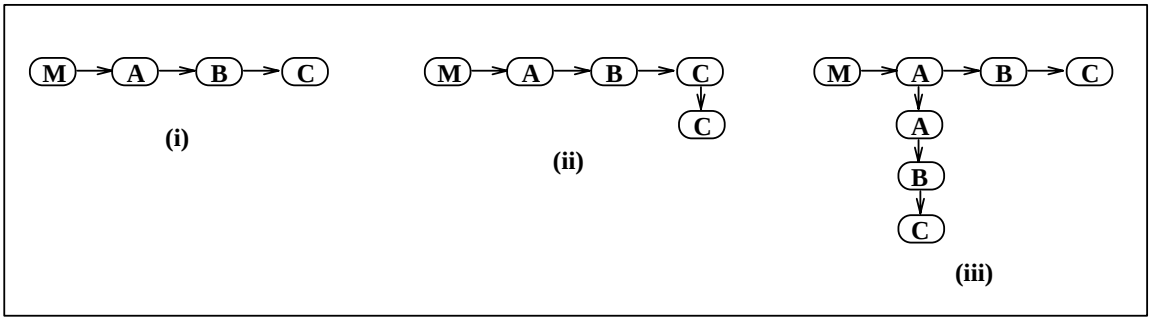

Figure 9. Extended Call Sequence Graph.

On the other hand, if such a node *is* found, then recursion has been detected[14]. In that case, we modify the ECSG as follows. First, the iterate list, rooted at $h=\omega_{0,j_l}$ is expanded by copying its root into it as well as all the procedures that correspond to the nodes of Ω satisfying $\{\omega_{i,j}: 0=<i\le n_j, j_l<j\le m\}$. Second, all iterate lists, $\{\omega_{l,j}: 1\le i\le n_j\}$ with $j>j_l$ are deleted. Furthermore, the fact that the procedure *U* was found in ECSG suggests that either we have only partially descended into it or have completed a first pass through it; therefore, instead of descending[15] into procedure *U*, we reflect the partial summary of *U* into the corresponding call site in *V* and resume solution of *V*. One example, assuming the use of ECSG in Figure 9(ii), a call from *C* to procedure *A* would yield the ECSG of Figure 9(ii).

Finally, a procedure *V* is deleted from the backbone, if and only if, the *entire procedure* has been processed. At the same time an intra-slice is performed and the summary[16] information that is obtained is reflected to its (known) call sites. Moreover, if the iterate list rooted at *V* is not empty, then this is a signal that iteration should be performed over the procedures in the iterate list.

Initially, the summary information calculated for a procedure in the iterate list is incomplete. We term this incomplete information a *partial summary*. The main concept of the iteration algorithm is that as the algorithm iterates over each procedure in the iterate list, this partial information is reflected onto the call-sites, which in turn, is used in the calculation of subsequent partial summaries. Eventually, when no new dependencies are found, this partial summary becomes a complete summary.

An iteration over a procedure is defined as follows. We descend into the procedure and calculate the dependencies as normal, *except* that as call-sites are encountered, only the (possibly partial) summary information is reflected onto the call-site; no descents are made from the procedure. When the procedure has been processed, the summary information is calculated and reflected to all known (encountered) call-sites of the procedure. It should be noted here that the correct calculation of dependencies requires that when partial dependencies (in which the effects of actual-out variables are *unknown* are involved), the reaching definitions for those actual-out variables are *killed*. Of course, the classification of the actual-out nodes will change as the partial summary becomes more complete.

---

[14] Although recursion has been detected, the extent (the procedures involved in the recursion) has not yet been determined. As the extent of recursion is determined, the root may change.

[15] Additional descents will be made at the time of iteration.

[16] It is important to note that if the procedure deleted is not a terminal procedure, its summary will be partial (i.e., incomplete).

This iteration is performed over the set of procedures contained within the iterate list until no changes to the calculated dependencies of the set are found. At this point, the procedures in the iterate list are solved.

As an example consider the call sequence in Table 5.

$M \rightarrow A\ B$
$A \rightarrow B\ D$
$B \rightarrow C\ E$
$C \rightarrow C\ A$
$D \rightarrow \wedge$
$E \rightarrow F$
$F \rightarrow \wedge$

Table 5. Program Abstraction.

We begin by descending into the main procedure (procedure *M*). The first procedure encountered is the unsolved procedure *A* which does not exist in the backbone; hence, $\partial\sigma M$ is saved, we insert *A* into the backbone, and descend into it. When a call to the unsolved procedure *B* is encountered, the backbone is searched; but, the procedure is not found. So *B* is inserted there, the partial solution $\partial\sigma A$ is preserved, and we descend into *B*. Similarly, when we encounter the call to the unsolved procedure *C*, we suspend solution of *B*, save $\partial\sigma B$, insert *C* into the backbone, and descend into it. Figure 9(i) illustrates the status of the ECSG (merely the backbone) up to this point whereas equation (6.e) indicates the solution steps so far.

$$\partial\sigma M \rightarrow \quad \partial\sigma A \rightarrow \quad \partial\sigma B \rightarrow \quad \partial\sigma C \tag{6.e}$$

During the solution of *C* a call to *C* is encountered; *C* exists in the backbone so we reflect the partial summary to the call-site (in this case the partial summary is empty) $\partial^2\sigma C = \partial\sigma C \bigcup \rho\partial\Sigma C$, then the set of nodes of the ECSG from *C* to the end of the list are copied and appended as an iterate list at *C* (Figure 9(ii). Processing of *C* is continued until the call to procedure *A* is encountered. A search of the ECSG reveals that *A* exists; hence, the partial summary $\partial\Sigma A$ (in this case the partial summary is also empty) is reflected to its call-site in *C*, i.e, $\partial^3\sigma C = \partial^2\sigma C \bigcup \rho\partial\Sigma A$; and, the iterate list rooted at *A* is updated (Figure 9 (iii)). Since we did not descend, processing of *C* continues until its end is encountered in which case it is marked as having been visited[17], its (partial) summary information ($\rho\partial^2\Sigma C$) is calculated; and, this summary information is reflected to the call site in *B*. This yields $\partial^2\sigma B = \partial\sigma B \bigcup \rho\partial^2\Sigma C$. *C* is then deleted from the backbone and processing returns to procedure *B* as indicated by the tail of the backbone (Figure 10(i)). *B* now calls *E* which in turn calls *F* as indicated by Figure 10(ii).

In symbols (6.e) has yielded (6.f)

$$\partial\sigma M \rightarrow \quad \partial\sigma A \rightarrow \quad \partial^2\sigma B \rightarrow \quad \partial\sigma E \rightarrow \quad \sigma F \tag{6.f}$$

Now *F*, being terminal and solved, can be reflected, (i.e., $\partial^2\sigma E = \partial\sigma E \bigcup \rho\Sigma F$); and, now *E* can be solved. Its summary is reflected via $\partial^3\sigma B = \partial^2\sigma B \bigcup \rho\Sigma E$.

The partial summary $\partial\Sigma B$ is calculated and reflected into *A*; in symbols, $\partial^2\sigma A = \partial\sigma A \bigcup \rho\partial\Sigma B$. Notice that as a consequence of the previous steps, equation (6.g) has become

$$\partial\sigma M \rightarrow \quad \partial^2\sigma A \tag{6.g}$$

[17] If recursion is not present, marking the procedure denotes the procedure as solved. A marked procedure will not be descended into, only its summary information need be reflected. In the case of recursion, marking the function only denotes that the procedure has, at best, a partial solution. However, the mark prevents the procedure from being descended into until the iteration stage.

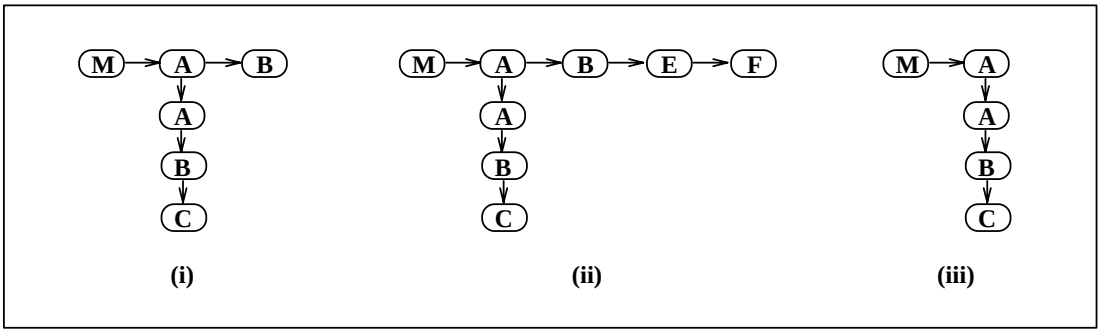

Figure 10. Extended Call Sequence Graph.

Figure 10(iii) shows the state of the graph up to this point.

Now, during processing of *A*, a call to *D* is encountered. The solution of *A* is suspended and we descend into *D* (Figure 11(i)).

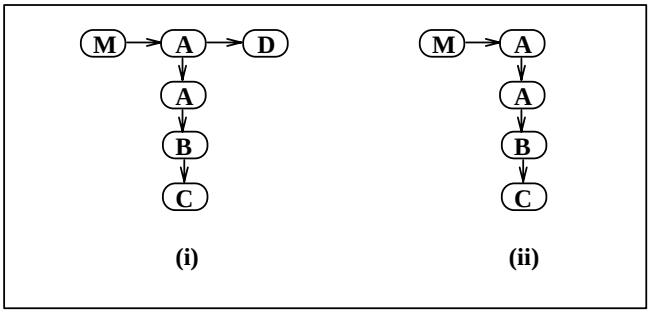

Figure 11. Extended Call Sequence Graph.

But, *D* being terminal is solved; its node is deleted from the backbone (Figure 11(ii)); and, its summary is calculated and reflected to its corresponding call site in *A* via $\partial^3\sigma A = \partial^2\sigma A \bigcup \rho\Sigma D$. Processing resumes with procedure *A*. When the end of *A* is encountered, it is marked as visited. When the end of *A* is reached, *A* is deleted from the backbone. But since the iterate list rooted at *A* was not empty, iteration over the union of the procedures of that iterate list is necessary; in symbols, $\iota(A \bigcup B \bigcup C)$. Notice that when the iteration has been completed, the *complete* summary of all procedures in the iterate list will have been obtained. In other words,

$$\iota(A \bigcup B \bigcup C) \rightarrow \sigma A \bigcup \sigma B \bigcup \sigma C$$

Once the recursion is solved, we return to finish procedure *M*. After calling *A*, *M* calls procedure *B*. However, since procedure *B* has already been solved, we are finished.

$$\partial^2\sigma M \xleftarrow{\rho\Sigma A}$$

$$\partial^3\sigma M \xleftarrow{\rho\Sigma B}$$

$$\rightarrow \quad \sigma M$$

Where ι is defined as an iteration on a set of procedures (σ, $\partial\sigma$, and ρ are defined in the previous example).

An Example

We will now give an example of how we build our System Dependence Graph. Consider the recursive program in Table 6.

```
[ 1]. void main()          [10]. void R(int *x, int *y)
[ 2]. {                    [11]. {
[ 3].     int x,y;         [12].     if (*y == 0)
[ 4].                      [13].         *x = *x + 1;
[ 5].     R(&x,&y);        [14].     else if (*y == 1) {
[ 6].                      [15].         *y = *y + *x;
[ 7].     x = x;           [16].         R(x,y);
[ 8].     y = y;           [17].         *x = *x + 1;
[ 9]. }                    [18].     }
                           [19].     else {
                           [20].         *x = *x - 1;
                           [21].         *y = *y - 1;
                           [22].         R(x,y);
                           [23].     }
                           [24]. }
```

Table 6. A sample program.

We begin by first building a parse tree representation[18] with the edges corresponding to control flow edges (Figure 12).

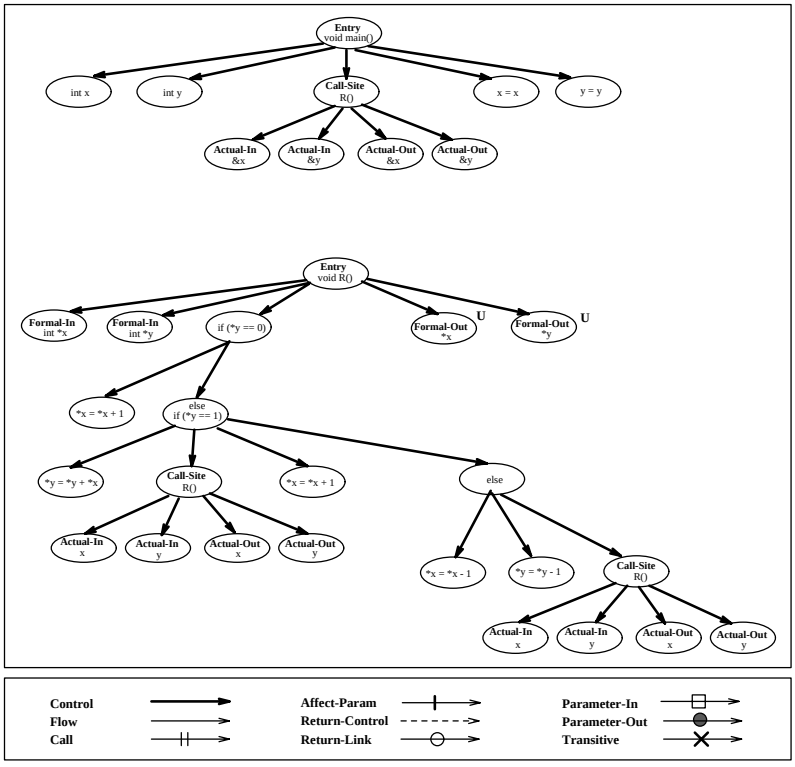

Figure 12. Construction of the System Dependence Graph.

[18] In order to simplify the figures, the SDG is depicted on the statement level.

Note that initially the formal-out nodes of `R` are marked as unknown. The first procedure to be executed in C is `main`; similarly, we begin by descending into `main`. In general, each statement is processed in a top-down, left-to-right fashion. However, contents of the looping statements are processed twice. At statement [5], we descend into procedure `R` (since it has not yet been summarized. If it were summarized, we would merely reflect the transitive summary.). We then process statements [10],[12],[13], and [15]. At statement [16], procedure `R` is already in the call sequence graph, so we do not descend. At this point, the partial summary is empty. However, we kill the reaching definitions for the variables contained in the actual-out nodes. At statement [17] since all the reaching definitions for variable `*x` have been killed, no flow edges are connected. We continue by processing statements [19],[20],[21], and [22]. As at statement [16], the partial summary is empty; the reaching definitions for the variables contained in the actual-out nodes are killed. Figure 13[19] shows the state of the SDG immediately before our first partial summary is calculated.

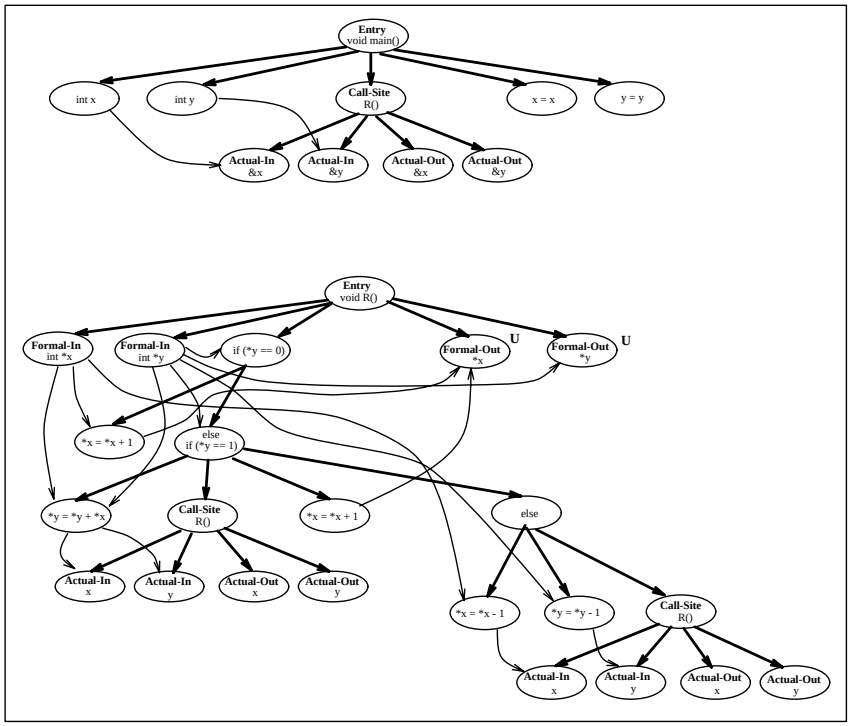

Figure 13. Construction of the System Dependence Graph.

Figure 14 shows the SDG after calculation of the (partial) summary information and after is has been reflected to all the call sites encountered so far (in this example, all the call sites have been encountered). Notice the formal-out node for variable `x` is now classified as *always modified* and the formal-out node for variable `y` is classified as *never modified*.

The iterate list for procedure `R` is `R`. Figure 15 shows the SDG after one iteration (following the initial pass). Notice that the classification of variable `y` has changed to *sometimes modified*. Additionally, new transitive edges have been added to the procedure's (partial) summary. Since the summary information has changed, we iterate again.

---

[19] To keep the graph from becoming ''busier'' than it already is, we have neglected to show declaration and interprocedural edges.

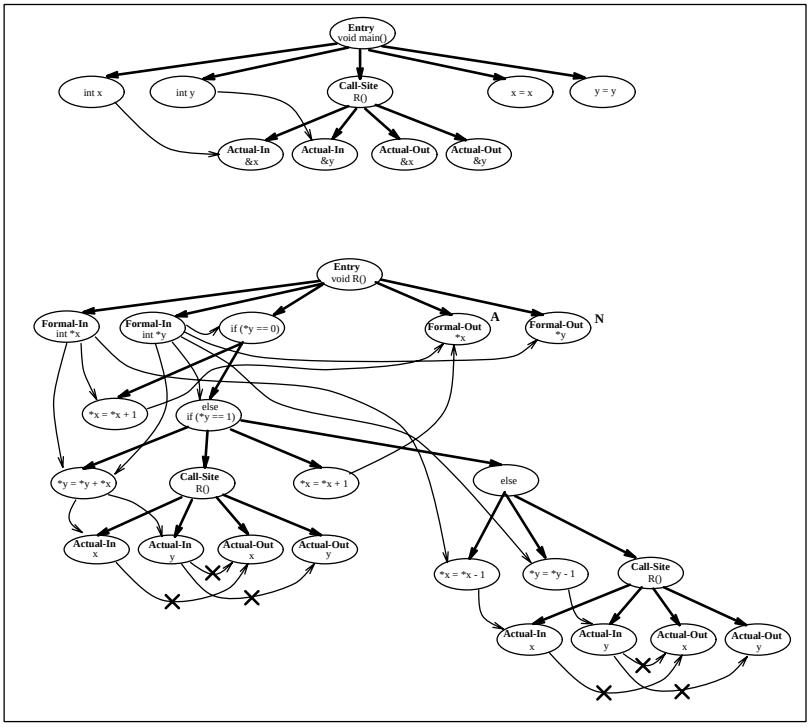

Figure 14. Construction of the System Dependence Graph.

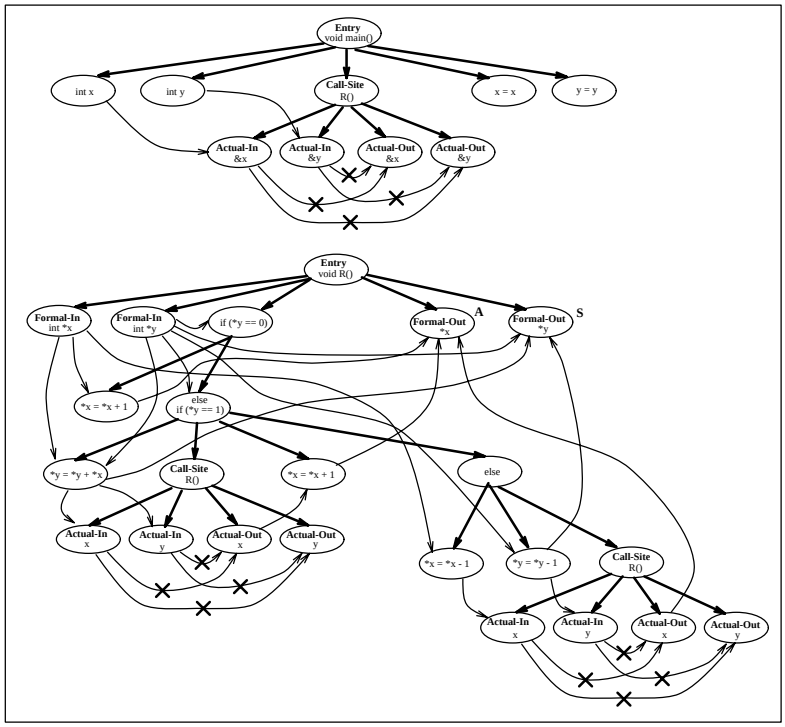

Figure 15. Construction of the System Dependence Graph.

Figure 16 shows the SDG after the second iteration has been completed. Another iteration is required because additional flow edges were added incident to the formal-out nodes. No additions were made during this third and final iteration and we ascend from the procedure.

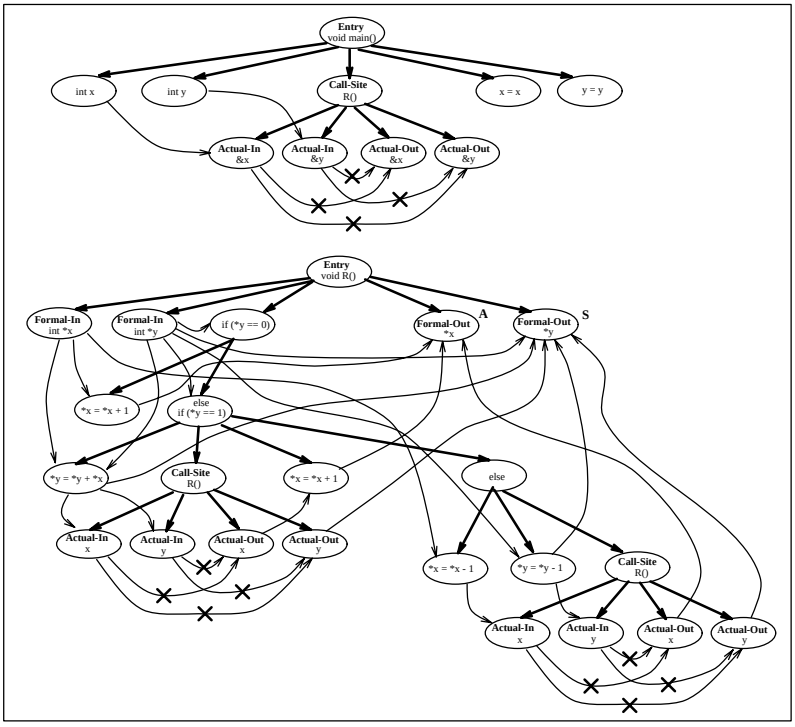

Figure 16. Construction of the System Dependence Graph.

Figure 17 shows the complete SDG whereas the algorithm for calculating interprocedural dependencies is presented in Table 7.

6.1. Recursion and Aliasing

If aliasing is present in a recursive procedure, the possibility exists that several alias configurations may be "spawned" as a result of the dependency calculation. Consider the procedure fragment in Table 8. This procedure (when called without aliased parameters) will "spawn" three distinct aliased configurations. They are: `R.(1,1,3)`, `R.(1,2,2)` and `R.(1,1,1)`. This does not present a problem since each alias configuration gives rise to a distinctly named function. In this case the algorithm will iterate over the set of four procedures (the non-aliased configuration as well as the aliased ones).

7. Calculating Reaching Definitions Using the SDG

Another application for which the SDG can be used is to calculate reaching definitions. Finding reaching definitions can be thought of as a restricted form of slicing; i.e., computing a slice of only one "iteration" backwards. Intuitively, each flow edge is followed backward from the target node and the nodes that have been reached are marked as being in the reaching definition set. This works for an intraprocedural case. For the interprocedural case, we would like to identify definitions that span one or more procedure boundaries. For this to occur, we must take into account both the passing of variables by reference and their subsequent "return" and the actual *return statement* mechanism. The interprocedural algorithm is shown in Table 9. Note that this algorithm requires two phases; it is similar to the slicing algorithm. The screen dump that was

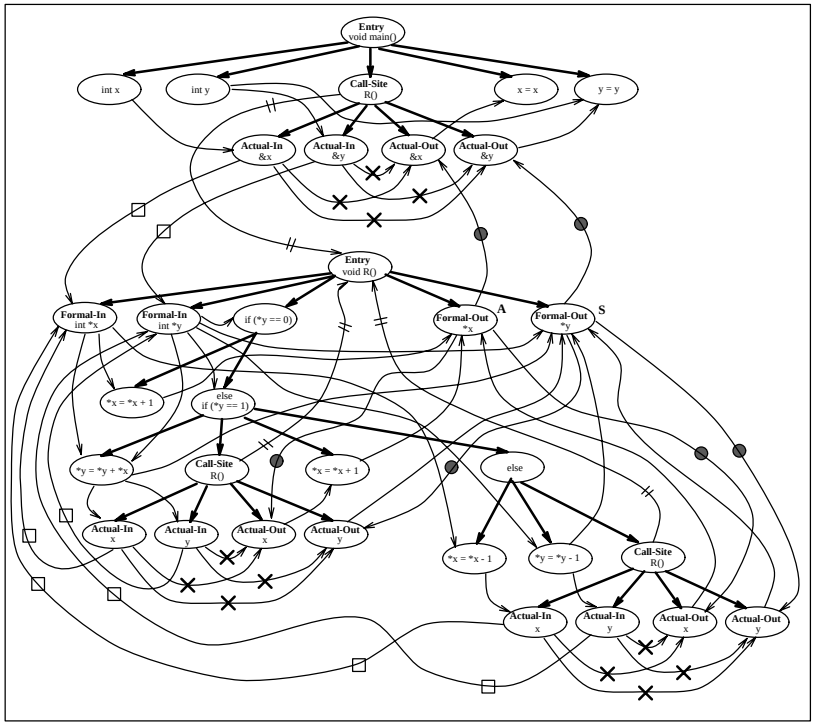

Figure 17. Construction of the System Dependence Graph.

provided from our Ghinsu tool in Figure 18 illustrates the result of finding the reaching definitions relative to the statement *sum = sum*.

8. The Interprocedural Ripple Analysis Algorithm

Ripple analysis is slicing in the forward direction. Whereas a slice relative to a particular variable in a particular statement is the set of all statements that may affect the value of the variable, ripple analysis will show the potential effect of changing a variable at a selected statement.

Like slicing, ripple analysis is accomplished in two phases. The first phase consists of a traversal of a particular set of edges starting at a selected node. In the second phase, traversal of a different set of edges is applied to each node visited during the first phase. The union of the nodes visited in both phases is the interprocedural slice. In general, all edges are (recursively) followed *forward*. The edges followed in the first phase are control, data flow, declaration, return-control, parameter-in, transitive, affect-param, and call edges. The second phase follows the control, data flow, declaration, return-control, parameter-out, transitive, affect-param, and return-link edges. These are the same sets of edges followed in slicing, but in the forward direction. Note that in forward slicing, there is no need for the "short circuit" operation when following return-control edges.

9. Dicing and Additional Tools

Dicing is an automatic bug location heuristic. As proposed in [Lyl87], a slice is generated from an incorrectly computed variable at a particular statement. If there exists another variable that is computed correctly, then the dicing heuristic may be employed. The bug is likely to be associated with the slice on the incorrectly computed variable minus those associated with the slice on the correctly computed variable. Dicing can be used iteratively to locate a program bug.

```
global IterateFlag, change;

main()
{
    Construct parse tree;
    Rename global and static variables;
    Descend(main);
}

Descend(function)
{
    OpenScope(function);
    // this includes accounting for aliased parameters
    Process formal parameters and create Formal-Out nodes for each pass-by-reference parameter;
    Process local variables;

    for each (statement in function) {
        case (statement) {
            assignment-statement:   CalcAssignment(statement);
            return-statement:       Calc-Return(statement);
            if-statement:           Calc-If(statement);
            while-statement:        Calc-While(statement);
        }
    }
    Link Flow edges based on Formal-Out variables;
    CloseScope(function);
}

CalcAssignment(statement)
{
    Parse statement to determine defined variable and used variables;

    for each (function call in statement) {
       if ((function is an alias configuration) AND (the alias configuration does not exist in the SDG))
            Create a distinct function for the alias configuration;

        if (IterateFlag == FALSE)
            CalcFunction(function,callsite);
        for each (Actual-Out node) {
            Update reaching definition table (based on Sometimes/Always/Never/Unknown information);
            Reflect summary information to the callsite;
        }
    }

    Link Flow and Declaration edges based on defined and used variables;
    Update reaching definition table (based on defined variables);
    Link Return-Control edges based on "reaching" return definitions;
}

CalcFunction(function,callsite)
{
    found = AddToCallSequence(function);

    Link Call edge (if one does not already exist);

    if ((found == FALSE) AND (function is NOT marked)) {
        Descend(function);
        ComputeSummaryInformation(function);

        // call sites are 'known' by following call edges backwards
        Reflect summary information to all (currently known) call sites;
        Mark function;

       Iterate();
    }

    if (found == FALSE)
        DeleteFromCallSequence(function);
}
```

Table 7(contnd).

The implementation of the dicing algorithm is straightforward. The dice is computed in two phases as in calculation of a slice. The only difference is that the action of the slicing algorithm is *reversed.* Instead of marking nodes as being contained in the slice, the encountered nodes are marked as *not* being in the slice.

```
int AddToCallSequence(function)
{
    // Attempt to locate the function in the calling sequence starting from the head of the backbone.

    root = head(ECSG);
    while (root != NULL) {
       if ((function is found in root) OR (function is found in root's iterate list))
            break;
       root = next_backbone_node(root);
    }

    if (root == NULL) { // not in the list
        add function to the tail of the list;
        return FALSE;
    }

    Move all elements of the iterate lists between (and including) root and the tail to
    root's iterate list;
    Make a copy of all elements in the backbone between (and including) root and the tail
    and add to root's iterate list;

    return TRUE;  // found in the list
}

DeleteFromCallSequence(function)
{
    if (function is the tail node in the call sequence graph)
        Remove the tail node;
}

Iterate()
{
    IterateFlag = TRUE;
    do {
        change = FALSE;
        for each (function in iterate list) {
            Descend(function);
            ComputeSummaryInformation(function,change);
            Reflect summary information to all (currently known) call sites;
        }
    } while (change == FALSE);
}

CalcReturn(statement)
{
    Parse statement to determine used variables;
    Link Flow edges based on used variables;
    Link Return-Control edges based on "reaching" return statements;
    // if there are any changes in the function's return summary, set change = TRUE
    IntraSlice, add all marked Formal-In nodes to function's return summary;
    Link Flow edges based on Formal-Out variables;
    Update "reaching" return definition table;
    ComputeSummaryInformation(function);
    Reflect summary information to all call sites;
    Kill all reaching definitions;
}

ReflectSummaryInformation(function)
{
    Connect Parameter-In, Parameter-Out, and Return-Link Edges;
    Connect Affect-Param Edges based on function's return summary;
    Connect Transitive Edges based on function's transitive summary;
}

ComputeSummaryInformation(function,change)
{
    for each (Formal-Out node in function) {
        // if there are any changes in the transitive summary, set change = TRUE
        IntraSlice;
        Associate all Formal-In nodes in the slice with the Formal-Out node and add to function's transitive summary;
        Classify Formal-Out node as Unknown, Never, Sometimes, or Always modified and add to transitive summary.
    }
}
```

Table 7. The Algorithm for Calculating Interprocedural Dependencies.

```
void R(int *x, int *y, int *z)
{
    if (-)
        ---;
    else if (-) {
        R(x,x,z);
        ---;
        R(x,y,y);
    }
}
```

Table 8. A procedure fragment.

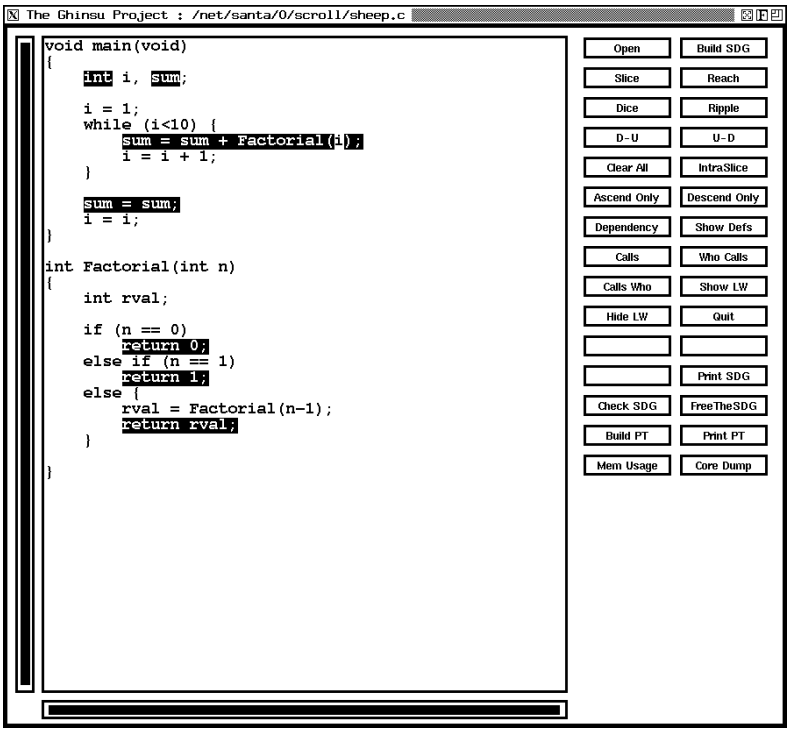

Figure 18. The Ghinsu tool. The reaching definitions relative to the statement *sum = sum* are shown.

The SDG is a versatile internal representation. Additional tools can be constructed that can extract useful information from the SDG. Some of the minor functions or tools that have been developed for use in the Ghinsu system are described below.

Dependency Analysis of a variable at a particular statement produces a list of variables that the selected variable is dependent upon. Additional information about each variable in the list is also displayed. This information is extracted from the slice of the variable.

An *intraprocedural slice* can be used when the maintainer wishes to limit his/her view to the scope of only one function.

The *ascend only* tool allows the maintainer to limit the slice to only the function selected and the functions that call the selected function. This operation corresponds to slicing phase one only. Correspondingly, the *descend only* tool allows the maintainer to limit the slice to only the function selected and the functions called by the selected function. This operation corresponds to slicing phase two only.

```
ReachPhase1(node)
{

    if (node is marked)
        return;

    MarkNodeInReachingSet(traverse);

    for each (Parameter-In edge connected to traverse) {
        ReachPhase1(FollowParameterInEdge());
    }

    for each (Flow edge connected to node) {
        traverse = FollowFlowEdge();
        MarkNodeInReachingSet(traverse);

        for each (Parameter-In edge connected to traverse) {
            ReachPhase1(FollowParameterInEdge());
        }
    }

    for each (Affect-Parameter edge connected to traverse) {
        traverse = FollowAffectParameterEdge();
        MarkNodeInReachingSet(traverse);

        for each (Parameter-In edge connected to traverse) {
            ReachPhase1(FollowParameterInEdge());
        }
    }
}

ReachPhase2(node)
{
    if (node is marked)
        return;

    MarkNodeInReachingSet(traverse);

    for each (Parameter-Out edge connected to traverse) {
        ReachPhase2(FollowParameterOutEdge());
    }

    for each (Return-Link edge connected to traverse) {
        MarkNodeInReachingSet(FollowReturnLink());
    }

    for each (Flow edge connected to node) {
        traverse = FollowFlowEdge();
        MarkNodeInReachingSet(traverse);

        for each (Parameter-Out edge connected to traverse) {
            ReachPhase2(FollowParameterOutEdge());
        }

        for each (Return-Link edge connected to traverse) {
            MarkNodeInReachingSet(FollowReturnLink());
        }

    }

    for each (Affect-Parameter edge connected to traverse) {
        traverse = FollowAffectParameterEdge();
        MarkNodeInReachingSet(traverse);

        for each (Parameter-Out edge connected to traverse) {
            ReachPhase2(FollowParameterOutEdge());
        }

        for each (Return-Link edge connected to traverse) {
            MarkNodeInReachingSet(FollowReturnLink());
        }

    }
}
```

Table 9. Algorithm for Calculating Reaching Definitions Using the SDG .

Show definitions displays all the definitions of a particular variable.

10. Related Work

Weiser[Wei84] has built slicers for FORTRAN and an abstract data language called Simple-D. His slices were based on flow-graph representation of programs. As far as we know, no other operational slicers have been built. In addition, Weiser's method does not produce an optimum slice across procedure calls because it cannot keep track of the calling context of a called procedure. Methods for more precise interprocedural slicing have been developed by Horwitz [Hor88] where parameters are passed by value-result. This is an extension of the program dependence graph presented in [Fer87]. However, this models a simple language that supports scalar variables, assignment statements, conditional statements, and while loops.

The dependence graph developed by Horwitz differentiates between loop-independent and loop-carried flow dependency edges. Our method treats these as a single type of edge -- the data flow edge -- which simplifies construction of the program dependence graph.

Our method of calculating interprocedural dependences does not use linkage grammar as used in Horwitz's algorithm[Hor90]. Our algorithm is conceptually much simpler. The linkage grammar utilized by Horwitz includes one nonterminal and one production for each procedure in the system. The attributes in the linkage grammar correspond to the input and output parameters of the procedures. After constructing the linkage grammar, the algorithm determines the procedure which does not call any other procedure and calculates its transitive dependencies and reflects them to other procedures. Our method descends to the called procedures in the order of their call in the program. When a called procedure does not call any other procedure, its transitive dependencies are reflected on the other procedures which called this procedure. Recursion is handled by a method of iteration over the recursive procedure(s). The called procedure always returns to the correct address in the calling procedure. This completely eliminates the use of linkage grammar and construction of subordinate characteristic graphs which makes our algorithm more efficient.

Harrold, et. al., [Har89] calculate interprocedural data dependencies in the context of interprocedural data flow testing. Their algorithm requires an invocation ordering of the procedures. Additionally, when recursive procedures are present, processing may visit each node p times where p is the number of procedures in the program. As above, we do not need to calculate an invocation ordering. Also, we need to iterate over only the recursive procedures, not the entire program.

Agrawal[Agr89] has provided algorithms for intraprocedural dynamic slicing. The Ghinsu tool, however, uses the concept of static slicing. The main disadvantage of dynamic slicing is that the program dependence graph depends upon the test data, i.e. only those portions of the graph will be built which fall on the execution path of the test data. In contrast, a static slice is independent of the test data. A dynamic slice can be considered to be a subset of a static slice. A technique for handling slices for recursive procedures has been suggested by Hwang [Hwa88] which constructs a sequence of slices of the system - where each slice of the sequence essentially permits only one additional level of recursion - until a fixed point is reached. Moreover, this algorithm solves only self-recursive procedures and has no mechanism for handling mutually recursive procedures.

11. Future Work

We intend to investigate the problems of pointer variables in the context of the internal representation that we employ. Since the aliasing problems with pointer variables in C are in general unsolvable, we are exploring user-guided approximations. Additionally, we intend to handle C-like, unstructured `goto` statements. This will require a flow-graph to be imposed on the parse

tree representation.

12. References

[Agr89] H. Agrawal. and J.R. Horgan. *''Dynamic Program Slicing''*, Technical Report SERC-TR-56-P, Software Engineering Research Center, Computer Science Dept., Purdue University.

[Aho74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *''The Design and Analysis of Computer Algorithms''*, Addison-Wesley, Reading, MA.

[Aho85] A.V. Aho, R. Sethi, and J.D. Ullman. *''Compilers: Principles, Techniques and Tools''*, Addison-Wesley, Reading, MA.

[Bad88] L. Badger and M. Weiser. *''Minimizing Communications for Synchronizing Parallel Dataflow Programs''*, In Proceedings of the 1988 International Conference on Parallel Processing, Penn State University Press, PA.

[Ban79] 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.

[Boe75] B.W. Boehm. *''The High Cost of Software, Practical Strategies for Developing Large Software Systems''*, E. Horowitz (ed.). Reading, Mass: Addison-Wesley.

[Cal88] D. Callahan. *''The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis''*, In Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation, Atlanta Georgia, June 22-24, 1988.

[Fer87] J. Ferrante, K. Ottenstein, and J. Warren. *''The Program Dependence Graph and its Use in Optimization''*, ACM TOPLAS, July 1987.

[Har89] M. J. Harrold and M. L. Soffa. *''Selecting Data for Integration Testing*

[Hor88] S. Horwitz, J. Prins, and T. Reps. *''Integrating Non-interfering Versions of Programs''*, in Proceedings of the 15th ACM Symposium of Programming Languages, ACM Press, N. York.

[Hor89] S. Horwitz, J. Prins, and T. Reps. *''Integrating Non-interfering Versions of Programs''*, ACM TOPLAS, July 1989.

[Hor90] S. Horwitz, T. Reps, and D. Binkley. *''Interprocedural Slicing Using Dependence Graphs''*, ACM TOPLAS, January 1990.

[Hwa88] J.C. Hwang, M.W. Du, C.R. Chou. *''Finding Program Slices for Recursive Procedures''*, In Proceedings of the IEEE COMPSAC 88, IEEE Computer Society, 1988.

[Kas80] Kastens, U. *''Ordered Attribute Grammars''*. Acta Inf. 13,3, 1980.

[Ker88] B.W. Kernigham and D. M. Ritchie. *''The C Programming (ANSI C) Language''*, 2nd. Edition, Prentice Hall, Englewood Cliffs, New Jersey.

[Leu87] H.K.N. Leung and H.K. Reghbati. *''Comments on Program Slicing''*, IEEE Transactions on Software Engineering, Vol. Se-13 No. 12, December 1987.

[Liv91] P.E. Livadas and S. Croll. *''The C-Ghinsu Tool''*, Technical Report, Software Engineering Research Center, SERC-TR-55-F, December 1991.

[Lyl86] J.R. Lyle and M. Weiser. *''Experiments in Slicing-based Debugging Aids''*, In Elliot Soloway and Sitharama Iyengar, editors, Empirical Studies of Programmers, Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[Lyl87] J.R. Lyle and M. Weiser. *''Automatic Program Bug Location by Program Slicing''*, In Proccedings of the 2nd International Conference on Computers and Applications, June 1987.

[Ott84] K.J. Ottenstein and L.M. Ottenstein. *''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 Notices 19,5, May 1984.

[Par86] G. Parikh. *''Handbook of Software Maintenance''*, Wiley-Interscience, New York, New York 1986.

[Reps88] T. Reps and W. Yang. *''The Semantics of Program Slicing''*, TR-777, Computer Sciences Dept., University of Wisconsin, Madison, June 1988.

[Reps89] T. Reps and T. Bricker. *''Illustrating Interference in Interfering Versions of Programs''*, TR-827, Computer Sciences Dept., University of Wisconsin, Madison, March 1989.

[Wei81] M. Weiser. *''Program Slicing''*, In Proceedings of the Fifth International Conference on Software Engineering, San Diego, CA, March 1981.

[Wei82] M. Weiser. *''Programmers Use Slices When Debugging''*, CACM July 1982.

[Wei84] M. Weiser. *''Program Slicing, IEEE Transactions on Software Engineering, July 1984.*

[Yang89] W. Yang, S. Horwitz, and T. Reps. *''Detecting Program Components With Equivalent Behaviors''*, TR-840, Computer Sciences Dept., University of Wisconsin, Madison, June 1989.