Computing Science Technical Report #73

Constructing the Call Graph of a Program

Barbara G. Ryder

December 12, 1978

Constructing the Call Graph of a Program

Barbara G. Ryder

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The proliferation of large software systems written in high level programming languages insures the utility of analysis programs which examine interprocedural communications. Often these analysis programs need to reduce the dynamic relations between procedures to a static data representation. This paper presents one such representation, a directed, acyclic graph named the call graph of a program. We delineate the programs representable by an acyclic call graph and present an algorithm for constructing it using the property that its nodes may be linearly ordered. We prove the correctness of the algorithm and discuss the results obtained from an implementation of the algorithm in the PFORT Verifier [1].

December 12, 1978

Constructing the Call Graph of a Program

Barbara G. Ryder

Bell Laboratories
Murray Hill, New Jersey 07974

I. Introduction

To do a static analysis of a program written in a high level programming language (e.g., FORTRAN), we must first examine the execution-time relationships between procedures in the program. We collapse these dynamic relations into the form of a directed graph, called the *call graph* of the program [2]. The nodes of the graph are the procedures of the program; each edge represents one or more invocations of a procedure P_j by a procedure P_i. For a nonrecursive language such as FORTRAN, with reasonable assumptions about the structure of a program, the call graph will be a directed, acyclic graph. In this paper we address the problem of constructing these call graphs.

The call graph is a useful data representation for control and data flow programs which investigate interprocedural communication (i.e., how procedures exchange information). It contains all the relationships among the procedures in a program and can contain auxiliary information concerning the data within each procedure and global data shared among procedures.

The construction of the call graph for a program without procedure parameters is very simple. (Note: We use *procedure parameters* to refer to parameters which are used as procedures in contrast to being used as variables.) One pass through the input data enables us to discover all the nodes and all the edges in the graph simply by making a table of all procedures and the references they contain. Each procedure must be analyzed only once; the order in which they are examined is not significant.

When procedure parameters are present, however, the work done in constructing the call graph is dependent upon the order in which procedures are analyzed. In programs containing procedure parameters, it is possible to have a reference to a formal procedure parameter which may represent invocations of several distinct external procedures.

In order to ascertain all possible invocations which result from such a reference in a procedure P_i, we need to know all external procedures associated with that formal procedure parameter. (Note: We define the concepts of reference, invocation, and parameter association in Section II.) Therefore, we must examine all procedures along paths in the graph leading to P_i, before we can complete the examination of P_i and its references. If some information necessary for this analysis is not core-resident, then the order in which we examine procedures is significant to the input/output channel time and processor time needed to complete the graph (see Section V).

The construction algorithm presented here was designed for use in the PFORT Verifier [1], a program which checks FORTRAN programs for conformance to PFORT [1], a portable subset of ANS FORTRAN [3,4,5]. In the Verifier, we are most interested in the sharing and setting of data through the use of parameter lists and global data areas (i.e., COMMON). The Verifier was designed to be portable across a wide range of machines. The Verifier itself is written in PFORT. No machine dependent input/output facilities are used (e.g., FORTRAN subprograms for implementing random access secondary storage). The use of procedures to set and extract information stored as bits was judged too costly to be relied upon heavily.

The portability constraint also meant that in-core storage was limited. In order to construct the call graph (in the nontrivial case) it is necessary to extract procedure definitions from

the input (i.e., descriptions of each procedure, its formal parameters and its references). The storage requirements for this information cannot be computed *a priori;* they are directly related to the number of procedures as well as the number of references they contain. The potential quantity of this data precluded in-core storage.

Given these considerations, we had to use sequential secondary storage for the procedure definition tables. We sought to minimize the number of searches through these tables, while still determining all possible procedure invocations in a program. In the algorithm described here, we pass once through the tables to determine the nodes in the graph. In addition, we must search through the tables n times, where n is the number of procedures in the program, each time looking for a table corresponding to a distinct procedure. Thus we have minimized the necessary searches of these tables, accruing thereby a savings in processor and channel time (see Section V.).

In the remainder of this paper, Section II presents definitions of basic concepts. Section III presents the algorithm with examples. It also contains an analysis of the algorithm and comments on the limitations of the algorithm. Section IV proves that the algorithm is correct when applied to a well-formed, nonrecursive program. Section V discusses our experience with the algorithm in the PFORT Verifier. It also presents some data on the use of procedures in a sample of the FORTRAN programs which have been processed by the PFORT Verifier. Section VI discusses other methods for constructing the call graph, compares our method to them and presents our conclusions.

II. Definitions

In this section, we first consider how procedures can be used in the programming languages we wish to consider. We describe procedure invocation and its accompanying parameter association process for procedure parameters. Second, we define the call graph of a program and show how a call graph embodies the interprocedural relationships in its structure. Third, we define the level of a node, a characteristic which can contain information about interprocedural relations. Fourth, we consider how to restrict a program in order to insure that it has an acyclic call graph. Finally, we define a class of programs for which we can prove the correctness of our algorithm.

Procedures and Their References

We consider a *program* to be a finite set of mutually external procedures $\{P_i\}_{i=1}^{n}$. We do not allow nested procedure definitions. Each procedure has a unique name and an optional ordered list of *formal parameters* which appear within the procedure definition. In our discussions, we use the term *external procedure* to refer to one of the P_i's.

Procedures communicate with one another through the use of references. There are two types of references which can occur in a procedure P_i. If the name of an external procedure, P_j, followed by a possibly empty list of *actual parameters* appears in the definition of procedure P_i, then we say there is a *direct reference* to P_j in P_i (i.e., P_i can invoke P_j). If the name of a formal procedure parameter Q of P_i, followed by an optional list of actual parameters appears in the definition of P_i, then we say there is a *formal reference* to Q in P_i.

A procedure is restricted to appearing as an actual procedure parameter in a reference and/or as the invoked procedure in a reference. We assume that a procedure invocation will occur if it is represented in the definition of the invoking procedure by a direct reference or a formal reference.

Procedure Invocation and Parameter Association

In order to understand the meaning of a formal reference we first must consider what procedure invocation and its accompanying process of parameter association mean in terms of execution-time flow of control in a program. When a procedure is invoked, control is passed to it and the execution of its definition commences. If we examine an executing program at a

particular point in time, we will find the procedures of that program fall into two disjoint categories. The first category is a set of procedures which have been invoked and have not yet been returned from. We call these *active procedures.* There must be one and only one procedure in this set which is not invoked by any other procedure in the program, but is the entry procedure to the program. There also is a unique procedure in this set whose definition is being executed at the moment execution is stopped. The second category is a set of procedures not active at this time (i.e., all remaining procedures in the program).

Whenever a procedure P_i becomes active through an invocation and has a list of formal parameters in its definition, the reference representing the invocation must have a corresponding list of actual parameters. At the time the invocation is executed, these actual and formal parameterlists are "matched together" through the following process of *parameter association.*

A one-to-one correspondence is set up between the list of actual parameters of the reference and the list of formal parameters of the invoked procedure. We say each actual parameter *is associated with* its corresponding formal parameter. The association is maintained for as long as procedure P_i remains active as a result of this invocation. We stipulate that the number of actual parameters and their usages (i.e., variable or procedure) must agree with the number and usages of the formal parameters for the invocation to occur.

If a formal parameter of the invoked procedure is a procedure parameter, the actual parameter associated with it must be an external procedure in the program *or* a formal procedure parameter of the invoking procedure. We require that all procedures referenced within a program must be defined in the program. A formal procedure parameter Q in procedure P_i, may become associated with several external procedures during the course of execution of the program; however, whenever P_i is active there can be only one external procedure associated with Q.

Let us consider the semantics of the process of procedure parameter association described above, in terms of the subsequent execution of the program. Suppose that as a result of an invocation of procedure P_i, the external procedure P_j used as an actual parameter becomes associated with the formal procedure parameter Q of P_i. Then, P_i will be executed as if P_j appeared in the definition of P_i wherever Q appeared (i.e., think of macro substitution of P_j for Q).

Likewise, suppose that as a result of an invocation of procedure P_l by procedure P_k the formal procedure parameter S of P_k is associated with the formal procedure parameter R of P_l. At the time of the invocation of P_l, S is already associated with some external procedure P_m, because P_k is active. The effect of the association of S with R is that P_l will be executed as if P_m appeared in the definition of P_l wherever R appeared (i.e., macro substitution of P_m for R).

Reference Expansion

To this point our discussions have been directed towards the meaning of procedure parameter association in the case of a single formal procedure parameter. In general, a procedure P_i can contain m $(0 \leqslant m \leqslant n)$ formal procedure parameters. Whenever P_i becomes active, there is an m-tuple of external procedures, corresponding to the m actual procedure parameters in the invocation, which becomes associated with the m-tuple of formal procedure parameters of P_i. We now refer to Figure 1 to explain this situation. For the sake of clarity we have included *only* procedure parameters; clearly, the presence of non-procedure parameters would not change the discussions below because of the one-to-one correspondence demanded between formal and actual parameterlists.

In Figure 1, P_i is directly referenced with m external procedures $(P_1, \ldots, P_m)$, used as actual procedure parameters. These correspond to the F_r's for $1 \leqslant r \leqslant m$, m formal procedure parameters of P_i. Each B_l, $0 \leqslant l \leqslant k$, can be either one of the F_r's or an external procedure in the program. The value of B_0 clearly determines whether there is a direct or a formal reference in P_i. The G_l's are the formal procedure parameters of P_j.

```
begin
    procedure P1,P2, . . . , Pm;
    Pi (P1,P2, . . . , Pm);
    .
    .
    .
end;

procedure Pi (F1,F2, . . . , Fm);
begin
    .
    .
    .
    B0(B1,B2, . . . , Bk);
    .
    .
    .
end;

procedure Pj (G1,G2, . . . , Gk);
    .
    .
    .
```

Figure 1

We can see that, in general, any reference in a program can be written as $B_0(B_1, \ldots, B_k)$, for $k \geqslant 0$. Let us assume that P_i is currently active as a result of the direct reference shown in Figure 1. Then each F_r is associated with P_r for $1 \leqslant r \leqslant m$. Also assume the reference to B_0 is about to be executed. The process of *reference expansion* refers to how each B_l, for $0 \leqslant l \leqslant k$, now is associated with an external procedure in the program. We have noted that when B_0 is an F_r, there is a formal reference in P_i; when B_0 is an external procedure there is a direct reference in P_i. Thus, by explaining the meaning of reference expansion we will explain the meaning of a formal reference.

Suppose B_0 is one of the F_r's, say F_j. When the reference to B_0 in P_i is executed, it will be an invocation of P_j because F_j is currently associated with P_j. We have an *expanded formal reference* to P_j in P_i. We must consider what k-tuple of external procedures $(P_{j_1}, \ldots, P_{j_k})$ will become associated with the k-tuple $(G_1, \ldots, G_k)$, as a result of this invocation of P_j. We know that the k-tuple $(P_{j_1}, \ldots, P_{j_k})$ corresponds to the k-tuple $(B_1, \ldots, B_k)$; therefore, let us examine the B_l's.

For each l, $1 \leqslant l \leqslant k$

(i) if $B_l = F_r$ for some r, $1 \leqslant r \leqslant m$ then, since F_r is currently associated with P_r, $P_{j_l} = P_r$;

(ii) if $B_l = P_q$ for some q, $1 \leqslant q \leqslant n$ then $P_{j_l} = P_q$.

Once the $(P_{j_1}, \ldots, P_{j_k})$ is known, the meaning of the association of $(P_{j_1}, \ldots, P_{j_k})$ with $(G_1, \ldots, G_k)$ in terms of the subsequent execution of P_j, follows from our previous discussions. The reference $B_0(B_1, \ldots, B_k)$ has thus been expanded into the reference $P_{j_0}(P_{j_1}, \ldots, P_{j_k})$ where $P_{j_0}=P_j$ in our example.

Since a procedure can become active more than once during the execution of a program, the process of expanding a formal reference may occur several times. Thus, a formal reference in P_i can be thought of as corresponding to a set of direct references (i.e., the set of expanded formal references).

It is evident that each procedure P_i defined with m formal procedure parameters has a set of m-tuples of external procedures which, during the execution of the program, can be associated with its m-tuple of formal procedure parameters. We call this the *procedure vector set* for P_i. (Note: For m=0, the set is empty.) In order to construct the call graph of a program, we must determine the procedure vector set for each procedure in the program. It should be obvious that the procedure vector set of P_i is *not* equivalent to the set of cross products of the sets of external procedures which can be associated with each formal procedure parameter of P_i.

```
      EXTERNAL B,C,D,E
10    CALL A(B,C)
15    CALL A(D,E)
      STOP
      END

      SUBROUTINE A(X,Y)
      EXTERNAL Y
20    CALL X(Y)
      RETURN
      END

      SUBROUTINE B(Z)
30    CALL Z
      RETURN
      END

      SUBROUTINE D(W)
40    CALL W
      RETURN
      END

      SUBROUTINE C
      RETURN
      END

      SUBROUTINE E
      RETURN
      END
```

Figure 2

The definitions presented above are illustrated in Figure 2 in a FORTRAN setting. Statement 10, a direct reference to subroutine A, causes the 2-tuple of subprograms (B,C) to become associated with formal procedure parameters (X,Y). As a result of this association, the formal reference at statement 20 is expanded into a reference to subroutine B with C as an actual parameter and the formal reference at statement 30 is expanded into reference to C. The direct reference at statement 15 causes the 2-tuple (D,E) to become associated with (X,Y) of subroutine A. As a result, statement 20 becomes "CALL D(E)" and statement 40 becomes "CALL E". Thus, in this program segment, A references B and D, B references C, and D references E. The procedure vector set for A is {(B,C),(D,E)}, for B is {(C)} and for D is {(E)}.

There is a particular usage of external procedures as actual arguments which is significant in our algorithm so we define it here. When an external procedure P_l appears in a procedure P_i used *only* as an actual procedure parameter, then there is a *referral* to P_l in P_i. The main program in Figure 2 illustrates the definition of referral in a FORTRAN setting. There are

referrals to B, C, D and E in the main program, since these procedures are only used as actual procedure parameters. We see that the referrals to B and D cause the formal reference in statement 20 to be expanded into a reference to subroutine B and a reference to subroutine D. Thus, the referral to B, for example, represents the first step in a sequence of procedure references and accompanying procedure parameter associations which end with a reference to B.

Call Graph

Given the above description of the use of procedures, to form the call graph of a program we must examine all procedure definitions and their references, and determine all possible formal procedure parameter associations with external procedures.

The reference relations between procedures in a program can be represented by a directed graph $G = \{N,E\}$ called a *call graph* where:

(i) each node N_i corresponds in a one-to-one manner to a procedure P_i and its procedure vector set;

(ii) if P_i contains a reference $B_0(B_1, \ldots, B_k)$ then for each expansion $P_{j_0}(P_{j_1}, \ldots, P_{j_k})$ of that reference, there is a directed edge (N_i,N_{j_0}) in the graph and $(P_{j_1}, \ldots, P_{j_k})$ is in the procedure vector set of N_{j_0}.

It is clear from the above definition that in order to determine all edges from a node N_i in the call graph, we must fully expand all references in P_i; thus, we must have the procedure vector set of P_i. To obtain the procedure vector set of P_i we must know all the predecessors of N_i in the call graph and their procedure vector sets. Our construction algorithm (see Section III) adds a node to the graph (i.e., "processes" a node) only *after* all of its predecessors have been processed (see Section IV); thus we *can* ascertain all edges from N_i as we add N_i to the graph.

If in a procedure P_i, a reference to P_j contains a reference to P_k used as an actual parameter (e.g., $P_j(P_k(x))$ *or* as part of an expression used as an actual parameter (e.g., $P_j(P_k(x)+1)$), then P_i references P_j and P_k (i.e., the reference to P_k is expanded in the procedure in which it appears). In a call-by-name language like ALGOL-60, P_i would not reference P_k in this situation.

Figure 3 shows the call graph corresponding to the program segment in Figure 2.

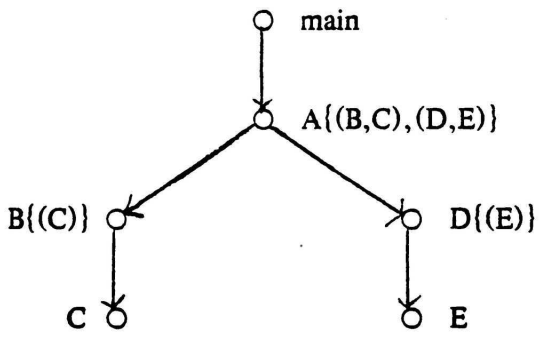


Figure 3

Level

Whenever a call graph of a program is acyclic, we can assign an integer called *level* to each node such that if (N_i,N_j) is a directed edge in the graph then level(N_i) < level(N_j). Obviously, if there is a non-zero length path from N_i to N_j then level(N_i) < level(N_j).

Recursion

We confine our attention to programs which contain no dynamic or static recursion. *Dynamic recursion* occurs when, in the execution of a program, an already active procedure is invoked. *Static recursion* occurs when there are two or more execution paths involving the same procedures invoked in a different order, and because of these paths a cycle is formed in the call graph. Consider the program fragment in Figure 4. We see that b invokes a and a invokes b, but neither a nor b is invoked when it is already active.

```
begin
    a(b);
    b(a);
end;

procedure a(x);
begin procedure c;
    x(c);
end;

procedure b(y);
begin procedure c;
    y(c);
end;
```

Figure 4

In the course of execution there is no recursion, yet a cycle appears in the call graph as shown in Figure 5. We disallow static recursion because we cannot distinguish between it and dynamic recursion in the call graph.

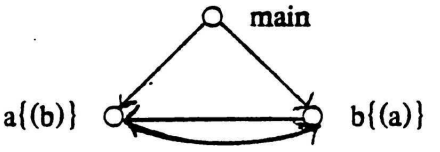


Figure 5

Well-Formed Program

We must make further restrictions about the use of procedures and procedure parameters in a program in order to guard against the construction of spurious cycles in the call graph by our algorithm. Here we define a *well-formed* program and show in Section IV that our algorithm correctly constructs the call graph of a well-formed, nonrecursive program.

For a program to be well-formed, each formal procedure parameter Q of procedure P_i must be used in P_i as the referenced procedure in a formal reference (e.g., $Q(x)$) and/or as an actual procedure parameter in a reference (e.g., $P_j(Q)$).

We note that in a well-formed, nonrecursive program, an external procedure used as an actual procedure parameter will be associated, through a sequence of procedure invocations and accompanying parameter associations, with a formal procedure parameter which is used as the referenced procedure in a formal reference.

III. Algorithm

The algorithm for the construction of the call graph of a well-formed, nonrecursive program (see Section II) is presented in this section. First, we discuss concepts necessary to the algorithm description. Second, we present the algorithm followed by a FORTRAN example of "how it works". Third, we discuss the complexity of the algorithm. Finally, we present additional thoughts on recursion and call-by-name parameter passing, two programming concepts which we have chosen not to handle in our algorithm.

The algorithm consists of an initialization phase followed by a construction phase. We assume that the program has been parsed to obtain tables of the references in each procedure. These tables are available as input to our algorithm. The initialization phase builds the set of nodes corresponding to the set of procedures in the program; it places definitional information about each procedure and its formal parameters at each node. It marks the nodes unprocessed. Edges corresponding to direct references are inserted and the level of each node is set accordingly. For programs containing no procedure parameters, the initialization phase completely constructs the call graph.

The construction phase consists of n steps where n is the number of procedures in the program. At each step, a current node of minimal level is processed; that node and all of its outgoing edges are added to the subgraph of the call graph already constructed, consisting of all processed nodes and their outgoing edges. In the Theorem (iv) in Section IV, we show that the order in which the nodes are processed for a well-formed, nonrecursive program insures that N_i is processed before N_j if P_i references P_j through a direct reference or an expanded formal reference. We also show that for a well-formed, nonrecursive program the graph constructed is the call graph of the program.

Level Updating

The level of a node is the criterion which determines when it is processed by the construction phase of the algorithm. Thus we must consider how the algorithm manipulates the level of a node. Initially the level of each node is zero. As the algorithm determines relationships between the procedure represented by one node and other procedures in the program, the levels of the nodes are adjusted to reflect these relationships (see Section II). During these adjustments, when the level of a node is increased, the levels of its *successors* (i.e., nodes along non-zero length paths from that node) may have to be increased as well. (Note: If node N_j is a successor of N_i, then we say N_i is a *predecessor* of N_j.) We refer to this process often in our algorithm description below; therefore, we present in Figure 6 an ALGOL-like definition of the process called *level updating* performed on nodes N_i and N_j. In our subsequent discussions we refer to this process as "update(N_i,N_j)". We assume that there is a global stack, initially empty, available for use in update to record paths in the graph and that "push" and "pop" operators exist.

In our algorithm described below, we refer to a construction device called *temporary edges* between nodes of the graph. They are needed during the algorithm to insure that the levels of unprocessed nodes correctly reflect possible relationships which might result from the use of procedure parameters. For example, by examining statement 10 in Figure 2, we can tell that B may become a successor of the main program in the call graph; a temporary edge (main, B) would be created during the initialization phase to reflect this relationship. During the construction phase, this temporary edge would be replaced by the path {(main, A), (A, B)}. We note that for a well-formed, nonrecursive program we show in the Theorem (ii) in Section IV that a temporary edge (N_i,N_j) constructed by our algorithm always represents a non-zero length path from N_i to N_j in the call graph of the program.

All edges (temporary or otherwise) are treated equivalently by the level update process. During the algorithm there may be paths involving unprocessed nodes in the graph which contain temporary edges; however, there are no temporary edges in the subgraph of processed nodes, because the temporary edges from a node are removed when it is processed.

```
procedure update(i,j)
begin
    if (j in stack) then report a cycle in the graph
       and halt algorithm;
    if (level(i) ≥ level(j)) then
        begin push j onto stack;
                    level(j) = level(i)+1;
                    for each l, an immediate successor of j, do
                    update(j,l);
                    pop j from stack;
        end;
    return;
end;
```

Figure 6

In a well-formed, nonrecursive program, the number of edges incident to a node during the algorithm never decreases. That is, assume the temporary edge (N_i,N_l) is in the graph (e.g., a formal procedure parameter Q of P_i is associated with P_l). When N_i is processed, the temporary edge is replaced by either a non-temporary edge (N_i,N_l) (e.g., if P_i contains a formal reference to Q) *or* a non-temporary edge (N_i,N_j) and a temporary edge (N_j,N_l) (e.g., if P_i references P_j with Q used as an actual procedure parameter).

When the algorithm is applied to a well-formed, nonrecursive program, the level of a node N_i at any point in time is equal to the length of the current maximal length path to N_i from a root. Paths can contain temporary and non-temporary edges. Thus, in the complete call graph the level of a node N_i is its maximal distance from the root representing the entry procedure.

The update process detects cycles in the graph; this necessitates the use of the stack in procedure **update**. We do not allow cycles in the call graph; we terminate the construction algorithm upon finding a cycle. In the Theorem (iii) in Section IV we prove that no cycles are generated for a well-formed, nonrecursive program. When the algorithm detects a cycle, if the cycle contains no temporary edges, then dynamic or static recursion is present in the program (see Section II). If the cycle contains at least one temporary edge, then we know that either the cycle will become, in the course of the algorithm, a cycle representing dynamic or static recursion or the cycle represents noncompliance with our restrictions on the use of procedures (i.e., the program is not well-formed, see Sections II, IV).

Algorithm

The following is a description of our algorithm.

phase 1: initialization

(i) mark all nodes unprocessed and set their levels to 0.

(ii) for each procedure P_i in the program:

  (a) for each direct reference in procedure P_i to procedure P_j, construct edge (N_i,N_j) and update(N_i,N_j).

  (b) for each referral in procedure P_i to procedure P_l, construct temporary edge (N_i,N_l) and update(N_i,N_l).

phase 2: construction

(i) if all nodes are processed, stop; else pick an unprocessed node N_i of minimal level (Note: the choice is not necessarily unique.)

(ii) using the procedure vector set of N_i, expand each reference $B_0(B_1, \ldots, B_k)$ in P_i into its set of corresponding references each of which is of the form $P_{j_0}(P_{j_1}, \ldots, P_{j_k})$; for each such reference:

  (a) construct edge (N_i,N_{j_0}) and update(N_i,N_{j_0}).

  (b) for each l, $1 \leqslant l \leqslant k$, construct temporary edge (N_{j_0},N_{j_l}) and update(N_{j_0},N_{j_l}).

  (c) add $(P_{j_1}, \ldots, P_{j_k})$ to the procedure vector set of N_{j_0} (Note: don't keep duplicate k-tuples.)

(iii) remove all temporary edges eminating from N_i

(iv) mark N_i processed and return to phase 2(i).

Example

Here we present the algorithm "at work" on the FORTRAN program given in Figure 2. We list the levels of the nodes at different times during the algorithm in Figure 7. We give views of the partially constructed call graph in Figure 8 with temporary edges appearing as dotted lines.

Level Table

level(MAIN)	*0*	*0*	*0*	*0*	*0*
level(A)	*0*	*1*	*1*	*1*	*1*
level(B)	*0*	*1*	*2*	*2*	*2*
level(D)	*0*	*1*	*2*	*2*	*2*
level(C)	*0*	*1*	*2*	*3*	*3*
level(E)	*0*	*1*	*2*	*3*	*3*

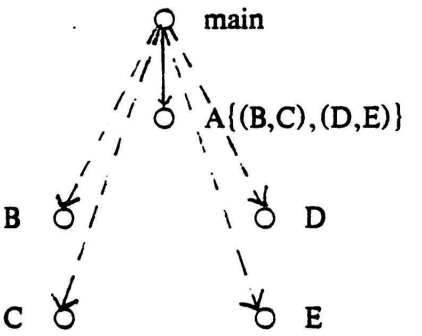
column 1: levels initially
column 2: levels after phase 1
column 3: levels after MAIN has been processed
column 4: levels after A has been processed
column 5: levels after phase 2

Figure 7

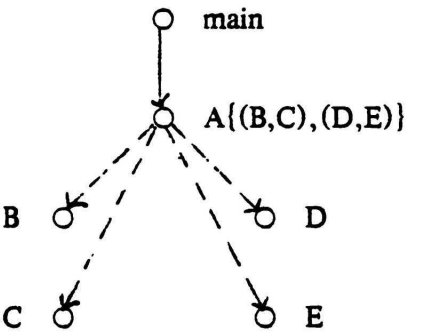
Analysis

We wish to analyze the algorithm as applied to a well-formed, nonrecursive program. There are three distinct contributions to the total work done by the algorithm. The first contribution is made by the parse of the program, which occurs before the algorithm commences. It builds tables describing the procedures and their references. Phases 1 and 2 each search this information once for each node. The work involved in these activities is proportional to the maximal number of references in any one procedure in the program, r_{max}, and the number of procedures, n; that is, a term of $O(nr_{max})$.

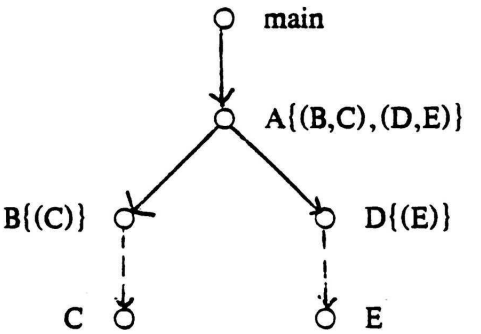
The second contribution stems from the insertion of edges into the graph and the ensuing node visits during level updating. Consider the graph $G^* = \{N,E^*\}$, where N is the set of nodes of the call graph of the program and E^* is the set of all temporary and non-temporary edges created during the algorithm. Let e^* be the number of edges in E^*. The work done in actually inserting edges into the graph is obviously proportional to e^*.



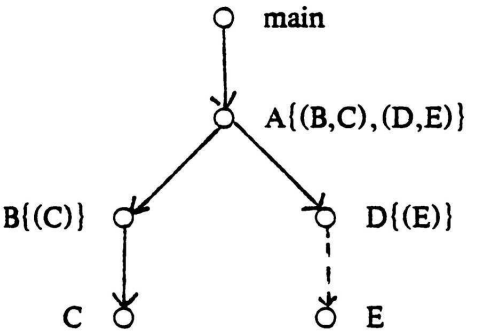
call graph after phase 1



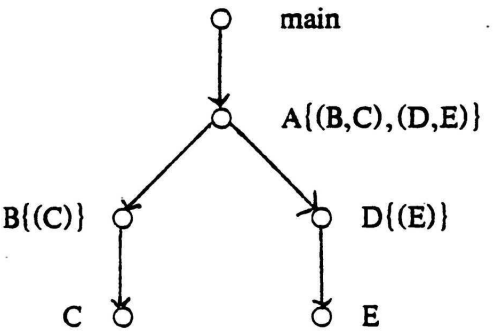
call graph after MAIN is processed



call graph after A is processed



call graph after B is processed



call graph after D is processed
(the complete call graph)

Figure 8

In our algorithm a level update is performed whenever a new edge is added to the graph. We describe the work done during these updates in terms of the number of visits to nodes in the graph. It is clear from Figure 6 that node N_j is visited during updating only when the level of an immediate predecessor N_i is increased or when a new edge is added incident to N_j. Assume lev(N_j) is the level of node N_j in the final call graph. Then an upper bound on the number of visits to N_j during level updating, V_j, is given by

$$V_j \leqslant \sum_{\{i \mid (N_i,N_j) \in E^*\}} (lev(N_i)+1).$$

Therefore, the total number of visits to all nodes during updating, V, is bounded by

$$V = \sum_{j=1}^{n} V_j \leqslant \sum_{\{(N_i,N_j) \in E^*\}} (lev(N_i)+1).$$

For any edge $(N_i,N_j) \in E^*$, we have

$$lev(N_i) < lev(N_j) \leqslant d$$

where

$$d = \max_{1 \leqslant k \leqslant n} lev(N_k)$$

is called the depth of the graph. It follows immediately that

$$V \leqslant de^*.$$

Finally, since

$$d \leqslant n-1$$

$$e^* \leqslant \binom{n}{2} = \frac{1}{2}n(n-1)$$

we have

$$V \leqslant \frac{1}{2}n(n-1)^2.$$

The third contribution to the work done is derived from the building of the procedure vector sets for all the nodes through the process of reference expansion. The work of expansion in a procedure P_i is proportional to the number of references in P_i, r_i, and the size of the procedure vector set of P_i, s_i. If r_{max} and s_{max} are the maxima of r_i,s_i for $1 \leqslant i \leqslant n$, then the work of reference expansion is bounded above by:

$$O\left(\sum_{i=1}^{n} r_i s_i\right) \leqslant O\left(nr_{max}s_{max}\right)$$

There is additional work done in recording the expanded m-tuples in the procedure vector sets of the nodes. Let t_{ij} be the number of expanded references to P_j in P_i. Let $f(s_j)$ be a bound on the work of recording a single m-tuple in the procedure vector set of N_j or rejecting it as a duplicate. Then the work done in recording m-tuples at N_j, w_j, is bounded by

$$w_j \leqslant \sum_{\{i|(N_i,N_j) \in E\}} t_{ij} f(s_j)$$

and the total work of constructing procedure vectors sets, W, is bounded by

$$W = \sum_{1 \leqslant j \leqslant n} w_j \leqslant \sum_{\{(N_i,N_j) \in E\}} t_{ij} f(s_j) \leqslant Tf(s_{max})$$

where T is the total number of expanded references in the program. It is clear that

$$T \leqslant \sum_{1 \leqslant i \leqslant n} r_i s_i \leqslant nr_{max}s_{max}.$$

Therefore

$$W \leqslant nr_{max}s_{max}f(s_{max}).$$

Since s_{max} is usually quite small, we use a sorted linear linked list to store the procedure vector sets at each node. Hence $f(s)=O(s)$, and we have

$$W = O(nr_{max}s_{max}^2).$$

If s_{max} were expected to be large, we could use a 2-3 tree instead, to achieve $f(s)=O(\log s)$

and

$$W = O(nr_{max}s_{max}log(s_{max})).$$

The bounds of this section imply that the algorithm may have to work harder to construct "deep" as opposed to "broad" graphs, "thickly connected" as opposed to "sparsely connected" ones.

Additional Remarks

In Section II we remarked that references which appear themselves as actual parameters or within expressions used as actual parameters are assumed not to be implemented by call-by-name. Our algorithm could be altered to handle a call-by-name language, but the definitions of procedure references and call graph would have to be changed. Additional parsing of the input to determine how formal variable parameters are used would have to be performed.

The basic notion of our algorithm, that of processing predecessors of a node before the node itself, cannot be satisfied in the presence of a cycle. There are approaches to call graph construction to handle recursion [6,9,10,14]. We did not attempt to handle recursion because our application language, FORTRAN, is dynamically nonrecursive.

IV. Correctness Proof

In this section we prove a theorem characterizing our algorithm and showing that it correctly constructs the call graph of a well-formed program which contains no static or dynamic recursion.

Theorem: For a well-formed, nonrecursive program, the algorithm presented in Section III correctly constructs the call graph of the program and terminates upon its completion.

We present a proof in four parts.

(i) Every non-temporary edge generated by the algorithm is in the call graph of the program.

(ii) If a temporary edge (N_i,N_l) is created by the algorithm, then there is a non-zero length path from N_i to N_l in the call graph of the program.

(iii) The graph constructed by the algorithm is acyclic; therefore, all the nodes are processed.

(iv) Every edge in the call graph is constructed as a non-temporary edge by the algorithm. (As part of this proof we will show that if P_i references P_j through a direct or expanded formal reference, then N_i will be processed before N_j by the algorithm. We will also show that the algorithm has completely calculated the procedure vector set of N_i before processing N_i.)

(i) Non-temporary edges are constructed by phase 1(ii)a and phase 2(ii)a. From the definition of the call graph we can see that the former non-temporary edges are in the call graph. The latter non-temporary edges are formed by calculating m-tuples of external procedures which correspond to the m-tuple of formal procedure parameters of P_i, recording these m-tuples in the procedure vector set of N_i and then expanding formal references in P_i using this set. It can easily be shown by induction that P_i can be referenced with each of these m-tuples; therefore, by definition of a call graph, these non-temporary edges are also in the call graph. **qed**

(ii) The algorithm creates a temporary edge in phase 2(ii)b and phase 1(ii)b. In the first case, the temporary edge (N_{j_0},N_{j_l}) is constructed because a formal procedure parameter Q of P_{j_0}, becomes associated with P_{j_l} through the expansion of a reference to P_{j_0}. The definition of a well-formed program demands that P_{j_0} references Q and/or P_{j_0} references another procedure, P_{k_1}, with Q used as an actual procedure parameter. If P_{j_0} references Q, then non-temporary edge (N_{j_0},N_{j_l}) is in the call graph. If P_j references P_{k_1} with Q as an actual procedure parameter, then the number of successive occurrences of this condition cannot exceed the number of procedures in the program without creating recursion. Therefore, there is a finite sequence:

$$P_{j_0} = P_{k_0}, P_{k_1}, \ldots, P_{k_p}, P_{j_l}$$

such that each P_{k_i} $(0 \leqslant i \leqslant p-1)$ references $P_{k_{i+1}}$ with P_{j_l} becoming associated with a formal procedure parameter of $P_{k_{i+1}}$ as a result of this reference. In addition, as a result of this sequence of references, there is an expanded formal reference to P_{j_l} in P_{k_p}. Therefore, there is a non-zero length path from N_{j_0} to N_{j_l} in the call graph.

In the second case, the temporary edge (N_i,N_l) is constructed because there is a referral to P_l in P_i. Thus, there is a reference to a procedure P_j in P_i, in which P_l appears as an actual argument. By the same reasoning as in the first case there will be a non-zero length path from N_j to N_l. Since a non-temporary edge (N_i,N_j) will exist, there will be a non-zero length path from N_i to N_l. **qed**

(iii) (by contradiction) Suppose the algorithm finds a cycle during the construction of the call graph. By part (i) of this theorem, non-temporary edges constructed by the algorithm are edges in the call graph; therefore, a cycle containing only non-temporary edges represents static or dynamic recursion. If the cycle contains a temporary edge, part (ii) of this theorem shows that a temporary edge (N_i,N_l) corresponds to a non-zero length path from N_i to N_l in the call graph. So a cycle containing temporary edges also corresponds to recursion. Since the program contains no recursion, the algorithm does not produce a cycle and does process all the nodes. **qed**

(iv) From the definition of a call graph, non-temporary edges correspond to expanded references in the program. Phase 1(ii)a constructs all the non-temporary edges corresponding to direct references (i.e., B_0 is a P_j). Phase 2(ii)a constructs the non-temporary edges corresponding to expanded formal references (i.e., $B_0=F_j$ and F_j is associated with a P_j). We must show that all the non-temporary edges corresponding to expanded formal references are constructed, that is, the procedure vector set of a node is complete when we use it to expand references at that node. This is equivalent to showing if P_i can reference P_j through a direct reference or expanded formal reference, then N_i is processed before N_j.

(by contradiction) Assume P_j is the first procedure such that P_i references P_j but N_j is processed when N_i is still unprocessed. Whether P_i directly references P_j (phase 1(ii)a) or P_i contains an expanded formal reference to P_j (phase 2(ii)a), a non-temporary edge (N_i,N_j) is created and update (N_i,N_j) performed to ensure that level(N_i) < level(N_j). The update procedure will maintain this ordering relation except if a path develops between N_j and N_i. This can only happen if a cycle develops in the graph and part (iii) of this theorem proves this cannot occur. Therefore, phase 2(i) will select N_i for processing before N_j. This conclusion contradicts the assumption; therefore, no such P_j exists. **qed**

V. Experience with the PFORT Verifier

We recall that the algorithm described here was designed for use in the PFORT Verifier [1], a program which checks FORTRAN programs for compliance with a portable subset of ANS FORTRAN. In this section, we briefly describe the uses of the call graph in the Verifier. We also analyze the input/output costs involved in our implementation of the algorithm and present timing data to support our analyses. Finally, we comment on the use of procedure parameters which we have encountered in FORTRAN programs.

The PFORT Verifier performs the parsing function of a compiler, checking syntax within subprograms; however, it *also* checks interprocedural communications through the use of parameter lists and COMMON. First, the Verifier checks the legality of procedure linkage using the call graph (i.e., the matching of parameter number, usage, type and structure).

Second, the Verifier uses the call graph to check for unsafe references; these are situations in interprocedural communication in FORTRAN, which may have implementation dependent results and which involve parameters which are set [1, p. 361]. We use a worst case analysis of each procedure to determine whether any formal parameters or global data areas can be set by the procedure. If a procedure P_i references procedure P_j we determine whether a

formal parameter which is set in P_j is associated with a formal parameter of P_i or a global variable. If so, we consider the parameter of P_i or the global variable as set. Such associations are collected in a bottom up fashion and stored in the call graph.

Third, the call graph is used to check the sharing of global data areas. The occurrence of a global data area in a procedure is recorded at its corresponding node in the graph. A graph walking algorithm checks the legality of COMMON block usage [1, p. 375].

Finally, a suitably printed version of the call graph provides a useful documentation and debugging aid. In our experience, the Verifier has proved especially useful for large programming packages where linkages are difficult to check manually. We find that users working with such packages prefer to check *all* procedure linkages; the requirement that all programs be well-formed does not unnecessarily restrict these users. Similarly, providing dummy definitions for library procedures in order to check links to them is not a source for serious complaint.

The above discussion shows the utility of the call graph. We now examine the Verifier implementation of our algorithm in an effort to demonstrate its utility. Consider the sequential secondary store to be a circular list of n tables. We access each entry in the list once in defining the nodes of the graph. In processing nodes, we search the list once for each entry, in an order determined by the algorithm (i.e., so that each node is processed after all its ancestors are already processed). We must search no more than n entries in the list in order to find any particular entry. Thus, we examine no more than n^2 entries in total. (Note: Since most programs are written in a top-down manner, the number of entries searched is usually much less than n^2).

We have shown in Section III that the in-core work of the algorithm is bounded by $O(n^3)$. Here we see that the input/output work is bounded by $O(n^2)$. Because in-core operations can be performed much faster than input/output fetches, we observe that the in-core operations of the algorithm show, in practice, shorter execution times than the input/output operations. In the data we timed using [11], the construction algorithm described here represents about 10% of the total processor time used by the Verifier. The processes of level updating and reference expansion (see Sections II, III) were consistently overshadowed in terms of processor time costs by the time required for input/output access to secondary storage. In one data set, these time requirements differed by a factor of 5.

During our timing studies we gathered some examples of programs which make heavy use of procedures as parameters. Two of these are numerical analysis packages with error recovery, output, and function evaluation all user-defined and passed to the package through procedure parameters. Program 1 consists of 57 procedures of which 10 contain an average of 3 formal procedure parameters each [12]. Program 2 consists of 17 procedures of which 6 contain an average of 2 formal procedure parameters each [13]. Both programs contain approximately 150 references with less than 5% being formal references. Only about 6% of all references in these programs contain actual procedure parameters. We also observed that as a consequence of the functional meaning of procedure parameters, often only single m-tuples of external procedures resulted from multiple references to one procedure by others.

In spite of the fact that these two programs make far greater use of procedure parameters than the average program processed by our Verifier, the time spent in the construction algorithm represented only about 10% of the total processor time of the Verifier run. The total run time for program 1 was 124 seconds; for program 2 it was 37.6 seconds. Thus, given the portability considerations in our application, the algorithm performs well.

VI. Comparisons and Conclusions

Control and data flow analysis programs which examine programs written in high level languages need a static representation of interprocedural relations. Here, we have presented one choice of representation, the call graph, and have carefully defined the set of programs to which it corresponds. We also have presented an algorithm for constructing the call graph of a program, analyzed it, and proved its correctness when applied to well-formed, nonrecursive

programs. We have discussed our experience with the algorithm in the PFORT Verifier. We have summarized our timing results which indicate that our algorithm is efficient given our implementation constraints. We wish to conclude by considering other approaches to call graph construction and comparing them to our algorithm.

To review, our construction algorithm performs some initialization and then adds nodes to the call graph one-by-one in such a way that nodes are added after all of their predecessors are already in the graph. Thus, when a node N_i is processed (i.e., added to the graph), the complete set of procedures it references can be determined since its procedure vector set is known. Therefore, when a node is added to the call graph all of the edges from that node are known.

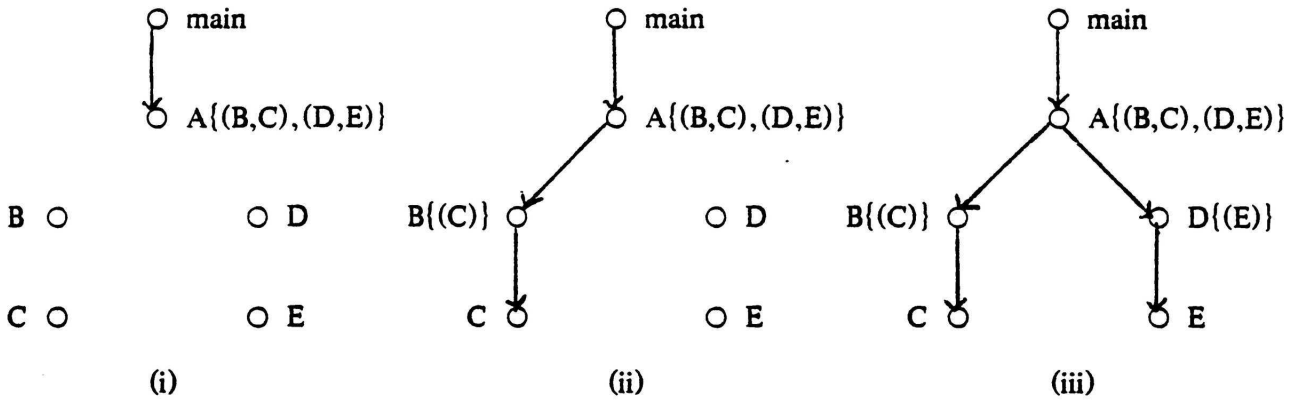
Another approach to construction might be to first determine all nodes and edges corresponding to direct references in the program. Then we could search through all references looking for those containing external procedures as parameters. We would form an $m+1$-tuple $P_{j_0}(P_{j_1}, \ldots, P_{j_m})$ from such a reference and examine P_{j_0} to determine if there are references containing the formal procedure parameters of P_{j_0} used as actual procedure parameters. Then, we could determine the external procedure $k+1$-tuples $P_{l_0}(P_{l_1}, \ldots, P_{l_k})$ obtained by expanding those references in P_{j_0} with respect to the m-tuple $(P_{j_1}, \ldots, P_{j_m})$. An algorithm recursively repeating this process, would construct the call graph by tracing paths of procedure parameter association in a depth-first manner. We would stop when there are no more references which can be generated (i.e., there are no new procedure m-tuples whose paths have yet to be fully traced). We note that it is possible to search for the table corresponding to a node many times during this algorithm.

A third approach might be to construct the call graph by tracing procedure parameter associations in a breadth-first manner. That is, determine all nodes and edges corresponding to direct references. Then form $m+1$-tuples $P_{j_0}(P_{j_1}, \ldots, P_{j_m})$ from each reference containing external procedures as parameters and record the m-tuple at N_{j_0}. The first pass through the graph would, for each P_{j_0} which has at least one m-tuple so recorded, expand any references within it using the m-tuples currently at N_{j_0} and mark those m-tuples "used". Each newly expanded reference might result in a new $k+1$-tuple $P_{l_0}(P_{l_1}, \ldots, P_{l_k})$ which will be recorded at P_{l_0} and marked "unused." The second pass through the graph would again visit each node and expand references there using the "unused" m-tuples currently recorded and marking them "used". We repeat these passes until there are no nodes which received new m-tuples in the last expansion pass. We note that it is possible to search for the table corresponding to a node several times in this algorithm; at one visit, expansions are only performed with respect to the "unused" m-tuples.

Figure 9 compares these two alternative approaches by applying them to the FORTRAN program of Figure 2.

Both of the alternative approaches add edges to the call graph in a manner where one cannot see a completed subgraph of nodes and all edges from those nodes at any point in time during the algorithm. We note that an analysis of a procedure which is dependent upon knowing its predecessors and immediate successors can be integrated with our algorithm, but not with the others.

Also, both alternatives may make multiple visits to a node, expanding references and using procedure definition tables while doing so. We believe other applications share our storage constraints on the procedure tables necessary for construction of the graph. The information in these tables may be embedded in data necessary for subsequent analysis which uses the call graph. Thus minimizing access to this information seems a justifiable goal and is realized in our algorithm. Since the call graph is widely cited as a useful data structure in data flow analysis [1,2,6,7,8,9,10], we feel our algorithm is of general utility.

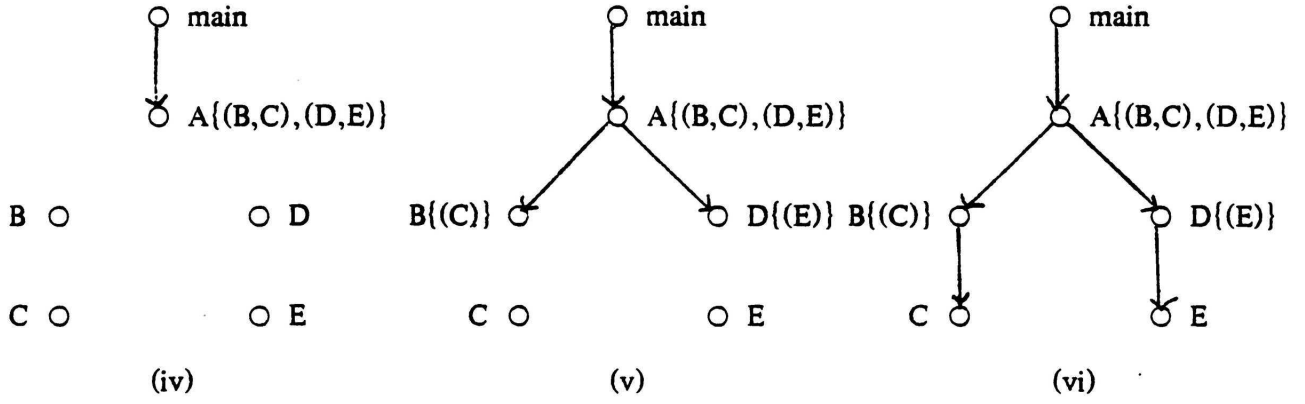


Alternative 1:

(i) after initial pass

(ii) after tracing "A(B,C)"

(iii) after tracing "A(D,E)"



Alternative 2:

(iv) after initial pass

(v) after recording (B,C), (D,E) in A and expanding statement 20

(vi) after recording (C) in B and (E) in D and expanding statements 30, 40

Figure 9

Acknowledgments

We wish to thank our colleagues A. D. Hall, Jr. and B. S. Baker for their helpful comments on this project and our colleagues A. V. Aho, S. R. Bourne, W. S. Brown, M. D. McIlroy, N. L. Schryer, and D. D. Warner for their help with this paper.

REFERENCES

1. Ryder, B. G., "The PFORT Verifier", Software Practice and Experience, Vol. 4, October-December 1974, pp. 359-377.
2. Allen, F. E., "Interprocedural Data Flow Analysis", Proceedings of IFIPS Conference 1974 (Software), pp. 398-402.
3. American National Standard FORTRAN, American National Standards Institute, New York, 1966.
4. "Clarifications of FORTRAN Standards-Initial Progress", Communications of the ACM, Vol. 12, pp. 289-294.
5. "Clarifications of FORTRAN Standards-Second Report", Communications of the ACM, Vol. 14, pp. 628-642.
6. Spillman, T. C., "Exposing Side — Effects in a PL/I Optimizing Compiler", Proceedings of IFIPS Conference 1971 (Computer Software), pp. 56-60.
7. Osterweil, L. J. and Fosdick, L. D., "Data Flow Analysis in Software Reliability," University of Colorado Computer Science Department, Technical Report No. CU-CS-087-76, May 1976.
8. Lomet, D. B., "Data Flow Analysis in the Presence of Procedure Calls," IBM Research Report RC 5728, November 21, 1975.
9. Rosen, B. K., "Data Flow Analysis for Recursive PL-I Programs," IBM Research Report RC 5211, January 9, 1975.
10. Allen, F. E. and Schwartz, J. T., "Determining the Data Relationships in a Collection of Procedures", unpublished, 1974.
11. Hall, A. D., "FDS: A FORTRAN Debugging System — Overview and Installer's Guide", Computing Science Technical Report #29, Bell Telephone Laboratories, April 1975.
12. Schryer, N. L., "A User's Guide to DODES, a Double Precision Ordinary Differential Equation Solver", Computing Science Technical Report #33, Bell Telephone Laboratories, August 1975.
13. Eldredge, B.D., Warner, D.D., "An Implementation of the Differential Correction Algorithm", Bell Laboratories internal memorandum.
14. Barth, Jeffrey, M., "Interprocedural Data Flow Analysis Based on Transitive Closure", University of California at Berkeley, Computer Science Department, Technical Report No. UCB-CS-76-44, September 1976.