An Optimal Algorithm for the Construction of the System Dependence Graph

Panos E. Livadas
Theodore Johnson

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

ABSTRACT

Program slicing can be used to aid in a variety of software maintenance activities including *code understanding*, *code testing*, *debugging*, and *program reengineering*. Program slicing (as well as other program analysis functions including forward slicing) can be efficiently performed on an internal program representation called a *system dependence graph* (SDG). The construction of the SDG depends primarily on the calculation of the transitive dependences which in turn depends in the calculation of the data dependences. In this paper we demonstrate the correctness and the optimality of our method of calculating the transitive data dependences. Furthermore, this method requires neither the (explicit) 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, a beneficial side effect of this method is that it provides us with a new method for performing *interprocedural*, flow-sensitive data flow analysis.

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, maintenance costs seem to be between two and four times higher than development costs for large embedded systems. Furthermore, according to internal conversations with our industrial affiliates, approximately 60% of their maintainer's time is spent *looking* at code. We therefore aim to reduce maintenance costs by developing tools that can assist in activities that focus on *understanding* code.

Program slicing provides one such useful tool for the maintenance programmer. Let *P* be a program, let *p* be a point in *P*, and let *v* be a variable of *P* that is either defined or used at *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 value of the variable *v* at the point *p*. This definition is less general than the one given in [Wei84]; but, it is sufficient [Hor90]. Program slices could be used in a variety of ways to aid in several software engineering activities. Weiser [Wei82] has shown that programmers use slices when debugging. Program slicing provides a meaningful way to decompose a large program into smaller components and can therefore aid in program understanding; and, can also be used in code reusability. *Dicing*, a method based on static slicing, can be used to aid in debugging by allowing certain program bugs to be *automatically* located [Lyl87]. 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. 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.

We noted in [Liv90] that the SDG provides us with a suitable form of internal program representation (IPR) that could be employed in the development of an integrated software maintenance environment. That is, an environment that integrates a number of tools that alone or in conjuction with one another would aid in various programn understanding and maintenance tasks.

Realizing the versatility of this IPR, we embarked in the development of a prototype that accepts programs written in a subset of ANSI C [Liv94a] including macro support[1] which generates an SDG [Croll94]. We have also implemented a number of tools such as a static slicer, a dicer, a forward slicer among others that can utilize this SDG [Liv94]. We should also note that the prototype incorporates the methods and algorithms discussed in this paper.

Despite that our prototype accepts C programs that may span multiple files and nearly every C construct in this paper we will restrict our grammar to a subset of ANSI C defined as follows. First, declarations of local, global, and static scalar 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 and can contain expressions that may include variables and are modeled after the `return` statements in the language C. Fourth, we distinguish between functions that return values as opposed to those that do not. Fifth, all C constructs are ''handled'' except `goto`, `break`, `continue`, and long jumps.

The main contribution of this paper is the presentation of a new 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[2]. There is no need to build an attribute grammar or calculate the corresponding subordinate graphs for the determination of the transitive dependences even in the presence of recursion; actual-out nodes that are deemed *N*-nodes are identified as such *during* the SDG construction and dependence calculation that makes the explicit calculation of the GMOD and GREF sets of all procedures unnecessary. Hence, as we show, our algorithm is not only conceptually simpler than other alghoritms but it is also optimal.

---

[1] The macro preprocessor is discussed in [Liv94b]

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

The remaining part of the paper is organized as follows. Section 2 describes the system dependence graph associated with our grammar. In section 3 we present the interprocedural slicing algorithm whereas in section 4 we provide with a classification of the formal parameters. Section 5 describes our algorithm and in section 6 we prove its the correctness and optimality. Finally, the handling of aliasing in the present of recursion is discussed in section 7, our prototype is briefly presented in section 8, related work is presented in section 9 and the paper concludes with section 9 where our current and future work is discussed.

2. The Program Dependence Graph and the System Dependence Graph

This section briefly describes the program dependence graph and the intraprocedural slicing algorithm, it follows with a discussion of the system dependence graph, and concludes by presenting the transformations that are employed in the presence of global and static variables and aliasing. We should note that the terminology and notation used in the sequel is the same as that in [Liv94] unless otherwise noted.

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 a declaration, an assignment statement, and a control predicate; there is also a special node called the *entry* node.

Edges represent several kinds of dependences among the nodes which can be distinguished by the label attached to them. Specifically, three dependences are distinguished[3]: *control*, *data flow*, and *declaration*, each of which will be briefly discussed below. 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. In this case, we write

$$v_1 \xrightarrow{cd} v_2$$

We note that every component of P that is not subordinate to any control predicate is control dependent on the entry node. Given the constructs[4] of the grammar under consideration, control dependences reflect the program's *nesting structure*.

We say that v_2 is *data flow dependent* on v_1, if and only if v_2 uses a variable α that v_1 defines[5], and there exists an execution path from v_1 to v_2 where variable α is not redefined. The above relationship can be denoted by using the following notation:

$$v_1 \xrightarrow{dd} v_2$$

Declaration dependences are considered as special kinds of data flow dependences that exist 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; and, we write:

$$v_1 \xrightarrow{de} v_2$$

---

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

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

[5] We say that v_1 defines a variable a if and only if execution of v_1 causes the memory location represented by a to be written (i.e. if $v_1 : a = b + 3$ then a is defined at v_1). A variable a is said to be used at a statement v_2 if and only if execution of v_2 causes the memory location represented by a is read. For example, b is used at statement v_1.

Let s_0 and w be two nodes in G_P. An *intraslice-path* from w to s_0 is a path on G_p denoted by $S_w^{s_0}$ and defined by

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

Let now s_0 be a node of G_P that defines a variable v. The slice[6] 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, the nodes of the slice are defined as follows

$$V(G_P/s_0) = \left\{ w \varepsilon V(G_P) : \exists S_w^{s_0} \right\}$$

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). 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[7] $0 \le k \le K = L$. The extension of the PDG to the 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. When a call to a function F^k is encountered, a *call-site node* is created that is denoted by $cs_j(F^k)$ (j is a nonnegative integer employed to enumerate the *static* calls to F^k). Then for each actual parameter a node, the *actual_in*, is created and an additional node, the *actual_out* node, is created for each actual parameter that is passed by reference. The sets of actual_in and actual_out nodes (corresponding to the call to F^k with i parameters are denoted by $a\_in_{i,j}(F^k)$ and $a\_out_{l,j}(F^k)$, where l is equal to the number of parameters of the function F^k that are passed-by-reference), are built. By definition, all such nodes are control dependent to the call-site node. In symbols,

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

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, two additional sets of nodes, referred to as the *formal-in* nodes ($f\_in_i(F^k)$) and the *formal-out* nodes ($f\_out_l(F^k)$) are built. 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 $f\_out_l(f^k)$. In addition:

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

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

---

[6] The definition that we use inn this paper is the following. 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 the value of v.

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

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

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,

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

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[8] from the formal-in node corresponding to the former node. Note that these edges may exist *only* between actual-in nodes and actual-out nodes. In other words, for each fixed k

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

where G_{F^k} denotes the procedure dependence graph of the function F^k.

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-return 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,

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

Two new types of edges, an intraprocedural edge (*return-control edge*) and an interprocedural edge (*return-link edge*) are needed to properly handle the `return` statements. We would like to note here that our method for determining control dependences is based on a syntax-directed method (hence we do not handle such constructs such as *gotos*). If another, more precise method was used (such as the one described in [Aho86], there would be no need to include return-control edges in the SDG.

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[9]:

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

A *return-link edge*[10] connects a return node to the corresponding function call-site. Specifically,

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

---

[8] The definition of reachability in the presence of procedures is more general than the one presented in the previous section and is given below.

[9] We note that the definition of return-control dependence does not coincide with the control dependence defined elsewhere. Our definition combined with the grammar that we employ makes computation of control dependences unnecessary.

[10] It should be noted that the return value could also be modeled as a parameter. We have chosen to use return-link edges in this paper.

Given now that our grammar allows call-by-reference parameters, `return` statements, as well as functions that may return values, we can define 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-return dependences, and return-link dependences. Therefore,

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

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

Figure 1 presents the system dependence graph of the program shown in Table 1. Declaration edges are not shown to keep the graph from becoming ''busier'' than it already is.

```
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.

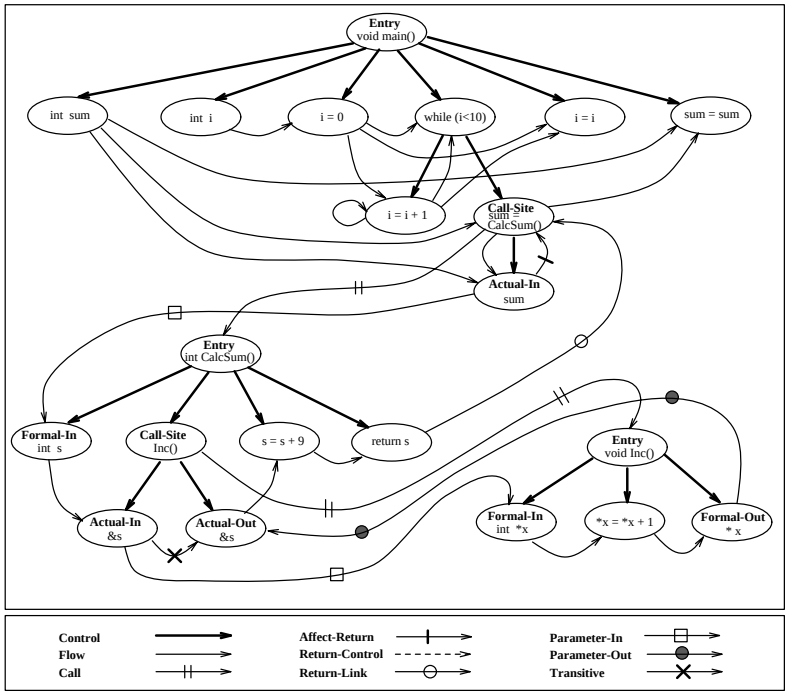

Figure 1. The program dependence graph corresponding to the program in Table 1.

At this time, we should further note that the nodes of the SDG (in the figures) are shown to be ''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 [Liv94]. 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 to be resolved at the statement level.

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 the formal-in nodes $f\_in_l(F^k)$ 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 2.1 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 2.1 is extended to a path denoted by $\overline{S}_w^{s_0}$ so that it also includes edges that represent the transitive dependences and affect-return edges associated with the call-sites of $cs_{j_k}(F^{m_k})$ as well as the return-control edges. Hence,

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

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-return and return-control edges must be taken into account.

2.3. Global Variables, Static Variables, and Aliasing

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

2.3.1. Global Variables

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.

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.

```
[ 1]. int g;

[ 2]. void main(void)     [ 7]. int Inc(int g)      [12]. void IncGlobal(void)
[ 3]. {                   [ 8]. {                   [13]. {
[ 4].     int i = 4;      [ 9].    IncGlobal();     [14].   g = g + 1;
[ 5].     i = Inc(i);     [10].    return g+1;      [15]. }
[ 6]. }                   [11]. }
```

Table 2. Illustration of global naming conflicts.

2.3.2. 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`.

2.3.3. 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[11] `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.

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 $i_1 \leq i_2 \leq \ldots \leq i_k$ 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

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

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[12]; the (alias-free) abstract syntax tree representing `alias` is copied so it is rooted at

---

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

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

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-return, 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-return, 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

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

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[13] 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. As described in [Hor90], 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

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

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[14] (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

---

[13] The short-circuiting is done by simply following the control dependence edge and continuing the slicing algorithm from its origin.

[14] For a procedure *P*, the set GMOD(*P*) is defined as the set of variables that might be *modified* by *P* itself or by a procedure (transitively) called from *P*. The set GREF(*P*) is defined as the set of variables that might be *referenced* by *P* itself or by a procedure (transitively) called from *P* [Hor90].

contained in the procedure's dependence graph. Furthermore, as we will show in the next section, we will derive this information *during* construction of the SDG.

In particular, we will consider four cases of a formal-out node that corresponds to a pass-by-reference parameter. The first case occurs when the variable is passed to the procedure and is *never* modified (i.e., there is no execution path where the variable is defined). The second occurs when the variable is passed and is *always* modified (i.e., the variable is defined on every execution path). The third occurs when the variable is passed and may *sometimes* be 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. The fourth case, the *unknown* case, is the initial condition before the nodes are classified. This case may also exist while the dependences for recursive procedures are being calculated. Note that this case will not exist after the calculation has been completed. The *unknown* case will be discussed in the next section.

We can summarize as follows: Let A, S, N, and U denote the set of formal-out nodes of F^k that are always, sometimes, never modified, and unknown, 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. $(f\_out_l(F^k) \in S) \leftrightarrow ((f\_in_{i_l} F^k \xrightarrow{dd} f\_out_l(F^k)) \wedge (indegree_{dd}(f\_out_i(F^k)) > 1))$,
3. $(f\_out_l(F^k) \in N) \leftrightarrow ((f\_in_{i_l}(F^k) \xrightarrow{dd} f\_out_l(F^k)) \wedge (indegree_{dd}(f\_out_i(F^k)) = 1))$,
4. $(f\_out_l(F^k) \in U) \leftrightarrow (indegree_{dd}(f\_out_i(F^k)) = 0)$.

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

5. 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[15]. 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 calculation that makes the explicit 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.

---

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

The Algorithm

In [Liv94], we presented and algorithm that in the absence of recursion correctly recursion computes the transitive dependences in a terminal procedure, and use it to compute both data dependences and transitive dependences of a SDG in a single pass. Moreover, 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 $\bar{\sigma}F^\tau$ 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 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 was modified as follows.

The *extended call sequence graph (ECSG)* is employed to detect 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 3 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.

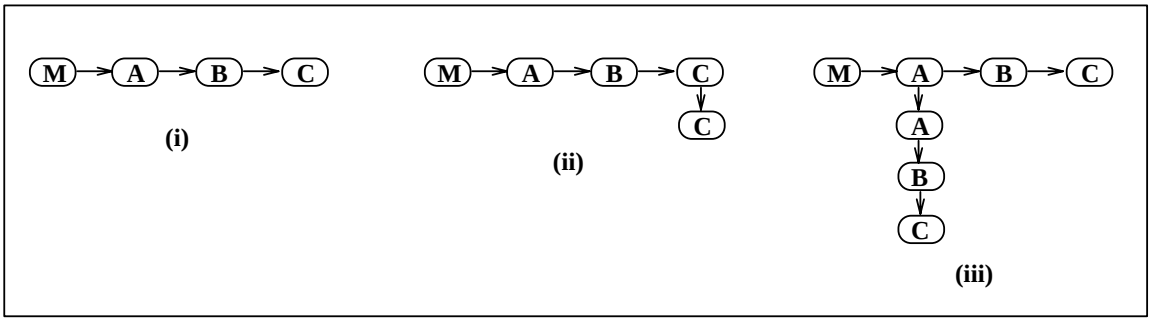

Figure 3. Extended Call Sequence Graph.

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.

On the other hand, if such a node *is* found, then recursion has been detected[16]. 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 $\{\omega_{i,j}: 0=<i\leq n_j, j_l<j\leq m\}$. Second, all iterate lists, $\{\omega_{l,j}: 1\leq i\leq n_j\}$ 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[17] 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 3(ii), a call from *C* to procedure *A* would yield the ECSG of Figure 3(iii).

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 information that is obtained is reflected to its (known) call sites. It is important to note that if the procedure deleted is not a terminal procedure, its summary will be partial (i.e., incomplete). 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 dependences 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 dependences 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 dependences requires that when partial dependences (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.

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

The algorithm just described is presented in a procedural language in Table 3. The SDG for program `Prog` is computed by calling `solve_program`. This procedure initializes the ECSG and the procedure solutions and then calls `solve_procedure` on the main procedure. In `solve_procedure`, each line of the the procedure `P` is solved, using the algorithm described in [Liv94]. Whenever an unsummarized procedure call `Q` is encountered, the partial solution of `P` is saved and `solve_procedure` is executed on `Q`. If `P` and `Q` do not call each other, then

[16] 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.

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

```
algorithm solve_program(Prog){
  ECSG = { Main }
  set all partial procedures solutions to empty
  solve procedure(Main)
}

solve_procedure(P)
  for every line l in P
     if l contains a procedure call to unsummarized procedure Q
       if Q does not appear on ECSG
           save the partial solution of P
           push Q onto ECSG
           solve_procedure(Q)
       else
         n=ECSG_POS(Q)
         m=ECSG_POS(P)
         For every procedure R such that n≤ECSG_POS(R)≤m,
          or R is on the iterate list of T, n≤ECSG_POS(T)≤m
            Place R on the iterate list of Q
         Delete all iterate lists for procedure T, n≤ECSG_POS(T)≤m
         Process l using the existing partial summary for Q
     else
       Process l
  If the iterate list of P is not empty
     compute_recursion(iterate_list(ECSG_POS(P))
  Remove P and its iterate list from ECSG
}

compute_recursion(procedure_list)
  Let changed=procedure_list
  while changed ≠ ∅
    for every P in changed
       solve(P)
       if the summary summary information of P changed
           add every Q in procedure_list that calls P to New_changed
    New_changed=changed
```

Table 3. The algorithm

when the solve_procedure on Q terminates,Q will be summarized and the solution of P can proceed. Otherwise, the recursive chain involving P and Q will be detected when Q (or a procedure that Q calls) makes a call to P is on the backbone of the ECSG. The iterate lists on the ECSG are updated, making use of the ECSG_POS procedure which returns the index of a procedure on the ECSG backbone. When a recursion is detected, the called procedure can't be solved immediately, so whatever partial summary that exists is used. If all lines in P have been solved, and P contains an iterate list on ECSG, then P is a member of a set of recursive procedures that must be solved, using the compute_recursion procedure.

The compute_recursion procedure accepts an iterate list as input, and repeatedly solves the procedures on the list until no new transitive dependences are found. The transitive dependences in a procedure can change only if the transitive dependences in a procedure that it calls has changed. Therefore, we only need to solve procedures that call procedures whose summary information was changed in a previous step.

We illustrate As an example consider the algorithm in detail by considering the call sequence in Table 4.

$$
\begin{aligned}
M &\rightarrow A\ B \\
A &\rightarrow B\ D \\
B &\rightarrow C\ E \\
C &\rightarrow C\ A \\
D &\rightarrow \varepsilon \\
E &\rightarrow F \\
F &\rightarrow \varepsilon
\end{aligned}
$$

Table 4. 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 3(i) illustrates the status of the ECSG (merely the backbone) up to this point whereas (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) $\partial^2\sigma C = \partial\sigma C \bigcup \rho\partial\bar{\sigma}C$, 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 3(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 $\partial\bar{\sigma}A$ (in this case the partial summary is also empty) is reflected to its call-site in *C*, i.e, $\partial^3\sigma C = \partial^2\sigma C \bigcup \rho\partial\bar{\sigma}A$; and, the iterate list rooted at *A* is updated (Figure 3(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[18], its (partial) summary information ($\rho\partial^2\bar{\sigma}C$) is calculated; and, this summary information is reflected to the call site in *B*. This yields $\partial^2\sigma B = \partial\sigma B \bigcup \rho\partial^2\bar{\sigma}C$. *C* is then deleted from the backbone and processing returns to procedure *B* as indicated by the tail of the backbone (Figure 4(i)). *B* now calls *E* which in turn calls *F* as indicated by Figure 4(ii).

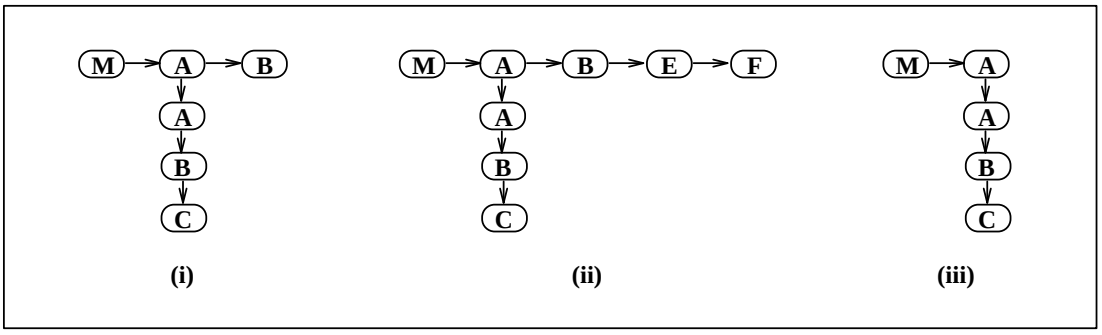

Figure 4. Extended Call Sequence Graph.

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., $\partial^2\sigma E = \partial\sigma E \bigcup \rho\bar{\sigma}F$); and, now *E* can be solved. Its summary is reflected via $\partial^3\sigma B = \partial^2\sigma B \bigcup \rho\bar{\sigma}E$.

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

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

[18] 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 4(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 5(i)).

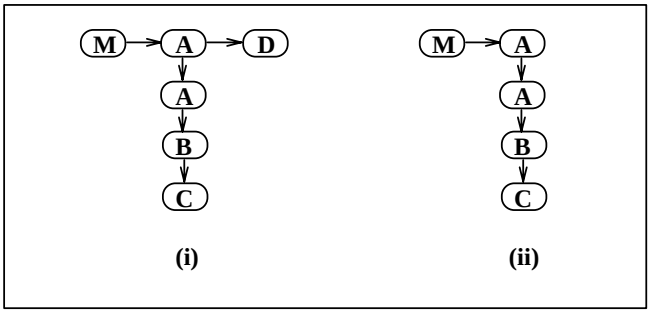

Figure 5. Extended Call Sequence Graph.

But, *D* being terminal is solved; its node is deleted from the backbone (Figure 5(ii)); and, its summary is calculated and reflected to its corresponding call site in *A* via $\partial^3\sigma A = \partial^2\sigma A \bigcup \rho\bar{\sigma}D$. 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, $\iota(A \bigcup B \bigcup C)$ (ι denotes the iteration operation). 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,

$$\iota(A \bigcup B \bigcup C) \rightarrow \bar{\sigma}A \bigcup \bar{\sigma}B \bigcup \bar{\sigma}C$$

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.

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

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

$$\rightarrow \quad \sigma M$$

6. Correctness and Optimality

In this section, we demonstrate some properties of our algorithm, and use them to show both that the algorithm is correct, and that it is optimal.

6.1. Recursive Biconnected Components

Let $G_P = (V_P, E_P)$ be a directed graph of calling dependences of a program *P*. The vertex set V_P is the set of procedures in the program. If procedures $A, B \in P$ and *A* calls procedure *B*, then we will add edge $\overline{(A,B)}$ to E_P. A *recursive biconnected component* of a graph G_P denoted by G_r is a maximal set of vertices, $V' \subseteq V_P$ such that there is a path between every pair of vertices *v* and *u* in *V'*.

The transitive dependences of a set of procedures in a biconnected component must be calculated together, because every procedure in the component can call every other, directly or indirectly. Furthermore, if *A* is in recursive biconnected component G_r, and $\overline{(A,B)} \in E_P$ with $B \notin G_r$, then the correct computation of the transitive dependences in G_r depends on the transitive dependences of *B*. Therefore, *B* should be solved before any procedure in G_r is solved.

The algorithm that computes the extended call sequence graph collects recursive biconnected components into an *iterate list* (i.e., a set of vertices $\omega_{i,j}$, $i=1..n_j$). Since all vertices on the right hand side of the ECSG are solved first (i.e, if $ECSG\_POS(A) \le ECSG(B)$ then A is solved before B), all procedures $S \notin G_r$ called by a procedure in G_r have been solved when the iteration starts on G_r.

6.2. Transitive Edge Depth

By the previous argument, we can restrict our attention to the calculation of the transitive dependences in the procedures in recursive biconnected component G_r. Let x and y be two parameters of a procedure A. We need a definition of a ''natural'' path in a program that defines a transitive dependence. We say that there is a transitive dependence of actual-out node x on actual-in node y if there is a path in $\hat{S}_w^{s_0}$, where:

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

In this case, we say that y is *interslice reachable* from x.

If there is a transitive edge e from the actual in node x to actual out node y then one can infer that y is interslice reachable from x (where we distinguish between edges incident to Sometimes and Always nodes). Moreover, given that a transitive edge can be generated by more than one interslice paths, we define $P^*(e)$ be the set of all such paths. For each path $P \in P^*(e)$ we define its *length*, denoted by $len(P)$, to be the number of procedures (or equivalently call sites) that are encountered along P including the initial procedure and all recursive copies. Moreover, we define the *recursive depth* of transitive edge e, denoted by $r(e)$, to be

$$\min_{P \in P^*(e)} \left\{ len(P) \right\}.$$

We give one more definition. Let e be a transitive edge in procedure A of recursive biconnected component G_r. The edge e will be found and inserted to the summary of A at some iterative step i when G_r is solved. We define the *iterative depth* of an edge e, denoted by $i(e)$, to be the iteration at which e is added to the graph. We may now state and prove the following theorem.

Theorem: *For any transitive edge $e \in A$ with $A \in G_r$ we have that its recursive and iterative depths are equal. In symbols, $r(e) = i(e)$.*

Proof: We prove the theorem by induction on $r(e)$. If $r(e)=1$, then the procedure is terminal and therefore it can be solved. Hence, the transitive dependence e of y on x will be found in the first iteration. Then e will be added on the first iteration and consequently, $i(e)=1$.

Assume now that $r(e)=i(e)$ for $r(e)<k$ for some $k>1$. If now $r(e)=k$, then there is a path $P \in P^*(e)$ such that $len(P)=k$. Therefore if P is restricted to procedure A, there is a intra-slice path that leads from x to y, and all of the transitive dependence edges e' in P are such that $r(e')<k$, and there is at least one transitive edge e_0 such that $r(e_0)=k-1$. The induction hypothesis tells us that all of the transitive edges in A are in place by the kth iteration, and one of then was added on the $k-1$st iteration. Therefore e is added to the SDG on the kth iteration, and $i(e)=r(e)=k$.□

6.3. Correctness

The correctness of the transitive dependence calculation follows if the algorithm terminates, and if all and only those transitive dependences that exist in the program are added as edges in the SDG. It is easy to see that the algorithm terminates, because only a finite number of transitive edges can be added. It is also easy to see, by an inductive argument, that only those edges that exist in the program are added to the SDG.

In the base case, the calculation of dependences within a single procedure is correct [Liv94], so only those edges e such that $r(e)=1$ are added in the first iteration. In the kth iteration, a transitive edge $e=x\rightarrow y$ will be added if in A the path from x to y crosses a transitive dependency edge e' such that $i(e')=r(e')=k-1$. But by our inductive hypothesis, an edge with iterative depth less than k is added to the SDG only if it exists in the program. Therefore, since the algorithm for calculating dependences in a single procedure is correct, only those edges e such that $r(e)=k$ are added on the kth iteration.

Next, we need to show that all transitive edges are added to the SDG. Let e be a transitive edge. Then e has a recursive depth, say $r(e)=k$. Let P be a path for e such that $len(P)=k$. Therefore the calling procedure A makes a call to procedure B with parameters w and z such that there is a transitive edge $e'=w\rightarrow z$ and $r(e')=k-1$. Continuing this way, we can see that in P there are transitive edges with recursive depth 1 through $k-1$. Since $i(e)=r(e)$, there was an edge added to the SDG on iterations 1 through $k-1$. Therefore if at step l no transitive edge is added, there is no transitive edge with recursive depth $l+1$ or greater.

6.4. Optimality

A worst case analysis of the number of iterations required to solve a recursive biconnected component G_r will produce a very pessimistic result, because one can construct examples in which only a single transitive edge is added at every iteration. If *param* is the maximum number of parameters in any procedure in G_r, then up to $O(|G_r|param^2)$ transitive edges can be added, so solving G_r might require $O(|G^2_{r|param})$ iterations. Consider the example in Table 5:

```
A(w,x,y,z){
   B(z,w,x,y)
}

B(w,x,y,z){
   C(w,x,y,z)
}

C(w,x,y,z){
   D(w,x,y,z)
}

D(w,x,y,z){
  if(w>0)
    A(w,x,y,z)
  else
    y = z
```

Table 5. A program with a large number of transitive dependences.

The program in Table 5 contains 48 dependencies, as in each procedure `A, B, C, D`, there is dependency between each pair of parameters. The transitive dependencies are shown in Figure 6, and are labeled with the number of iterations required to find the dependency. To simplify the figure, we use the parameter name to represent both the formal in and the formal out nodes. In procedure `D`, the edges $w\rightarrow y$ and $z\rightarrow y$ are added on the first iteration, because w and z directly affect y. These transitive dependences propagate through the recursive biconnected component (the match between the parameters of the calling and the called parameters is indicated by dashed lines). The last dependence to be determined is $e = w\rightarrow x$ in A, after 20 iterations.

Recall the theorem that we just proved, that $i(e)=r(e)$. Since 20 iterations are required to solve the set of procedures, there is an edge with iterative depth 20 and therefore an edge e with recursive depth 20. Since e has recursive depth 20, any algorithm that finds e must search through 20 instances of the procedures. We can now state the following theorem:

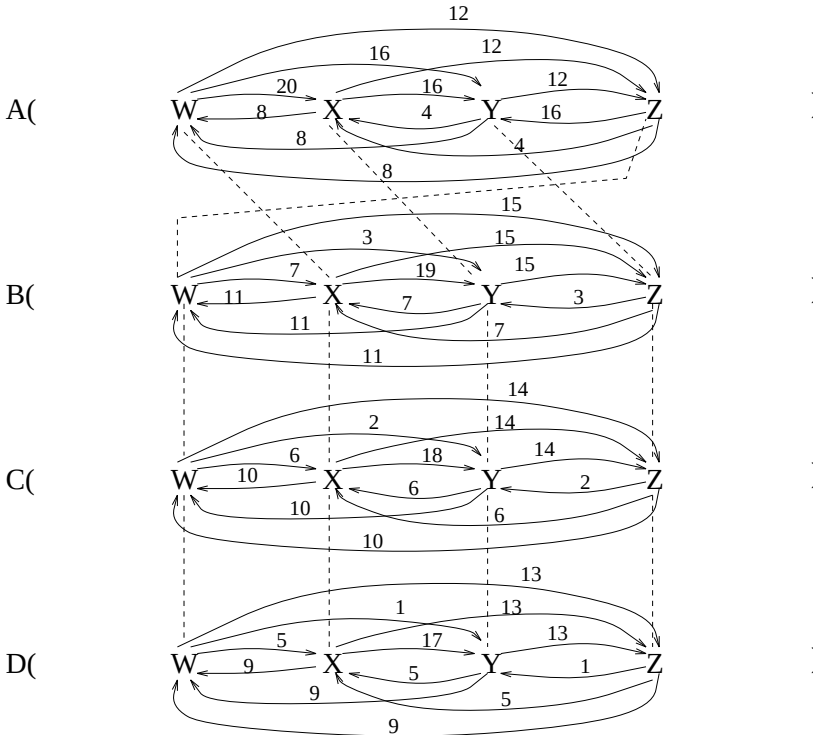

Figure 6. Transitive dependences for the Program in Table 5.

Theorem: *The algorithm for finding transitive edges in recursive biconnected components is optimal with respect to the longest chain of procedures that must be solved.*

Proof: Suppose that the algorithm requires k iterations. Then there is an transitive edge e in the recursive biconnected component with recursive depth k. Therefore, any algorithm which finds e must search through k procedure instances.□

We note that the algorithm in Table 3 can be optimized somewhat to reduce the number of iterations and the space overhead. The algorithm actually used in our implementation is listed in Table 7. The idea is to collapse some iterations by propagating changes within an iterative step. We delayed the presentation of the actual algorithm because the algorithm in Table 3 is easier to analyze, and has the same behavior.

```
compute_recursion(procedure_list)
  mark all procedures on procedure_list
  while there is a marked procedure on procedure_list
    for every P in procedure_list
       if P is marked
           solve(P)
       if the summary summary information of P changed
           mark every Q in procedure_list that calls P
```

Table 7. The modified `compute_recursion` procedure.

6.4.1. Recursive Depth

The optimality of our algorithm shows that any algorithm for computing an SDG can require $O(|G_r|param^2)$ iterations when confronted with a recursive biconnected component. Intuitively, only a few iterations will be required, because a well written program will contain only a few

procedures in each recursive biconnected component, and the procedures will add most of their transitive edges on the first iteration. Therefore, the number of iterations should be proportional to the number of procedures in the component.

Let us recall the example in Table 5. It is difficult to determine just what computation the program carries out, because so many calls must be examined to determine where the information flows. A more typical example of a recursive procedure is shown in Table 8. This program has a recursive depth of 2, and it is not difficult to trace the execution of the recursive procedure. In general, a recursive biconnected component that contains edges with a large recursive depth will be more difficult to read than a recursive biconnected component that only contains transitive edges with a small recursive depth.

We propose a new software metric: the *recursive depth*. The recursive depth of a recursive biconnected component is the largest iterative depth on any transitive edge in the component, and the recursive depth of a program is the largest recursive depth of any recursive biconnected component in the program. A large recursive depth indicates a difficult to understand program.

```
[ 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]. }                    [16].         R(x,y);
                           [17].         *x = *x + 1;

                           [18].     }
                           [19].     else {
                           [20].         *x = *x - 1;
                           [21].         *y = *y - 1;
                           [22].         R(x,y);
                           [23].     }
                           [24]. }
```

Table 8. A sample program.

7. 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 9. 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).

```
[ 1]. void R(int *x, int *y, int *z)
[ 2]. {
[ 3].     if (-)
[ 4].         ---;
[ 5].     else if (-) {
[ 6].         R(x,x,z);
[ 7].         ---;
[ 8].         R(x,y,y);
[ 9].     }
[10]. }
```

Table 9. A procedure fragment.

8. An Integrated Software Maintenance Environment

As we indicated in the introduction we have developed a prototype that integrates a number of tools. Three of those, reaching definitions calculation, forward slice, and dice, are briefly discussed below.

The SDG Can be used to find *reaching definitions* and/or *du-pairs* even when they cross procedure boundaries. 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 similar to the slicing algorithm in that it requires two phases. The screen dump that was provided from our Ghinsu tool in Figure 7 illustrates the result of finding the reaching definitions relative to the statement `*y=*y+*x`. Notice, that this way we are able to also detect data anomalies. For example, we can see that the declaration of `y` ‘‘reaches’’ the use of `y` in the statement `*y=*y+*x`; therefore, it is detected an uninitialized variable.

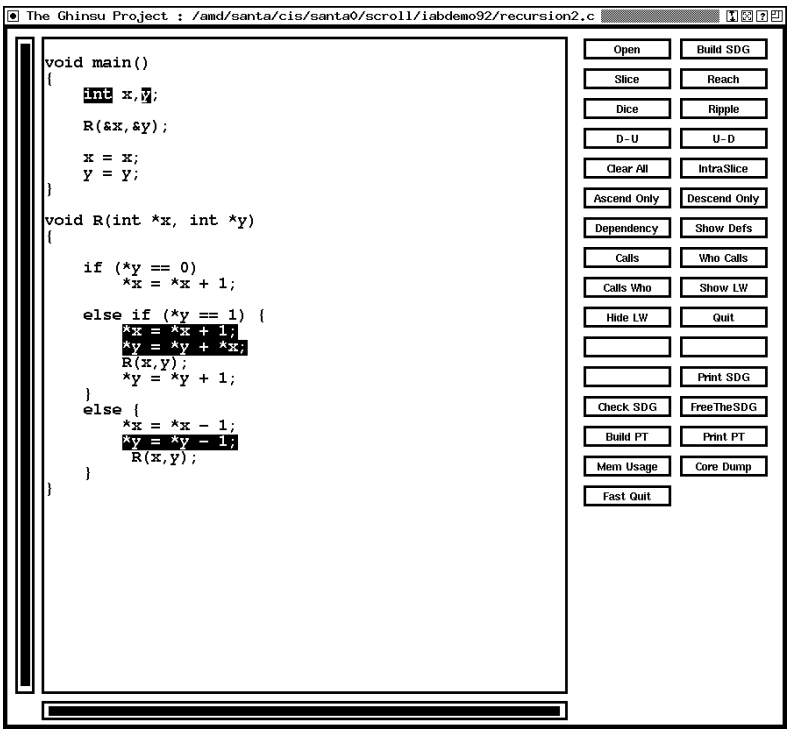

Figure 7. The Ghinsu tool. The reaching definitions relative to the statement `*y=*y+*x` are shown.

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, a *forward slice* will capture the potential effect of changing a variable at a selected statement [Hor90].

Like slicing, forward slicing 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-return, and call edges. The second phase follows the control, data flow, declaration, return-control, parameter-out, transitive, affect-return, 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.

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.

9. 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 operational slicers for C 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 dependences 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 dependences 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 dependences 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.

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.

10. Current and Future Work

We presently have produced a prototype that builds an SDG for virtually every construct of ANSI C (except long jumps) with the restriction that only single-level pointers are allowed and in that case analysis induced by the presence of the pointers is detected and handled. For arbitrary pointers our prototype permits interprocedural analysis by inline expanding each function. We are currently developing methods to *summarize* procedures in the case of arbitrary level pointers in the context of the method that we employ in the determination of the du-chains. When this is accomplished will enable us to build the SDG.

11. References

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

[Aho86] 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.

[Liv94] Panos E. Livadas, Stephen Croll. *''A New Algorithm for the Calculation of Transitive Dependences''*, Journal of Software Maintenance, Vol 6, 1994; pp. 100-127.

[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.