Measurement based automatic analysis of FORTRAN programs *

E. C. RUSSELL and G. ESTRIN

University of California
Los Angeles, California

INTRODUCTION

A graph model of computer programs has been developed in a series of studies directed toward improving analysis of the structure of programs executed on different computer configurations. One inherent weakness of the model has been the need for estimates of the mean number of times a program would cycle around its loop structures and estimates of branching probabilities. Extensive improvements were made in the model on the assumption that good estimates would be inserted during a manual transformation of a given program into a computer processable graph representation. The combination of improved tools for measurement of program activities and recently developed analysis programs now permit automatic analysis of source programs. The automatic analysis is based on more reliable measured a priori statistics. This paper discusses a valuable by-product of this measurement and analysis which directs attention toward those parts of a program which are leading candidates for application of optimization techniques. In particular we present an example of the automatic analysis of programs written in the FORTRAN IV language. FORTRAN was selected as a first target for analysis because there exists a large number of time-consuming programs written entirely in FORTRAN.

The first part of the paper presents a simple summary of the graph model of programs. The second part deals with a particularization of this model to FORTRAN programs. The last part presents the results of an experiment which illustrates the use of the graph model based on measurements. Automatic analysis procedures obtain an ordering of FORTRAN statements according to their frequency and time-of-execution along with other structural data.

The computational model

A principal goal of the several investigators who have developed a graph model for representing computer programs has been to study the structure of the program for analysis of its space and time requirements on a variety of hardware configurations. This model is illustrated in Figure 1.

The salient properties of the model are:

1. There is one original vertex ($v_1$)
2. There is one terminal vertex ($v_n$)

(When either of these conditions does not exist, a pseudo-vertex is appended. Thus multiple terminal vertices are connected to a pseudo terminal vertex and multiple origin vertices are preceded by a pseudo-origin vertex.)

3. Each vertex is connected to other vertices by directed arcs. There may be more than one arc directed either to or from a vertex, in which case the interaction of the arcs is specified by a logic condition. Arcs may enter a vertex or leave a vertex in a simultaneous (“AND”) fashion or in an exclusive (“OR”) fashion. The “AND” condition indicates situations in which multiple-processors may be used to advantage.

4. Cycles are represented by arcs which are directed from $v_j$ to $v_i$, where $j \leq i$. This implies that care is taken in the choice of numbers for vertices.

Thus far only structural properties of the program are described. The remainder of the description is concerned with the definition of the operation to be performed at each vertex. Such a description includes:

* This research was sponsored by the Atomic Energy Commission [AT(11-1) Gen 10 Project 141, the Advanced Research Projects Agency ARPA SD 184, and the Office of Naval Research, Information Systems Division. Reproduction in whole or part is permitted for any purpose of the United States Government.
5. The block of instructions required to perform the computation.

6. The identification of required input data.

7. The identification of required output data.

8. The amount of intermediate (temporary) storage required.

One of the goals of this modelling process is the a priori analysis of the execution of a program on a particular hardware configuration. In order to complete the program description for this purpose, the following additional parameters are required:

9. For each “exclusive or” output, a probability for each arc.

10. The iteration factor for each cycle (These two parameters may be derived from a single “frequency count” for each vertex.)

This model has the advantage over some others (Rodriguez), that only a single type of vertex exists. Variations such as input connections, output connections, actual computations, etc., are described parametrically. This leads quite naturally to the description of the graph in terms of Boolean matrices.

Additional generalizations are also possible. In the experiments performed, FORTRAN statements are taken as the primitive computations represented as vertices. It may not always be desirable to make such a choice. In highly parallel computations, a refinement or “blowup” of a particular vertex may reveal parallel processing potential within complex statements. Conversely, large portions of the program may represent such a minor portion of the computation time, that it may be helpful to collapse portions of the graph into single vertices. Such transformations are possible, given that all the above descriptive parameters are available. For the present an entire FORTRAN program is analyzed and represented as a single graph. If subprograms are referenced, they must be described parametrically prior to their inclusion. The parametric description of a subprogram can be developed independently and a library of subprograms prepared.

**Particularization to FORTRAN programs**

**Nomenclature**

**Vertex definition**

The application of the model to FORTRAN will be made at the statement level. That is, each executable FORTRAN statement will be assigned to a unique vertex. Other choices could be made: for example, if more than one vertex per statement were permitted then each arithmetic operator could be assigned to a unique vertex and analysis of parallel arithmetic expressions could be performed. This would result in graphs with large numbers of vertices. On the other hand, a single vertex could be made to represent several statements or even an entire subprogram. This would provide for smaller graphs but could obscure potential parallelism. With an initial analysis at the statement level, a refined analysis could then be performed on those portions of the program which consume significant portions of execution time, whereas less significant portions can be collapsed into fewer vertices.

Each program will contain one origin vertex ($w_i$) which is predecessor to all other vertices in the graph. (A program with multiple entries will still have one pseudo-origin with immediate outbranchings to the various entry points.)

Each program will contain one terminal vertex ($w_n$) which is successor to all other vertices in the graph. (Multiple exits from the program will be tied to this one terminus.)

For each vertex, the input data set, $I_i$, consists of the names of those variables which are referenced by the statement represented by the vertex. For example, for an arithmetic assignment statement this includes all variables in the expression and any subscripts.
For each vertex, the output data set, \( O_i \), consists of the names of those variables which are modified by the execution of the statement represented by the vertex.

Arc definitions

Three categories of connections between vertices will be established: sequential connections, logical connections and loops.

Each executable FORTRAN statement which does not explicitly specify the successor statement implicitly "falls through" to the next executable statement in the program. These connections (of the form \( (w_i, w_{i+1}) \)) are designated sequential connections. There can be only one sequential successor vertex for each vertex.

Each executable FORTRAN statement which does specify explicitly one or more successor statements produces connections which are designated logical connections. The connections are of the form \( (w_i, w_j) \) where \( i \neq j \).

Some of the arcs of a program represent the return path of a programmed loop. Of course, any one arc in a cycle can be selected as the return path but often one is recognizable from the syntax of the language (e.g., the FORTRAN "DO" statement). Each arc which closes a programmed loop is designated as a feedback connection.

Graph representation of FORTRAN IV

A complete specification for the graph representation of FORTRAN statements is given in the Appendix.

Experimental measurements

Two attributes of the vertices of a graph are determined from experimental measurement. These are: (1) vertex activity number and (2) vertex single-execution time.

Vertex activity in a sequential program

Vertex activity can be determined by making some minor modifications to the source program before execution. These modifications are similar to those outlined in "SNUMPER COMPUTER, A Computer Instrumentation Automaton," Estrin. For these experiments, however, only vertex activity and loop activity are desired and not all arc activity as in the reference. Because of this, the artifact introduced is considerably reduced. The required measurements are as follows:

1. the number of times the program is entered.
2. the number of times each labeled statement is executed with special treatment of DO loops and unusual branches.
3. the number of times the auxiliary statement of a "logical if" is executed.

From these measurements, the entire set of vertex activities and loop factors may be calculated.

Introduction of artifact

The artifact introduced into FORTRAN programs consists of a series of subroutine calls. Routine EMIT \((i)\) is introduced to perform the activity of counting the number of times vertex \( i \) is executed. For efficient execution, the subroutine EMIT is written so as to perform the emit table address calculation and then modify the calling instructions to perform the counter incrementing with "inline" code after the first encounter of that call.

The monitoring of vertex activity must be introduced at the following points in a FORTRAN program:

1. Measurement of origin vertex:
   Following the SUBROUTINE or FUNCTION statement or preceding the first executable statement of main program, insert CALL EMIT \((i)\). In order to properly monitor entries into a program via the ENTRY statement, the CALL EMIT which follows an ENTRY must not be executed if control "falls through" from above the ENTRY statement. For example:

   \[
   A = B \\
   ENTRY FIRST \\
   CALL EMIT (i) \\
   C = D
   \]

   would record an erroneous measure of entry to the program at FIRST. In such a case the following is a correct modification of the program:

   \[
   A = B \\
   GO TO XX \\
   ENTRY FIRST \\
   CALL EMIT (i) \\
   XX CONTINUE \\
   C = D
   \]

2. Measurement of labeled statement:
   Replace labelled (executable) statement \( L : s \) by \( L : CALL EMIT (n) \) where \( L \) is the state-
ment label (external formula number), s is the statement and n is the vertex number assigned to s.

3. Measurement of DO loops:
   Insert a CALL EMIT \( (n+1) \) after the DO statement where n is the vertex number assigned to the DO statement.

4. Measurement of unusual branches:
   In the case where CALL or READ statements provide unusual returns, the activity of the succeeding statement must be monitored. For example:
   
   \[
   \text{CALL SUB1 (A,B,}\$2) \\
   A = D \\
   2 \ B = C
   \]
   The activity of the statement \( A = D \) is not known unless specifically monitored.

5. Measurement of logical IF statements:
   In order to monitor the conditional statement, it is necessary to introduce "CALL EMIT" but the restriction is a single statement as the conditionally executed branch. Therefore the following scheme is proposed (as in Estrin\(^9\)).
   
   a. Negate the logical condition
   b. Replace the conditional statement by a 'GO TO' statement branching around the original conditional statement and an inserted 'CALL EMIT.' For example:
   
   \[
   \text{IF } \neg \neg (t) \text{ s becomes } \text{IF} \neg (t) \text{ GO TO XXX} \\
   \text{CALL EMIT } (i+1)
   \]
   
   XXX \text{ CONTINUE}
   
   c. On the other hand, if s cannot pass control to the next executable statement, the control is simpler:
   
   \[
   \text{IF } (t) \text{ s becomes } \text{IF } (t) \text{ s} \\
   \text{CALL EMIT } (i+2)
   \]
   where \( w \) is the vertex assigned to the IF statement.

Two problems in implementation now occur. If this logical IF is the terminal statement of a DO loop, the program meaning has been altered. Also, the statement label XXX may already have been used. If the logical IF is the terminal statement of a DO loop, the DO reference should be changed to XXX. (This might be most easily implemented by adding a unique terminal for each DO of the form XXX \text{ CONTINUE}.) The uniqueness of statement labels can be guaranteed by keeping track of all statement label references or generations during the source program scan.

Subprogram inclusion

There are at least two methods of including subprograms in the analysis. The first is to do an independent analysis of the subprograms and provide only the aggregate attributes of the subprogram for inclusion in the programs which call it.

The second method is to combine the subprogram with its dynamic ascendants replacing CALL statement by GO TO XX statements where XX is the first executable statement of the subroutine (now a part of the same graph). The RETURN from a subroutine is replaced by a GO TO \((n_1,n_2,n_3,\ldots)\) \(XX\) where \(n_1, n_2, \ldots\) are the statements after each CALL to the subroutine and \(XX\) is an index specifying which particular call is being executed. In the process of including a subprogram in the graph, all of the COMMON statements can be systematically eliminated. This involves transforming variables used in the subprogram to their calling program equivalent names and generating unique names for local subprogram variables.

Discussion of experiment

A process flow chart describing the experimental procedure is illustrated in Figure 2. The program to be analyzed goes through a process which produces two types of outputs. The first type of output (on the left) is the actual graph model of the program in machine-processable form. The second type is the modified source FORTRAN program ready for compilation and for execution during which raw frequency data is to be gathered. The post-execution program analyzer accepts the two sets of output data and produces the listed vertex and program attributes.

Figure 3 gives a listing of our example program which was written to simulate the UCLA Boolean Analyzer.\(^9\) It has 127 vertices as defined in Part 2 of this paper. These 127 vertices are numbered on the right of the listing. Figure 4 shows the computer gen-
When it is desired to study the detailed structure of the execution time, in this example further analysis indicated that a change in sequencing of the two subroutines account for 76 percent of total time. These vertices are listed in Table I and show from vertex to vertex i. The connection matrix has been triangularized such that non-triangular elements would represent program loops. The actual feedback arcs of the program loops are represented as a list of ordered pairs and shown in the upper triangular portion. Figure 6 is a list of vertex attributes. The headings are defined in the legend at the top of the table. Figure 7 shows computer generated plots of the frequencies of execution and computation times of the vertices. They are ordered to produce a monotonically decreasing plot using a logarithmic vertical ordinate.

Now consider a programmer seeking to make use of the above analysis results. First, he must have reasonable confidence in the set of input data used to obtain vertex measurements. The plot of ordered execution times (Figure 7) then permits him to select the most important vertices as candidates for optimization and to determine their characteristics from the list of vertex attributes (Figure 6). In the present example, seven of the vertices use 93.4 percent of the total time. These vertices are listed in Table I and show that two small subroutines account for 76 percent of the execution time. In this example further analysis indicated that a change in sequencing of the two subroutines should in itself produce considerable improvement and validating measurements are in process. When it is desired to study the detailed structure of the routines represented by vertices other analysis tools come into play. For example a question was raised in one study as to the distribution of machine language instruction types executed in a subroutine. A machine language instrumentation program was utilized to obtain the answer.

Figure 2—The measurement and analysis processor

Figure 3—FORTRAN example (the Boolean analyzer simulator)
REFERENCES

1 B Bussell
Properties of a variable structure computer system in the solution of parabolic partial differential equations
PhD Dissertation University of California Los Angeles 1962

2 G ESTRIN R TURN
Automatic assignment of computations in a variable structure computer system
Trans IEEE EC-12 December 1963 756-773

3 D F MARTIN G ESTRIN
Experiments on models of computations and systems
Trans IEEE EC-16: February 1967 59-69

4 G R WOOD
Application of a restructurable computer system to numerical weather prediction computations
UCLA Report No 66-14 February 1966

5 F C RUSSELL
Automatic assignment of computational tasks in a variable structure computer
UCLA Report No 63-45 August 1963

6 J L BAER
Graph models of computations for computer systems
PhD Dissertation Department of Engineering University of California UCLA Report No 68-46 October 1968

7 D P BOVET
Memory allocation in computer systems
PhD Dissertation Department of Engineering University of California UCLA Report No 68-17 June 1968

8 J E RODRIGUEZ
A graph model for parallel computations
PhD Dissertation Massachusetts Institute of Technology September 1967

9 G ESTRIN D HOPKINS B COGGAN
S CROCKER
Sniper computer—a computer instrumentation automaton
Proc S J C C 1967

10 M A MARIN
Investigation of the field of problems for the Boolean analyzer
PhD Dissertation Department of Engineering University of California UCLA Report No 68-28 June 1968

APPENDIX

Complete specification of the graph representation of FORTRAN IV

In the following specification for the representation of FORTRAN statements, these symbols will be used:

1. \( w_i \) represents the vertex assigned to the current statement.
2. \( w_{i+1} \) represents the next sequential statement.
3. \( w_n \) represents the statement with FORTRAN label \( n \).
4. \( \langle w_i, w_j \rangle \) represents an arc directed from \( w_i \) to \( w_j \).
5. A variable will be represented in a dictionary by its attributes:

\[ p_j \text{—the number of bytes of storage (precision)} \]
\[ n_j \text{—the name of the variable} \]
Figure 5—Connection matrix of Boolean analyzer
### Vertex Description Data

<table>
<thead>
<tr>
<th>VERTEX NO.</th>
<th>PANEL NO.</th>
<th>LIFE</th>
<th>LOGIC</th>
<th>LOCAL</th>
<th>STORED</th>
<th>EXECUTION TIME</th>
<th>COMPUTATION TIME</th>
<th>RELATIVE IMPORTANCE</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.1</td>
<td>0.2</td>
<td>127</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>+2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.2</td>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>+3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1.5</td>
<td>0.2</td>
<td>42</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>+4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
<td>0.2</td>
<td>51</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>+5</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.2</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>+6</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1.1</td>
<td>0.2</td>
<td>127</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>+7</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.2</td>
<td>31</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>+8</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
<td>0.2</td>
<td>51</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>+9</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.2</td>
<td>31</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>+10</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1.1</td>
<td>0.2</td>
<td>127</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>+11</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.2</td>
<td>31</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>+12</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
<td>0.2</td>
<td>51</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>+13</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.2</td>
<td>31</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>+14</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>1.1</td>
<td>0.2</td>
<td>127</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>+15</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.2</td>
<td>31</td>
<td>15</td>
</tr>
</tbody>
</table>

**Total Predicted Execution Time:** 22693792.0

**Total Predicted Local Memory:** 13.0

**Total Predicted Program Storage:** 143.0

### Figure 6—Vertex attributes of Boolean analyzer
Program delimiters

Three FORTRAN statements will be processed as program delimiters:

1. SUBROUTINE name (a1,a2,a3, ... ,an). will be represented as vertex Wi with input data set (a1,a2,a3, ... ,an). This input data set is given a special designation Ip, the parameter list.
2. type FUNCTION name *s (a1,t12,a3, ... ,an) is represented just as in (1) with the following additions. The function name is added to a list of functions to be recognized as distinct from input variables.
3. END will be represented as the terminal vertex wz. All statements which would terminate the execution of the program will be connected to wz. The output data set of wz is Ip.

Specification statements

The specification statements are non-executable. They provide information about the type, precision and dimensionality of variables, together with possible sharing of storage between variables.

1. type *s a1 *s1 (k1)/x1/, a2 *s2 (k2)/x2/, ... ,an *sn (kn)/xn/. Each variable in a type statement will be entered in the dictionary according to the following conventions:
   a. name (ni) is stored directly
   b. type (ti) is taken from the statement type (INTEGER, REAL, LOGICAL, COMPLEX)
   c. precision (pi) is either determined from the statement (if included) or defaults to the following values:

<table>
<thead>
<tr>
<th>type</th>
<th>t_i</th>
<th>p_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>REAL</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

d. size si is computed as product of ki expression.

2. DIMENSION a1 (k1), a2 (k2), ... , an (kn) Dimensioned variables will be entered into the dictionary (if not previously defined by a type of COMMON statement) and the size (si) is computed from ki.

3. COMMON /r1/a1 (k1), ... /r2/an (kn) Common variables are entered into the dictionary (if not previously defined by a type or dimension statement) and the size (si) is computed from ki. In addition, all variables in COMMON blocks are placed in a special set, the common list L.

4. EQUIVALENCE (a1, a2, a3, ... ), (a4, a5, ... ), ... The equivalence statement has two effects on the program description. First, the storage allocation for defined variables is altered. If the variables in an equivalence class are not completely overlapped then some modification of the calculation of the amount of storage is required. This is especially important when such variables lie in COMMON blocks. If storage allocation were the only reason for using equivalence statements, they could be ignored in this analysis. However, the values of variables may be referenced by any name which happens to be assigned to the cell. Thus, it is imperative to include all “aliases” when preparing the input and output data sets. A well-written program for a single processor may obscure potential parallelism due to storage sharing.

Control statements

The control statements are the source of the majority of logical control arcs. The FORTRAN statements included in this group are the GO TO statements (unconditional, computed and assigned), the IF statement (arithmetic and logical), the DO, CONTINUE, PAUSE, RETURN, CALL, and STOP statements.

1. GO TO statements
   a. GO TO n
      the unconditional transfer statement pro-
duces an arc of the form \((w_{i-1}, w_i)\). This statement does not result in a new vertex.

b. GO TO \((n_1, n_2, \ldots, n_k)\)
the computed GO TO statement is represented by a vertex \((w_i)\) with multiple out-branchings \((w_i, w_{i+1}), (w_i, w_{i+2}), \ldots, (w_i, w_{i+k})\). \(w_i\) has exclusive-OR output logic. \(I_i = \{j\}\)

c. GO TO \(j, (n_1, n_2, \ldots, n_k)\)
the assigned GO TO statement is represented exactly the same as (b)

2. IF statements
a. IF \((e)\) \(n_1, n_2, n_3\)
The arithmetic if statement is represented by a vertex \((w_i)\) with exclusive or output logic and out-branching arcs \((w_i, w_{i+1}), (w_i, w_{i+2})\). The input data set consists of all variables referenced in the arithmetic expression "e."

b. IF \((e)\) \(s\)
the logical if statement is represented by a vertex \((w_i)\) with exclusive or output logic and outbranching arcs \((w_i, w_{i+1}), (w_i, w_{i+2})\). The input data set consists of all variables referenced in the logical expression "e." Statement "s" is treated as a separate vertex \((w_{i+1})\).

3. DO \(n i = m_1, m_2, m_3\)
The DO statement is represented by a vertex \((w_i)\) with \(I_i = \{m_1, m_2, m_3\}\) and \(O_i = \{i\}\). Arcs produced are \((w_i, w_{i+1})\) and \((w_i, w_{i+2})\). The latter arc is a feedback connection for the loop.

4. CONTINUE
The CONTINUE statement is represented by a vertex \((w_i)\) with \(I_i = \emptyset, O_i = \emptyset\) and outbranching arc \((w_i, w_{i+1})\).

5. PAUSE
The PAUSE statement is represented by a vertex \((w_i)\) with \(I_i = \emptyset, O_i = \emptyset\) and outbranching arc \((w_i, w_{i+1})\).

6. RETURN
The RETURN statement produces an arc of the form \((w_{i-1}, w_i)\). This statement does not result in a new vertex. \((w_i, \text{pseudo-terminus of the program})\)

7. STOP
The STOP statement is represented just the same as a return.

8. CALL name \((a_1, a_2, \ldots, a_n)\)
The CALL statement is represented by a vertex \((w_i)\). In the usual case there is a single outbranching \((w_i, w_{i+1})\). However, it is possible to specify other return paths as parameters to the subroutine in the form CALL name \((a_1, \ldots, \& a_n)\) in which \&a_n and \&a_n represent statement numbers in the calling program. In this case, multiple outbranching arcs are generated \((w_i, w_{i+1}), (w_i, w_{i+2}), \ldots, (w_i, w_{i+n})\) with output logic exclusive or. In the absence of further information, all parameters are assumed to be members of both \(I_i\) and \(O_i\). Also any COMMON variables must be considered part of both sets. Thus

\[\{I_i = a_1, a_2, \ldots, a_n, I_e\}\]

\[\{O_i = a_1, a_2, \ldots, a_n, I_e\}\]

Input/output statements
There are five I/O statements which will be included in this discussion: READ, WRITE, END FILE, REWIND and BACKSPACE.

1. READ \((a, b, END = n_1, ERR = n_2)\) \(\)\text{list} \(\)
This statement is represented by a vertex \((w_i)\) with input data set \(I_i = \{a, m_1, m_2, \ldots\}\) where \(\{m_1, m_2, \ldots\}\) represent any loop limits in the list. \(O_i = \{\text{list}\}\). The exclusive-outbranching arcs are \((w_i, w_{i+1})\) and optional arcs \((w_i, w_{i+2})\) and \((w_i, w_{i+3})\). Other variations on the READ statement are processed similarly.

2. WRITE \((a, b)\) \(\)\text{list} \(\)
This statement is represented by a vertex \((w_i)\) with input data set \(I_i = \{a, m_1, m_2, \ldots, \text{list}\}\) where \(\{m_1, m_2, \ldots\}\) represent any loop limits in \(\text{list}\) and \(\text{list}\) represents the variables to be output. \(O_i = \emptyset\) and there is a single outbranching \((w_i, w_{i+1})\).

3. ENDFILE \(a\)
REWIND \(a\)
BACKSPACE \(a\)
The structure of these three statements is the same. Each is represented by a vertex \((w_i)\) with \(I_i = \{a\}, O_i = \emptyset\) and a single outbranching arc \((w_i, w_{i+1})\).

Arithmetic statements
The arithmetic statement, most generally of the form \("a = e\)" is represented by a single vertex, \(w_i\) with \(O_i = \{a\}\) and \(I_i = \{k_1, \ldots, k_n, b_1, \ldots, b_n\}\) where the \(k\)'s are subscripts used anywhere in the statement and the \(b\)'s are variables in the arithmetic expression "e." A single outbranching arc \((w_i, w_{i+1})\) is produced.