MCR-75-390
NAS9-13616

Volume III

Phase II
Final
Report

NASA CR-
144530

October 1975

As-Built Specifications for the Prototype Language and Module Library

Scheduling Language and Algorithm Development Study

(NASA-CR-144530) SCHEDULING LANGUAGE AND ALGORITHM DEVELOPMENT STUDY. VOLUME 3, PHASE 2: AS-BUILT SPECIFICATIONS FOR THE PROTOTYPE LANGUAGE AND MODULE LIBRARY Final Report (Martin Marietta Corp.) 596 p

N76-11753

Unclas

G3/61 01861

MARTIN MARIETTA

FOREWORD

This is the final report for Phase II of the Scheduling Language and Algorithm Development Study (NAS9-13616). It is contained in three volumes. The objectives of Phase II were to implement prototypes of the Scheduling Language called PLANS and the scheduling module library that were designed and specified in Phase I.

Volume I of this report contains data and analyses related to a variety of algorithms for solving typical large-scale scheduling and resource allocation problems. The capabilities and deficiencies of various alternative problem solving strategies are discussed from the viewpoint of computer system design.

Volume II is an introduction to the use of the Programming Language for Allocation and Network Scheduling (PLANS). It is intended as a reference for the PLANS programmer.

Volume III contains the detailed specifications of the scheduling module library as implemented in Phase II. This volume extends the Detailed Design Specifications previously published in the Phase II Interim report (April 1975).

CONTENTS

Figure

Figure

Table

INTRODUCTION

During Phase I of the Scheduling Language and Algorithm Development Study, a computer programming language and a library of modules (subroutines) were designed and functionally specified. These functional specifications appear in Volume III of the Phase I Final Report. The interested reader can also refer to Volumes I and II of the Phase I Final Report which provide overview and usage information. The information in this introduction is intended to provide a brief background and a context for the detailed design specifications which follow in subsequent sections.

The products of this study are called PLANS (Programming Language for Allocation and Network Scheduling) and the PLANS Module Library. These products are designed to reduce substantially the costs and span times of implementing software to solve scheduling and resource allocation problems. Most programs associated with planning and/or managing the activities and resources in a large operational system are programs that should be implemented using the products of this study.

It should be understood that this study does not develop complete scheduling system application programs or a language in which a user communicates with a scheduling system. PLANS users are assumed to be charged with the design or modification of application programs related to scheduling and resource assignment (allocation), that could be part of a scheduling system. Also, potential users are assumed to have a problem orientation (as opposed to computer programming orientation) that is not limited to aerospace system applications. Thus, a language and associated basic data structure and routines are being developed to provide high-level but flexible programming capability to analysts with a wide variety of scheduling and resource allocation problems.

The Phase I study showed that increasing correspondence between the individual logical operations of the problem solution and the individual computer program statements, greatly increased the inherent usability of the language for the user concerned with scheduling/resource allocation problems. However, practical limits to doing this in the basic language must be recognized, because too much individual statement power would unnecessarily reduce the flexibility of the language.

To provide functions that have more power than available from individual PLANS statements, the Phase I study specified a flexible data structure especially suited for describing operating systems, and a library of subroutines called modules for use with the structure and PLANS (the scheduling language). This combination of a flexible language and data structure, plus a library of preprogrammed modules constitutes a software programming system. The PLANS Programming System consists of three elements which together simplify the development or modification of scheduling/resource allocation software.

In summary, these products are:

1) A high-level programming language for writing scheduling programs that

- Use typical arithmetic, transfer-of-control, conditional and iterative statements in logic and computational modules,
- access and manipulate the data structure for problem/module support,
- define the problem/objectives and manipulate the library modules;

2) A flexible data structure specially suited for describing the characteristics of the systems to be scheduled;

3) A library of preprogrammed logic modules to
   - access the data structure for system operations data,
   - implement frequently used scheduling/resource allocation problem solution algorithms.

This programming system meets the prime requirements for (1) substantially reducing software programming and reprogramming times, (2) desensitizing programs to problem changes, and (3) accommodating a wide range of problem types and applications with generic logic codes.

1.0 DETAILED DESIGN SPECIFICATION FOR PLANS PROGRAMMING LANGUAGE

PLANS has been implemented by a two-pass translator. The first pass performs a syntax check with appropriate error messaging and recovery. The second pass translates PLANS to PL/I. Both of these programs were developed using the Martin Marietta Aerospace Translator-Writing System (TWS). TWS is a system in which translators are automatically generated from an appropriate formal definition, in the form of an "augmented grammar", of the translation process. Since the augmented grammar definitions of the PLANS Syntax Checker and the PLANS Code Generator are more complete and rigorous than those achievable by most other means, they provide an excellent means of functional definition. Before they can be understood, however, it will be necessary for the reader to familiarize himself with TWS. A functional description of TWS is included in this document as Appendix 1. It is suggested that the Appendix be consulted before proceeding.

1.1 PLANS Lexical Analyzer

The state transition diagram for the PLANS lexical analyzer is shown in Fig. 1.1-1. This lexical analyzer is used in both the syntax checker and the code generator. Notice that comments are removed by the lexical analyzer, and need not be considered in the augmented grammars.

1.2 Specialized PLANS Output Routine

The output routine @OUT was changed for the PLANS code generator to allow the user to insert code prior to the current line of code in the code buffer. This allows the user to start outputting a line of data and at any point, output any number of lines prior to the current line by incrementing a grammar switch called NORMAL_MERGE_SWITCH. For example,the user can output a segment of line A, then decide a line B is needed prior to A. If while outputting line B the need for a line C before line B is needed the new line C can be output, then more added to B and output, as well as more added to A. This procedure can be nested to any arbitrary depth by incrementing and the decrementing NORMAL_MERGE_SWITCH, which corresponds to the current nesting level.

1.3 PLANS Syntax Checker

The error message declaration for the syntax checker is shown in Fig. 1.3-1. Fig. 1.3-2 contains the complete augmented grammar for the syntax checker. It is suggested that this grammar be studied before that of the code generator, which is somewhat more complex.

1.4 PLANS Code Generator

The error message declaration for the code generator is shown in Fig. 1.4-1. Fig. 1.4-2 contains the augmented grammar for the PLANS→PL/I translation pass.

1.5 Comparison of Implementation with Original Functional Specification

With minor exceptions, all the functional capabilities outlined in the PLANS language specification (Phase I Final Report, Vol III, September, 1974) have been provided in the implemented translator. In addition, numerous capabilities which were not specified have been provided. This section lists the deviations from the specifi-

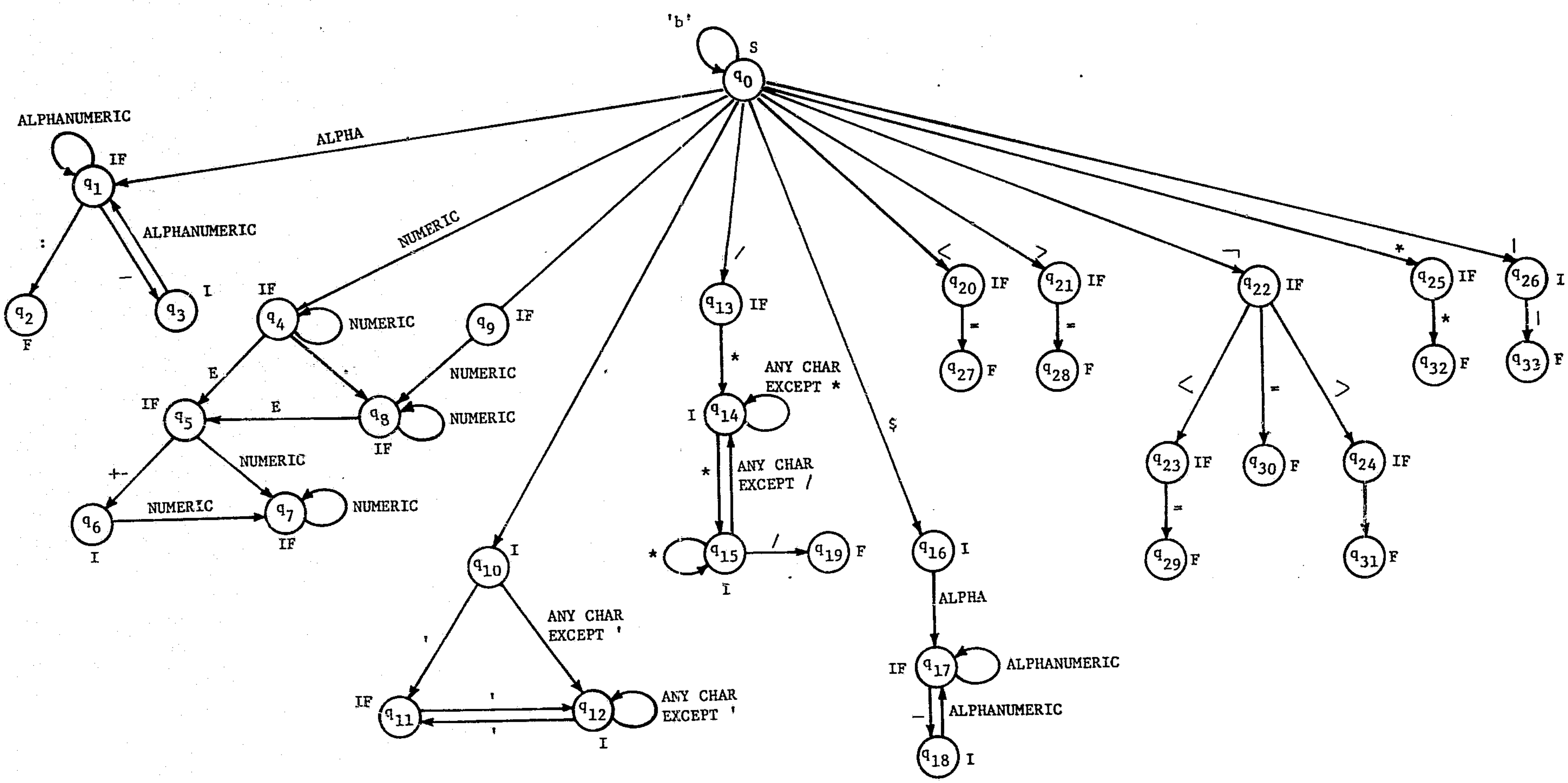


Figure 1.1-1 State Transition Diagram for PLANS Lexical Analyzer

```
/* ERROR MESSAGES */
  DECLARE @ERROR_MESSAGE(92) CHARACTER(60) VARYING STATIC INIT(
   'S:MAIN PROCEDURE NAME MISSING',                                   /*01*/
   'S:MISSING PROCEDURE STATEMENT',                                   /*02*/
   'S:INTERNAL PROCEDURE NAME MISSING',                               /*03*/
   'W:MISSING LEFT PARENTHESIS ASSUMED PRESENT',                      /*04*/
   'S:MISSING OR ILLEGAL MAIN PROCEDURE OPTION LIST',                 /*05*/
   'S:MISSING OR ILLEGAL PROCEDURE PARAMETER LIST',                   /*06*/
   'S:MISSING MAIN PROCEDURE END-STATEMENT',                          /*07*/
   'S:STATEMENTS AFTER END-STATEMENT',                                /*08*/
   'S:UNRECOGNIZABLE STATEMENT',                                      /*09*/
   'S:MISSING INTERNAL PROCEDURE END-STATEMENT',                      /*10*/
   'N:NONITERATIVE DO-GROUP EXECUTES MORE EFFICIENTLY HERE',          /*11*/
   'S:MISSING BEGIN-BLOCK END-STATEMENT',                             /*12*/
   'S:MISSING OR ILLEGAL BOOLEAN EXPRESSION',                         /*13*/
   'W:TRACE IGNORED -- TRACE OPTION NOT SELECTED',                    /*14*/
   'S:TRACE LEVEL NOT SPECIFIED',                                     /*15*/
   'S:"$NULL" CANNOT BE MODIFIED OR PASSED AS PARAMETER',             /*16*/
   'S:MISSING WORD "BEFORE"'                                          /*17*/
   'S:MISSING WORD "AT"',                                             /*18*/
   'S:UNINTERPRETABLE ITERATION CLAUSE IN DO-STATEMENT',              /*19*/
   'W:MISSING WORD "OF" ASSUMED PRESENT',                             /*20*/
   'S:MISSING OR ERRONEOUS TREE NODE EXPRESSION',                     /*21*/
   'W:MISSING WORD "USING" ASSUMED PRESENT',                          /*22*/
   'S:MISSING OR ERRONEOUS TREE NAME',                                /*23*/
   'W:MISSING WORD "ALL" ASSUMED PRESENT',                            /*24*/
   'N:LABEL IGNORED -- PLANS DOES NOT ALLOW MULTIPLE CLOSURE',        /*25*/
   'S:UNEXPECTED END OF FILE ENCOUNTERED',                            /*26*/
   'W:MISSING DECLARATION LIST',                                      /*27*/
   'S:EXTRANEOUS INFORMATION IN BOOLEAN EXPRESSION',                  /*28*/
   'W:SEMI-COLON AFTER IF-CLAUSE IGNORED',                            /*29*/
   'S:THEN-CLAUSE REQUIRES EXECUTABLE STATEMENT OR BLOCK',            /*30*/
   'S:ELSE-CLAUSE REQUIRES EXECUTABLE STATEMENT OR BLOCK',            /*31*/
   'S:MISSING THEN-CLAUSE IN IF-STATEMENT',                           /*32*/
   'W:MISSING WORD "AS" ASSUMED PRESENT',                             /*33*/
   'S:ILLEGAL MULTIPLE NODE REFERENCE',                               /*34*/
   'S:ELSE-CLAUSE NOT ASSOCIATED WITH IF-STATEMENT',                  /*35*/
   'S:DECLARATIONS ALLOWED ONLY AT START OF BLOCK',                   /*36*/
   'S:MISSING DO-GROUP END-STATEMENT',                                /*37*/
   'S:MISSING WORD "TAKEN"',                                          /*38*/
   'S:MISSING OR ERRONEOUS ARITHMETIC EXPRESSION',                    /*39*/
   'W:MISSING WORD "A" ASSUMED PRESENT',                              /*40*/
   'W:MISSING WORD "TIME" ASSUMED PRESENT',                           /*41*/
   'W:MISSING WORD "TO" ASSUMED PRESENT',                             /*42*/
   'S:MISSING OR ERRONEOUS LABEL REFERENCE',                          /*43*/
   'S:MISSING OR ERRONEOUS PROCEDURE REFERENCE',                      /*44*/
   'S:MISSING OR ERRONEOUS CALL ARGUMENT',                            /*45*/
   'S:EXTRA ARITHMETIC OPERATOR',                                     /*46*/
   'S:MISSING OR ILLEGAL OPERAND IN ARITHMETIC EXPRESSION',           /*47*/
   'S:MISSING OR ERRONEOUS "NUMBER"-FUNCTION ARGUMENT',               /*48*/
   'S:MISSING OR ILLEGAL TREE LABEL',                                 /*49*/
   'N:"NEXT" IS USED AS LABEL, NOT SUBSCRIPT KEYWORD',                /*50*/
   'S:"NEXT" ILLEGAL HERE',                                           /*51*/
   'N:"LAST" IS USED AS LABEL, NOT SUBSCRIPT KEYWORD',                /*52*/
   'S:"ALL" ILLEGAL HERE',                                            /*53*/
   'S:"LABEL"-FUNCTION CANNOT BE USED AS DIRECT TREE LABEL',          /*54*/
   'S:INDIRECT REFERENCE ILLEGAL HERE',                               /*55*/
   'S:MISSING OR ERRONEOUS "LABEL"-FUNCTION ARGUMENT',                /*56*/
   'S:MISSING OR ERRONEOUS STRING EXPRESSION',                        /*57*/
   'S:SUBSCRIPT KEYWORD "FIRST" NOT ALLOWED AS LABEL',                /*58*/
   'W:EXTRA COMMA IGNORED',                                           /*59*/
   'W:MISSING COMMA ASSUMED PRESENT',                                 /*60*/
   'S:MISSING OR UNINTERPRETABLE ELEMENT IN PROPERTY LIST',           /*61*/
   'S:MISSING OR ILLEGAL INDIRECT TREE NODE REFERENCE',               /*62*/
   'S:MISSING OR ERRONEOUS INPUT/OUTPUT LIST',                        /*63*/
   'S:MISSING RIGHT PARENTHESIS',                                     /*64*/
   'S:EXTRANEOUS OR UNINTERPRETABLE INFORMATION',                     /*65*/
   'S:MISSING OR ILLEGAL TREE OR VARIABLE NAME',                      /*66*/
   'S:NUMBER OF NODES INCORRECTLY SPECIFIED',                         /*67*/
   'S:NUMBER OF LABELS INCORRECTLY SPECIFIED',                        /*68*/
   'N:POSSIBLE ATTEMPT TO INSERT BEFORE ROOT NODE',                   /*69*/
   'S:"ALL" NOT ALLOWED AS LABEL QUALIFIER',                          /*70*/
   'S:MISSING OR ERRONEOUS BOOLEAN RELATION',                         /*71*/
   'S:MISSING INPUT OR OUTPUT FILE NAME',                             /*72*/
   'S:NEGATED BOOLEAN EXPRESSION MUST BE IN PARENTHESES',             /*73*/
   'S:"$COMBINATION" OR "$PERMUTATION" REQUIRES SUBSCRIPT',           /*74*/
   'S:ARITHMETIC EXPRESSION IS REQUIRED HERE',                        /*75*/
   'S:REMOVAL OF $COMBINATION OR $PERMUTATION SUBNODE ILLEGAL',/*76*/
   'S:COMBINATION AND PERMUTATION LOOPS MAY NOT BE NESTED',           /*77*/
   'W:CALL ARGUMENT MAY RESULT IN TYPE ERROR',                        /*78*/
   'S:PROCEDURE HAS CONFLICTING MAIN AND EXTERNAL OPTIONS',           /*79*/
   'S:NODES OPTION NOT ALLOWED IN EXTERNAL PROCEDURE',                /*80*/
   'S:PARAMETER LIST NOT ALLOWED IN MAIN PROCEDURE',                  /*81*/
   'S:RECURSIVE FEATURE NOT ALLOWED IN MAIN PROCEDURE',               /*82*/
   'W:MISSING ATTRIBUTE "LOCAL" ASSUMED PRESENT',                     /*83*/
   'S:MISSING PROPERTY LIST',                                         /*84*/
   'S:INDIRECT REFERENCE IS A LABEL,NOT SUBSCRIPT,QUALIFIER',         /*85*/
   'S:MISSING LEFT PAREN ON LABEL FUNCTION ARGUMENT',                 /*86*/
   'S:"GRAFT $TREE(ALL:  )" NOT PRESENTLY IMPLEMENTED',               /*87*/
   'S:"(GRAFT) INSERT $TREE(ALL:  )" NOT PRESENTLY IMPLEMENTED',
   'S:"ALL" CAN BE USED ONLY FOR CONDITIONAL ACCESS ("ALL:")', /*89*/
   'S:"FIRST:" IS SUBSCRIPT, NOT LABEL, QUALIFIER',                   /*90*/
   'S:"ALL:" IS SUBSCRIPT, NOT LABEL, QUALIFIER',                     /*91*/
   'S:MISSING OR ERRONEOUS EXPRESSION');                              /*92*/
```


Fig. I.3-I Error Messages for PLANS Syntax Checker

```
.OPTION_LIST (PUNCH_CODE = YES, PRINT_CODE = NO)
.AUG_GRAM  PLNSYN  (.INITIAL_CODE = SYNDCL, .FINAL_CODE = SYNWRP)

                /******************************/
                /* BASIC PROGRAM STRUCTURE */
                /******************************/

PLNSYN  :=
     /* .SET (ALLSUCH_FLAG = 0)                          */
     /* .SET (ARITH_OPERATION_FLAG = 0)                  */
     /* .SET (COMB_OR_PERE_LOOP_FLAG = 0)                */
     /* .SET (END_OF_FILE_FLAG = 0)                      */
     /* .SET (LABEL_FLAG = 0)                            */
     /* .SET (MAIN_EXTERNAL_NEITHER_SWITCH = 3)          */
     /* .SET (NODES_PARAM_NEITHER_SWITCH = 3)            */
     /* .SET (OUTPUT_NOTES_FLAG = 1)                     */
     /* .SET (PRUNING_FLAG = 0)                          */
     /* .SET (TAKE_STATISTICS_FLAG = 1)                  */
     /* .SET (TRACE_FLAG = 0)                            */
     /* .SET (UNFOUND_FOUND_SWITCH = 1)                  */
     ( .LABEL
       .DO ("@MAIN_PROCEDURE_NAME = ")
       .DO ("SUBSTR (@SYMBOL,1,LENGTH (@SYMBOL) -1) ;")
     | .MESSAGE (1)
       .EMPTY )
     "PROCEDURE"  .ERR (2)
     ( "("
       TREE_VARIABLE_LIST  .ERR (6,.SCANTO (MATCHING_PAREN|";"))
       ")"  .ERR (64)
       .SET (NODES_PARAM_NEITHER_SWITCH = 2)
     | .EMPTY )
     ( "OPTIONS"
       "("  .ERR (4)
       OPTION_ELEMENT
       $( "," OPTION_ELEMENT )
       ")"  .ERR (64)
     | .EMPTY )
     ( .TEST (MAIN_EXTERNAL_NEITHER_SWITCH = 2)
       ( .TEST (NODES_PARAM_NEITHER_SWITCH = 1)
         .MESSAGE (80)
       | .EMPTY )
     | .EMPTY
       ( .TEST (NODES_PARAM_NEITHER_SWITCH = 2)
         .MESSAGE (81)
       | .EMPTY ) )
     ( "RECURSIVE"
       .TEST (MAIN_EXTERNAL_NEITHER_SWITCH =2)
       .ERR (82)
     | .EMPTY )
     SEMI_COLON
     $( "DECLARE"  DECLARE_STATEMENT  SEMI_COLON )
     $ NON_END_UNIT
     ( "END"
       ( .ID  .MESSAGE (25) | .EMPTY )
       SEMI_COLON
     | .MESSAGE (7)
       .EMPTY )
     .PEEK ("¢¢¢¢")  .ERR (8)
     $( .PEEK ("¢¢¢¢")  .NEG
      | $ NON_END_UNIT
        ( "END"
          ( .ID .MESSAGE (25) | .EMPTY )
          SEMI_COLON
        | .EMPTY ) )  ;

OPTION_ELEMENT  :=
        "MAIN"
        ( .TEST (MAIN_EXTERNAL_NEITHER_SWITCH = 2)
          .MESSAGE (79,.SCANTO (MATCHING_PAREN|";"))
        | .EMPTY )
        .SET (MAIN_EXTERNAL_NEITHER_SWITCH = 1)
      | "NOTES"
      | "NONOTES"
        .SET (OUTPUT_NOTES_FLAG = 0)
      | "NOTRACE"
      | "TRACE"
        .SET (TRACE_FLAG = 1)
      | "STAT"
      | "NOSTAT"
        .SET (TAKE_STATISTICS_FLAG = 0)
      | "NODES"
        .SET (NODES_PARAM_NEITHER_SWITCH = 1)
        "("  .ERR (4)
        .NUM  .ERR (67)
        ","  .ERR (60)
        .NUM  .ERR (68)
        ")"  .ERR (64)
      | "EXTERNAL"
        ( .TEST (MAIN_EXTERNAL_NEITHER_SWITCH = 1)
          .MESSAGE (79,.SCANTO (MATCHING_PAREN|";"))
        | .EMPTY )
        .SET (MAIN_EXTERNAL_NEITHER_SWITCH = 2)
      | .MESSAGE (5,.SCANTO (MATCHING_PAREN|","|";"))  .EMPTY  ;

NON_END_UNIT  :=
       $ PROGRAM_UNIT
       ( .PEEK ("END")
         .NEG
         .RETURN
       | .PEEK ("¢¢¢¢")
         ( .TEST (END_OF_FILE_FLAG = 0)
           .MESSAGE (26)
           .EMPTY
           .SET (END_OF_FILE_FLAG = 1)
         | .EMPTY )
         .NEG
         .RETURN
       | .MESSAGE (9,.SCANBY (";"))
         .DO ("@STMT = @STMT + 1;")
         .EMPTY )  ;

PROGRAM_UNIT  :=
       .SET (LABEL_FLAG = 0)
       $( .LABEL  .SET (LABEL_FLAG = 1) )
       ( "PROCEDURE"
         .TEST (LABEL_FLAG = 1)  .ERR (3)
         PROCEDURE_BLOCK  .DO ("@COUNT (1) = @COUNT (1) + 1;")
```


Fig. 1.3-2 Augmented Grammar for PLANS Syntax Checker

```
        | STATEMENT
        | .EMPTY  .NEG  .RETURN ) ;

PROCEDURE_BLOCK  :=
        ( "("
          TREE_VARIABLE_LIST  .ERR(6,.SCANTO(MATCHING_PAREN|";"))
          ")"  .ERR(64)
        | .EMPTY )
        ( "RECURSIVE" | .EMPTY )
        .DO("@LEVEL = @LEVEL + 1;")
        SEMI_COLON
        $( "DECLARE"  DECLARE_STATEMENT  SEMI_COLON )
        $ NON_END_UNIT
        ( "END"  ( .ID .MESSAGE(25) | .EMPTY )
          .DO("@LEVEL = @LEVEL - 1;")
          SEMI_COLON
        | .PEEK("¢¢¢¢")
          .MESSAGE(10)  ) ;

 DECLARE_STATEMENT  :=
        .EMPTY  .DO("@COUNT(11) = @COUNT(11) + 1;")
        IGNORE_EXTRA_COMMAS
        ( .PEEK("LOCAL")  .MESSAGE(27)
        | .TREE
        | .ID )
        .ERR(27)
        $( ","  IGNORE_EXTRA_COMMAS
           ( .PEEK("LOCAL"|";")  .MESSAGE(59)
           | .TREE
           | .ID)
           .ERR(66,.SCANTO(","|";")) )
        "LOCAL"  .ERR(83) ;

                        /*******************/
                        /* STATEMENT TYPES */
                        /*******************/

 STATEMENT  :=
          "IF"  CONDITIONAL_STATEMENT
        | UNCONDITIONAL_STATEMENT
          SEMI_COLON ;

 UNCONDITIONAL_STATEMENT  :=
          .PEEK("END"|"¢¢¢¢")  .NEG  .RETURN
        | .PEEK(";")
        | "STOP"  .DO("@COUNT(14) = @COUNT(14) + 1;")
        | "RETURN"  .DO("@COUNT(15) = @COUNT(15) + 1;")
        | "TRACE"  .DO("@COUNT(26) = @COUNT(26) + 1;")
          .TEST(TRACE_FLAG = 1)  .ERR(14)
          ( "OFF"
          | "LOW"
          | "HIGH"
          | .MESSAGE(15,.SCANTO(";"))  .EMPTY )
        | "ELSE"
          .MESSAGE(35)
          ( "IF"  CONDITIONAL_STATEMENT
          | UNCONDITIONAL_STATEMENT
          | ( .PEEK("¢¢¢¢") | "END" )
            .MESSAGE(31)
          | .MESSAGE(9,.SCANTO(";"))
            .DO("@STMT = @STMT + 1;")
            .EMPTY )
        | "DECLARE"
          .MESSAGE(36)
          DECLARE_STATEMENT
        | "BEGIN"  BEGIN_BLOCK
        | "DO"  DO_GROUP
        | "GO"  "TO"  .ERR(42)
          .ID  .ERR(43,.SCANTO(";"))
          .DO("@COUNT(12) = @COUNT(12) + 1;")
        | "CALL"  CALL_STATEMENT
        | TREE_ASSIGNMENT_STATEMENT  .DO("@COUNT(16) = @COUNT(16) + 1;")
        | "LABEL"
          LABEL_ASSIGNMENT_STATEMENT  .DO("@COUNT(17) = @COUNT(17) + 1;")
        | "PRUNE"  PRUNE_STATEMENT  .DO("@COUNT(18) = @COUNT(18) + 1;")
        | "INSERT"  INSERT_STATEMENT  .DO("@COUNT(19) = @COUNT(19) + 1;")
        | "GRAFT"  .SET(PRUNING_FLAG = 1)
          ( "INSERT"  INSERT_STATEMENT  .DO("@COUNT(21) = @COUNT(21) + 1;")
          | GRAFT_STATEMENT )
        | "ADVANCE"  ADVANCE_STATEMENT
          .DO("@COUNT(56) = @COUNT(56) + 1;")
        | "ORDER"  ORDER_STATEMENT  .DO("@COUNT(22) = @COUNT(22) + 1;")
        | "READ"  READ_STATEMENT  .DO("@COUNT(23) = @COUNT(23) + 1;")
        | "WRITE"  WRITE_STATEMENT  .DO("@COUNT(24) = @COUNT(24) + 1;")
        | "DEFINE"  DEFINE_STATEMENT  .DO("@COUNT(27) = @COUNT(27) + 1;")
        | ARITHMETIC_ASSIGNMENT_STATEMENT
          .DO("@COUNT(25) = @COUNT(25) + 1;")  ;

                        /*****************/
                        /* BEGIN BLOCK */
                        /*****************/

BEGIN_BLOCK  :=
        .EMPTY  .DO("@COUNT(2) = @COUNT(2) + 1;")
        .DO("@LEVEL = @LEVEL + 1;")
        SEMI_COLON
        ( "DECLARE"  DECLARE_STATEMENT  SEMI_COLON
        | .EMPTY  .MESSAGE(11) )
        $( "DECLARE"  DECLARE_STATEMENT  SEMI_COLON )
        $ NON_END_UNIT
        ( "END"
        | .PEEK("¢¢¢¢")
          .MESSAGE(12) )
        ( .ID
          .MESSAGE(25)
        | .EMPTY )
        .DO("@LEVEL = @LEVEL - 1;")  ;

                        /**************/
                        /* DO GROUP */
                        /**************/

DO_GROUP  :=
        .EMPTY
        .DO("@NEST = @NEST + 1;")
```

Fig. 1.3-2 (cont)

```
    ( "WHILE"  WHILE_CLAUSE  .DO("@COUNT(5) = @COUNT(5) + 1;")
    | "FOR"
      ( "ALL"
          ( "SUBNODES"  SUBNODE_CLAUSE
            .DO("@COUNT(6) = @COUNT(6) + 1;")
          | COMBINATION_CLAUSE
            .RETURN
          | .EMPTY  .MESSAGE(19,.SCANTO(";")) )
      | .MESSAGE(19,.SCANTO(";"))  .EMPTY )
    | INCREMENT_CLAUSE  .DO("@COUNT(4) = @COUNT(4) + 1;")
    | .EMPTY  .DO("@COUNT(3) = @COUNT(3) + 1;") )
    DO_BODY_THROUGH_END  ;

WHILE_CLAUSE  :=
    .EMPTY
    "("  .ERR(4)
    BOOLEAN_EXPRESSION  .ERR(13,.SCANTO(")"|";"))
    ")"  .ERR(64,.SCANTO(";") )  ;

SUBNODE_CLAUSE  :=
    .EMPTY
    "OF"  .ERR(20)
    SOFT_TREE_NODE  .ERR(21,.SCANTO("USING"|";"))
    "USING"  .ERR(22)
    .TREE  .ERR(23,.SCANTO(";") )  ;

COMBINATION_CLAUSE  :=
    ( "COMBINATIONS"  .DO("@COUNT(7) = @COUNT(7) + 1;")
    | "PERMUTATIONS"  .DO("@COUNT(8) = @COUNT(8) + 1;") )
    .TEST(COMB_OR_PERM_LOOP_FLAG = 0)  .ERR(77)
    .SET(COMB_OR_PERM_LOOP_FLAG = 1)
    "OF"  .ERR(20)
    SOFT_TREE_NODE  .ERR(21,.SCANTO("TAKEN"|";"))
    "TAKEN"  .ERR(36,.SCANTO(";") )
    ARITH_EXPRESSION  .ERR(39,.SCANTO("AT"|";"))
    "AT"  .ERR(18,.SCANTO("A"|"TIME"|";") )
    "A"  .ERR(40,.SCANTO("TIME"|";") )
    "TIME"  .ERR(41)
    DO_BODY_THROUGH_END
    .SET(COMB_OR_PERM_LOOP_FLAG = 0)  ;

INCREMENT_CLAUSE  :=
    .ID
    ( "="
      .MESSAGE(19,.SCANTO(";"))
      .EMPTY
      .RETURN )
    ARITH_EXPRESSION  .ERR(39,.SCANTO(","|"TO"|"BY"|";"))
    $( ","  ARITH_EXPRESSION  .ERR(39,.SCANTO(","|"TO"|"BY"|";")) )
    ( "TO"
      ARITH_EXPRESSION  .ERR(39,.SCANTO("BY"|";"))
      ( "BY"
        ARITH_EXPRESSION  .ERR(39,.SCANTO(";"))
      | .EMPTY )
      $( ","  ARITH_EXPRESSION  .ERR(39,.SCANTO(","|";")) )
    | .EMPTY)
    ( "WHILE"  WHILE_CLAUSE
    | .EMPTY )  ;

DO_BODY_THROUGH_END  :=
    .EMPTY
    .PEEK(";")  .ERR(19,.SCANTO(";"))
    $ NON_END_UNIT
    .DO("@NEST = @NEST - 1;")
    ( "END"
    | .PEEK("¢¢¢¢")  .MESSAGE(37)  .EMPTY )
    ( .ID  .MESSAGE(25)
    | .EMPTY )  ;

               /**************************/
               /* CONDITIONAL STATEMENT */
               /**************************/

CONDITIONAL_STATEMENT  :=
    .EMPTY
    BOOLEAN_EXPRESSION  .ERR(13,.SCANTO("THEN"|";"))
    ( .PEEK("THEN")
    | ";"  .MESSAGE(29)
    | .MESSAGE(28,.SCANTO("THEN"|";"))
      .EMPTY )
    .DO("@STMT = @STMT + 1;")
    ( "THEN"
      ( STATEMENT
      | "END"
        .MESSAGE(30)
        SEMI_COLON
      | .PEEK("¢¢¢¢")
        .MESSAGE(30)
      | .MESSAGE(9,.SCANBY(";"))
        .DO("@STMT = @STMT + 1;")
        .EMPTY )
    | .MESSAGE(32,.SCANBY(";"))
      .DO("@STMT = @STMT + 1;")
      .EMPTY )
    ( "ELSE"  .DO("@COUNT(10) = @COUNT(10) + 1;")
      ( STATEMENT
      | "END"
        .MESSAGE(31)
        SEMI_COLON
      | .PEEK("¢¢¢¢")
        .MESSAGE(31)
      | .MESSAGE(9,.SCANBY(";"))
        .DO("@STMT = @STMT + 1;")
        .EMPTY )
    | .EMPTY  .DO("@COUNT(9) = @COUNT(9) + 1;") )  ;

                  /*******************/
                  /* CALL STATEMENT */
                  /*******************/

CALL_STATEMENT  :=
    .EMPTY  .DO("@COUNT(13) = @COUNT(13) + 1;")
    .ID  .ERR(44,.SCANTO("("|";"))
    ( "("  IGNORE_EXTRA_COMMAS
      CALL_ARGUMENT  .ERR(45,.SCANTO(","|MATCHING_PAREN|";"))
```

Fig. I.3-2 (cont)

```
         $( "," IGNORE_EXTRA_COMMAS
              ( .PEEK(")"|";")
                .MESSAGE(59)
              | CALL_ARGUMENT
              | .EMPTY
                .MESSAGE(45,.SCANTO(","|MATCHING_PAREN|";"))) )
           ")" .ERR(64)
         | .EMPTY )  ;

CALL_ARGUMENT  :=
       HARD_TREE_NODE
     | .PEEK("LABEL")
       ARITH_EXPRESSION
       ( .TEST(ARITH_OPERATION_FLAG = 1)
         .MESSAGE(78)
       | .EMPTY
         .MESSAGE(45) )
     | ARITH_EXPRESSION
       ( .TEST(ARITH_OPERATION_FLAG = 1)
         .MESSAGE(78)
       | .EMPTY ) ;

                /******************************/
                /* TREE ASSIGNMENT STATEMENT */
                /******************************/

TREE_ASSIGNMENT_STATEMENT  :=
       HARD_TREE_NODE
       ( "="
       | .MESSAGE(9,.SCANTO(";"))   .EMPTY   .RETURN )
       SIMPLE_EXPRESSION  .ERR(21,.SCANTO(";"))  .EMPTY  ;

               /*******************************/
               /* LABEL ASSIGNMENT STATEMENT */
               /*******************************/

LABEL_ASSIGNMENT_STATEMENT  :=
       .EMPTY
       "("  .ERR(86)
       HARD_TREE_NODE  .ERR(56,.SCANTO(MATCHING_PAREN|"="|";"))
       ")"  .ERR(64,.SCANTO("="|";"))
       ( "="
       | .MESSAGE(9,.SCANTO(";"))   .EMPTY   .RETURN )
       SIMPLE_EXPRESSION  .ERR(92,.SCANTO(";"))  .EMPTY  ;

                   /*******************/
                   /* PRUNE STATEMENT */
                   /*******************/

PRUNE_STATEMENT  :=
       .EMPTY
       .SET(PRUNING_FLAG = 1)
       IGNORE_EXTRA_COMMAS
       SOFT_TREE_NODE  .ERR(21,.SCANTO(";"))
       $( "," IGNORE_EXTRA_COMMAS
          ( .PEEK(";")
            .MESSAGE(59)
          | SOFT_TREE_NODE
          | .EMPTY
            .MESSAGE(21,.SCANTO(";"|","))) )
       .SET(PRUNING_FLAG = 0)  ;

                   /*******************/
                   /* GRAFT_STATEMENT */
                   /*******************/

GRAFT_STATEMENT  :=
       .DO("@COUNT(20) = @COUNT(20) + 1;")
       .EMPTY
       SIMPLE_EXPRESSION  .ERR(21,.SCANTO("AT"|";"))
       .TEST(ALLSUCC_FLAG = 0)  .ERR(87)
       .SET(PRUNING_FLAG = 0)
       "AT" .ERR(18)
       HARD_TREE_NODE  .ERR(21,.SCANTO(";"))  .EMPTY  ;

                   /*********************/
                   /* ADVANCE STATEMENT */
                   /*********************/

ADVANCE_STATEMENT  :=
       .EMPTY
       .TREE  .ERR(23,.SCANTO(";"))  ;

                   /*******************/
                   /* ORDER STATEMENT */
                   /*******************/

ORDER_STATEMENT  :=
       .EMPTY
       SOFT_TREE_NODE  .ERR(21,.SCANTO("BY"|";"))
       ( "BY"
         IGNORE_EXTRA_COMMAS
         ( .PEEK(";")  .MESSAGE(84)
         | ORDER_ARGUMENT )
         $( ","
            ( .PEEK(","|";")  .MESSAGE(59)
            | ORDER_ARGUMENT )
          | .PEEK(.ID|"-")  .MESSAGE(60)
            ORDER_ARGUMENT )
       | .EMPTY )  ;

ORDER_ARGUMENT  :=
       ( "-" | .EMPTY )
       ( "$ELEMENT" | .ID )  .ERR(61,.SCANTO(","|";"))
       $( "." SOFT_QUALIFIER_BY_LABEL
        | "(" SOFT_QUALIFIER_BY_SUBSCRIPT ")" .ERR(64) )  ;

                   /********************/
                   /* INSERT STATEMENT */
                   /********************/
```

Fig. 1.3-2 (cont)

```
INSERT_STATEMENT  :=
     .EMPTY
     SIMPLE_EXPRESSION   .ERR(21,.SCANTO("BEFORE"|";"))
     .TEST(ALLSUCH_FLAG = 0)   .ERR(88)
     .SET(PRUNING_FLAG = 0)
     "BEFORE"   .ERR(17)
     HARD_TREE_NODE   .ERR(21,.SCANTO(";"))
     .TEST(UNQUALIFIED_TREE_FLAG = 0)   .ERR(69)   .EMPTY   ;

                    /******************/
                    /* READ STATEMENT */
                    /******************/

READ_STATEMENT  :=
     IGNORE_EXTRA_COMMAS
     FILE_COMPRESSED_OPTIONS
     ( .PEEK(";")   .MESSAGE(63)   .EMPTY
     | READ_ELEMENT )
     $( ","
        ( .PEEK(";"|",")   .MESSAGE(59)
        | READ_ELEMENT )
      | .PEEK(.ID|.TREE)   .MESSAGE(60)
        READ_ELEMENT )   ;

FILE_COMPRESSED_OPTIONS  :=
     ( "FILE"
       "("   .ERR(4)
       .ID   .ERR(72,.SCANTO(")"|";"))
       ")"   .ERR(64)
     | .EMPTY )
     ( "COMPRESSED"
     | .EMPTY )   ;

READ_ELEMENT  :=
     .ID
   | HARD_TREE_NODE
   | .MESSAGE(63,.SCANTO(","|";"))   .EMPTY   ;

                    /*******************/
                    /* WRITE STATEMENT */
                    /*******************/

WRITE_STATEMENT  :=
     IGNORE_EXTRA_COMMAS
     FILE_COMPRESSED_OPTIONS
     ( .PEEK(";")   .MESSAGE(63)   .EMPTY
     | WRITE_ELEMENT )
     $( ","
        ( .PEEK(";"|",")   .MESSAGE(59)
        | WRITE_ELEMENT )
      | .PEEK(.ID|.TREE)   .MESSAGE(60)
        WRITE_ELEMENT )   ;

WRITE_ELEMENT  :=
     "LABEL"  LABEL_STRING
   | .STRING
   | .ID
```



```
   | SOFT_TREE_NODE
   | .MESSAGE(63,.SCANTO(","|";"))   .EMPTY   ;

                    /********************/
                    /* DEFINE STATEMENT */
                    /********************/

DEFINE_STATEMENT  :=
     .EMPTY
     .TREE   .ERR(23,.SCANTO("AS"|";"))
     "AS"   .ERR(33)
     HARD_TREE_NODE   .ERR(21,.SCANTO(";"))   ;

                    /***********************************/
                    /* ARITHMETIC ASSIGNMENT STATEMENT */
                    /***********************************/

ARITHMETIC_ASSIGNMENT_STATEMENT  :=
     .ID
     ( "="
     | .MESSAGE(9,.SCANTO(";"))
       .EMPTY
       .RETURN )
     ARITH_EXPRESSION   .ERR(39,.SCANTO(";"))  ;

                    /**********************/
                    /* GENERAL EXPRESSION */
                    /**********************/

SIMPLE_EXPRESSION  :=
     .SET(ALLSUCH_FLAG = 0)
     ( SOFT_TREE_NODE
       .SET(TREE_STRING_ARITH_SWITCH = 1)
     | CHAR_STRING
       .SET(TREE_STRING_ARITH_SWITCH = 2) )
     ( .PEEK("+"|"-"|"*"|"/"|"**")
       .SET(UNFOUND_FOUND_SWITCH = 2)
       ARITH_EXPRESSION
       .SET(TREE_STRING_ARITH_SWITCH = 3)
     | .EMPTY )
   | ARITH_EXPRESSION
     .SET(TREE_STRING_ARITH_SWITCH = 3)   ;

                    /**********************/
                    /* BOOLEAN EXPRESSION */
                    /**********************/

BOOLEAN_EXPRESSION  :=
     BOOLEAN_PRIMARY
     $( ( "|" | "&" )
        BOOLEAN_PRIMARY   .ERR(13,.SCANTO("THEN"|"&"|"|"|";")) )

BOOLEAN_PRIMARY  :=
     "("
     BOOLEAN_EXPRESSION
```

Fig. 1.3-2 (cont)

```
          .ERR(13,.SCANTO(MATCHING_PAREN|"THEN"|"&"|"|"|";"))
         ")"  .ERR(64)
       | "¬"
         "("  .ERR(73)
         ( BOOLEAN_EXPRESSION
         | .EMPTY  .NEG  .RETURN )
         ")"  .ERR(64)
       | SIMPLE_EXPRESSION
         .DO("TSA_LEFT_SWITCH = TREE_STRING_ARITH_SWITCH;")
         ( TREE_RELATION
           .SET(TSA_RELATION_SWITCH = 1)
         | ARITH_RELATION
           .SET(TSA_RELATION_SWITCH = 3)
         | .EMPTY
           .MESSAGE(71) )
         SIMPLE_EXPRESSION  .ERR(13,.SCANTO("THEN" | "&" | "|" | ";"))
         .DO("TSA_RIGHT_SWITCH = TREE_STRING_ARITH_SWITCH;")
         DETECT_TYPE_CONVERSIONS  ;

ARITH_RELATION  :=
         "<" | "<=" | "=" | ">=" | ">" | "¬<" | "¬<=" | "¬=" | "¬>=" |
         "¬>" ;

TREE_RELATION  :=
         ( "¬" | "NOT" | .EMPTY )
         ( "IDENTICAL"  ( "TO" | .EMPTY )
           .DO("@COUNT(51) = @COUNT(51) + 1;")
         | "SUBSET"  ( "OF" | .EMPTY )
           .DO("@COUNT(52) = @COUNT(52) + 1;")
         | "ELEMENT"  ( "OF" | .EMPTY )
           .DO("@COUNT(53) = @COUNT(53) + 1;")
         | .EMPTY  .NEG  .RETURN ) ;

DETECT_TYPE_CONVERSIONS  :=
         .TEST(TSA_RELATION_SWITCH = 1)
         ( .TEST(TSA_LEFT_SWITCH = 2)
           .DO("@COUNT(47) = @COUNT(47) + 1;")
         | .TEST(TSA_LEFT_SWITCH = 3)
           .DO("@COUNT(49) = @COUNT(49) + 1;")
         | .EMPTY )
         ( .TEST(TSA_RIGHT_SWITCH = 2)
           .DO("@COUNT(47) = @COUNT(47) + 1;")
         | .TEST(TSA_RIGHT_SWITCH = 3)
           .DO("@COUNT(49) = @COUNT(49) + 1;")
         | .EMPTY )
       | .EMPTY  ;       /* TO BE EXTENDED */

                    /*********************/
                    /* TREE EXPRESSION */
                    /*********************/

HARD_TREE_NODE  :=
         ("$NULL"  .MESSAGE(16) | COMBINATION_OR_PERMUTATION_TREE | .TREE)
         .DO("@COUNT(28) = @COUNT(28) + 1;")
         .SET(UNQUALIFIED_TREE_FLAG = 1)
         $( "."  HARD_QUALIFIER_BY_LABEL  .SET(UNQUALIFIED_TREE_FLAG = 0)
          | "("
            HARD_QUALIFIER_BY_SUBSCRIPT
            ")"  .ERR(64,.SCANTO("="|"."|"("|";"))
            .SET(UNQUALIFIED_TREE_FLAG = 0) )  ;

SOFT_TREE_NODE  :=
         ( COMBINATION_OR_PERMUTATION_TREE | .TREE )
         .DO("@COUNT(29) = @COUNT(29) + 1;")
         .SET(UNQUALIFIED_TREE_FLAG = 1)
         $( "."  SOFT_QUALIFIER_BY_LABEL  .SET(UNQUALIFIED_TREE_FLAG = 0)
          | "("
            SOFT_QUALIFIER_BY_SUBSCRIPT
            ")"
            .ERR(64,.SCANTO("¬"|"."|"("|";"|"+"|"-"|"*"|"/"|"**"|"&"|"|"|
                  "<"|">"|"="|"¬="|"<="|">="|"¬<="|"¬>="|"NOT"))
            .SET(UNQUALIFIED_TREE_FLAG = 0) )  ;

COMBINATION_OR_PERMUTATION_TREE  :=
         ( "$COMBINATION" | "PERMUTATION" )
         ( "("
           ( ( "NEXT" | "LAST" | "FIRST" | "ALL" | "FIRST:" | "ALL:" )
             .MESSAGE(75,.SCANTO(MATCHING_PAREN | ";"))
           | .EMPTY
             ARITH_EXPRESSION  .ERR(39,.SCANTO(MATCHING_PAREN | ";")) )
           ")"  .ERR(64)
         | .MESSAGE(74)  .EMPTY )
         ( .TEST(PRUNING_FLAG = 1)
           .PEEK("." | "(")  .ERR(76)
         | .EMPTY )  ;

HARD_QUALIFIER_BY_LABEL  :=
         NODE_LABEL  .DO("@COUNT(30) = @COUNT(30) + 1;")
       | "#"  INDIRECT_CHAR_STRING  .DO("@COUNT(31) = @COUNT(31) + 1;")
       | .MESSAGE(49)  .EMPTY  ;

SOFT_QUALIFIER_BY_LABEL  :=
         NODE_LABEL  .DO("@COUNT(37) = @COUNT(37) + 1;")
       | "#"
         ( .TEST(PRUNING_FLAG = 1)
           .SET(PRUNING_FLAG = 0)
           INDIRECT_CHAR_STRING
           .SET(PRUNING_FLAG = 1)
         | INDIRECT_CHAR_STRING )
         .DO("@COUNT(38) = @COUNT(38) + 1;")
       | .EMPTY
         .MESSAGE(49,.SCANTO("."|"("|";"))  ;

NODE_LABEL  :=
         "NEXT"  .MESSAGE(50)
       | "LAST"  .MESSAGE(52)
       | "FIRST" .MESSAGE(58)
       | "FIRST:" .MESSAGE(90)
       | "ALL" .MESSAGE(70)
       | "ALL:" .MESSAGE(91)
       | "LABEL"  .MESSAGE(54)  LABEL_STRING
       | .ID  ;

HARD_QUALIFIER_BY_SUBSCRIPT  :=
         "NEXT"  .DO("@COUNT(35) = @COUNT(35) + 1;")
       | "LAST"  .DO("@COUNT(34) = @COUNT(34) + 1;")
       | "FIRST"  .DO("@COUNT(32) = @COUNT(32) + 1;")
       | "ALL"  .MESSAGE(89)  .EMPTY
```

Fig. 1.3-2 (cont)

```
    | "FIRST:"  .DO("@COUNT(33) = @COUNT(33) + 1;")
      BOOLEAN_EXPRESSION  .ERR(13,.SCANTO(MATCHING_PAREN|";"|"."|","))
    | "ALL:"  .MESSAGE(34)
      BOOLEAN_EXPRESSION  .ERR(13,.SCANTO(MATCHING_PAREN|";"|"."|","))
    | "$"  .MESSAGE(85)
      INDIRECT_CHAR_STRING
    | ARITH_EXPRESSION  .DO("@COUNT(36) = @COUNT(36) + 1;")
    | .MESSAGE(39,.SCANTO(MATCHING_PAREN|";"|"."|","))  .EMPTY  ;

SOFT_QUALIFIER_BY_SUBSCRIPT  :=
      "NEXT"  .MESSAGE(51)  .EMPTY
    | "LAST"  .DO("@COUNT(43) = @COUNT(43) + 1;")
    | "FIRST"  .DO("@COUNT(39) = @COUNT(39) + 1;")
    | "ALL"  .MESSAGE(89)
    | "FIRST:"  .DO("@COUNT(40) = @COUNT(40) + 1;")
      ( .TEST(PRUNING_FLAG = 1)
        .SET(PRUNING_FLAG = 0)
        BOOLEAN_EXPRESSION  .ERR(13,.SCANTO(MATCHING_PAREN|";"|"."|","))
        .SET(PRUNING_FLAG = 1)
      | BOOLEAN_EXPRESSION )
      .ERR(13,.SCANTO(MATCHING_PAREN|"."|","|";"))
    | "ALL:"  .DO("@COUNT(42) = @COUNT(42) + 1;")
      ( .TEST(PRUNING_FLAG = 1)
        .SET(PRUNING_FLAG = 0)
        BOOLEAN_EXPRESSION  .ERR(13,.SCANTO(MATCHING_PAREN|";"|"."|","))
        .SET(PRUNING_FLAG = 1)
      | BOOLEAN_EXPRESSION )
      .ERR(13,.SCANTO(MATCHING_PAREN|";"|"."|","))
    | "$"  .MESSAGE(85)
      ( .TEST(PRUNING_FLAG = 1)
        .SET(PRUNING_FLAG = 0)
        INDIRECT_CHAR_STRING
        .SET(PRUNING_FLAG = 1)
      | INDIRECT_CHAR_STRING )
    | ( .TEST(PRUNING_FLAG = 1)
        .SET(PRUNING_FLAG = 0)
        ARITH_EXPRESSION  .ERR(39,.SCANTO(MATCHING_PAREN|";"|"."|","))
        .DO("@COUNT(44) = @COUNT(44) + 1;")
        .SET(PRUNING_FLAG = 1)
      | ARITH_EXPRESSION  .DO("@COUNT(44) = @COUNT(44) + 1;") )
    | .MESSAGE(39,.SCANTO(MATCHING_PAREN|";"|"."|","))  .EMPTY  ;


                    /***********************/
                    /* STRING EXPRESSION */
                    /***********************/

CHAR_STRING  :=
       .STRING
     | SOFT_TREE_NODE
     | "LABEL"  LABEL_STRING  ;

LABEL_STRING  :=
     .EMPTY
     "("  .ERR(86)
     SOFT_TREE_NODE  .ERR(56,.SCANTO(MATCHING_PAREN|";"))
     ")"  .ERR(64)  ;

INDIRECT_CHAR_STRING  :=
```

```
          .EMPTY
          ( "LABEL"  LABEL_STRING
          | "("
            ( SOFT_TREE_NODE
            | "LABEL"
              .MESSAGE(62)
              LABEL_STRING
            | .EMPTY
              .MESSAGE(62,.SCANTO(MATCHING_PAREN|";")) )
            ")"  .ERR(64)
          | .EMPTY  .MESSAGE(62,.SCANTO(";")) )  ;


                    /**************************/
                    /* ARITHMETIC EXPRESSION */
                    /**************************/

ARITH_EXPRESSION  :=
          .SET(ARITH_OPERATION_FLAG = 0)
          ARITH_TERM
          $( ( "+" | "-" ) .SET(ARITH_OPERATION_FLAG = 1)
             $( ("*"|"/"|"**")  .MESSAGE(46) )
             ( .PEEK(";"|")"|",")  .MESSAGE(46)
             | ARITH_TERM
             | .MESSAGE(47,.SCANTO("+"|"-"|";"))  .EMPTY ) )  ;

ARITH_TERM  :=
          ARITH_FACTOR
          $( ( "*" | "/" ) .SET(ARITH_OPERATION_FLAG = 1)
             $( ("*"|"/"|"**")  .MESSAGE(46) )
             ( .PEEK(";"|")"|",")  .MESSAGE(46)
             | ARITH_FACTOR
             | .MESSAGE(47,.SCANTO("+"|"-"|"*"|"/"|";"))  .EMPTY ) )  ;

ARITH_FACTOR  :=
          ARITH_PRIMARY
          $( "**"  .SET(ARITH_OPERATION_FLAG = 1)
             $( ("*"|"/"|"**")  .MESSAGE(46) )
             ( .PEEK(";"|")"|",")  .MESSAGE(46)
             | ARITH_FACTOR
             | .MESSAGE(47,.SCANTO("+"|"-"|"*"|"/"|";")) .EMPTY ) )  ;

ARITH_PRIMARY  :=
          "NUMBER"  .SET(ARITH_OPERATION_FLAG = 1)
          "("  .ERR(4)
          SOFT_TREE_NODE  .ERR(48,.SCANTO(MATCHING_PAREN|";"))
          ")"  .ERR(64)
        | "LABEL"  LABEL_STRING  .DO("@COUNT(48) = @COUNT(48) + 1;")
        | .ID
        | .SET (ARITH_OPERATION_FLAG = 1)
          .NUM
        | SOFT_TREE_NODE  .DO("@COUNT(46) = @COUNT(46) + 1;")
        | "("
          ARITH_EXPRESSION  .ERR(39,.SCANTO(MATCHING_PAREN|";"|","))
          ")"  .ERR(64)  .SET(ARITH_OPERATION_FLAG = 1)
        | .TEST(UNFOUND_FOUND_SWITCH = 2)
          .SET(UNFOUND_FOUND_SWITCH = 1)
        | ( "+" | "-" )
          ARITH_PRIMARY  .ERR(47,.SCANTO("+"|"-"|"*"|"/"|";"))  ;
```

Fig. I.3-2 (cont)

```
                    /******************/
                    /* MISCELLANEOUS */
                    /******************/

IGNORE_EXTRA_COMMAS  :=
      .EMPTY
      $( ","  .MESSAGE(59) )  ;

SEMI_COLON  :=
      .PEEK("¢¢¢¢")
    | .EMPTY
      .PEEK(";")   .ERR(65,.SCANTO(";"))
      .DO("@STMT = @STMT + 1;")
      ( ";" | .EMPTY )  ;

TREE_VARIABLE_LIST  :=
      IGNORE_EXTRA_COMMAS
      ( .TREE  |  .ID  |  .EMPTY  .NEG  .RETURN )
      $( ","  IGNORE_EXTRA_COMMAS
          ( .TREE
          | .ID
          | .PEEK(")"|";")   .MESSAGE(59)
          | .EMPTY   .MESSAGE(66,.SCANTO(","|";"|MATCHING_PAREN)) )  ;

.END
```

Fig. 1.3-2 (concl)

```
/* ERROR MESSAGES */
   DECLARE @ERROR_MESSAGE (10) CHAR(60) VARYING STATIC INIT(
   'S:POINTER NAME CONFLICTS WITH TREE NAME USED ELSEWHERE',   /*01*/
   'S:OUTPUT STACK LEFT NONEMPTY',                              /*02*/
   'S:PROCEDURE NAME CONFLICTS WITH VARIABLE NAME USED ELSEWHERE',
   'N:POSSIBLE USER ERROR:  POINTER USED OUTSIDE OF SUBNODE LOOP',
   'N:POSSIBLE INSERTION BEFORE ROOT NODE');                    /*05*/
```

Fig. I.4-I Error Messages for PLANS Code Generator

```
.OPTION_LIST(PUNCH_CODE=YES, PRINT_CODE=NO)
.AUG_GRAM  PLNSPM  (.INITIAL_CODE=SPMDCL, .FINAL_CODE=SPMWRP)

/* INCLUDES ONLY TREE ASSIGNMENT AND PRUNE APPLICATION OF "ALL:" */
/* STATISTICS THROUGH 40, 56 */

                    /*****************************/
                    /* BASIC PROGRAM STRUCTURE */
                    /*****************************/

PLNSPM  :=
   /* FLAG INITIALIZATION */
     /** .SET(FIRST_PRIMARY_FOUND_FLAG = 0)    **/
     /** .SET(MAIN_EXTERNAL_SWITCH = 1)        **/
     /** .SET(NORMAL_MERGE_SWITCH = 1)         **/
     /** .SET(OUTPUT_NOTES_FLAG = 1)           **/
     /** .SET(PERIOD_ALREADY_FOUND_FLAG = 0)   **/
     /** .SET(SET_SOFT_LINK_FLAG = 0)          **/
     /** .SET(TAKE_STATISTICS_FLAG = 0)        **/
     /** .SET(TRACE_ALREADY_OUTPUT_FLAG = 0)   **/
     /** .SET(TRACE_FLAG = 0)                  **/
   /* MAIN PROCEDURE */
     .LABEL
     .SAV("PUT FILE(STATIST) EDIT(''*N")  .CAT(*,"''") (A);")
     .SAV(*)
     .SEARCH_BLOCK  .IF_NEW(1,2,"PD")
     .DO("PROCEDURE_NAME = SUBSTR(@SYMBOL,1,LENGTH(@SYMBOL) - 1);")
     .DO("@PAGE_HEAD = @PAGE_HEAD||' OF PROCEDURE '||PROCEDURE_NAME;")
     "PROCEDURE"  .OUT(*)
     ( "("    $","  .OUT(*)
       PROCEDURE_ARGUMENT
       $( ","    $","  .OUT(",")  PROCEDURE_ARGUMENT )
       ")"  .OUT(*)
     | .EMPTY )
     ( "OPTIONS"
       ( "("  |  .EMPTY )
       OPTION_ELEMENT  $( ","  OPTION_ELEMENT )
       ")"
     | .EMPTY )
     .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
     ( .TEST(MAIN_EXTERNAL_SWITCH = 2)
       .OUT(#)
       .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
       ( "RECURSIVE"  .OUT(" RECURSIVE")
         .INIT_BLOCK  .FIND_NEXT(1,1,"P")  .ENTER(2,1,"R")
       | .EMPTY )
       .OUT(" REORDER;")
       .OUT("%INCLUDE PLANS(XNODES);")
       .OUT("%INCLUDE PLANS(XSTOFAG);")
       .OUT("%INCLUDE PLANS(XENTRY);")
     | .EMPTY  .OUT(#)
       .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
       .OUT(" OPTIONS(MAIN);")
       .OUT("DECLARE @NUMBER_OF_NODES FIXED BINARY(15,0) STATIC INIT(")
       .DO("CALL @OUT(NODES);")  .OUT(");")
       .OUT("DECLARE @NUMBER_OF_LABELS FIXED BINARY (15,0) INIT(")
       .DO("CALL @OUT(LABELS);")  .OUT(");")
       .OUT("DECLARE @STORAG(0: ")
       .DO("CALL @OUT(LABELS);")  .OUT(") CHARACTER(8) EXTERNAL;")
       .OUT("%INCLUDE PLANS(OBJDCL1);")
       .OUT("2 @BINARY_NODES(0: ")  .DO("CALL @OUT(NODES);")
       .OUT("),¢","3 @SON FIXED BINARY (15,0) INIT(0),¢")
       .OUT("3 @BROTHER FIXED BINARY (15,0) INIT(0),¢")
       .OUT("3 @LABEL FIXED BINARY (15,0) INIT(0);")
       .OUT("%INCLUDE PLANS(OBJDCL2);")  )
     SEMI_COLON
     $( "DECLARE"  DECLARE_STATEMENT )
     .OUT(#)
     ( .TEST(TAKE_STATISTICS_FLAG = 0)
       .OUT("PUT FILE(STATIST) EDIT(''*O'') (A);")
     | .EMPTY )
     $ PROGRAM_UNIT
     OUTPUT_DECLARATIONS
     "END"  ( .ID | .EMPTY )  SEMI_COLON
     .OUT("@PROC_EXIT: ;")
     PRUNE_LOCAL_TREES
     .TEST(NORMAL_MERGE_SWITCH = 1)  .ERR(2)
     .DO("CALL @INCLUDE_PROCEDURES;")
     .DO("CALL @INCLUDE_DECLARATIONS;")
     ( .TEST(TAKE_STATISTICS_FLAG = 1)
       .OUT("%INCLUDE PLANS(OBJWRP);")
     | .EMPTY )
     .OUT("END;")
     .PEEK("¢¢¢¢")  ;

OPTION_ELEMENT  :=
       "MAIN"
     | "EXTERNAL"
       .SET(MAIN_EXTERNAL_SWITCH = 2)
     | "NOTES"
     | "NONOTES"
       .SET(OUTPUT_NOTES_FLAG = 0)
     | "NOTRACE"
       .SET(TRACE_FLAG = 0)
     | "TRACE"
       .SET(TRACE_FLAG = 1)
     | "STAT"
       .SET(TAKE_STATISTICS_FLAG = 1)
     | "NOSTAT"
       .SET(TAKE_STATISTICS_FLAG = 0)
     | "NODES"
       ( "("  |  .EMPTY )
       .NUM  .DO("NODES = @SYMBOL;")
       $ ","
       .NUM  .DO("LABELS = @SYMBOL + MOD(@SYMBOL,4) + 3;")
       ")"  .SET(DEFAULT_NODES_FLAG = 0)  ;

OUTPUT_DECLARATIONS  :=
     .EMPTY
     .INIT_BLOCK
     .FIND_NEXT(1,1,"P")
     ( .TABLE_TEST(2,1,"R")  .SET(RECURSIVE_FLAG = 1)
     | .EMPTY  .SET(RECURSIVE_FLAG = 0) )
     .INIT_BLOCK
     $( .FIND_NEXT(1,1,"B")
        .OUT("DECLARE ",**,"@ POINTER STATIC;")
        .OUT("DECLARE ",**,"$ FIXED BINARY (15,0) ")
```

Fig. I.4-2 Augmented Grammar for PLANS Code Generator

```
        .OUT("BASED (",**,"@);") )
      .INIT_BLOCK
      $( .FIND_NEXT(1,1,"T")
        .OUT("DECLARE ",**," FIXED BINARY (15,0)")
        ( .TABLE_TEST(2,1,"P")
        | STATIC_IF_NOT_RECURSIVE  .OUT(" INIT(0)") )
        .OUT(";") )
     .INIT_BLOCK
     $( .FIND_NEXT(1,1,"$")
       .OUT("DECLARE ",**," FIXED BINARY (15,0)")
       STATIC_IF_NOT_RECURSIVE
       .OUT(";") )
     .INIT_BLOCK
     $( .FIND_NEXT(1,1,"V")
       .DO("IF INDEX('IJKLMN',SUBSTR(@SYM_NAME(@SYMBOL_LEVEL,")
       .DO("@SYMBOL_NUMBER),1,1)) = 0 ")
       .DO("THEN SUBSTR(@ASS_INFO(@SYMBOL_LEVEL,@SYMBOL_NUMBER),1,1)")
       .DO("= 'R';")
       .OUT("DECLARE ",**)
       ( .TABLE_TEST(1,1,"R")  .OUT(" DECIMAL FLOAT (6)")
       | .EMPTY  .OUT(" FIXED BINARY (15,0)") )
       ( .TABLE_TEST(2,1,"P")
       | STATIC_IF_NOT_RECURSIVE )
       .OUT(";") )  ;

STATIC_IF_NOT_RECURSIVE  :=
      .TEST(RECURSIVE_FLAG = 0)
      .OUT(" STATIC")
    | .EMPTY  ;

PROGRAM_UNIT  :=
      $( .LABEL
         ( .PEEK("PROCEDURE")
         | .EMPTY .OUT(";") )
         .OUT(*) )
      .SET(TRACE_ALREADY_OUTPUT_FLAG = 0)
      ( .PEEK("END")  .NEG  .RETURN
      | PROCEDURE_BLOCK
      | STATEMENT ) ;

PROCEDURE_BLOCK  :=
      .PEEK("PROCEDURE")
      .DO("@SYMBOL = SUBSTR(@SYMBOL,1,LENGTH(@SYMBOL) - 1);")
      .SEARCH_ALL  .IF_NEW(1,2,"PD")
      .TABLE_TEST(1,1,"P")  .ERR(3)
      .ENTER(1,2,"PD")
      "PROCEDURE"  .OUT(*)
      .BLKENTER
      .SEARCH_BLOCK  .ENTER(1,2,"PD")
      ( "("   $",")  .OUT(*)
        PROCEDURE_ARGUMENT
        $( ","    $","  .OUT(",")  PROCEDURE_ARGUMENT )
        ")"  .OUT(*)
      | .EMPTY )
      ( "RECURSIVE"  .OUT(" RECURSIVE")
        .INIT_BLOCK  .FIND_NEXT(1,1,"P")  .ENTER(2,1,"R")
      | .EMPTY )
      .OUT(";")
      .DO("@LEVEL = @LEVEL + 1;")
      TRACE_OUTPUT
      .SET(STATISTICS_SWITCH = 1)
      TAKE_STATISTICS
      SEMI_COLON
      $( "DECLARE"  DECLARE_STATEMENT )
      $ PROGRAM_UNIT
      OUTPUT_DECLARATIONS
      "END"
      TRACE_OUTPUT
      .OUT("@PROC_EXIT: ;")
      PRUNE_LOCAL_TREES
      ( .ID | .EMPTY )
      .DO("@LEVEL = @LEVEL - 1;")
      SEMI_COLON  .OUT("END;")
      .BLKEXIT ;

PROCEDURE_ARGUMENT  :=
      .TREE  .OUT(*)
      .SEARCH_BLOCK  .IF_NEW(1,2,"TP")
    | .ID  .OUT(*)
      .SEARCH_BLOCK  .IF_NEW(1,2,"VP") ;

DECLARE_STATEMENT  :=
      .SET(STATISTICS_SWITCH = 11)
      TAKE_STATISTICS
      $ ","
      DECLARE_ITEM
      $( ","  $ ","  DECLARE_ITEM )
      "LOCAL"
      SEMI_COLON ;

DECLARE_ITEM  :=
      .PEEK("LOCAL")
    | .TREE
      .SEARCH_BLOCK  .IF_NEW(1,2,"$D")
    | .ID
      .SEARCH_BLOCK  .IF_NEW(1,2,"VD")  ;

PRUNE_LOCAL_TREES  :=
      .TEST(MAIN_EXTERNAL_SWITCH = 2)
      .DO("BLOCK_LEVEL_COUNT = @BLK_LEVEL_CNT;")
      (  .TEST(BLOCK_LEVEL_COUNT = 1)
         .INIT_BLOCK
         $( .FIND_NEXT(1,1,"T")
            (  .TABLE_TEST(2,1,"P")
            |  .EMPTY
               .OUT("@SOFT_LINK_ADDR = ADDR(",**,");")
               .OUT("CALL @PRUNE;")  .SEARCH_PROCEDURE("@PRUNE") ) )
      |  PRUNE_DECLARED_TREES )
    |  PRUNE_DECLARED_TREES  ;

PRUNE_DECLARED_TREES  :=
      .EMPTY
      .INIT_BLOCK
      $( .FIND_NEXT(1,2,"TD")
         .OUT("@SOFT_LINK_ADDR = ADDR(",**,");")
         .OUT("CALL @PRUNE;")  .SEARCH_PROCEDURE("@PRUNE") )  ;
```


Fig. I.4-2 (cont)

```
                              /*********************/
                              /* STATEMENT TYPES */
                              /*********************/

STATEMENT  :=
      .SNIR_USE(SOFT*01,NUM*01,STR*01)
      .RFSET(SOFT*01,NUM*01,STR*01)
      TRACE_OUTPUT
      .SET(AS_GR_IN_GRIN_PF_IF_OTHR_SWITCE=7)
      ( CONDITIONAL_STATFMFNT
      | UNCONDITIONAL_STATEMENT  SEMI_COLON )
      .RFSET(SOFT*01,NUM*01,STR*01)  ;

UNCONDITIONAL_STATEMENT  :=
      .PFEK("END")   .NEG   .RETURN
    | .PFFK(";")   .OUT(";")
    | "STOP"
      .SFT(STATISTICS_SWITCH = 14)
      TAKE_STATISTICS
      ( .TEST(TAKE_STATISTICS_FLAG = 1)
        .OUT("GO TO @EXIT;")
      | .EMPTY
        .OUT("RETURN;") )
    | "RETURN"
      .SFT(STATISTICS_SWITCP = 15)
      TAKE_STATISTICS
      .OUT("GO TO @PROC_EXIT:")
    | "TRACE"
      .SET(STATISTICS_SWITCE = 26)
      TAKE_STATISTICS
      ( .TEST(TRACP_FLAG = 1)
        .OUT("@TRACE = ")
        ( "HIGH"   .OUT("2;")
        | "LOW"   .OUT("1;")
        | "OFF"   .OUT("0;") )
      | .ID )
    | "BEGIN"  BEGIN_BLOCK
    | "DO"  DO_GROUP
    | "GO"   ( "TO"  |  .EMPTY )   .ID
      .SET(STATISTICS_SWITCE = 12)
      TAKE_STATISTICS
      .OUT("GO TO ",*,";")
    | "CALL"  CALL_STATEMENT
    | TREE_ASSIGNMENT_STATEMENT
    | "LABEL"  LABEL_ASSIGNMENT_STATEMENT
    | "PRUNE"  PRUNE_STATEMENT
    | "GRAFT"  GRAFT_STATEMENT
    | "INSERT"  .SET(INSERT_GRAFTINSERT_SWITCE = 1)
      .SET(STATISTICS_SWITCE = 19)
      TAKE_STATISTICS
      INSERT_STATEMENT
    | "ADVANCE"  ADVANCE_STATEMENT
    | INPUT_OUTPUT_STATEMENT
    | "DEFINE"  DEFINE_STATEMENT
    | "ORDER"  ORDER_STATEMENT
    | ARITHMETIC_ASSIGNMENT_STATEMENT  ;

                                   /*****************/
```

```
                              /* BFGIN BLOCK */
                              /*****************/

BFGIN_BLOCK  :=
      .BLKENTFR
      .SFARCP_BLOCK  .IF_NEW(1,2,"PD")
      SEMI_COLON
      .OUT("BFGIN;")
      .SET(STATISTICS_SWITCP = 2)
      TAKP_STATISTICS
      $( "DECLARE"  DECLARE_STATEMENT )
      $ PROGRAM_UNIT
      OUTPUT_DECLARATIONS
      "END"
      TRACE_OUTPUT
      .DO("@LEVEL = @LEVEL - 1;")
      .OUT("END;")
      .BLKEXIT  ;

                              /*************/
                              /* DO GROUP */
                              /*************/

DO_GROUP  :=
      "WHILE"  DO_WHILE_GROUP
    | "FOR"  "ALL"
      ( "SUBNODES"  DO_SUBNODE_GROUP
      | "COMBINATIONS"  DO_COMBINATION_GROUP
      | "PERMUTATIONS"  DO_PERMUTATION_GROUP )
    | DO_INCREMENT_GROUP
    | .SFT(STATISTICS_SWITCP = 3)
      TAKE_STATISTICS
      .OUT("DO;")
      DO_BODY_THROUGH_END
      .OUT("END;")  ;

DO_SUBNODE_GROUP  :=
      .SNIR_USE(SOFT*01)   .RESET(SOFT*01)
      "OF"
      .SET(STATISTICS_SWITCF = 6)
      TAKE_STATISTICS
      SOFT_TREE_NODE
      "USING"
      BASED_TREE_NAME
      .ENTER(3,1,"A")
      .OUT(*,"@ = ADDR(@SON(",SOFT*01,"));")
      .OUT("DO WHILE (",*,"$ > 0);")
      .SAV(*)
      .CAT("@ = ADDF(@BROTHER(",*,"$));")
      .RESET(SOFT*01)
      DO_BODY_THROUGH_END
      .DO("DO @I=@BLK_SYMBOL_CNT(@BLK_LEVEL_CNT) TO 1 BY -1¢")
      .DO("WHILE(SUBSTR(@ASS_INFO(@BLK_LEVEL_CNT,@I),3,1) ¬= 'A');")
      .DO("END;")
      .DO("SUBSTR(@ASS_INFO(@BLK_LEVEL_CNT,@I),3,1) = ' ';")
      .OUT(#,"END;")  ;

DO_INCREMENT_GROUP  :=
```

Fig. I.4-2 (cont)

```
        .SNIR_USE(LAB*01,NUM*01)
        .RESET(LAB*01,NUM*01)
        .ID
        .SAV("")
        ( .TEST(TRACE_FLAG = 1)
          .CAT("IF @TRACE > 1 THEN PUT EDIT (''TRACE: VARIABLE'',''")
          .CAT(*," - '',",*,") (COL(70),A,COL(77),A,E(13,6));")
        | .EMPTY ).
        .SET(STATISTICS_SWITCH = 4)
        TAKE_STATISTICS
        .OUT("DO ",*,"=")
        .SET(TREE_STRING_ARITH_SWITCH = 3)
        "="
        CONSTRAINED_EXPRESSION
        $( "," .OUT(*) CONSTRAINED_EXPRESSION )
        ( "TO" .OUT(" TO ")
          CONSTRAINED_EXPRESSION
          ( "BY" .OUT(" BY ")
            CONSTRAINED_EXPRESSION
          | .EMPTY )
          $( "," .OUT(*) CONSTRAINED_EXPRESSION )
        | .EMPTY )
        .OUT(";")
        ( "WHILE" "(" .OUT("IF ")
          BOOLEAN_EXPRESSION
          ")" .OUT(" THEN; ELSE GO TO ",LAB*01,";")
          .RESET(NUM*01)
          .OUT(#)
          DO_BODY_THROUGH_END
          (.ID | .EMPTY)
          .OUT("END;")
          .LAB(LAB*01)
          ( .TEST(TRACE_FLAG = 1)
          | .EMPTY .OUT(";") )
          .RESET(LAB*02)
        | .OUT(#)
          .RESET(NUM*01)
          DO_BODY_THROUGH_END
          .OUT("END;") ) ;

DO_COMBINATION_GROUP :=
        .SNIR_USE(LAB*02)
        .SET(STATISTICS_SWITCH = 7)
        TAKE_STATISTICS
        ( "OF" | .EMPTY )
        SORT_TREE_NODE
        "TAKEN"
        .OUT("@COMBINATION_SIZE=")
        ARITH_EXPRESSION
        "AT"
        ( "A" | .EMPTY )
        ( "TIME" | .EMPTY )
        .OUT(";","IF @FIRST_COMBIN(",SORT*01,") THEN GO TO ",LAB*02,";")
        .LAB(LAB*01)
        .RESET(SORT*01)
        DO_BODY_THROUGH_END
        .OUT("IF @NEXT_COMBIN THEN GO TO ",LAB*01,";")
        .LAB(LAB*02)  .OUT(";")
        .SEARCH_PROCEDURE("@COMBIN")  ;

DO_PERMUTATION_GROUP :=
        .SNIR_USE(LAB*02)
        .SET(STATISTICS_SWITCH = 8)
        TAKE_STATISTICS
        ( "OF" | .EMPTY )
        SORT_TREE_NODE
        "TAKEN"
        .OUT("@COMBINATION_SIZE=")
        ARITH_EXPRESSION
        "AT"
        ( "A" | .EMPTY )
        ( "TIME" | .EMPTY )
        .OUT(";","IF @FIRST_PERMUT(",SORT*01,") THEN GO TO ",LAB*02,";")
        .LAB(LAB*01)
        .RESET(SORT*01)
        DO_BODY_THROUGH_END
        .OUT("IF @NEXT_PERMUT THEN GO TO ",LAB*01,";")
        .LAB(LAB*02)  .OUT(";")
        .SEARCH_PROCEDURE("@COMBIN")
        .SEARCH_PROCEDURE("@PERMUT")  ;

DO_WHILE_GROUP :=
        .SNIR_USE(LAB*02,SORT*01,NUM*01,STR*01)
        .RESET(SORT*01,NUM*01,STR*01)
        .SET(STATISTICS_SWITCH = 5)
        TAKE_STATISTICS
        .LAB(LAB*01)
        ( "(" | .EMPTY )  .OUT("IF ")
        BOOLEAN_EXPRESSION
        ")"  .OUT(" THEN; ELSE GO TO ",LAB*02,";")
        .RESET(SORT*01,NUM*01,STR*01)
        DO_BODY_THROUGH_END
        .OUT("GO TO ",LAB*01,";")
        .LAB(LAB*02)  .OUT(";")  ;

DO_BODY_THROUGH_END :=
        .DO("@NEST = @NEST + 1;")
        SEMI_COLON
        $ PROGRAM_UNIT
        "END"
        ( .ID | .EMPTY )
        TRACE_OUTPUT
        .DO("@NEST = @NEST - 1;")  ;

                        /*************************/
                        /* CONDITIONAL STATEMENT */
                        /*************************/

CONDITIONAL_STATEMENT :=
        .SNIR_USE(SORT*01,NUM*01,STR*01)
        .RESET(SORT*01,NUM*01,STR*01)
        "IF"
        .SET(STATISTICS_SWITCH = 9)
        TAKE_STATISTICS
        .OUT("IF ")
        BOOLEAN_EXPRESSION
        .DO("@STMT = @STMT + 1;")
```

Fig. I.4-2 (cont)

```
( ";" | .EMPTY )
"THEN"   .OUT(" THEN DO;")
.RESET(SORT*01,NUM*01,STR*01)
STATEMENT   .OUT("END;")
( "ELSE"   .OUT("ELSE DO;")   STATEMENT   .OUT("END;")
| .EMPTY )   ;

                    /********************/
                    /* CALL STATEMENT */
                    /********************/

CALL_STATEMENT  :=
     .SNIR_USE(HARD*01)
     .RESET(HARD*01)
     .SET(STATISTICS_SWITCH = 13)
     TAKE_STATISTICS
     .ID
     .SEARCH_ALL   .IF_NEW(1,2,"PU")
     .TABLE_TEST(1,1,"P")   .ERR(3)
     .OUT("CALL ",*)
     ( "("   .OUT(*)
       CALL_ARGUMENT
       $( ","   .OUT(*)   CALL_ARGUMENT )
       ")"   .OUT(*)
     | .EMPTY )
     .OUT(";")
     .RESET(HARD*01)   :

CALL_ARGUMENT  :=
     .SNIR_USE(HARD*01)
     .RESET(HARD*01)
     .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
     HARD_TREE_NODE
     .OUT(HARD*01,"@ = @HARD_LINK_ADDR;")
     .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
     .OUT(HARD*01,"$")
     .RESET(HARD*02)
   | .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
     .SET(TREE_STRING_ARITH_SWITCH = 3)
     CONSTRAINED_EXPRESSION   ;

                    /*********************/
                    /* ORDER STATEMENT */
                    /*********************/

ORDER_STATEMENT  :=
     .SNIR_USE  (SORT*01,LAB*01)
     .RESET(SORT*01)
     .DO("@SORT_POSITION = 3;")
     .SAV("CALL PLISRTD('' SORT FIELDS=(")
     SORT_TREE_NODE
     "BY"
     .OUT("@TEMP_ADDR = ADDR(@SON(",SORT*01,"));")
     .OUT(SORT*01," = @SON(",SORT*01,");")
     .LAB(LAB*01)
     .OUT("PROCEDURE RETURNS(CHAR(95));")
     .OUT("IF ",SORT*01," <= 0 ¢")
     .OUT("THEN DO; CALL PLIRETC(8); RETURN(@SORT_DATA); END;")
     .OUT("ELSE CALL PLIRETC(12);")
     .OUT("@SORT_TEMP = ",SORT*01,";")
     .OUT("UNSPEC(@SORT_CHAR) = UNSPEC(",SORT*01,");")
     .OUT("SUBSTR(@SORT_DATA,1,2) = @SORT_CHAR;")
     .RESET(SORT*01)
     $","
     ORDER_ARGUMENT
     $","
     $(.PEEK(":")  .NEG
      | .RESET(SORT*01)  .CAT(",")  ORDER_ARGUMENT $"," )
     .OUT(SORT*01," = @BROTHER(@SORT_TEMP);")
     .OUT("RETURN(@SORT_DATA);")
     .OUT("END;")
     .OUT(#,") '','' RECORD TYPE=F,LENGTH=(95)")
     .OUT(" '',20000,@SORT_RETURN_CODE,",LAB*01,",@SORT_OUT);")
     .SEARCH_PROCEDURE("@SOROUT")   ;

ORDER_ARGUMENT  :=
     .SNIR_USE(SORT*01)
     .RESET(SORT*01)
     ( "-" .DO("@SORT_DIRECTION = 'A';")
     | .EMPTY .DO("@SORT_DIRECTION = 'D';")   )
     ( "$ELEMENT" | .SET(PERIOD_ALREADY_FOUND_FLAG = 1)
       SORT_QUALIFIER_BY_LABEL )
     $(.RESET(SORT*01)  SORT_QUALIFIER_BY_LABEL
     | .RESET(SORT*01)  SORT_QUALIFIER_BY_SUBSCRIPT)
     .OUT("@SORT_TEMP_NUM = @GET_VALUE_STRING(",SORT*01,");")
     .OUT("SUBSTR(@SORT_DATA,")
     .DO("PUT STRING(@SORT_TEMP_STRING) EDIT  (@SORT_POSITION)")
     .DO("(F(10));")
     .DO("CALL @OUT(")
     .DO("SUBSTR(@SORT_TEMP_STRING,VERIFY(@SORT_TEMP_STRING,' ')));")
     .OUT(",10)  = @SORT_TEMP_OVER;")
     .DO("PUT STRING(@SORT_TEMP_STRING) EDIT  (@SORT_POSITION)")
     .DO("(F(10));")
     .DO("CALL @CAT(")
     .DO("SUBSTR(@SORT_TEMP_STRING,VERIFY(@SORT_TEMP_STRING,' ')));")
     .CAT(",10,FL,")
     .DO("CALL @CAT(@SORT_DIRECTION);")
     .DO("@SORT_POSITION = @SORT_POSITION + 10;")   ;

                    /*******************************/
                    /* TREE ASSIGNMENT STATEMENT */
                    /*******************************/

TREE_ASSIGNMENT_STATEMENT  :=
     HARD_TREE_NODE
     "="
     .SET(AS_GR_IN_GRIN_ER_IF_OTER_SWITCH=1)
     .SET(SET_SOFT_LINK_FLAG = 0)
     .SET(TREE_STRING_ARITH_SWITCH = 1)
     CONSTRAINED_EXPRESSION
     .SET(SET_SOFT_LINK_FLAG = 1)
     SET_LABEL_REPLACE_FLAG
     ( .TEST(REAL_DUMMY_SWITCH = 1)
       .OUT("CALL @COPY(",SORT*01,");")
       .SEARCH_PROCEDURE("@COPY")   .SEARCH_PROCEDURE("@PRUNE")
```

Fig. I.4-2 (cont)

```
  | .EMPTY
    .OUT ("CALL @GRAFT;")
    .SEARCH_PROCFDURF ("@GRAFT")   .SFARCH_PROCFDURF ("@PRUNE") )
  .SPT (STATISTICS_SWITCP = 16)
  TAKE_STATISTICS   ;

                    /*********************************/
                    /* LABEL ASSIGNMENT STATEMENT */
                    /*********************************/

LABEL_ASSIGNMENT_STATEMENT   :=
      .SPT (STATISTICS_SWITCP = 17)
      TAKP_STATISTICS
      "("  FARD_TREE_NODE  ")"  "="
      .OUT ("CALL @RFSPRVP (")
      .SPT (TREE_STRING_ARITE_SWITCH = 2)
      CONSTRAINED_EXPRESSION
      .OUT (",@LABEL (@HARD_LINK) );")
      ( .TEST (TRACE_FLAG = 1)
        .OUT ("IF @TRACE > 1 THEN CALL @NODE_TRACE (@PARD_LINK);")
      | .EMPTY )   ;

                         /*********************/
                         /* PRUNP STATEMFNT */
                         /*********************/

PRUNE_STATEMENT   :=
      .SNIR_USP (SOFT*01)   .RPSET (SOFT*01)
      .SET (STATISTICS_SWITCH = 18)
      TAKE_STATISTICS
      .SPT (AS_GR_IN_GRIN_PR_IF_OTER_SWITCH=5)
      .SET (SET_SOFT_LINK_FLAG = 1)
      SOFT_TFPE_NODE
      .OUT ("CALL @PRUNE;")   .SEARCF_PROCFDURE ("@PRUNF")
      $( .PESET (SOFT*01)
         ","
         SOFT_TREE_NODP
         .OUT ("CALL @PRUNE;") )
         .SET (SET_SOFT_LINK_PLAG = 0)   ;

                         /*********************/
                         /* GPAFT STATEMENT */
                         /*********************/

GRAFT_STATEMENT   :=
      .SPT (SFT_SOFT_LINK_FLAG = 1)
    ( "INSERT"   .SET (INSERT_GRAFTINSERT_SWITCP= 2)
      .SPT (STATISTICS_SWITCP = 21)
      TAKE_STATISTICS
      .SET (SET_SOFT_LINK_FLAG = 1)
      INSERT_STATPMENT
      .SPT (SPT_SOFT_LINK_FLAG = 0)
    | .SET (TREE_STRING_ARITE_SWITCH = 1)
      .SPT (STATISTICS_SWITCH = 20)
      TAKE_STATISTICS
      .SET (SET_SOFT_LINK_FLAG = 1)
      CONSTRAINED_EXPRESSION
      .SET (SPT_SOFT_LINK_FLAG = 0)
      "AT"
      HARD_TREE_NODE
      SPT_LABEL_REPLACE_FLAG
      .OUT ("CALL @GRAFT;")   .SPARCH_PROCFDURE ("@GPAFT")
                              .SEARCH_PROCFDURE ("@PRUNP")   ) ;

                         /*********************/
                         /* INSERT STATEMENT */
                         /*********************/

INSERT_STATEMENT   :=
      .SNIR_USE (SOFT*01)
      .RESET (SOFT*01)
      .SET (TREE_STRING_ARITE_SWITCH = 1)
       CONSTRAINED_EXPRESSION
       ( "BEFORE" | .EMPTY )
       .SET (POSSIBLE_ROOT_NODE_FLAG = 0)
       HARD_TREE_NODE
       .TEST (POSSIBLE_ROOT_NODE_FLAG = 0)   .ERR (5)
       .OUT ("IF @SON (@HARD_LINK) ¬= 0 | @LABEL (@HARD_LINK) ¬= 0 ")
       .OUT ("THEN DO ;  @TEMP_SAVE = @HARD_LINK;")
       .OUT ("@HARD_LINK = @NEW_NODE;")
       .OUT ("@BROTHER (@HARD_LINK) = @TEMP_SAVE;")
       .OUT ("END;")
       .OUT ("@LABEL_REPLACE_FLAG = ''T'';")
     ( .TEST (INSERT_GRAFTINSERT_SWITCH = 1)
       ( .TEST (REAL_DUMMY_SWITCH = 1)
         .OUT ("CALL @COPY (",SOFT*01,") ;")
         .SEARCH_PROCEDURE ("@COPY")   .SEARCH_PROCEDURE ("@PRUNE")
       |  .EMPTY   .OUT ("CALL @GRAFT;")
          .SEARCH_PROCEDURE ("@GRAFT")   .SEARCH_PROCEDURE ("@PRUNE")   )
   |  .EMPTY   .OUT ("CALL @GRAFT;")
      .SEARCH_PROCEDURE ("@GRAFT")   .SEARCH_PROCEDURE ("@PRUNE")   ) ;

                         /***********************/
                         /* ADVANCE STATEMENT */
                         /***********************/

ADVANCE_STATEMENT   :=
      .EMPTY
      .SET (STATISTICS_SWITCH = 56)
      TAKE_STATISTICS
      BASED_TREE_NAME
      .OUT (*,"@ = ADDR (@BROTHER (",*,"$) );")   ;

                    /******************************/
                    /* INPUT/OUTPUT STATEMENT */
                    /******************************/

INPUT_OUTPUT_STATEMENT   :=
      .SNIR_USE (SOFT*01)
      .RESET (SOFT*01)
      "READ"
      .SET (STATISTICS_SWITCH = 23)
```

Fig. I.4-2 (cont)

```
        TAKE_STATISTICS
        ( "FILE"
          ( "(" | .EMPTY )
          .ID   .OUT("@INPUT = ",*,";")
          ")"
        | .EMPTY   .OUT("/* @INPUT = SYSIN */;") )
        ( "COMPRESSED"   .SET(NORMAL_COMPRESSED_SWITCH = 2)
        | .EMPTY   .SET(NORMAL_COMPRESSED_SWITCH = 1) )
        $","
        $( INPUT_ELEMENT  $"," )
      | "WRITE"
        .SET(STATISTICS_SWITCH = 24)
        TAKE_STATISTICS
        ( "FILE"
          ( "(" | .EMPTY )
          .ID   .OUT("@OUTPUT = ",*,";")
          ")"
        | .EMPTY   .OUT("/* @OUTPUT = SYSPRINT */;") )
        ( "COMPRESSED"   .SET(NORMAL_COMPRESSED_SWITCH = 2)
        | .EMPTY   .SET(NORMAL_COMPRESSED_SWITCH = 1) )
        $","
        $( OUTPUT_ELEMENT  $"," ).  ;

INPUT_ELEMENT  :=
        .ID
        .OUT("GET EDIT(", *, ") (COL(1),F(20,0));")
        .OUT("PUT SKIP EDIT(''INPUT:'',", *, ") (A,COL(11),F(20,0));")
      | HARD_TREE_NODE
        ( .TEST(NORMAL_COMPRESSED_SWITCH = 1)
          .OUT("CALL @INPUT_NODE;")   .SEARCH_PROCEDURE("@INPODE")
        | .EMPTY
          .OUT("CALL @INPUT_NODE_COMPRESSED;")
          .SEARCH_PROCEDURE("@INESED") )
        .SEARCH_PROCEDURE("@PRUNE")
        .SEARCH_PROCEDURE("@OUTODE")   ;

OUTPUT_ELEMENT  :=
        STRING_EXPRESSION
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
        .OUT("PUT SKIP EDIT(''OUTPUT:'',")
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
        .OUT(") (A,COL(11),A);")
      | .ID
        .OUT("PUT SKIP EDIT(''OUTPUT:'',", *, ") (A,COL(12),F(13,6));")
      | .SET(SET_SOFT_LINK_FLAG = 0)
        SOFT_TREE_NODE
        ( .TEST(NORMAL_COMPRESSED_SWITCH = 1)
          .OUT("CALL @OUTPUT_NODE(",SOFT*01,");")
          .SEARCH_PROCEDURE("@OUTODE")
        | .EMPTY
          .OUT("CALL @OUTPUT_NODE_COMPRESSED(",SOFT*01,");")
          .SEARCH_PROCEDURE("@OUTSED") )   ;

                    /*********************/
                    /* DEFINE STATEMENT */
                    /*********************/

DEFINE_STATEMENT  :=
        .EMPTY
        BASED_TREE_NAME
        .SAV(*)
        "AS"
        HARD_TREE_NODE
        .OUT(#,"@ = @HARD_LINK_ADDR;")   ;

                    /************************************/
                    /* ARITHMETIC ASSIGNMENT STATEMENT */
                    /************************************/

ARITHMETIC_ASSIGNMENT_STATEMENT  :=
        .ID  .OUT(*)
        .SAV("")
        ( .TEST(TRACE_FLAG = 1)
          .CAT("IF @TRACE > 1 THEN PUT EDIT (''TRACE: VARIABLE'',''")
          .CAT(*," - '',",*,") (COL(70),A,COL(77),A,E(13,6));")
        | .EMPTY )
        "="  .OUT(*)
        .SET(TREE_STRING_ARITH_SWITCH = 3)
        CONSTRAINED_EXPRESSION
        .OUT(";",#)
        .SET(STATISTICS_SWITCH = 25)
        TAKE_STATISTICS  ;

                    /************************/
                    /* GENERAL EXPRESSIONS */
                    /************************/

CONSTRAINED_EXPRESSION  :=
        .SKIP_USE(SOFT*01,STR*01,NUM*01)
        .RESET(SOFT*01,STR*01,NUM*01)
        .TEST(TREE_STRING_ARITH_SWITCH = 1)
        ( SOFT_TREE_NODE
          ( PEEK_FOR_ARITH_OPERATOR
            .RESET(SOFT*02,NUM*01)
            .OUT(STR*01," = @GET_VALUE_ARITH(",SOFT*01,")")
            ARITH_EXPRESSION
            .OUT(";")
            .RESET(SOFT*01,STR*01)
            .SET(TREE_STRING_ARITH_SWITCH=2)
            PUT_VALUE_ON_DUMMY_NODE
          | .EMPTY )
        | .OUT(STR*01," = ")
          ( STRING_EXPRESSION
            ( PEEK_FOR_ARITH_OPERATOR
              ARITH_EXPRESSION
            | .EMPTY )
          | ARITH_EXPRESSION )
          .OUT(";")
          .RESET(SOFT*01,STR*01)
          .SET(TREE_STRING_ARITH_SWITCH=2)
          PUT_VALUE_ON_DUMMY_NODE )
      | .TEST(TREE_STRING_ARITH_SWITCH = 2)
        ( STRING_EXPRESSION
          ( PEEK_FOR_ARITH_OPERATOR
            ARITH_EXPRESSION
```

Fig. I.4-2 (cont)

```
      | .EMPTY )
    | .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
      SORT_TREE_NODE
      .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
      .OUT("@GET_VALUE_STRING(",SORT*01,")")
      .RESET(SORT*01)
    | .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
      ARITH_EXPRESSION )
  | ARITH_EXPRESSION ;

UNCONSTRAINED_EXPRESSION :=
     .STIR_USE(SORT*01,NUM*01,STR*01)   .RESET(SORT*01,NUM*01,STR*01)
     .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
     SORT_TREE_NODE
     ( PEEK_FOR_ARITH_OPERATOR
       .OUT(NUM*01," = @GET_VALUE_ARITH(",SORT*01,")")
       ARITH_EXPRESSION  .OUT(";")
       .RESET(SORT*01,STR*01,NUM*02)
     | .EMPTY
       .SET(TREE_STRING_ARITH_SWITCH=1) )
     .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")

   | STRING_EXPRESSION
     ( PEEK_FOR_ARITH_OPERATOR
       .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
       .OUT(NUM*01," =")
       .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
       ARITH_EXPRESSION  .OUT(";")
       .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
       .RESET(SORT*01,STR*01,NUM*02)
     | .EMPTY   .RESET(SORT*01,STR*02)
       .SET(TREE_STRING_ARITH_SWITCH=2)
       .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
       .OUT(STR*01," =")
       .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")  .OUT(";")
       .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")  )

   | ARITH_EXPRESSION  .RESET(SORT*01,STR*01,NUM*02)
     .SET(TREE_STRING_ARITH_SWITCH=3)
     .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
     .OUT(NUM*01," =")
     .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")  .OUT(";")
     .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")  ;


                         /**********************/
                         /* BOOLEAN EXPRESSION */
                         /**********************/

BOOLEAN_EXPRESSION :=
     BOOLEAN_TERM
     $( "|"  .OUT(*)  BOOLEAN_TERM )  ;

BOOLEAN_TERM :=
     BOOLEAN_PRIMARY
     $( "&"  .OUT(*)  BOOLEAN_PRIMARY )  ;

BOOLEAN_PRIMARY :=
     .STIR_USE(SORT*02,NUM*02,STR*02)
     .RESET(SORT*01,NUM*01,STR*01)
     "("  .OUT(*)
     BOOLEAN_EXPRESSION
     ")"  .OUT(*)
   | "¬"  .OUT(*)  BOOLEAN_EXPRESSION
   | UNCONSTRAINED_EXPRESSION
     ( ( "="  .SAV("=")
       | "¬="  .SAV("¬=") )
       ( .TEST(TREE_STRING_ARITH_SWITCH=1)
         UNCONSTRAINED_EXPRESSION
         ( .TEST(TREE_STRING_ARITH_SWITCH=3)
           .OUT("@GET_VALUE_ARITH(",SORT*01,")  ",#,"= ",NUM*01)
         | .OUT(#,"@COMPARE_STRING_OR_ARITH(@GET_VALUE_STRING(")
           .OUT(SORT*01)
           ( .TEST(TREE_STRING_ARITH_SWITCH=2)
             .OUT("),",STR*01,")")
           | .EMPTY
             .OUT("),@GET_VALUE_STRING(",SORT*02,"))")  )  )
       | .TEST(TREE_STRING_ARITH_SWITCH=2)
         UNCONSTRAINED_EXPRESSION
         ( .TEST(TREE_STRING_ARITH_SWITCH=3)
           .OUT(STR*01,#,"=",NUM*01)
         | .OUT(#,"@COMPARE_STRING_OR_ARITH(",STR*01)
           ( .TEST(TREE_STRING_ARITH_SWITCH=2)
             .OUT(",",STR*02,")")
           | .EMPTY
             .OUT(",@GET_VALUE_STRING(",SORT*01,"))")  )  )
       | .OUT(NUM*01,#,"=")
         UNCONSTRAINED_EXPRESSION
         ( .TEST(TREE_STRING_ARITH_SWITCH=3)
           .OUT(NUM*02)
         | .TEST(TREE_STRING_ARITH_SWITCH=2)
           .OUT(STR*01)
         | .EMPTY
           .OUT("@GET_VALUE_ARITH(",SORT*01,")")  )  )
     | ARITH_RELATION
       ( .TEST(TREE_STRING_ARITH_SWITCH = 1)
         .OUT("@GET_VALUE_STRING(",SORT*01,")",#)
         .SET(TREE_STRING_ARITH_SWITCH = 3)
         CONSTRAINED_EXPRESSION
       | .TEST(TREE_STRING_ARITH_SWITCH = 2)
         .OUT(STR*01,#)
         .SET(TREE_STRING_ARITH_SWITCH = 3)
         CONSTRAINED_EXPRESSION
       | .OUT(NUM*01,#)
         CONSTRAINED_EXPRESSION )
     | ( .TEST(TREE_STRING_ARITH_SWITCH = 1)
       | .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
         .RESET(SORT*01)
         ( .TEST(TREE_STRING_ARITH_SWITCH=2)   .RESET(STR*01)
         | .EMPTY   .RESET(NUM*01) )
         PUT_VALUE_ON_DUMMY_NODE
         .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;") )
       .RESET(SORT*01)
       ( ("¬" | "NOT")   .OUT("¬")
       | .EMPTY )
       TREE_RELATION
       .SET(TREE_STRING_ARITH_SWITCH = 1)
       .RESET(SORT*02)
```

Fig. I.4-2 (cont)

```
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
        CONSTRAINED_EXPRESSION
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;") )   ;

ARITH_RELATION  :=
      ( "<" | "<=" | ">=" | ">" | "¬<" | "¬>" )
      .SAV(*)
    | "¬<="   .SAV(">")
    | "¬>="   .SAV("<")   ;

TREE_RELATION  :=
      .SNIR_USE(SOFT*01)
      ( "IDENTICAL"   ( "TO" | .EMPTY )
        .OUT("@COMPARE_IDENTICAL")   .SEARCH_PROCEDURE("@COMPIDE")
      | "SUBSET"   ( "OF" | .EMPTY )
        .OUT("@COMPARE_SUBSET")    .SEARCH_PROCEDURE("@COMSET")
                                   .SEARCH_PROCEDURE("@COMPIDE")
      | "ELEMENT"   ( "OF" | .EMPTY )
        .OUT("@COMPARE_ELEMENT")   .SEARCH_PROCEDURE("@COMENT")
                                   .SEARCH_PROCEDURE("@COMPIDE") )
      .OUT(" (",SOFT*01,",",SOFT*02,")")   ;

                          /*********************/
                          /* TREE EXPRESSIONS */
                          /*********************/

 HARD_TREE_NODE  :=
      ( ( COMBINATION_SUBSCRIPT
            .OUT("CALL @HARD_SUBSCRIPT_COMBIN;")
        | PERMUTATION_SUBSCRIPT
            .OUT("CALL @HARD_SUBSCRIPT_PERMUT;") )
        .SET(STATISTICS_SWITCH = 36)
        TAKE_STATISTICS
      | .TREE   .SEARCH_ALL   .IF_NEW(1,2,"TU")
        ( .TABLE_TEST(1,1,"$")   .ENTER(1,1,"T")
        | .EMPTY )
        ( .TABLE_TEST(1,1,"T")
          .OUT("IF ",*," = 0 THEN ",*," = @NEW_NODE;")
          .OUT("@HARD_LINK_ADDR = ADDR(",*,");")
          ( .TABLE_TEST(2,1,"P")
          | .SET(POSSIBLE_ROOT_NODE_FLAG = 1)
            .EMPTY )
        | .TABLE_TEST(1,1,"B")
          .TABLE_TEST(3,1,"A")   .ERR(4)
          .OUT("@HARD_LINK_ADDR = ",*,"@;")
          .OUT("IF @HARD_LINK = 0 THEN @HARD_LINK = @NEW_NODE;") ) )
      .SET(STATISTICS_SWITCH = 28)
      TAKE_STATISTICS
      .SET(ADDITIONAL_QUALIFIER_LEGAL_FLAG = 1)
      .SET(SET_SOFT_LINK_FLAG = 0)
      .SET(LABEL_SUBSCRIPT_SWITCH = 2)
      $( HARD_QUALIFIER_BY_SUBSCRIPT   .SET(POSSIBLE_ROOT_NODE_FLAG = 0)
       | HARD_QUALIFIER_BY_LABEL  .SET(POSSIBLE_ROOT_NODE_FLAG = 0) )  ;

  SOFT_TREE_NODE  :=
      .SNIR_USE(SOFT*01)   .RESET(SOFT*01)
      ( COMBINATION_SUBSCRIPT
          .OUT("CALL @SOFT_SUBSCRIPT_COMBIN(",SOFT*01,");")
      | PERMUTATION_SUBSCRIPT
          .OUT("CALL @SOFT_SUBSCRIPT_PERMUT(",SOFT*01,");") )
      .SET(STATISTICS_SWITCH = 29)
      TAKE_STATISTICS
      .SET(STATISTICS_SWITCH = 44)
      TAKE_STATISTICS
      .SET(REAL_DUMMY_SWITCH = 1)
      $( .RESET(SOFT*01)
         SOFT_QUALIFIER_BY_SUBSCRIPT
       | .RESET(SOFT*01)
         SOFT_QUALIFIER_BY_LABEL )
      .RESET(SOFT*02)
    | ( "$ELEMENT"   .SEARCH_ALL   .IF_NEW(1,2,"BU")
        .SET(ELEMENT_FLAG = 1)
        ( .TABLE_TEST(1,1,"B")
        | .EMPTY   .MESSAGE(1) )
      | .TREE   .SEARCH_ALL   .IF_NEW(1,2,"TU")   .SET(ELEMENT_FLAG = 0) )
      ( .TABLE_TEST(1,1,"$")   .ENTER(1,1,"T")
      | .EMPTY )
      ( .TABLE_TEST(1,1,"B")
        ( .TABLE_TEST(3,1,"A")
          .TEST(ELEMENT_FLAG = 0)   .ERR(4)
        | .EMPTY )
      | .EMPTY )
      .SET(STATISTICS_SWITCH = 29)
      TAKE_STATISTICS
      ( .TABLE_TEST(1,1,"T")
        .OUT(SOFT*01," = ",*,";")
        ( .TEST(SET_SOFT_LINK_FLAG = 1)
          .OUT("@SOFT_LINK_ADDR = ADDR(",*,");")
        | .EMPTY )
      | .OUT(SOFT*01," = ",*,"$;")
        ( .TEST(SET_SOFT_LINK_FLAG = 1)
          .OUT("@SOFT_LINK_ADDR = ",*,"@;")
        | .EMPTY ) )
      .SET(REAL_DUMMY_SWITCH = 1)
      ( .TEST(SET_SOFT_LINK_FLAG = 1)
        $( .SET(SET_SOFT_LINK_FLAG = 1)
           .RESET(SOFT*01)
           SOFT_QUALIFIER_BY_SUBSCRIPT
         | .RESET(SOFT*01)
           SOFT_QUALIFIER_BY_LABEL )
      | $( .RESET(SOFT*01)
           SOFT_QUALIFIER_BY_SUBSCRIPT
         | .RESET(SOFT*01)
           SOFT_QUALIFIER_BY_LABEL ) )
      .RESET(SOFT*02)   ;

HARD_QUALIFIER_BY_SUBSCRIPT  :=
      .SNIR_USE(NUM*01,SOFT*01)
      .RESET(NUM*01,SOFT*01)
      "("
      .SET(LABEL_SUBSCRIPT_SWITCH = 2)
      ( "LAST"
        .SET(STATISTICS_SWITCH = 34)
        TAKE_STATISTICS
        .OUT("CALL @HARD_LAST;")   .SEARCH_PROCEDURE("@HARAST")
      | "NEXT"
        .SET(STATISTICS_SWITCH = 35)
```


Fig. I.4-2 (cont)

```
      TAKE_STATISTICS
      .OUT ("CALL @HARD_NEXT;")   .SEARCH_PROCEDURE ("@HARNEXT")
    | HARD_FIRST_QUALIFIER
    | .OUT ("@SUBSCRIPT = ")
      ARITH_EXPRESSION
      .OUT (";","CALL @HARD_SUB_ACCESS;")
      .SET (STATISTICS_SWITCH = 36)
      TAKE_STATISTICS
      .SEARCH_PROCEDURE ("@HARESS") )
    ") "
    .RESET (NUM*01,SOFT*01)   ;

HARD_FIRST_QUALIFIER  :=
    .SNIP_USE(LAB*01)
    .RESET (LAB*01)
    "FIRST"
    .OUT ("@SUBSCRIPT = 1;")
    .OUT ("CALL @HARD_SUB_ACCESS;")
    .SEARCH_PROCEDURE ("@HARESS")
    .SET (LABEL_SUBSCRIPT_SWITCH = 2)
    .SET (STATISTICS_SWITCH = 32)
    TAKE_STATISTICS
  | "FIRST:"
    .RESET (LAB*02)
    .OUT ("$ELEMENT@ = ADDR(@SON(@HARD_LINK));")
    .OUT ("DO WHILE ($ELEMENT$ > 0);")
    .OUT ("IF ")
    BOOLEAN_EXPRESSION
    .OUT (" THEN GO TO ",LAB*01,";")
    .OUT ("$ELEMENT@ = ADDR(@BROTHER($ELEMENT$));")
    .OUT ("END;")
    .OUT ("$ELEMENT$ = @NEW_NODE;")
    .LAB (LAB*01)
    .OUT ("@HARD_LINK_ADDR = $ELEMENT@;")
    .SET (STATISTICS_SWITCH = 33)
    TAKE_STATISTICS   ;

SOFT_QUALIFIER_BY_SUBSCRIPT  :=
    .SNIP_USE(SOFT*01,NUM*01)
    .RESET (SOFT*01,NUM*01)
    "("
    (  SOFT_FIRST_QUALIFIER
    |  SOFT_ALL_QUALIFIER
    |  "LAST"
       SET_UP_SOFT_SON
       .OUT ("DO WHILE(@BROTHER(",SOFT*01,") ¬= 0);")
       SOFT_LINK_BROTHER
       .OUT (SOFT*01," = @BROTHER(",SOFT*01,");")
       .OUT ("END;")
    |  .RESET(SOFT*02)
       .OUT ("@SUBSCRIPT = ")
       (  .TEST (SET_SOFT_LINK_FLAG=1)
          .SET (SET_SOFT_LINK_FLAG=0)
          ARITH_EXPRESSION
          .SET (SET_SOFT_LINK_FLAG=1)
       |  ARITH_EXPRESSION  )
       .OUT (";")
       (  .TEST (SET_SOFT_LINK_FLAG=1)
          .OUT ("CALL @SSUBLK(",SOFT*01,");")
          .SEARCH_PROCEDURE ("@SSUBLK")
       |  .EMPTY
          .OUT ("CALL @SSUB(",SOFT*01,");")
          .SEARCH_PROCEDURE ("@SSUB")  )  )
    ")"
    .RESET (NUM*01)   ;

SOFT_FIRST_QUALIFIER  :=
    .SNIP_USE(SOFT*01,LAB*01)
    .RESET (SOFT*01,LAB*01)
    "FIRST"
    .SET (STATISTICS_SWITCH = 39)
    TAKE_STATISTICS
    SET_UP_SOFT_SON
    .SET (STATISTICS_SWITCH = 40)
    TAKE_STATISTICS
  | "FIRST:"
    .SET (STATISTICS_SWITCH = 40)
    TAKE_STATISTICS
    .RESET (SOFT*02,LAB*02)
    .OUT ("$ELEMENT@ = ADDR(@SON(",SOFT*01,"));")
    ( .TEST (SET_SOFT_LINK_FLAG = 1)
      .OUT ("@SOFT_LINK_ADDR = $ELEMENT@;")
    | .EMPTY )
    .OUT ("DO WHILE ($ELEMENT$ > 0);")
    .OUT ("IF ")
    ( .TEST (SET_SOFT_LINK_FLAG = 1)
      .SET (SET_SOFT_LINK_FLAG = 0)
      BOOLEAN_EXPRESSION
      .SET (SET_SOFT_LINK_FLAG = 1)
    | BOOLEAN_EXPRESSION )
    .OUT (" THEN GO TO ",LAB*01,";")
    .OUT ("$ELEMENT@ = ADDR(@BROTHER($ELEMENT$));")
    ( .TEST (SET_SOFT_LINK_FLAG = 1)
      .OUT ("@SOFT_LINK_ADDR = $ELEMENT@;")
    | .EMPTY )
    .OUT ("END;")
    .LAB (LAB*01)
    .OUT (SOFT*01," = $ELEMENT$;")
    .RESET (SOFT*01)   ;

SOFT_ALL_QUALIFIER  :=
    .SNIP_USE(SOFT*02)
    "ALL:"
    .SET (STATISTICS_SWITCH = 42)
    TAKE_STATISTICS
    SET_UP_SOFT_SON
    (  .TEST (AS_GR_IN_GRIN_PR_IF_OTHR_SWITCH=1)
       .OUT ("@TEMP_ADDR = @HARD_LINK_ADDR;")
       .OUT ("$ELEMENT@ = ADDR(",SOFT*01,");")
       .OUT (SOFT*02," = @NEW_NODE;")
       .OUT ("@HARD_LINK_ADDR = ADDR(@SON(",SOFT*02,"));")
       .OUT ("@LABEL_REPLACE_FLAG = ''T'';")
       .OUT ("DO WHILE($ELEMENT$ > 0);")
       .OUT ("IF ")
       BOOLEAN_EXPRESSION
       .OUT (" THEN DO;")
       .OUT ("@HARD_LINK = @NEW_NODE;")
       .OUT ("CALL @COPY($ELEMENT$);")
```


Fig. I.4-2 (cont)

```
            .SEARCH_PROCEDURE("@COPY")   .SEARCH_PROCEDURE("@PRUNE")
            .OUT("@HARD_LINK_ADDR = ADDR(@BROTHER(@HARD_LINK));")
            .OUT("END;")
            .OUT("$ELEMENT@ = ADDR(@BROTHER($ELEMENT$));")
            .OUT("END;")
            .OUT("@SOFT_LINK_ADDR = ADDR(",SOFT*02,");")
            .OUT("@HARD_LINK_ADDR = @TEMP_ADDR;")
            .SET(REAL_DUMMY_SWITCH=2)
         |  .TEST(AS_GR_IN_GRIN_PR_IP_OTHR_SWITCH=5)   .RESET(SOFT*02)
            .OUT("$ELEMENT@ = ADDR(",SOFT*01,");")
            .OUT("DO WHILE($ELEMENT$ > 0);")
            .OUT("IF ")
            BOOLEAN_EXPRESSION
            .OUT(" THEN DO;")
            .OUT("@SOFT_LINK_ADDR = $ELEMENT@;")
            .OUT("CALL @PRUNE;")   .SEARCH_PROCEDURE("@PRUNE")
            .OUT("END;")
            .OUT("ELSE $ELEMENT@ = ADDR(@BROTHER($ELEMENT$));")
            .OUT("END;")
         |  .EMPTY   .MESSAGE("S:NOT YET IMPLEMENTED")   )
         .RESET(SOFT*01)   ;

HARD_QUALIFIER_BY_LABEL   :=
         .SNIR_USE(STR*01,SOFT*01)
         .RESET(STR*01,SOFT*01)
         "."
         .SET(LABEL_SUBSCRIPT_SWITCH = 1)
         .OUT("@LABEL_STRING = ")
         ( .ID   .OUT("'",*,"';")
           .SET(STATISTICS_SWITCH = 30)
         | INDIRECT_CHAR_STRING   .OUT(";")
           .SET(STATISTICS_SWITCH = 31) )
         .OUT("CALL @HARD_LABEL;")   .SEARCH_PROCEDURE("@HARDBEL")
         .RESET(STR*01,SOFT*01)   ;

SOFT_QUALIFIER_BY_LABEL   :=
         .SNIR_USE(SOFT*02,STR*01)
         .RESET(SOFT*01,STR*01)
        ("." | .TEST(PERIOD_ALREADY_FOUND_FLAG = 1)
         .SET(PERIOD_ALREADY_FOUND_FLAG = 0) )
         .OUT("@LABEL_STRING = ")
         ( .ID   .OUT("'",*,"'")
           .SET(STATISTICS_SWITCH = 37)
         | .RESET(SOFT*02)
           .TEST(SET_SOFT_LINK_FLAG = 1)
           INDIRECT_CHAR_STRING
           .SET(STATISTICS_SWITCH = 38)
           .SET(SET_SOFT_LINK_FLAG = 1)
         | INDIRECT_CHAR_STRING
           .SET(STATISTICS_SWITCH = 38) )
         .OUT(";")
         TAKE_STATISTICS
         (  .TEST(SET_SOFT_LINK_FLAG=1)
            .OUT("CALL @SLABLK(",SOFT*01,");")
            .SEARCH_PROCEDURE("@SLABLK")
         |  .EMPTY
            .OUT("CALL @SLAB(",SOFT*01,");")
            .SEARCH_PROCEDURE("@SLAB")   )
         .RESET(SOFT*01,STR*01)   ;
```

Fig. I.4-2 (cont)

```
                                   /**********************/
                                   /* STRING EXPRESSIONS */
                                   /**********************/

STRING_EXPRESSION   :=
      .STRING   .OUT(*)
    | "LABEL"  LABEL_STRING   ;

LABEL_STRING   :=
      .SNIR_USE(SOFT*01)   .RESET(SOFT*01)
      .EMPTY
      .SET(SET_SOFT_LINK_FLAG = 0)
      .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
      "("  SOFT_TREE_NODE  ")"
      .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
      .OUT("@GET_LABEL(",SOFT*01,")")   ;

INDIRECT_CHAR_STRING   :=
      .SNIR_USE(STR*01,SOFT*01)
      .RESET(STR*01,SOFT*01)
      "$"
      .SET(SET_SOFT_LINK_FLAG = 0)
      ( "LABEL" LABEL_STRING
      | "("
        ( "LABEL" LABEL_STRING  ")"
        | .EMPTY
          .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
          SOFT_TREE_NODE  ")"
          .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
          .OUT("@GET_VALUE_STRING(",SOFT*01,")")) )   ;

                                /**************************/
                                /* ARITHMETIC EXPRESSION */
                                /**************************/

ARITH_EXPRESSION   :=
      ARITH_TERM
      $( ( "+" | "-" )   .OUT(*)   ARITH_TERM )   ;

ARITH_TERM   :=
      ARITH_FACTOR
      $( ( "*" | "/" )   .OUT(*)   ARITH_FACTOR )   ;

ARITH_FACTOR   :=
      ARITH_PRIMARY
      ( "**"   .OUT(*)   ARITH_FACTOR   |   .EMPTY )   ;

ARITH_PRIMARY   :=
      .SNIR_USE(SOFT*01,NUM*01)   .RESET(SOFT*01,NUM*01)
      "INFINITY"   .OUT("1.E+71")
    | "NUMBER"
      .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
      "("  SOFT_TREE_NODE  ")"
      .OUT(NUM*01," = 0;")
      .OUT(SOFT*01," = @SON(",SOFT*01,");")
      .OUT("DO WHILE (",SOFT*01," > 0);")
```

```
        .OUT(NUM*01," = ",NUM*01," + 1;")
        .OUT(SORT*01," = @BROTHER(",SORT*01,");")
        .OUT("END;")
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
        .OUT(NUM*01)
        .RESET(SORT*01,NUM*02)
      | "(" .OUT(*)
        ARITH_EXPRESSION
        ")" .OUT(*)
      | STRING_EXPRESSION
      | .NUM  .OUT(*)
      | .ID  .OUT(*)
      | .TEST(FIRST_PRIMARY_FOUND_FLAG = 1)
        .SET(FIRST_PRIMARY_FOUND_FLAG = 0)
      | ( "+" | "-" )  .OUT(*)
        ARITH_PRIMARY
      | .EMPTY
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
        SORT_TREE_NODE
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
        .OUT("@GET_VALUE_ARITH(",SORT*01,")")  ;


                         /******************/
                         /* MISCELLANEOUS */
                         /******************/

BASE_TREE_NAME  :=
        .TREE  .SEARCH_ALL  .IF_NEW(1,2,"BU")
        ( .TABLE_TEST(1,1,"S")  .ENTER(1,1,"B")
        | .EMPTY )
        .TABLE_TEST(1,1,"B")  .ERR(1)  .EMPTY  ;

COMBINATION_SUBSCRIPT  :=
        "$COMBINATION"
        "("
        .OUT("@SUBSCRIPT = ")
        ARITH_EXPRESSION
        ")"
        .OUT(";")  ;

CONVERT_STRING_IF_ARITH  :=
        .SNIP_USE(STR*01)
        .RESET(STR*01)
        .TEST(TREE_STRING_ARITH_SWITCH = 3)
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH + 1;")
        .OUT("IF ",STR*01," = '''' THEN ",STR*01," = ''0'';")
        .DO("NORMAL_MERGE_SWITCH = NORMAL_MERGE_SWITCH - 1;")
      | .EMPTY  ;

PEEK_FOR_ARITH_OPERATOR  :=
        .PEEK("+" | "-" | "*" | "/" | "**")
        .SET(TREE_STRING_ARITH_SWITCH = 3)
        .SET(FIRST_PRIMARY_FOUND_FLAG = 1)  ;

PERMUTATION_SUBSCRIPT  :=
        "$PERMUTATION"
        "("
        .OUT("@SUBSCRIPT = ")
        ARITH_EXPRESSION
        ")"
        .OUT(";")  ;

PUT_VALUE_ON_DUMMY_NODE  :=
        .SNIP_USE(SORT*01,STR*01,NUM*01)
        .RESET(STR*01,NUM*01)
        .EMPTY
        .OUT(SORT*01," = @NEW_NODE;")
        .OUT("CALL @RESERVE(")
        ( .TEST(TREE_STRING_ARITH_SWITCH=2)
          .OUT(STR*01)
        | .EMPTY
          .OUT(NUM*01) )
        .OUT(",@SON(",SORT*01,"));")
        .OUT("@SON(",SORT*01,") = -@SON(",SORT*01,");")
        .OUT("@SORT_LINK_ADDR = ADDR(",SORT*01,");")
        .SET(REAL_DUMMY_SWITCH = 2)
        ( .TEST(TRACE_FLAG = 1)
          .OUT("IF @TRACE > 1 THEN CALL @NODE_TRACE(",SORT*01,");")
        | .EMPTY )  ;

SEMI_COLON  :=
        .EMPTY
        .DO("@STMT = @STMT + 1;")
        ";"  .ERR("",.SCANBY(";"))  ;

SET_LABEL_REPLACE_FLAG  :=
        .EMPTY
        .OUT("@LABEL_REPLACE_FLAG = ")
        ( ( .TEST(LABEL_SUBSCRIPT_SWITCH = 1)
          | .TEST(REAL_DUMMY_SWITCH=2) )
          .OUT("''F'';")
        | .EMPTY
          .OUT("''T'';") )  ;

SET_UP_SORT_SON  :=
        .EMPTY
        .OUT(SORT*01," = @SON(",SORT*01,");")
        .OUT("IF ",SORT*01," < 0  THEN ",SORT*01," = 0;")
        SORT_LINK_SON  ;

SORT_LINK_BROTHER  :=
        .TEST(SET_SORT_LINK_FLAG = 1)
        .OUT("@SORT_LINK_ADDR = ADDR(@BROTHER(@SORT_LINK));")
      | .EMPTY  ;

SORT_LINK_SON  :=
        .TEST(SET_SORT_LINK_FLAG = 1)
        .OUT("@SORT_LINK_ADDR = ADDR(@SON(@SORT_LINK));")
      | .EMPTY  ;

TAKE_STATISTICS  :=
        .TEST(TAKE_STATISTICS_FLAG = 1)
        .OUT("@COUNT(")
        .DO("CALL @OUT(STATISTICS_SWITCH);")
        .OUT(") = @COUNT(")
        .DO("CALL @OUT(STATISTICS_SWITCH);")
        .OUT(") + 1;")
      | .EMPTY  ;

TRACE_OUTPUT  :=
        .TEST(TRACE_ALREADY_OUTPUT_FLAG = 1)
      | ( .TEST(TRACE_FLAG = 0)
          .OUT("                              ")
        | .EMPTY
          .OUT("CALL @TRACE_STATEMENT(")
          .DO("CALL @OUT(@STMT);")
          .OUT(") ") )
        .OUT("                  /*")
        .DO("CALL @OUT(@STMT);")
        .OUT(" */;")  ;

      .END
```


Fig. I.4-2 (concl)

cation, and briefly discusses the reasons for those which do not represent additions to the capabilities of the language.

Additions

- Option list
  - Node, label-value storage specification
  - Statistics
  - Trace
  - Notes option
- External procedures
- Pointer variables
- DO FOR ALL SUBNODES statement
- DEFINE statement
- ADVANCE statement
- Fixed, floating point distinction
- Expanded incremental DO statement
- Keyword pointers useable outside their loops
- ALL: in PRUNE statement
- String literal output

Deletions

Indirect subroutine call -

As our problem-solving techniques evolved and the patterns of use of the language become clearer, it became evident that this capability is not required.

String expression as CALL argument -

Allowing the user to call a subroutine using a character string as an argument generates logical type conversion problems which were not forseen. While the problems can be solved, at considerable expense in translator sophistication, if the language is restricted to internal procedures, our expansion to include external procedures renders the solution invalid.

ALL label keyword -

This provides no additional functional capability and was deleted. Its inclusion in the functional specification was somewhat vestigial, since the language feature with which its beneficial use was associated had already been deleted from the language design for logical reasons.

Modifications

Negated boolean expression must be parenthesized -

This is necessary to avoid syntactically ambiguous expressions.

Equality relations may be string or arithmetic, as determined at <u>execution</u> time, and need not be differentiated by the programmer.

It proved to be implementation-feasible to provide this more powerful capability, which eliminates the need for separate, programmer-specified relations for the two data types.

FIRST, FIRST:, and ALL: changed from label to subscript keywords.

This was done to increase syntactic consistency and has no effect on the functional capabilities of the language.

2.0 DETAILED DESIGN SPECIFICATION FOR THE PLANS MODULE LIBRARY

This section extends the functional design specification for the PLANS Module Library Volume III of the Phase I Final Report. The extensions include details on how each module is to be implemented and thus constitute a detailed design specification. To provide a stand alone capability in Section 2, the four digit subsections previously published as functional specifications are repeated in this report followed by the additional subsections containing details of the implementation design. For reference, Table 2-1 indicates those subsections of this document that appeared as functional specifications in Volume III of the Phase I Final Report.

Section 2.1 provides a brief summary of the PLANS Module Library Contents and characterizes the types of functions that the modules perform. Section 2.2 discusses a general operations model which is a descriptive framework for defining scheduling and resource allocation problems. The use of the operations model conventions provides the greatest direct applicability of the Library Modules. Since the input and output of all library modules are compatible with the operations model, a cursory familiarity with it is necessary in order to understand the functional and detailed design of the individual modules. Section 2.3 discusses the format of presentation of the design details contained in the subsequent subsections.

2.1 Description of the Module Library Contents

The purpose of PLANS Module Library is to provide precoded logic that is common or frequently used in the programming of scheduling

TABLE 2-1 RELATIONSHIPS OF DETAILED DESIGN SPECIFICATIONS TO PREVIOUSLY PUBLISHED FUNCTIONAL SPECIFICATIONS

LIBRARY MODULES		SECTIONS OF VOL. III OF PHASE I FINAL REPORT CONTAINING FUNCTIONAL SPECIFICATIONS	SECTIONS OF THIS REPORT CONTAINING DETAILED DESIGN SPECIFICATIONS
2.4.1	DURATION	2.4.1.1 - 2.4.1.6	2.4.1.7 - 2.4.1.10
2.4.2	ENVELOPE	2.4.2.1 - 2.4.2.6	2.4.2.7 - 2.4.2.10
2.4.3	INTERVAL_UNION	2.4.3.1 - 2.4.3.6	2.4.3.7 - 2.4.3.10
2.4.4	INTERVAL_INTERSECT	2.4.4.1 - 2.4.4.6	2.4.4.7 - 2.4.4.10
2.4.5	FIND_MAX	2.4.5.1 - 2.4.5.5	2.4.5.6 - 2.4.5.9
2.4.6	FIND_MIN	2.4.6.1 - 2.4.6.5	2.4 6.6 - 2.4.6.9
2.4.7	CHECK_FOR_PROCESS_ DEFINITION	2.4.7.1 - 2.4.7.5	2.4.7.6 - 2.4.7.9
2.4.8	GENERATE_JOBSET	2.4.8.1 - 2.4.8.6	2.4.8.7 - 2.4.8.11
2.4.9	EXTERNAL_TEMP_RELATIONS	2.4.9.1 - 2.4.9.5	2.4.9.6 - 2.4.9.9
2.4.10	INTERNAL_TEMP_RELATIONS	2.4.10.1 - 2.4.10.5	2.4.10.6 - 2.4.10.9
2.4.11	ELEMENTARY_TEMP_RELATION	2.4.11.1 - 2.4.11.5	2.4.11.6 - 2.4.11.9
2.4.12	NEXTSET	2.4.12.1 - 2.4.12.6	2.4.12.7 - 2.4.12.9
2.4.13	RESOURCE_PROFILE	2.4.13.1 - 2.4.13.4	2.4.13.5 - 2.4.13.9
2.4.14	POOLED_DESCRIPTOR_ COMPATIBILITY	2.4.14.1 - 2.4.14.5	2.4.14.6 - 2.4.14.9
2.4.15	DESCRIPTOR_PROFILE	2.4.15.1 - 2.4.15.5	2.4.15.6 - 2.4.15.9
2.4.16	UPDATE_RESOURCE	2.4.16.1 - 2.4.16.6	2.4.16.7 - 2.4.16.9
2.4.17	WRITE_ASSIGNMENT	2.4.17.1 - 2.4.17.7	2.4.17.8 - 2.4.17.10
2.4.18	UNSCHEDULE	2.4.18.1 - 2.4.18.6	2.4.18.7 - 2.4.18.9
2.4.19	COMPATIBILITY_SET_ GENERATOR	2.4.19.1 - 2.4.19.6	
2.4.20	FEASIBILITY_PARTITION_ GENERATOR	2.4.20.1 - 2.4.20.7	
2.4.21	PROJECT_DECOMPOSER	2.4.21.1 - 2.4.21.8	2.4.21.9 - 2.4.21.12
2.4.22	REDUNDANT_PREDECESSOR_ CHECKER	2.4.22.1 - 2.4.22.8	2.4.22.9 - 2.4.22.12
2.4.23	CRITICAL_PATH_CALCULATOR	2.4.23.1 - 2.4.23.9	2.4.23.10 - 2.4.23.13

TABLE 2-1 (CONTINUED)

LIBRARY MODULES		SECTION OF VOL. III OF PHASE I FINAL REPORT CONTAINING FUNCTIONAL SPECIFICATIONS	SECTIONS OF THIS REPORT CONTAINING DETAILED DESIGN SPECIFICATIONS
2.4.24	PREDECESSOR_SET_ INVERTER	2.4.24.1 - 2.4.24.7	2.4.24.8 - 2.4.24.11
2.4.25	NETWORK_CONDENSER	2.4.25.1 - 2.4.25.8	2.4.25.9 - 2.4.25.12
2.4.26	CONDENSED_NETWORK_MERGER	2.4.26.1 - 2.4.26.8	2.4.26.9 - 2.4.26.12
2.4.27	NETWORK_ASSEMBLER	2.4.27.1 - 2.4.27.7	2.4.27.8 - 2.4.27.11
2.4.28	CRITICAL_PATH_PROCESSOR	2.4.28.1 - 2.4.28.7	2.4.28.8 - 2.4.28.12
2.4.29	NETWORK_EDITOR	2.4.29.1 - 2.4.29.8	2.4.29.9 - 2.4.29.12
2.4.30	CHECK_DESCRIPTOR_ COMPATIBILITY	2.4.30.1 - 2.4.30.6	2.4.30.7 - 2.4.30.10
2.4.31	ORDER_BY_PREDECESSORS	2.4.31.1 - 2.4.31.8	2.4.31.9 - 2.4.32.12
2.4.32	RESOURCE_ALLOCATOR	2.4.32.1 - 2.4.32.8	
2.4.33	RESOURCE_LEVELER	2.4.33.1 - 2.4.33.9	
2.4.34	HEURISTIC_SCHEDULING_ PROCESSOR	2.4.34.1 - 2.4.34.7	2.4.34.8 - 2.4.34.12
2.4.35	GUB_LP	2.4.35.1 - 2.4.35.9	
2.4.36	MIXED_INTEGER_PROGRAM	2.4.36.1 - 2.4.36.9	
2.4.37	PRIMAL_SIMPLEX	2.4.37.1 - 2.4.37.9	
2.4.38	DUAL_SIMPLEX	2.4.38.1 - 2.4.38.9	
2.4.39	INTEGER_PROGRAM	2.4.39.1 - 2.4.39.8	
2.4.40	REQUIREMENT_GROUP GENERATOR	2.4.40.1 - 2.4.40.6	2.4.40.7

and resource allocation software. A detailed description of how the functions performed by various modules were identified can be found in the Appendix of the Phase I Final Report previously published. The Library contains programs which perform the functions listed below:

PREPROCESSORS

PRELIMINARY PROCESSORS

ELEMENTARY FUNCTIONS

PERFORMANCE OR CONSTRAINT STATUS

DATA UPDATING

ALGORITHMS

In the Phase I Study, 39 modules were functionally specified. Those specified do not exhaust the possibilities for logic that could be preprogrammed. However, since Phase I was completed, four typical scheduling programs taken from NASA applications have been implemented and 86% of the code used was from the specified Module Library. This provides some degree of confidence that the functions provided by the modules are common to scheduling problems.

Table 2.1-1 provides a concise description of the function performed by each of the specified modules. Functional and detailed descriptions constitute the subsections of Section 2.4 in this document. Some examples of the use of modules can be found in Volume II of the Phase I Final Report previously published.

2.2 Description of the Operations Model and the Standard Data Structures

The Operations Model is a single set of descriptive conventions within which all varieties of scheduling and resource allocation

TABLE 2.1-1

CONCISE DESCRIPTION OF THE PLANS LIBRARY MODULES

MODULE CLASS	MODULE NAME	DESCRIPTION
PREPROCESSORS	CHECK_FOR_PROCESS_DEFINITION	Checks for input data consistency.
	NETWORK_EDITOR	Identifies and eliminates both redundant predecessors and cycles specified in the specification of precedence networks.
	REDUNDANT_PREDECESSOR_CHECKER	Identifies and eliminates redundant specifications of predecessors in $JOBSET.
PRELIMINARY PROCESSORS	GENERATE_JOBSET	Creates individual jobs for each occurrence of a process specified explicitly or via an operations sequence in $OBJECTIVES. Merges information contained in $OBJECTIVES, $OPSEQUENCE, and $PROCESS into a tree called $JOBSET.
	PREDECESSOR_SET_INVERTER	Creates, from a set of jobs with predecessors, an equivalent set of jobs with successors.
	ORDER_BY_PREDECESSOR	Produces a list of jobs where all jobs appear in the list only after all their predecessors have appeared.
	CRITICAL_PATH_PROCESSOR	Condenses, merges and computes critical path data for a master network.
	NETWORK_ASSEMBLER	Assembles a master network from subnetworks with interfacing events. The relations between the subnetworks may be more general than those describable by nesting operations sequences.
	CRITICAL_PATH_CALCULATOR	Calculates early and late start and finish times as well as total and free float in a network of jobs.
	CONDENSED_NETWORK_MERGER	Merges two condensed networks into a single composite condensed network, and computes the critical path data for the composite network.

TABLE 2.1- 1 (CONTD.)

MODULE CLASS	MODULE NAME	DESCRIPTION
PRELIMINARY PROCESSORS (Continued)	NETWORK_CONDENSER	Eliminates activities (jobs) from a network leaving only events linked by critical delays as branches.
	PROJECT_DECOMPOSER	Identifies all subprojects within a project description; i.e., finds sub-networks that contain app predecessors and successors of its member activities.
	COMPATIBILITY_SET_GENERATOR	Enumerates all compatible subsets of an input set using externally supplied compatibility criteria.
	FEASIBLE_PARTITION_GENERATOR	Generates all sets of integers with a given number of elements that sum to a given total.
	REQUIREMENT_GROUP_GENERATOR	Generates sets of jobs as a function of resource commonality group.
ELEMENTARY FUNCTIONS	DURATION	Calculates the duration of any standard (simple or multiple) interval.
	ENVELOPE	Calculates an interval that is the smallest cover of a given standard (simple or multiple) interval.
	ELEMENTARY_TEMP_RELATION	Checks satisfaction of a single binary temporal relation given specific assignments for the two jobs named in the temporal relation.
	WRITE_ASSIGNMENT	Writes a single assignment for a resource and adds the assignment node in chronological order in $RESOURCE.
	INTERVAL_UNION	Calculates a standard interval that is the union of two standard intervals, i.e., all points in the output standard interval are in one or both of the input standard intervals.
	INTERVAL_INTERSECTION	Calculates a standard interval that is the intersection of two standard intervals, i.e., all points in the output standard interval are in both the input standard intervals.

TABLE 2.1- 1 (CONTD.)

MODULE CLASS	MODULE NAME	DESCRIPTION
PERFORMANCE OR CONSTRAINT STATUS	EXTERNAL_TEMP_RELATIONS	Identifies temporal constraint violations that would occur if two sets of job assignments were merged.
	INTERNAL_TEMP_RELATIONS	Identifies temporal constraint violations that exist within a set of job assignments. Useful in finding constraint violations after multiple assignments have been made with temporal constraints relaxed.
	RESOURCE_PROFILE	Determines the profile of a resource pool over a given time interval for both 'normal' and 'contingency' levels. Determines the profile of the assigned portion of a pool and gives the jobs to which the resources are assigned.
	POOLED_DESCRIPTOR_COMPATIBILITY	Determines if a single assignment of a job using pooled resources with explicit descriptors is (will be) compatible with existing descriptors for resources required by that job.
	CHECK_DESCRIPTOR_COMPATIBILITY	Determines if a single assignment of a job using item-specific resources with explicit descriptors is (will be) compatible with existing descriptors for resources required by that job. Identifies scheduled jobs that change the incompatible descriptors.
	DESCRIPTOR_PROFILE	Determines the descriptors for an item-specific resource that are valid after a set of jobs involving those resources have been scheduled. Uses the assignment information in $RESOURCE to determine the descriptor set at a particular time.

TABLE 2.1- 1 (CONCL)

MODULE CLASS	MODULE NAME	DESCRIPTION
ALGORITHMS	FIND_MAX	Finds the maximum value in a numeric set and all the elements that have that maximum value.
	FIND_MIN	Finds the minimum value in a numeric set and all the elements that have that minimum value.
	HEURISTIC_SCHEDULING_PROCESSOR	Performs both time-progressive resource allocations/job scheduling and resource leveling.
	RESOURCE_ALLOCATOR	Allocates resources to jobs to satisfy all resource constraints and heuristically produce a minimum duration schedule.
	RESOURCE_LEVELER	Reallocates resources to smooth the usage of resources while maintaining schedule constraints.
	NEXTSET	Determines a set of specific resource items to meet the requirements of a job and permit the earliest possible execution of that job. Determines future times the job requirements can be met with any combination of appropriate resource types.
	INTEGER_PROGRAM	Solves the linear form of the binary decision making problem.
	MIXED_INTEGER_PROGRAM	Solves linear programs that contain both continuous and integer-valued decision variables.
DATA UPDATING	UPDATE_RESOURCE	Records the scheduling of a schedule unit (job) by writing assignments in $RESOURCE for all resources used in the schedule unit.
	UNSCHEDULE	Deletes assignments from $RESOURCE for all resources associated with a specified job to be deleted.

problems can be defined. Adherence to the operations model conventions necessitates a logical partitioning of the information associated with this class of problems according to the categories below.

PROCESSES: Information describing any activity that ultimately must be assigned to an interval on the timeline and which may or may not require the availability of resources for that interval. Examples: 'Train Crew', 'Transport Payload'.

RESOURCES: Information describing any elements in the system that are required for one or more processes to occur. Examples: Crewpersons, Payloads, Trucks.

OPERATIONAL SEQUENCES: Information on the relationships between processes or between the resources associated with different processes. Examples 'Load_Truck' precedes 'Move_Truck'. Truck used in 'Load_Truck' is same Truck as that used in 'Move_Truck'.

A schematic representation of how process, and resource, and relational descriptors can be assembled to constitute various problem models is shown in Figure 2.2-1.

Volume II of the Phase I Final Report should be consulted for a description of each element shown in the Figure.

Based on this simple partitioning of information, blue prints for tree structures that contain this information have been devised. These "standard data structures" if used to describe the problem being solved, permit direct use of the library modules. See Figures

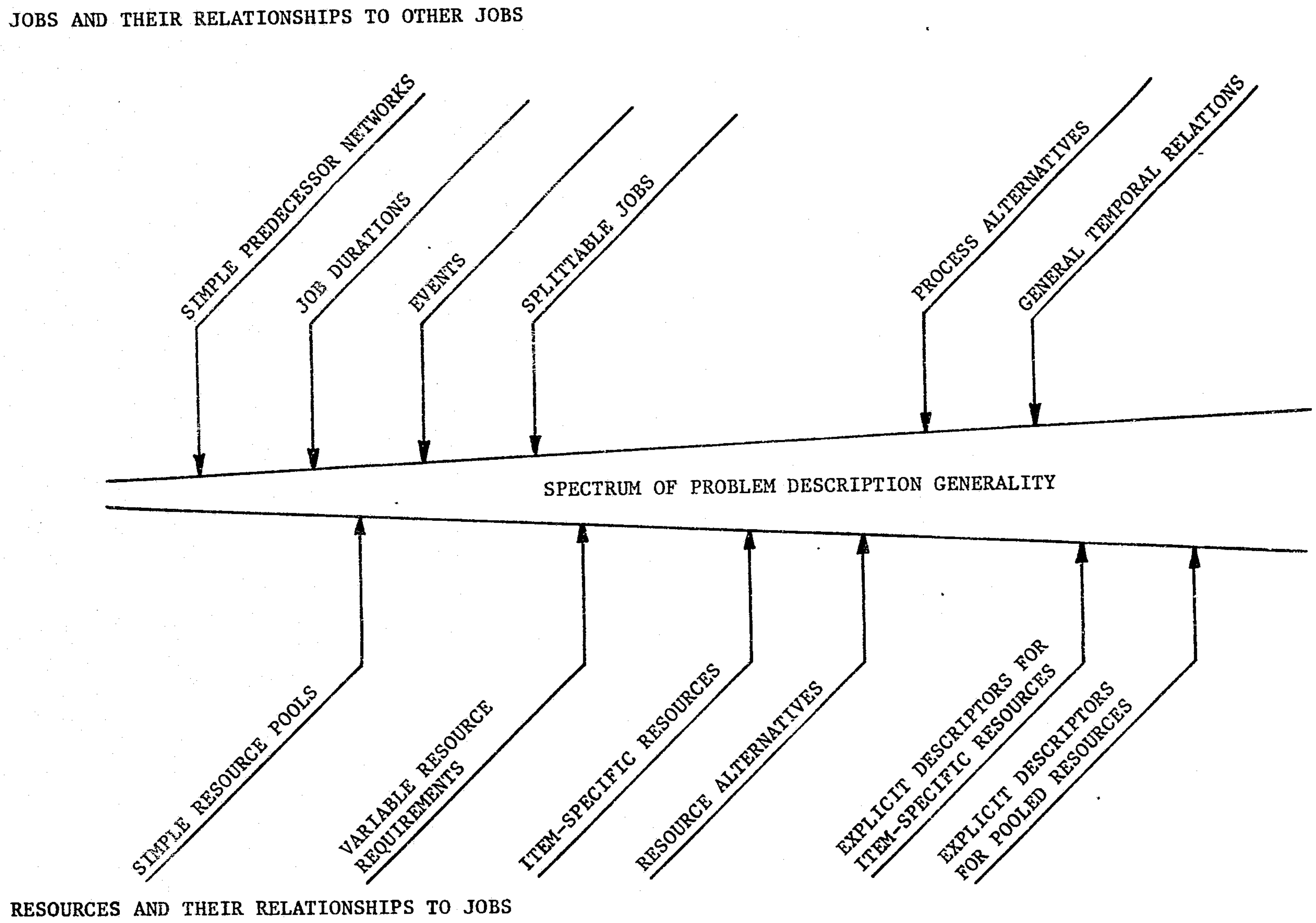
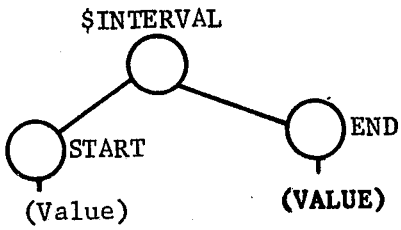


Figure 2.2-1 Problem Description Using the Operations Model

2.2-2 through 2.2-9. The modules, where appropriate, assume that they will get information in tree formats compatible to the standard data structures. It should be clearly understood that the standard data structures need not be used to code in PLANS, and that no module requires all the information shown in any of the standard data structures. However, the information required as input to the library modules is assumed to be arranged with the same relationships that are in the standard data structures. Thus a module requiring an interval could be called with $INTERNAL or with $RESOURCE.ORBITER.ID_006.ASSIGNMENT(1) where $INTERVAL is shown below.



The trees $RESOURCE, $PROCESS, and $OPSEQ contain input information about the sytem to be scheduled. $OBJECTIVES contains information on the particular problem to be solved, i.e. how many processes must be completed to constitute a solution, or other information constraining the solution. $JOBSET is a list of single occurrences of processes; each job in $JOBSET must be assigned time and resources before the problem is solved. $SCHEDULE is a standard format for storing a solution to a scheduling problem. Generally the problem formulator would not build $JOBSET or $SCHEDULE but rather would

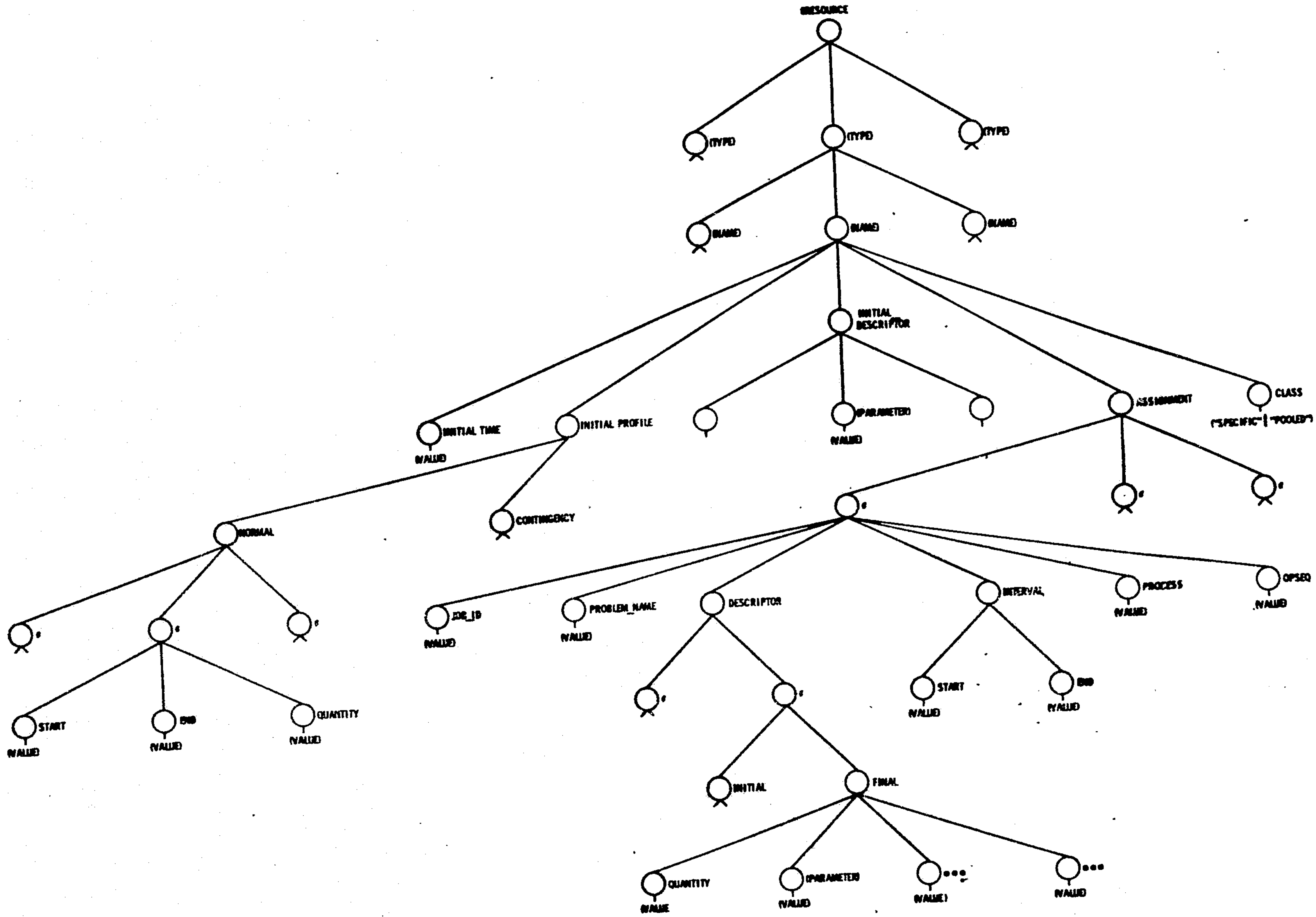


Fig. 2.2-2 $RESOURCE Standard Data Structure

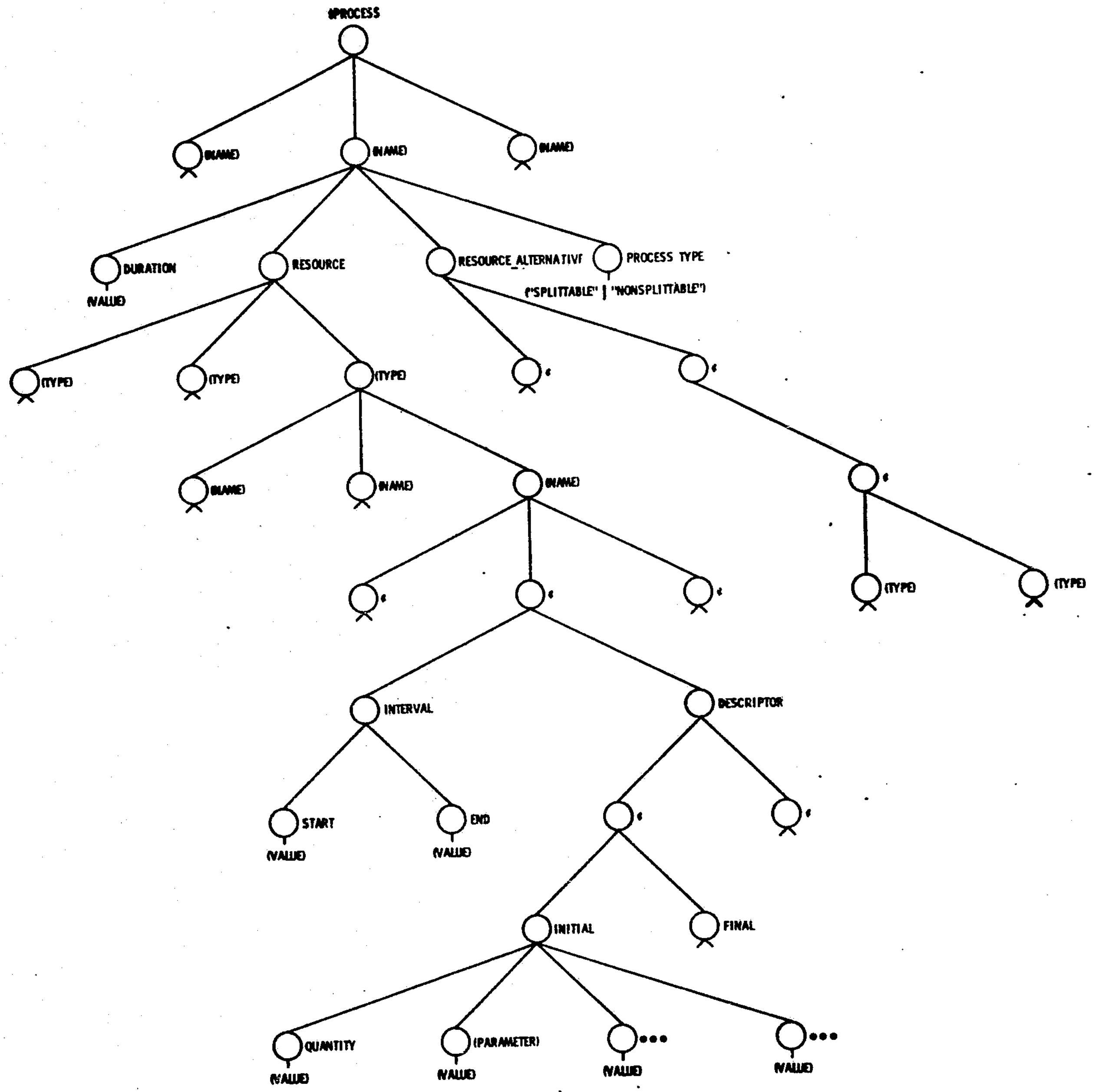


Fig. 2.2-3 $PROCESS Standard Data Structure

ORIGINAL PAGE IS OF POOR QUALITY

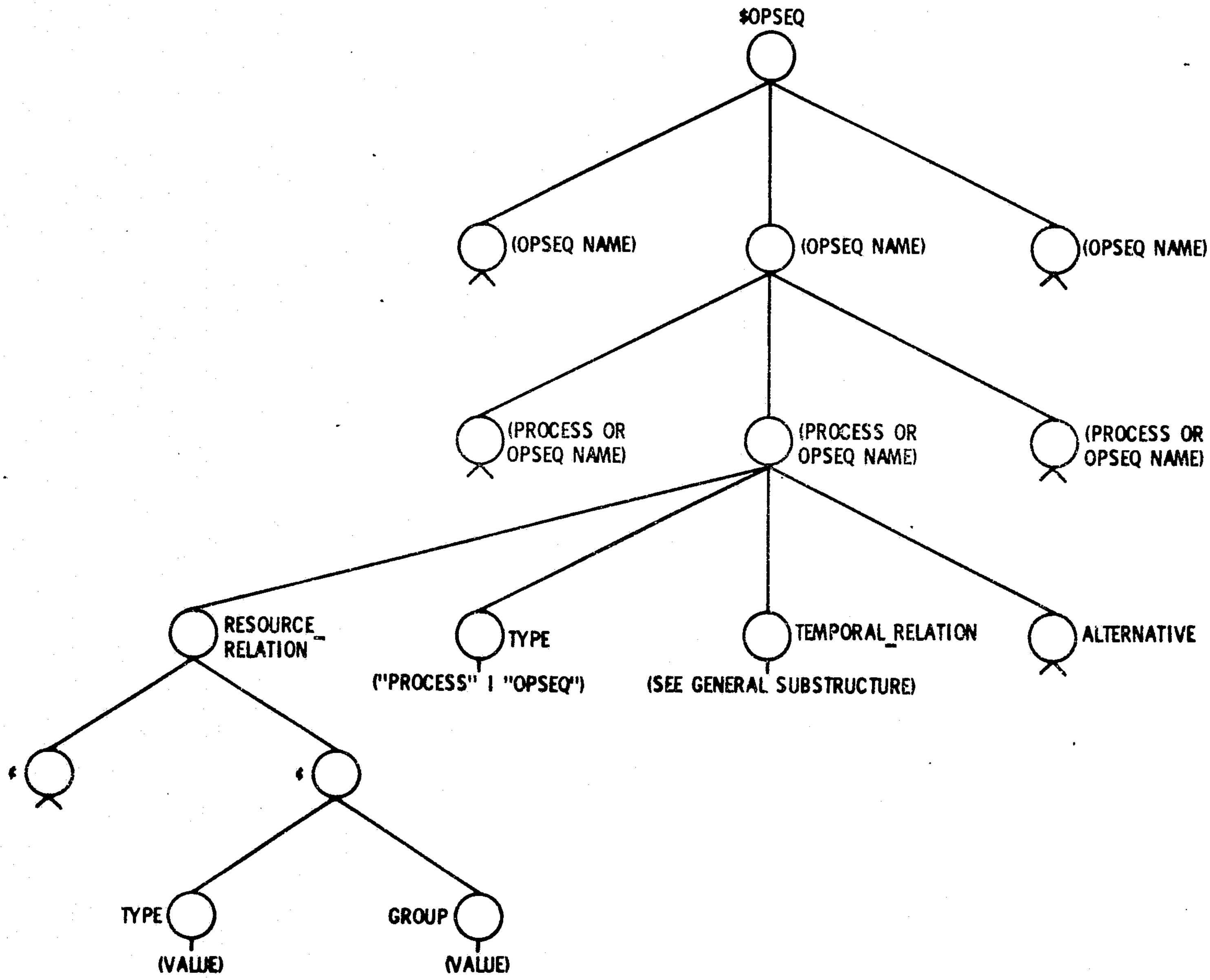


Fig. 2.2-4 $OPSEQ Standard Data Structure

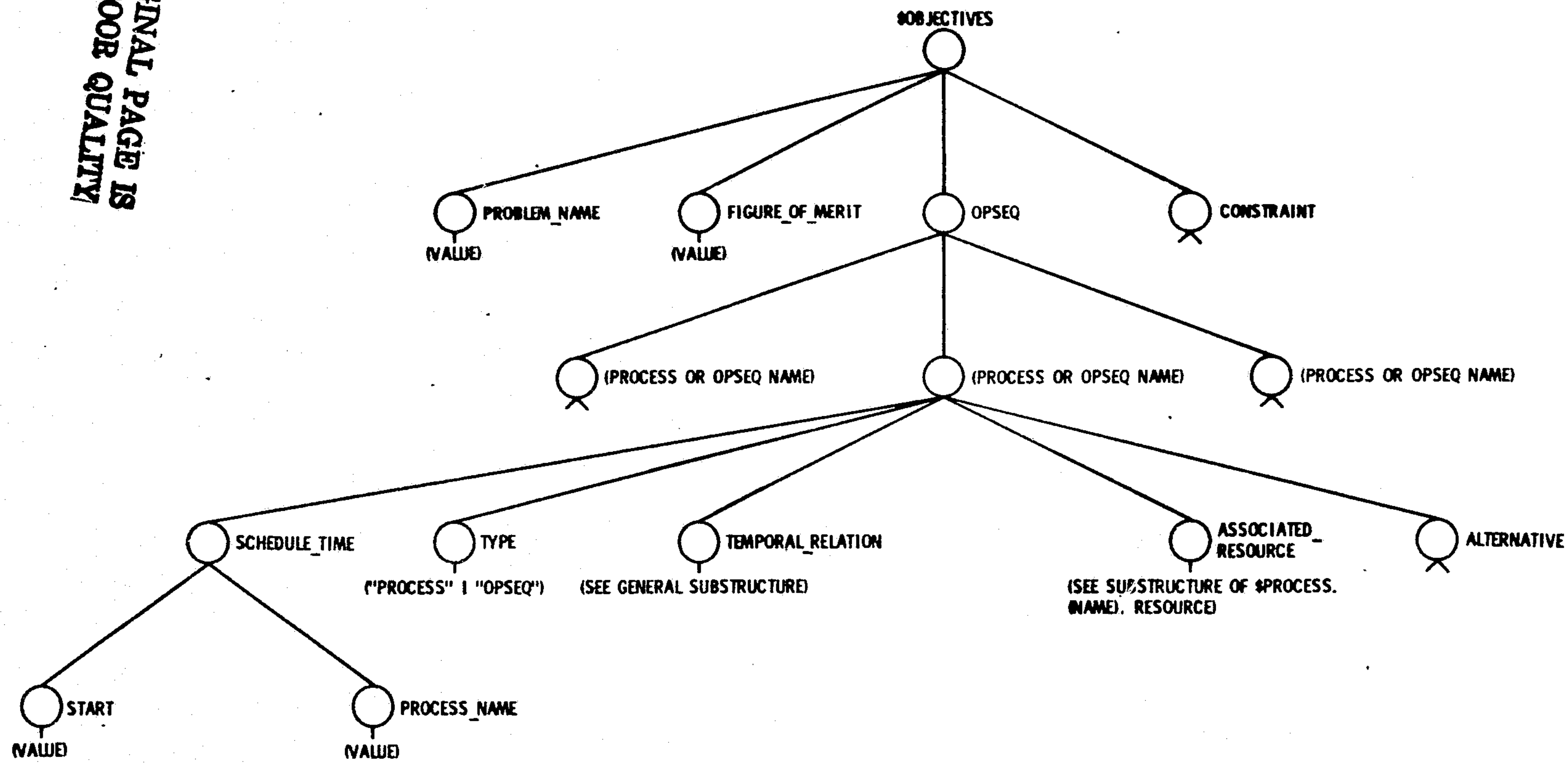


Fig. 2.2-5 $OBJECTIVES Standard Data Structure

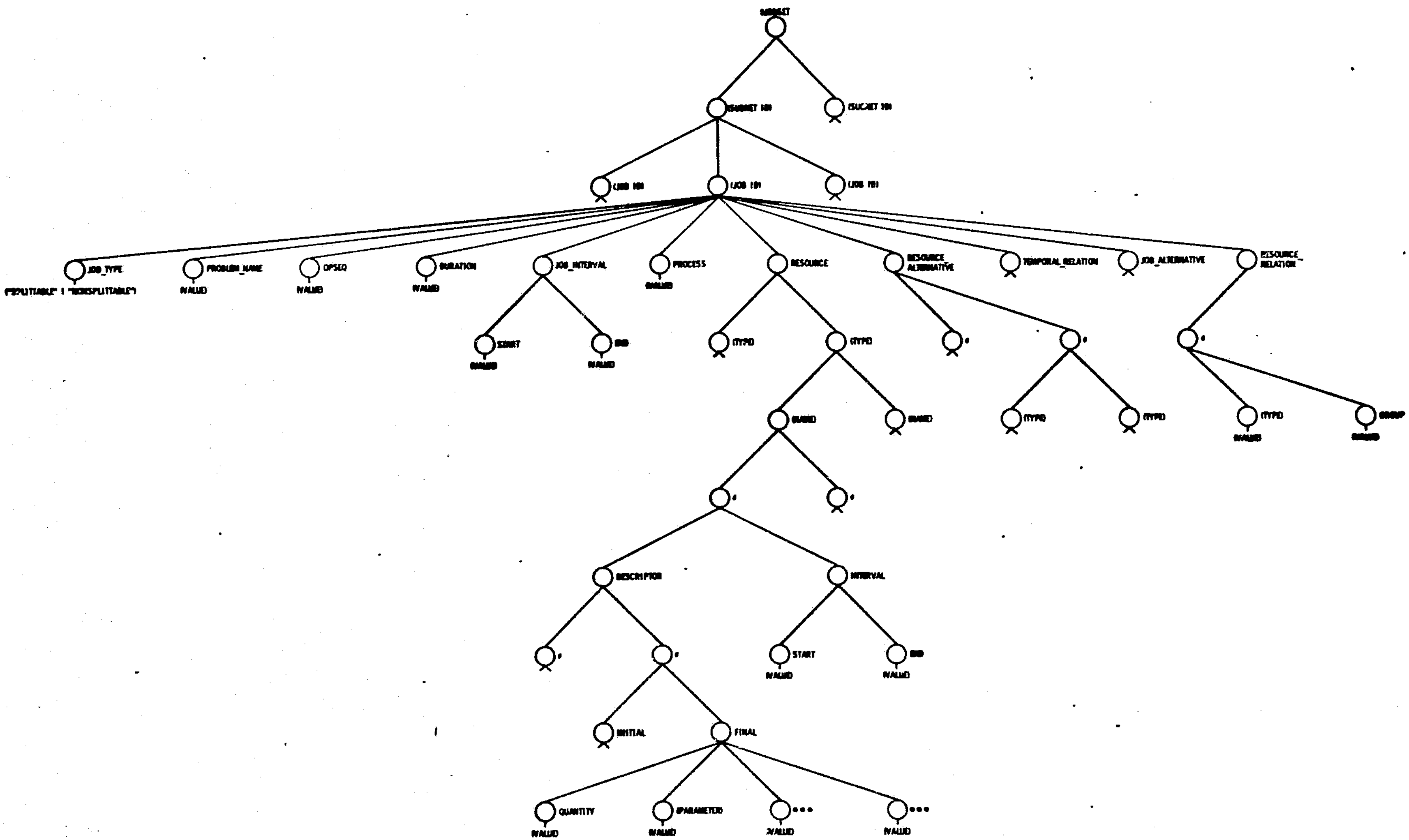


Fig. 2.2-6 $JOBSET Standard Data Structure

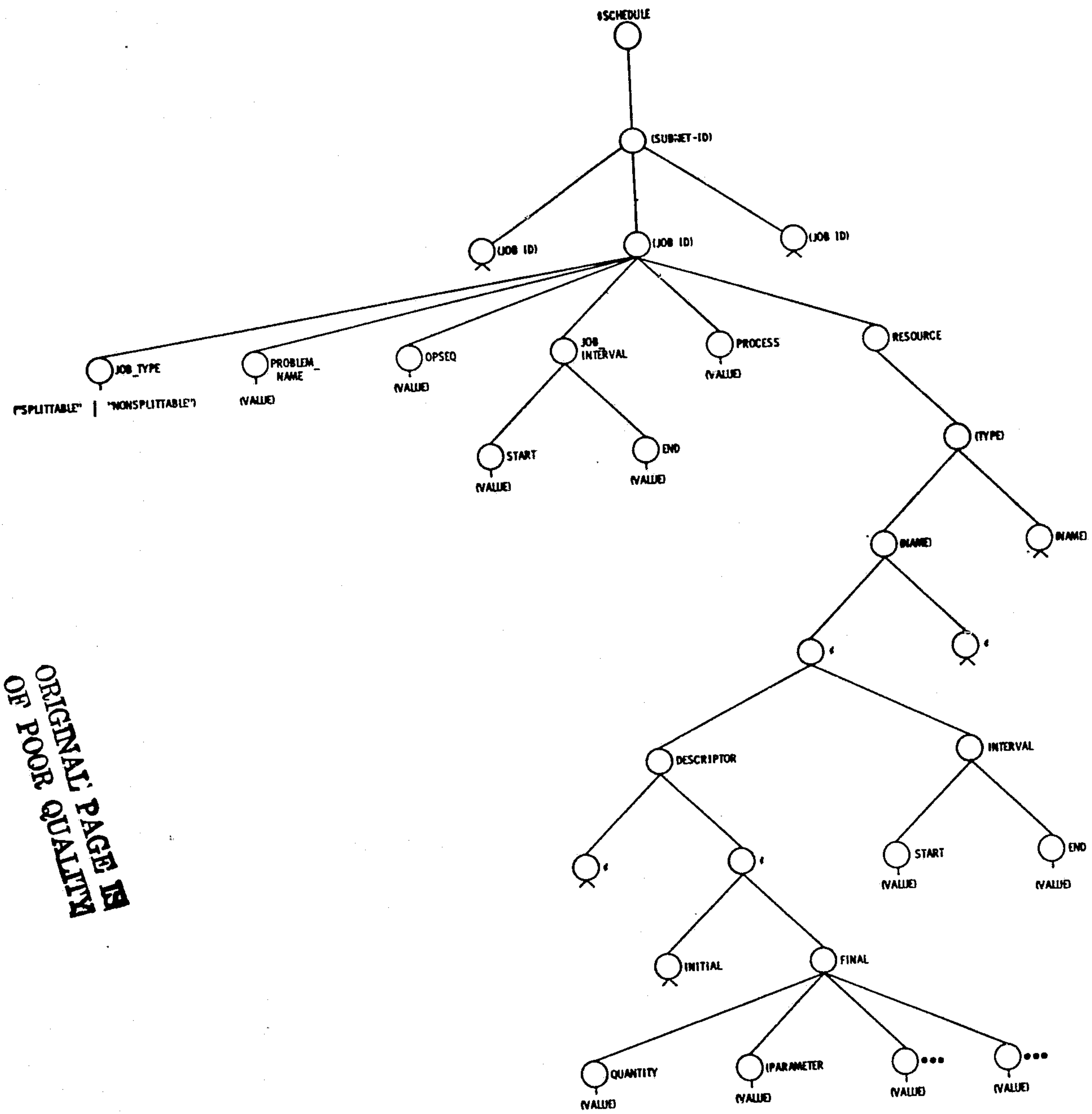


Fig. 2.2-7 $SCHEDULE Standard Data Structure

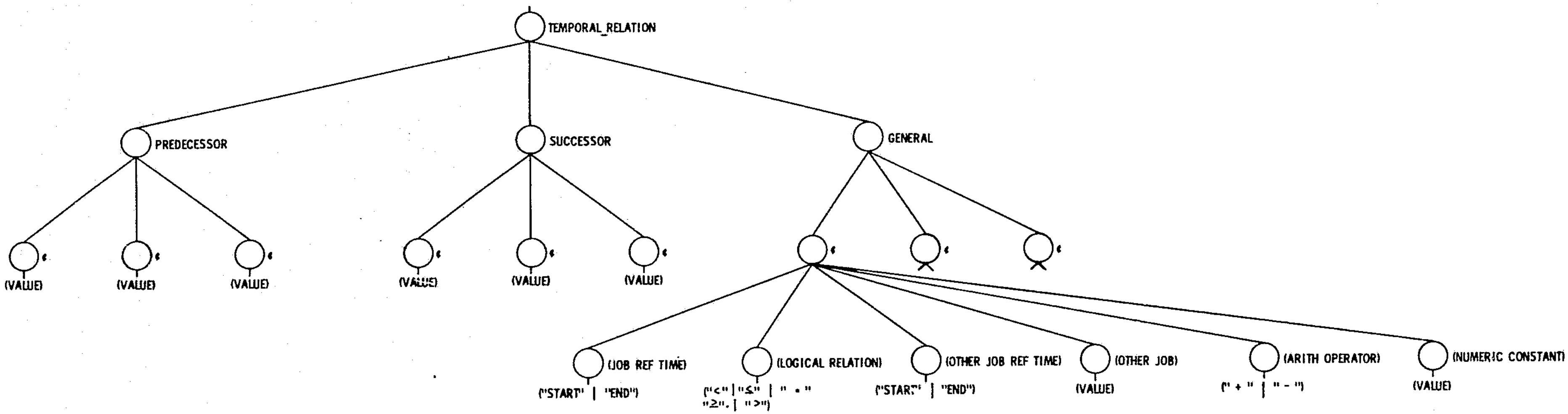
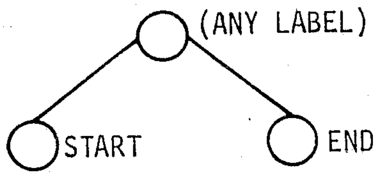


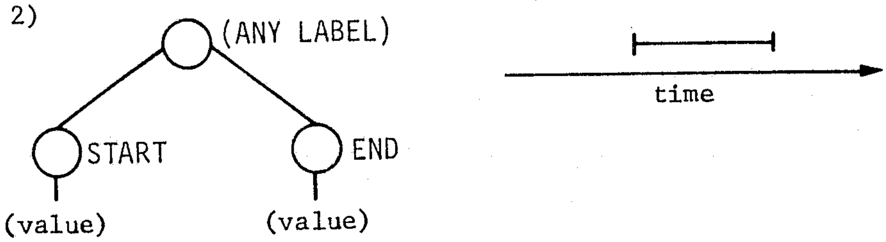
Fig. 2.2-8 TEMPORAL RELATION

STANDARD INTERVAL FORMS

1) 

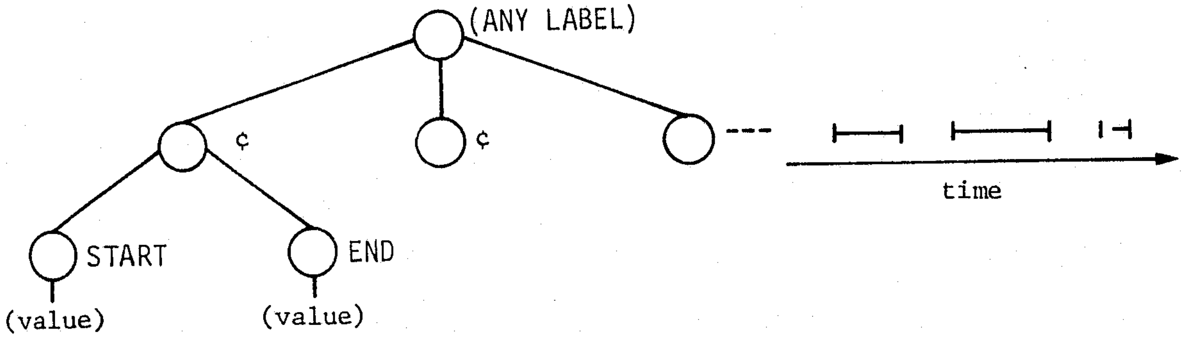
This is a null interval, equivalent to



2) 

This is an ordinary (single) interval. It can be of zero duration if

X.START = X.END

3) 

This is a multiple interval. In addition to the usual constraint X.START $\leq$ X.END, it is also required here that for all I < NUMBER(X), X(I).END $\leq$ X(I+1).START.

Fig. 2.2-9 STANDARD INTERVALS

cause them to be generated from $OBJECTIVE, $OPSEQ, $PROCESS, and $RESOURCE by calling library modules or writing his own PLANS code.

2.3 Format for Detailed Design Specifications

Because PLANS is a high-level language, single statements written in PLANS typically perform the logic contained in a single block of a functional flow chart. In fact, a design goal of PLANS was to virtually eliminate the need for a detailed design step in the software implementation; i.e. to make executable code easy enough to write that functional designers could use the language themselves. Since thirty-four of the modules functionally specified in Phase I are most appropriately implemented in PLANS, the detailed design specification for these modules would typically require little more detail than provided in the functional specification. Yet the intent of this document is to provide greater specificity than offered in previous documentation. To achieve this end, the appropriate format for unambiguous design of the modules seemed to be PLANS code itself. Therefore, a coding of the modules is included in this document. The implementation documentation that will succeed this report may show substantial modifications to the code presented here. Since the intent of this code is to provide a detailed design specification including all capabilities described in the functional specifications, it is not optimized for computational efficiency.

Five of the library modules are appropriate for FORTRAN coding rather than PLANS coding due to their highly algebraic nature. Two of these (INTEGER_PROGRAM, MIXED_INTEGER_PROGRAM) were already implemented and checked out as a feasibility check in Phase I. The documentation of these implemented codes has been provided to NASA under separate documentation. The Phase II effort does not include the prototype implementation of the three remaining mathematical programming modules (PRIMAL_SIMPLEX, DUAL_SIMPLEX, GUB_LP); therefore sections 2.4.35, 2.4.37, 2.4.38 in this document are identical to those in the functional specifications.

Only minor deviations from the functional specifications were deemed desirable in generating the detailed design for the remaining 34 modules. In all cases, these modifications extend the capabilities specified previously or add greater consistency between modules. All functional modifications are noted in a subsection entitled "Modifications to Functional Specifications and/or Standard Data Structures Assumed".

2.4 Library Module Specifications

The following sections present detailed functional and design specifications for the PLANS module library. Each section presents a different module. The numbering system used in the Phase I Final Report has been preserved in this volume.

2.4.1 DURATION

2.4.1 DURATION

2.4.1.1 Purpose and Scope

This module calculates and returns the duration of any standard interval, as identified in the section on Standard Data Structures. If the interval is null, the returned duration is zero. The duration of a multiple interval is defined here as the sum of the durations of its constituent simple intervals.

2.4.1.2 Modules Called

None

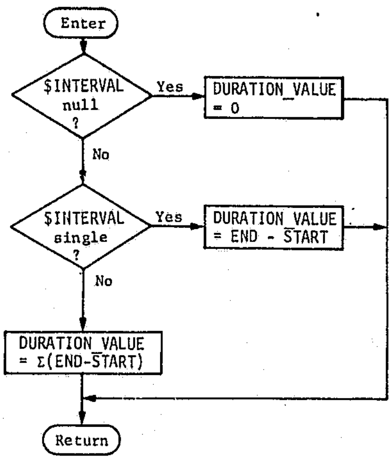
2.4.1.3 Module Input

$INTERVAL is any standard interval. See Fig. 2.4.1-1 for the minimum required data structure in generic form. The minimum required data structure from other Standard Data Structures is illustrated in Fig. 2.4.1-2.

2.4.1.4 Module Output

DURATION_VALUE is an arithmetic variable.

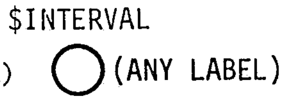
2.4.1.5 Functional Block Diagram



2.4.1.6 Typical Applications

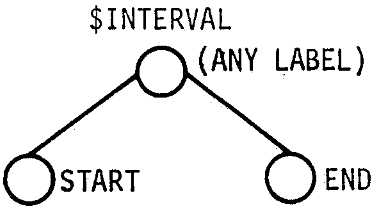
Any applications involving intervals.

STANDARD INTERVAL FORMS

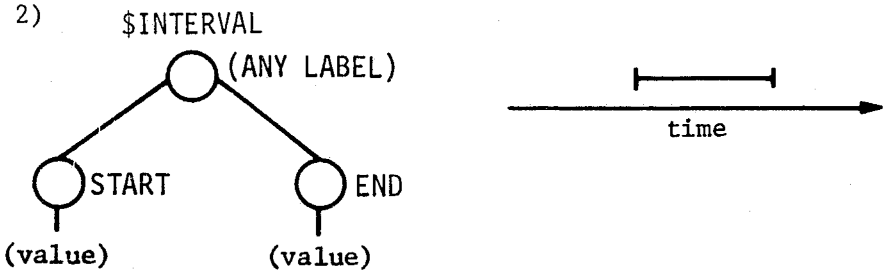


1)

This is a null interval, equivalent to



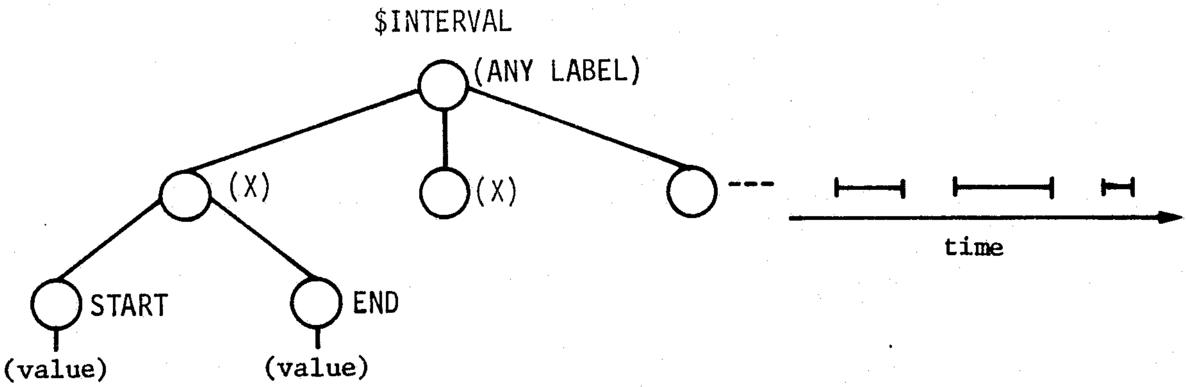
2)



This is an ordinary (single) interval. It can be of zero duration if

X.START = X.END

3)



This is a multiple interval. In addition to the usual constraint X.START $\leq$ X.END, it is also required here that for all I < NUMBER(X), X(I).END $\leq$ X(I+1).START

Fig. 2.4.1-1 Minimum Required Input Data Structures for Module: DURATION

__Note:__ Minimum (relevant) portion of required input standard Data Structures is shown. In all trees, any additional structure will be preserved.

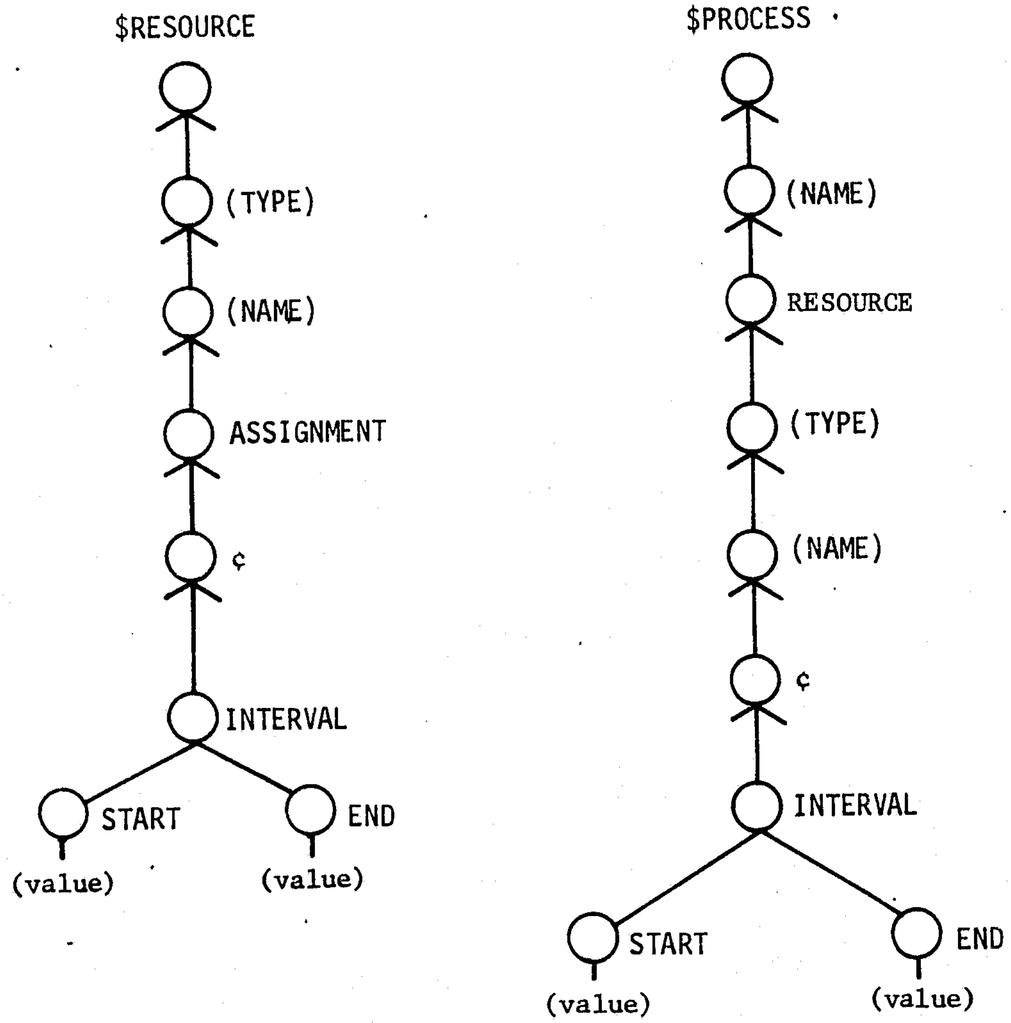


Fig. 2.4.1-2
Minimum Required Input Structures from Standard Data Structures for Module: DURATION

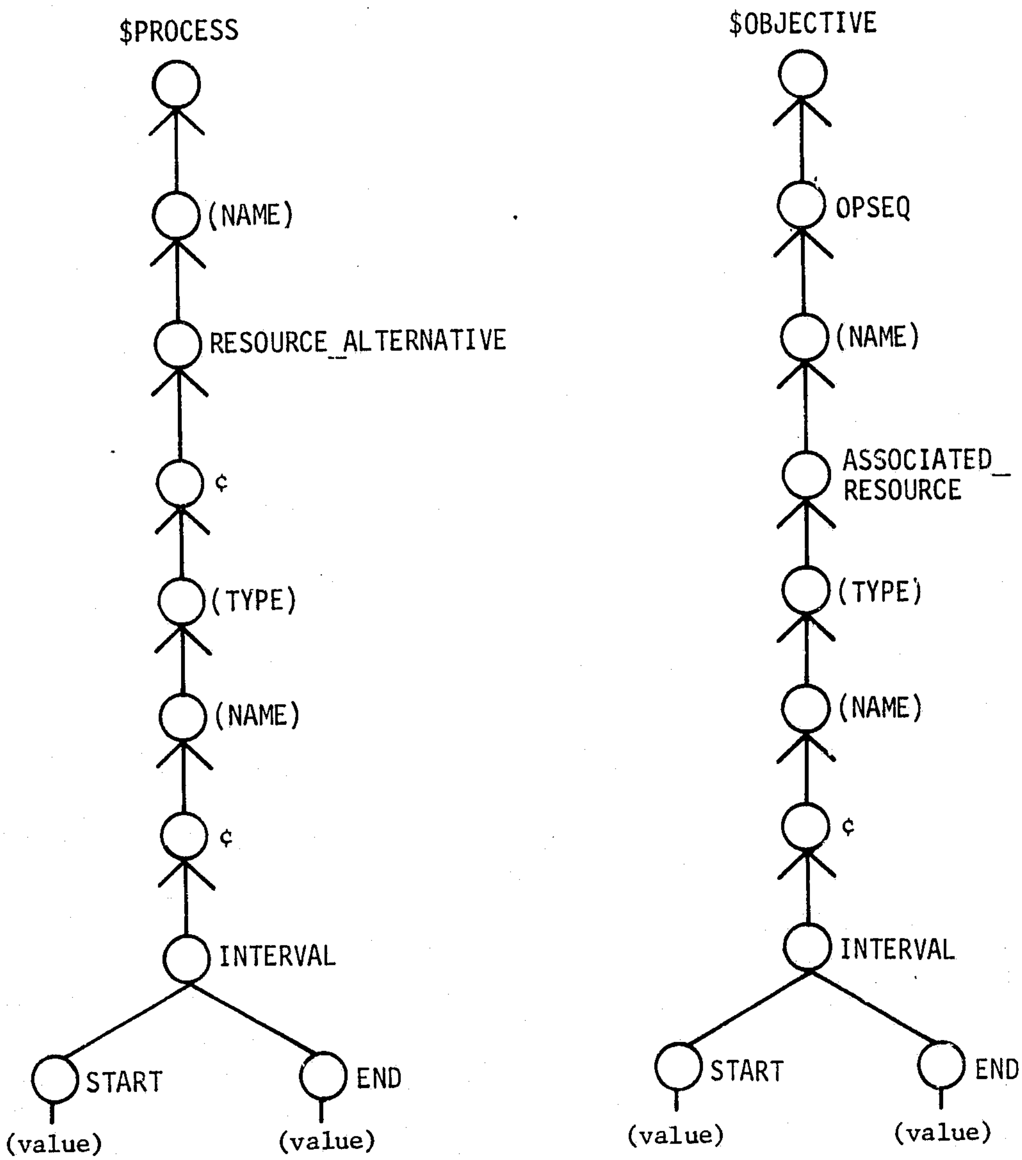


Fig. 2.4.1-2 (cont)

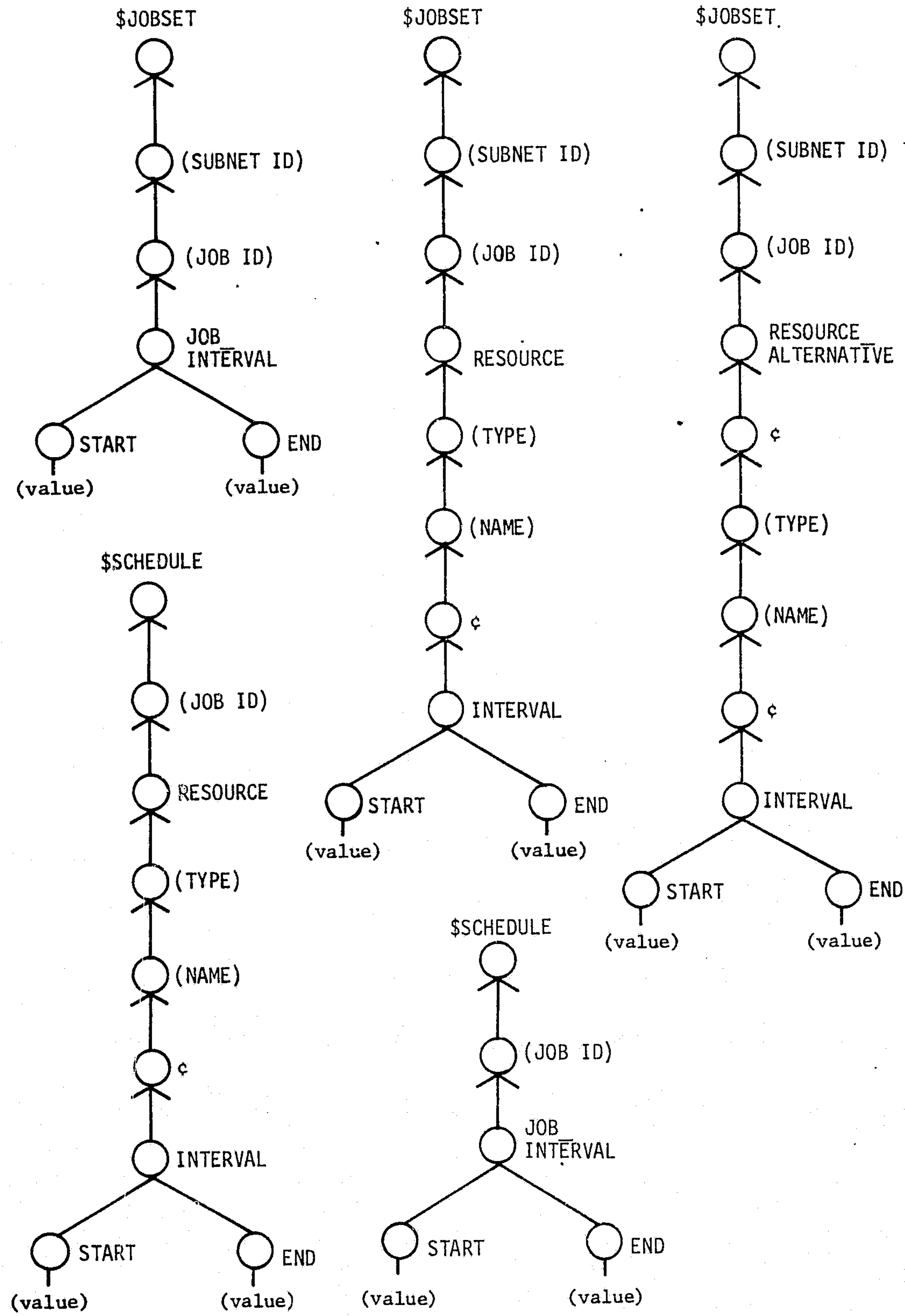


Fig. 2.4.1-2 (concl)

2.4.1.7 DETAILED DESIGN

The label on the first subnode of $INTERVAL is checked to see if it is equal to 'START'. If so, a single interval structure is assumed, otherwise, a multiple-interval structure is assumed.

2.4.1.8 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

DURATION_VALUE - used to return value of the total duration

$INTERVAL - the input data tree that contains the intervals to be summed

2.4.1.9 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES ASSUMED

None

2.4.1.10 COMMENTED CODE

```
DURATION:  PROCEDURE ($INTERVAL, DURATION_VALUE) OPTIONS(EXTERNAL);
/*********************************************************************/
/*                                                                   */
/*  THIS MODULE CALCULATES THE TOTAL DURATION OF ALL THE TIME        */
/*  INTERVALS SPECIFIED IN THE INPUT TREE, $INTERVAL.  THE SUM OF    */
/*  ALL THE INTERVAL DURATIONS IS RETURNED IN 'DURATION_VALUE'.      */
/*                                                                   */
/*********************************************************************/
   DECLARE $SUB_INTERVAL LOCAL ;
   DURATION_VALUE = 0.0 ;
   IF LABEL($INTERVAL(FIRST)) = 'START'
      THEN DURATION_VALUE = $INTERVAL.END - $INTERVAL.START ;
      ELSE DO FOR ALL SUBNODES OF $INTERVAL USING $SUB_INTERVAL ;
              DURATION_VALUE = DURATION_VALUE + $SUB_INTERVAL.END
                                             - $SUB_INTERVAL.START ;
              END ;
END;    /*   DURATION   */
```

2.4.2 ENVELOPE

2.4.2 ENVELOPE

2.4.2.1 Purpose and Scope

This module generates a simple (or null) interval, which is the smallest covering of a given (potentially multiple) interval. If the input interval is null or simple, the envelope interval is identical to the input interval. If the input interval is multiple, the envelope interval ranges from the start of its first constituent interval to the end of the last.

2.4.2.2 Modules Called

None

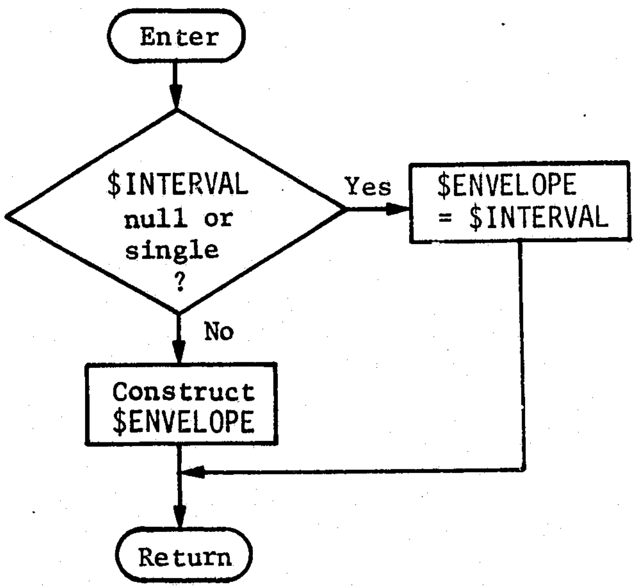
2.4.2.3 Module Input

$INTERVAL is any standard interval. See information under this section for module DURATION for minimum required data structure.

2.4.2.4 Module Output

$ENVELOPE is a simple (or null) interval.

2.4.2.5 Functional Block Diagram



2.4.2.6 Typical Applications

Any applications involving intervals.

2.4.2.7 DETAILED DESIGN

The label on the first subnode of $INTERVAL(1) is checked to see if it is equal to 'START'. If so, multiple intervals are assumed, otherwise, a single interval is assumed.

2.4.2.8 INTERNAL VARIABLE AND TREE NAME DIFINITIONS

$ENVELOPE - the output tree used to return the start and end values of the envelope

$INTERVAL - the input tree that contains the interval(s)

2.4.2.9 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES ASSUMED

None

2.4.2.10 COMMENTED CODE

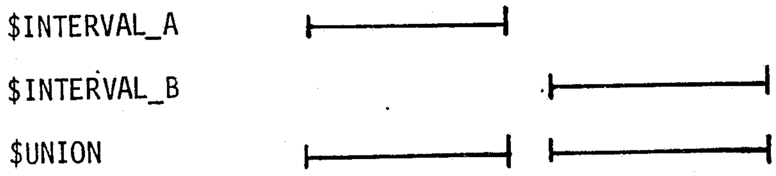
```
ENVELOPE:  PROCEDURE ($INTERVAL, $ENVELOPE) OPTIONS(EXTERNAL);
/*********************************************************************/
/*                                                                   */
/*  THIS MODULE DETERMINES THE START AND END VALUES OF AN "ENVELOPE" */
/*  THAT COVERS THE INTERVALS CONTAINED IN THE INPUT TREE, $INTERVAL.*/
/*  THESE VALUES ARE RETURNED IN THE OUTPUT TREE, $ENVELOPE.         */
/*                                                                   */
/*********************************************************************/
   IF LABEL($INTERVAL(FIRST)(FIRST)) ¬= 'START'
      THEN $ENVELOPE = $INTERVAL ;
      ELSE DO ;  $ENVELOPE = $INTERVAL(FIRST);
                 $ENVELOPE.END = $INTERVAL(LAST).END ;
           END ;
END;        /*    ENVELOPE    */
```

2.4.3 INTERVAL_UNION

2.4.3 INTERVAL_UNION

2.4.3.1 Purpose and Scope

Given two standard intervals, this module constructs a standard interval that represents their union, in the sense of the sketch below.



2.4.3.2 Modules Called

None

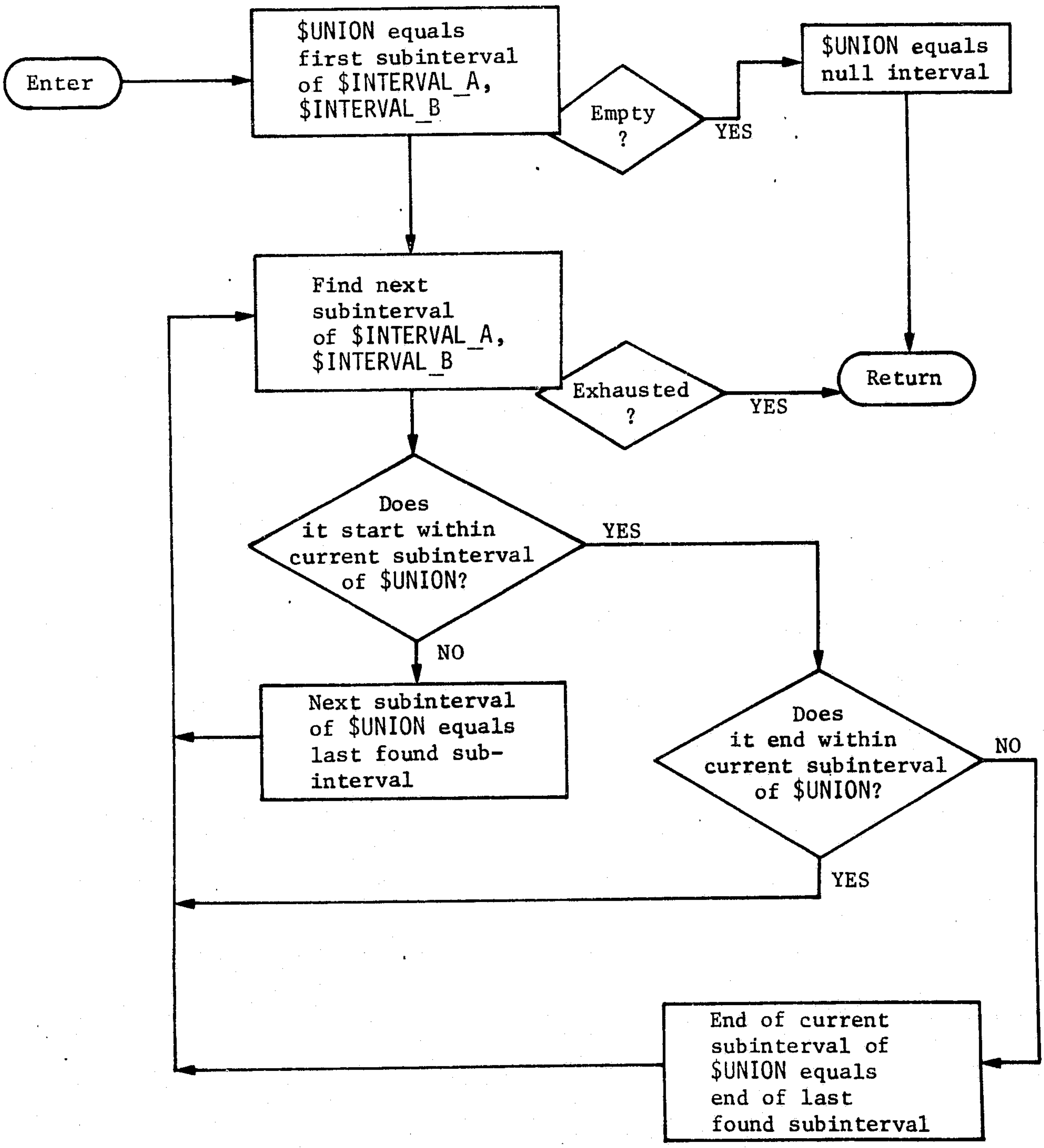
2.4.3.3 Module Input

$INTERVAL_A and $INTERVAL_B are standard intervals.

2.4.3.4 Module Output

$UNION is a standard interval.

2.4.3.5 Functional Block Diagram



2.4.3.6 Typical Applications

Any applications involving intervals.

2.4.3.7 DETAILED DESIGN

The inputs to this module can be of two different types, single or multiple standard interval tree structures. Since there are two inputs, there are four possible combinations of input structures. To avoid having to test for each one of these cases, INTERVAL_UNION transforms both inputs into multiple interval structures. Once this is done the structures can be interpreted by the same block of code without regard to their original form. Of course, before control is returned to the calling program both input structures, $INTERVAL_A and $INTERVAL_B, are transformed back to their original form. This approach necessitates only one general block of code for all cases and makes it appear that the input structures have not been changed.

In order to facilitate the scanning of both input intervals simultaneously, a dummy interval with a start time of INFINITY is placed at the end of each input tree. Since the intervals are selected on the basis of earliest start times, this insures that all intervals will be considered. As new intervals are created in $UNION, they are inserted before the first interval already in $UNION. Before returning, the module reverses the order of these intervals so that they will appear in chronological order.

2.4.3.8 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

I_A	-	used as a pointer into $INTERVAL_A
I_B	-	used as a pointer into $INTERVAL_B
$INTERVAL_A	-	an input standard interval

$INTERVAL_B	-	an input standard interval
$NEXT_INTERVAL	-	the interval currently under consideration
NUMBER_A	-	the total number of intervals in $INTERVAL_A plus one
NUMBER_B	-	the total number of intervals in $INTERVAL_B plus one
$TEMP	-	used as a temporary storage area
$UNION	-	the output standard interval containing the union of the intervals in $INTERVAL_A and $INTERVAL_B

2.4.3.9 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES

None

2.4.3.10 <u>COMMENTED CODE</u>

```
INTERVAL_UNION: PROCEDURE ($INTERVAL_A, $INTERVAL_B, $UNION)
                OPTIONS (EXTERNAL) ;
/***********************************************************************/
/*  INPUT TO THIS MODULE CONSISTS OF TWO STANDARD-INTERVAL TREE        */
/*  STRUCTURES, $INTERVAL_A AND $INTERVAL_B.  THE MODULE BUILDS A      */
/*  SET OF INTERVALS REPRESENTING THE UNION OF THE INPUT INTERVALS     */
/*  AND RETURNS IT IN $UNION.                                          */
/***********************************************************************/
   DECLARE I_AB_SWITCH,
         $POINTER_A,$POINTER_B,$NEXT,$NEXT_INTERVAL,$UNION_LAST LOCAL ;
/*  INITIALIZE VARIABLES AND TREE STRUCTURES.                          */
   PRUNE $UNION ;
   IF LABEL($INTERVAL_A(FIRST)) = 'START'
      THEN DEFINE $POINTER_A AS $INTERVAL_A ;
      ELSE DEFINE $POINTER_A AS $INTERVAL_A(FIRST) ;
   IF LABEL($INTERVAL_B(FIRST)) = 'START'
      THEN DEFINE $POINTER_B AS $INTERVAL_B ;
      ELSE DEFINE $POINTER_B AS $INTERVAL_B(FIRST) ;
   DEFINE $UNION_LAST AS $UNION(FIRST) ;
   $UNION_LAST.END = -INFINITY ;
   DO WHILE ($POINTER_A NOT IDENTICAL TO $NULL  |
             $POINTER_B NOT IDENTICAL TO $NULL) ;
/*  OF THE REMAINING INTERVALS, THE ONE WITH THE EARLIEST START        */
/*  TIME IS CONSIDERED NEXT.                                           */
      IF $POINTER_B IDENTICAL TO $NULL  |  ($POINTER_A
         ¬ IDENTICAL TO $NULL  &  $POINTER_A.START < $POINTER_B.START)
         THEN DO;  DEFINE $NEXT AS $POINTER_A;  I_AB_SWITCH = 1;  END;
         ELSE DO;  DEFINE $NEXT AS $POINTER_B;  I_AB_SWITCH = 2;  END;
      $NEXT_INTERVAL.START = $NEXT.START ;
      $NEXT_INTERVAL.END = $NEXT.END ;
/*  NOW THE INTERVALS IN $UNION CAN BE MODIFIED BASED ON THE START     */
/*  AND END TIMES OF THE INTERVAL UNDER CONSIDERATION.                 */
      IF $NEXT_INTERVAL.START > $UNION_LAST.END
         THEN DO ;  ADVANCE $UNION_LAST ;
                    GRAFT $NEXT_INTERVAL AT $UNION_LAST ;
                    END ;
         ELSE IF $NEXT_INTERVAL.END > $UNION_LAST.END
                 THEN $UNION_LAST.END = $NEXT_INTERVAL.END ;
      IF I_AB_SWITCH = 1
         THEN ADVANCE $POINTER_A ;
         ELSE ADVANCE $POINTER_B ;
      END ;
   PRUNE $UNION(FIRST) ;
END INTERVAL_UNION ;
```

2.4.4 INTERVAL_INTERSECT

.4.4 INTERVAL_INTERSECT

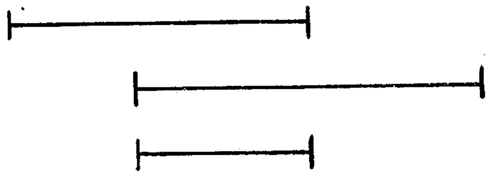
2.4.4.1 Purpose and Scope

Given two standard intervals, this module constructs a standard interval which represents their intersection, in the sense of the sketch below.

$INTERVAL_A

$INTERVAL_B

$INTERSECTION



2.4.4.2 Modules Called

None

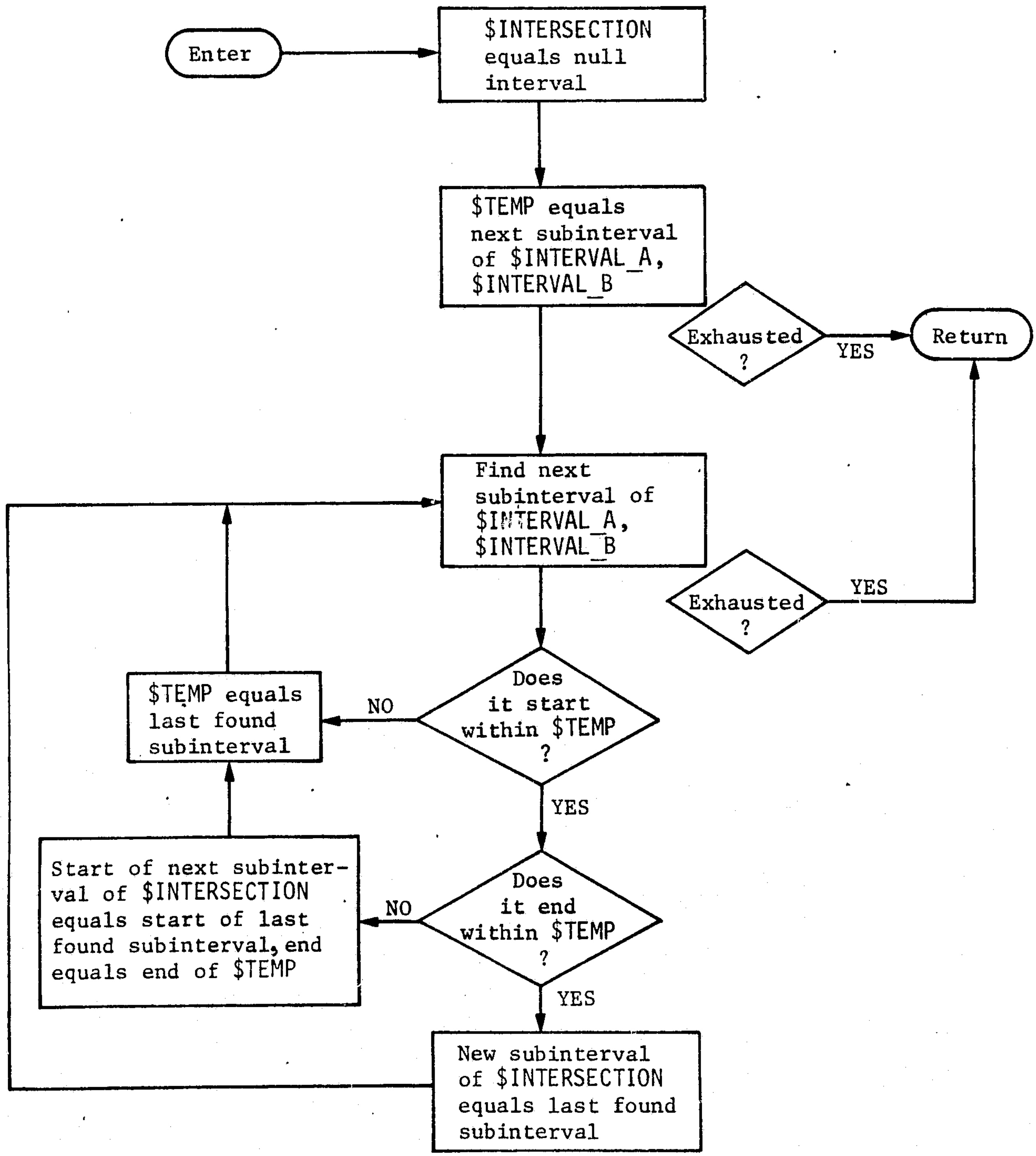
2.4.4.3 Module Input

$INTERVAL_A and $INTERVAL_B are standard intervals.

2.4.4.4 Module Output

$INTERSECTION is a standard interval. A point intersection will be detected by INTERVAL_INTERSECT thus the calling program must check for and prove zero duration intersections if they are not required.

2.4.4.5 Functional Block Diagram



2.4.4.6 Typical Applications

Any applications involving intervals.

2.4.4.7 **DETAILED DESIGN**

The inputs to this module can be of two different types, single or multiple standard interval tree structures. Since there are two inputs, there are four possible combinations of input structures. To avoid having to test for each one of these cases, INTERVAL_INTERSECTION transforms both inputs into multiple interval structures. Once this is done the structures can be interpreted by the same block of code without regard to their original form. Of course, before control is returned to the calling program both input structures, $INTERVAL_A and $INTERVAL_B, are transformed back to their original form. This approach necessitates only one general block of code for all cases and makes it appear that the input structures have not been changed.

In order to facilitate the scanning of both input intervals simultaneously, a dummy interval with a start time of INFINITY is placed at the end of each input tree. Since the intervals are selected on the basis of earliest start times, this insures that all intervals will be considered. As new intervals are created in $INTERSECTION, they are inserted before the first interval already in $INTERSECTION. Before returning, the module reverses the order of these intervals so that they will appear in chronological order.

2.4.4.8 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

I_A	-	used as a pointer into $INTERVAL_A
I_B	-	used as a pointer into $INTERVAL_B
$INTERSECTION	-	the output standard interval containing the intersection of the intervals in $INTERVAL_A and $INTERVAL_B
$INTERVAL_A	-	an input standard interval
$INTERVAL_B	-	an input standard interval
$NEXT_INTERVAL	-	the interval currently under consideration
NUMBER_A	-	the total number of intervals in $INTERVAL_A plus one
NUMBER_B	-	the total number of intervals in $INTERVAL_B plus one
$TEMP	-	used as a temporary storage area

2.4.4.9 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES

None

```
INTERVAL_INTERSECT:
/***********************************************************************/
/*                                                                     */
/*  INPUT TO THIS MODULE CONSISTS OF TWO STANDARD-INTERVAL TREE        */
/*  STRUCTURES, $INTERVAL_A AND $INTERVAL_B.  THE MODULE BUILDS A      */
/*  SET OF INTERVALS REPRESENTING THE INTERSECTION OF THE INPUT        */
/*  INTERVALS AND RETURNS IT IN $INTERSECTION.                         */
/*                                                                     */
/***********************************************************************/
PROCEDURE ($INTERVAL_A, $INTERVAL_B, $INTERSECTION) OPTIONS(EXTERNAL);
   DECLARE I_A,I_B,NUMBER_A,NUMBER_B,$TEMP,$NEXT_INTERVAL LOCAL ;
/*  INITIALIZE VARIABLES AND TREE STRUCTURES.                          */
   PRUNE $INTERSECTION ;
   IF  $INTERVAL_A(FIRST) IDENTICAL TO $NULL    |
      $INTERVAL_B(FIRST) IDENTICAL TO $NULL  THEN RETURN ;
   IF LABEL($INTERVAL_A(FIRST)) ='START'
   THEN DO;  GRAFT $INTERVAL_A AT $TEMP;
             GRAFT $TEMP AT $INTERVAL_A(FIRST);
        END;
   IF LABEL($INTERVAL_B(FIRST)) ='START'
   THEN DO;  GRAFT $INTERVAL_B AT $TEMP;
             GRAFT $TEMP AT $INTERVAL_B(FIRST);
        END;
   $INTERVAL_A(NEXT).START = INFINITY ;
   $INTERVAL_B(NEXT).START = INFINITY ;
   NUMBER_A = NUMBER($INTERVAL_A) ;  I_A = 1 ;
   NUMBER_B = NUMBER($INTERVAL_B) ;  I_B = 1 ;
   $TEMP.END = -INFINITY ;
   DO WHILE (I_A < NUMBER_A  |  I_B < NUMBER_B) ;
/*  OF THE REMAINING INTERVALS, THE ONE WITH THE EARLIEST START        */
/*  TIME IS CONSIDERED NEXT.                                           */
      IF $INTERVAL_A(I_A).START < $INTERVAL_B(I_B).START
         THEN DO ;  $NEXT_INTERVAL = $INTERVAL_A(I_A) ;  I_A = I_A+1 ;
              END ;
         ELSE DO ;  $NEXT_INTERVAL = $INTERVAL_B(I_B) ;  I_B = I_B+1 ;
              END ;
/*  NOW THE INTERVALS IN $INTERSECTION CAN BE MODIFIED BASED ON THE    */
/*  START AND END TIMES OF THE INTERVAL UNDER CONSIDERATION.           */
      IF $NEXT_INTERVAL.START >  $TEMP.END
         THEN GRAFT $NEXT_INTERVAL AT $TEMP ;
         ELSE IF $NEXT_INTERVAL.END < $TEMP.END
                 THEN GRAFT INSERT $NEXT_INTERVAL
                                 BEFORE $INTERSECTION(FIRST) ;
                 ELSE DO ;
                     INSERT $NEXT_INTERVAL BEFORE $INTERSECTION(FIRST);
                      $INTERSECTION(FIRST).END = $TEMP.END ;
                      GRAFT $NEXT_INTERVAL AT $TEMP ;
                      END ;
      END ;
/*  PUT THE INTERVALS OF $INTERSECTION IN CHRONOLOGICAL ORDER.         */
   GRAFT $INTERSECTION AT $TEMP ;
```

```
    DO I=1 TO NUMBER($TEMP) ;
       GRAFT INSERT $TEMP(FIRST) BEFORE $INTERSECTION(FIRST) ;
       END ;
    PRUNE $INTERVAL_A(LAST),$INTERVAL_B(LAST) ;
    IF NUMBER_A = 2
    THEN DO;  GRAFT $INTERVAL_A(FIRST) AT $TEMP;
              GRAFT $TEMP AT $INTERVAL_A;
         END;
    IF NUMBER_B = 2
    THEN DO;  GRAFT $INTERVAL_B(FIRST) AT $TEMP;
              GRAFT $TEMP AT $INTERVAL_B;
         END;
    PRUNE $TEMP ;

END INTERVAL_INTERSECTION ;
```

2.4.5 FIND_MAX

2.4.5 FIND_MAX

2.4.5.1 Purpose and Scope

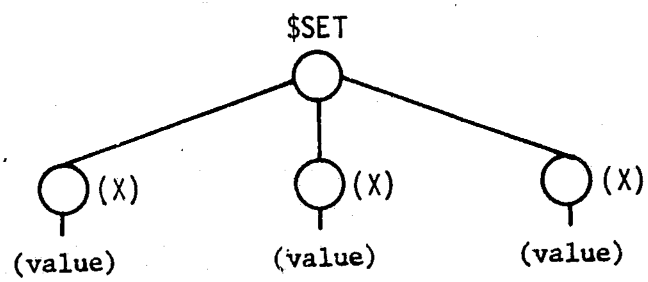
Given a set of numerical values (i.e., a node of a tree for which each of the next lower level subnodes is terminal and has a numerical value), find the maximum of the values and find the indices (i.e., the ordinal positions in the original set) of each of the subnodes for which the value equals the maximum.

2.4.5.2 Modules Called

None

2.4.5.3 Module Input

$SET is a tree of the form shown in the sketch. Minimum required data structure is a tree with at least one subnode at the next lower level.

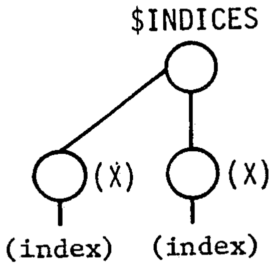


Each value is numeric.

2.4.5.4 Module Output

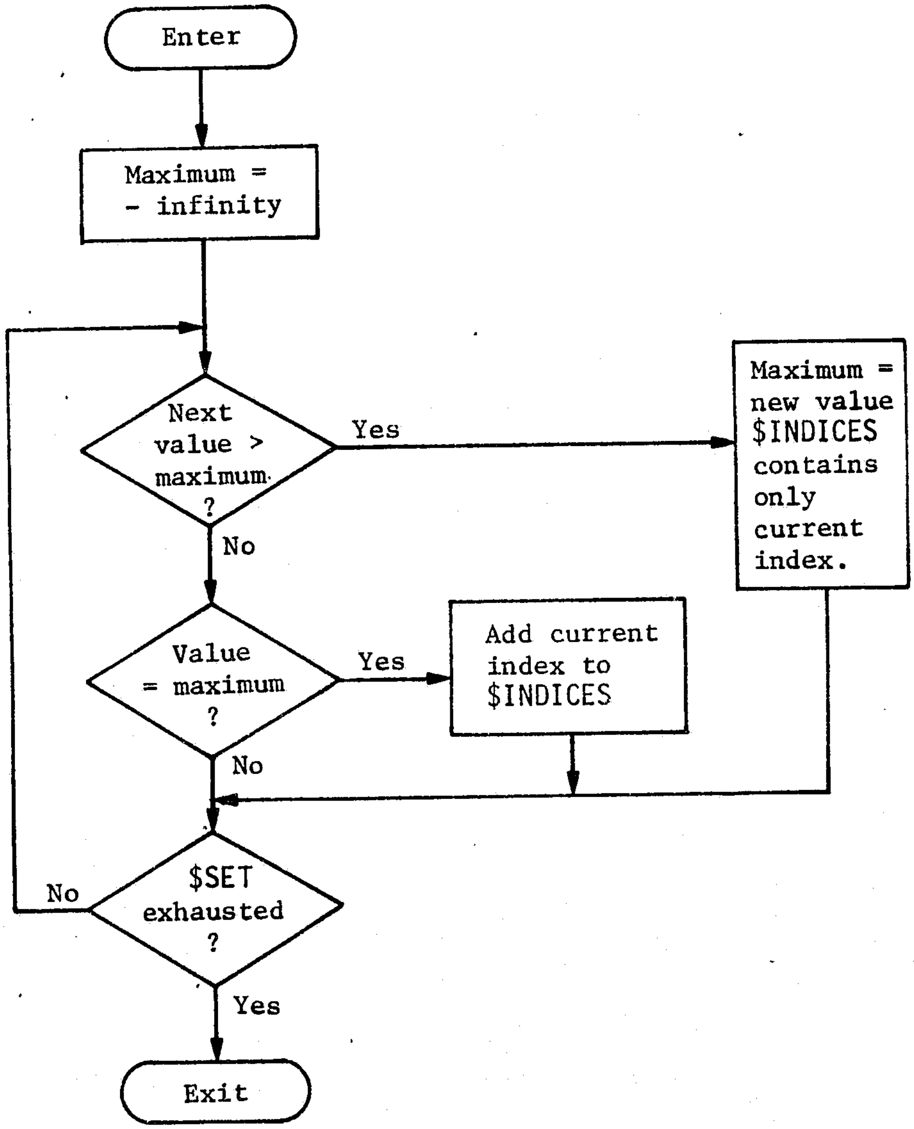
VALUE_MAXIMUM is an arithmetic variable whose value is the maximum of the values of $SET.

$INDICES is a tree of the form



where the indices are the ordinal positions in $SET of all nodes whose value equals VALUE_MAXIMUM.

2.4.5.5 Functional Block Diagram



2.4.5.6 DETAILED DESIGN

This module iteratively searches through the numeric values of the subnodes of $SET. As the subnodes are scanned, the current maximum is maintained by comparing each value with it. $INDICES and the VALUE_MAXIMUM are updated every time a larger value is encountered. If $SET is empty, -INFINITY is returned as the maximum value.

2.4.5.7 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

I	-	is a counter used to obtain subnodes indices
$INDICES	-	used to record the indices of the subnodes whose value equals the maximum
$SET	-	is the set of numeric values
$VALUE	-	is one of the subnodes of $SET
VALUE_MAXIMUM	-	used to return the maximum value to the calling program

2.4.5.8 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES

In addition to returning the appropriate subscripts in $INDICES, this tree is also used to return the node labels. Of course, node references by subscript are more efficient than references by label. Therefore, the user is not encouraged to use these labels to reference the subnodes of $SET since the corresponding indices are also available.

The variable name VALUE_MAXIMUM was substituted for MAXIMUM in order to allow for real as well as integer values.

2.4.5.9 COMMENTED CODE

```
FIND_MAX:       PROCEDURE ($SET, $INDICES, VALUE_MAXIMUM)
                OPTIONS(EXTERNAL);
/*********************************************************************/
/*                                                                   */
/*  THIS PROCEDURE FINDS THE LARGEST VALUE IN THE NUMERIC SET INPUT  */
/*  IN $SET AND RETURNS IT IN 'VALUE_MAXIMUM'.  IT ALSO OUTPUTS A    */
/*  TREE, $INDICES, CONTAINING THE LABELS AND SUBSCRIPTS THAT COR-   */
/*  RESPOND TO THE SUBNODES OF $SET WHOSE VALUES EQUAL THE MAXIMUM.  */
/*  IF $SET IS NULL, -INFINITY IS RETURNED AS THE 'VALUE_MAXIMUM'.   */
/*                                                                   */
/*********************************************************************/
   DECLARE I,$VALUE LOCAL ;  I = 0 ;
   VALUE_MAXIMUM = -INFINITY ;
   DO FOR ALL SUBNODES OF $SET USING $VALUE ;
       I = I+1 ;
       IF $VALUE > VALUE_MAXIMUM
          THEN DO ;  VALUE_MAXIMUM = $VALUE ;
                     PRUNE $INDICES ;
                     $INDICES.#LABEL($VALUE) = I ;
               END ;
          ELSE IF $VALUE = VALUE_MAXIMUM
               THEN DO ;  $INDICES(NEXT) = I ;
                          LABEL($INDICES(LAST)) = LABEL($VALUE) ;
                     END ;
       END ;
END;     /*   FIND_MAX        */
```

2.4.6 FIND_MIN

2.4.6 FIND_MIN

2.4.6.1 Purpose and Scope

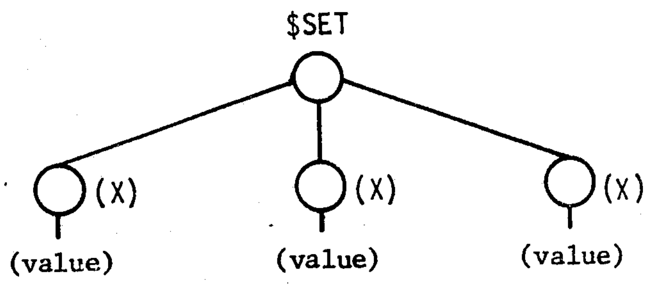
Given a set of numerical values (i.e., a node of a tree for which each of the next lower level subnodes is terminal and has a numerical value), find the minimum of the values and find the indices (i.e., the ordinal positions in the original set) of each of the subnodes for which the value equals the minimum.

2.4.6.2 Modules Called

None

2.4.6.3 Module Input

$SET is a tree of the form shown on the sketch with at least one subnode at the next lower level.

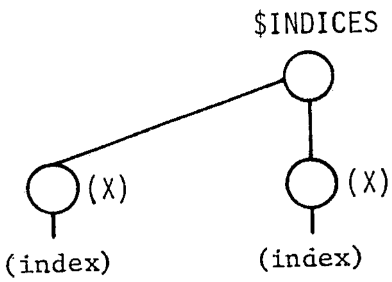


Each value is numeric.

2.4.6.4 Module Output

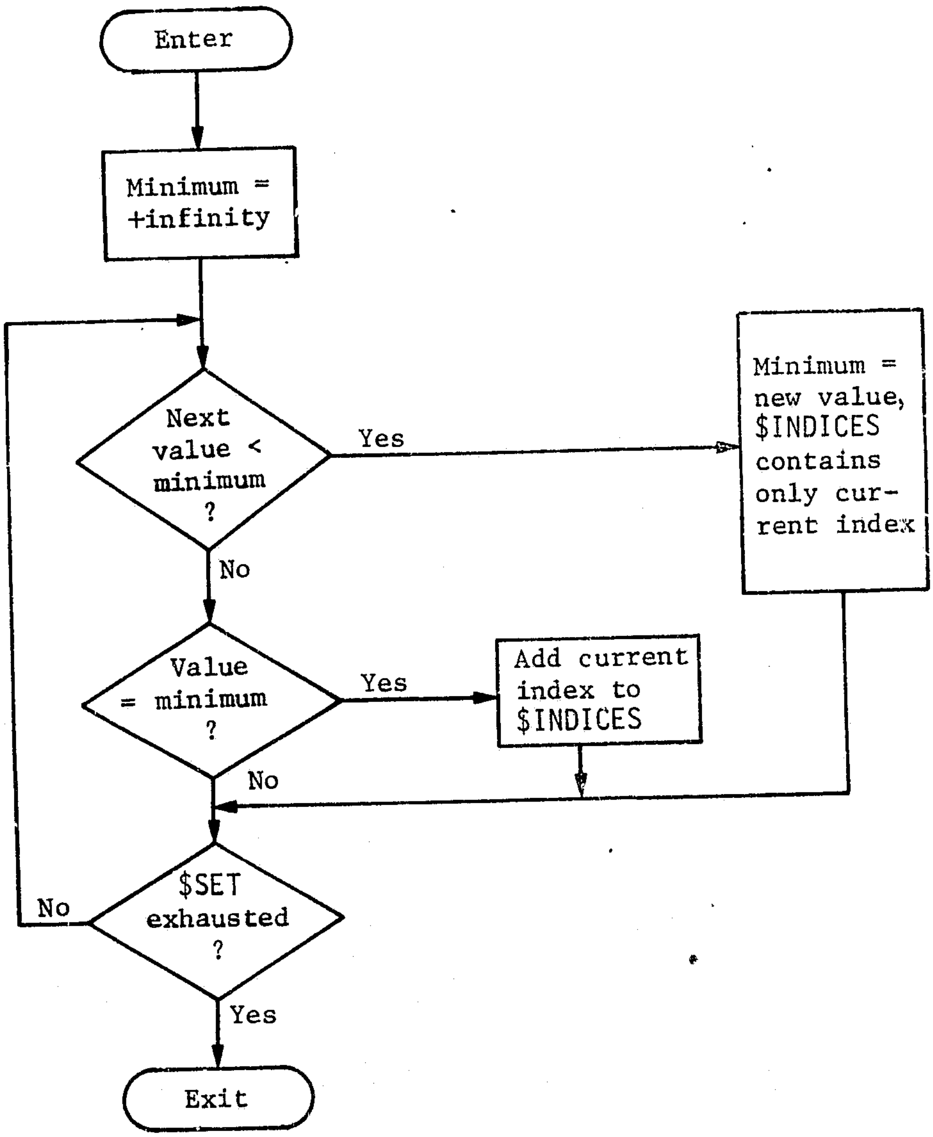
VALUE_MINIMUM is an arithmetic variable whose value is the minimum of the values of $SET.

$INDICES is a tree of the form



where the indices are the ordinal positions in $SET of all nodes whose value equals VALUE_MINIMUM.

2.4.6.5 Functional Block Diagram



2.4.6.6 DETAILED DESIGN

This module iteratively searches through the numeric values of the subnodes of $SET. As the subnodes are scanned, the current minimum is maintained by comparing each value with it. $INDICES and the VALUE_MINIMUM are updated every time a smaller value is encountered. If $SET is empty, +INFINITY is returned as the minimum value.

2.4.6.7 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

I	-	is a counter used to obtain subnodes indices
$INDICES	-	used to record the indices of the subnodes whose value equals the minimum
$SET	-	is the set of numeric values
$VALUE	-	is one of the subnodes of $SET
VALUE_MINIMUM	-	used to return the minimum value to the calling program

2.4.6.8 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES

In addition to returning the appropriate subscripts in $INDICES, this tree is also used to return the node labels. Of course, node references by subscript are more efficient than references by label. Therefore, the user is not encouraged to use these labels to reference the subnodes of $SET since the corresponding indices are also available.

The variable name VALUE_MINIMUM was substituted for MINIMUM in order to allow for real as well as integer values.

2.4.6.9 COMMENTED CODE

```
    FIND_MIN:  PROCEDURE ($SET, $INDICES, VALUE_MINIMUM)
               OPTIONS(EXTERNAL);
/*************************************************************************/
/*                                                                       */
/*  THIS PROCEDURE FINDS THE SMALLEST VALUE IN THE NUMERIC SET INPUT     */
/*  IN $SET AND RETURNS IT IN 'VALUE_MINIMUM'.  IT ALSO OUTPUTS A        */
/*  TREE, $INDICES, CONTAINING THE LABELS AND SUBSCRIPTS THAT COR-       */
/*  RESPOND TO THE SUBNODES OF $SET WHOSE VALUES EQUAL THE MINIMUM.      */
/*  IF $SET IS NULL, +INFINITY IS RETURNED AS THE 'VALUE_MINIMUM'.       */
/*                                                                       */
/*************************************************************************/
   DECLARE I,$VALUE LOCAL ;  I = 0 ;
   VALUE_MINIMUM = INFINITY ;
   DO FOR ALL SUBNODES OF $SET USING $VALUE ;
      I = I+1 ;
      IF $VALUE < VALUE_MINIMUM
         THEN DO ;  VALUE_MINIMUM = $VALUE ;
                    PRUNE $INDICES ;
                    $INDICES.#LABEL($VALUE) = I ;
              END ;
         ELSE IF $VALUE = VALUE_MINIMUM
                 THEN DO ;  $INDICES(NEXT) = I ;
                            LABEL($INDICES(LAST)) = LABEL($VALUE) ;
                      END ;
   END ;
END;     /*   FIND_MIN       */
```

C.2

2.4.7 CHECK_FOR_PROCESS_DEFINITION

2.4.7 CHECK_FOR_PROCESS_DEFINITION

2.4.7.1 Purpose and Scope

This module checks that all processes or operations sequences specified in $OBJECTIVE are defined in $PROCESS or $OPSEQ. These processes may be listed explicitly or contained in an operations sequence specified in $OBJECTIVES. If any processes are not included in $PROCESS, such information as process duration and required resources are not defined. Since this condition precludes successful execution of the problem, the missing processes should be identified. This module performs that identification function.

2.4.7.2 Modules Called

None

2.4.7.3 Module Input

Input to this module consists of $OBJECTIVES, $OPSEQ and $PROCESS. The minimum required data structure from these Standard Data Structures is illustrated in Fig. 2.4.7-1.

2.4.7.4 Module Output

This module will output a tree structure, $MISSING, with the names of unfound processes and operations sequences. If this tree is null, no missing definitions have been identified.

Note: Minimum (i.e. relevant) portion of required input Standard Data Structures is shown. In all trees, any additional structure will be preserved.

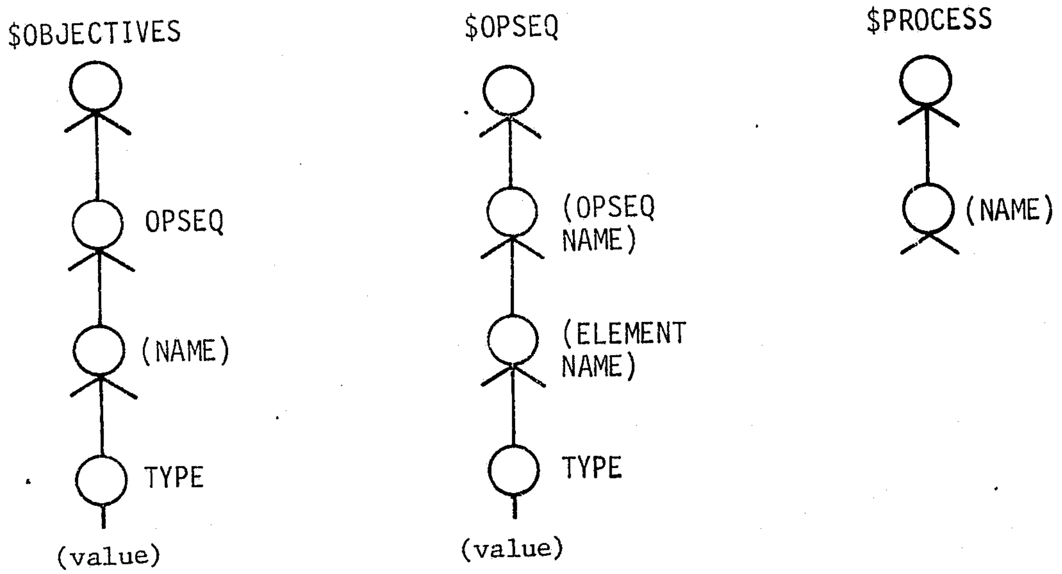
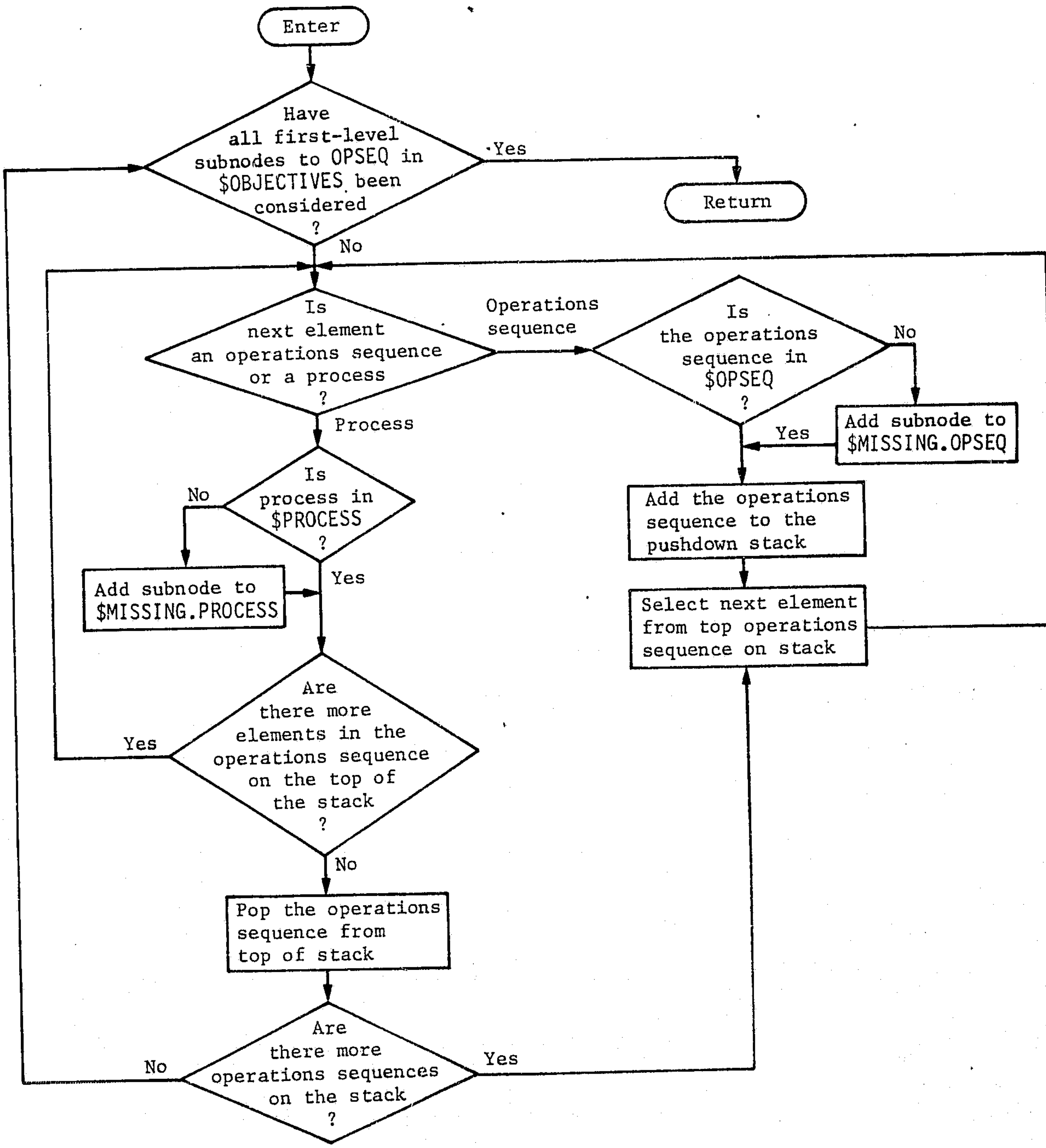


Fig. 2.4.7 -1
Minimum Required Input Structures from Standard Data Structures for Module: CHECK_FOR_PROCESS_DEFINITION

2.4.7.5 Functional Block Diagram



ORIGINAL PAGE IS OF POOR QUALITY

2.4.7.6 Typical Applications

This module is useful for initial problem processing, which checks for logical errors or incomplete data.

2.4.7.7 Detailed Design

The functional block diagram provides the flow chart for this module. The module selects each subnode under OPSEQ in the $OBJECTIVES tree. For each element an internal procedure, CHECK_PROC_RECURSIVE is called resursively. This procedure determines whether the current element is an operations sequence or a process. If the element is a process, $PROCESS is interrogated to verify the current element is included, if not, the current element label is added to $MISSING. If the element is determined to be an operations sequence, $OPSEQ is interrogated to verify the current element is included, if not the current element label is added to $MISSING. Otherwise each subnode is selected and the procedure CHECK_PROC_RECURSIVE is repeatedly called until all processes have been checked.

2.4.7.8 Internal Variable and Tree Names

None

2.4.7.9 Commented Code

```
CHECK_FOR_PROCESS_DEFINITION: PROCEDURE($OBJECTIVES,$OPSEQ,$PROCESS,
        $MISSING) OPTIONS(EXTERNAL);
     PRUNE $MISSING;
     DO I = 1 TO NUMBER($OBJECTIVES.OPSEQ);
     CALL CHECK_PROC_RECURSIVE($OBJECTIVES.OPSEQ(I));
  CHECK_PROC_RECURSIVE: PROCEDURE($OPSEQ_OR_PROC) RECURSIVE;
     DECLARE J LOCAL;
     IF $OPSEQ_OR_PROC.TYPE = 'OPSEQ' THEN DO;
  CHECK_OPSEQ:
     IF $OPSEQ.#LABEL($OPSEQ_OR_PROC)
      IDENTICAL TO $NULL
        THEN $MISSING.OPSEQ(NEXT) = LABEL($OPSEQ_OR_PROC);
           ELSE DO J = 1 TO NUMBER($OPSEQ.#LABEL($OPSEQ_OR_PROC));
           CALL CHECK_PROC_RECURSIVE($OPSEQ.#LABEL($OPSEQ_OR_PROC)
               (J));
           END;
           RETURN;
           END;
     ELSE IF $PROCESS.#LABEL($OPSEQ_OR_PROC)
      IDENTICAL TO $NULL
        THEN $MISSING.PROCESS(NEXT) = LABEL($OPSEQ_OR_PROC);
     RETURN;
     END ;        /*   CHECK_PROC_RECURSIVE   */
     END ;
     END ; /* CHECK_FOR_PROCESS_DEFINITION */
```

2.4.8 GENERATE_JOBSET

2.4.8 GENERATE_JOBSET

2.4.8.1 Purpose and Scope

This module will create a set of jobs by examining the contents of the data trees, $OBJECTIVES, $OPSEQ, and $PROCESS. Each job will represent a single occurrence of a process. It will create an output data tree that contains unique nodes for each job identified.

This module will build a data tree, $JOBSET, containing only the jobs that are identifiable from the input trees $OBJECTIVES, $OPSEQ, and $PROCESS. The jobs will be grouped under first-level subnodes, each of which represents the occurrence of an operations sequence. Because operations sequences may be nested, only those at the highest level of nesting will cause a first-level node of $JOBSET to be built. If, during the execution of a scheduling problem, implied jobs are scheduled, these jobs may be added to the trees created by this module; however, the GENERATE_JOBSET module *will not* put implied jobs into the jobset since implied jobs cannot be identified from $OBJECTIVES, $OPSEQ, and $PROCESS.

The output of this module will be ordered by a set of unique job identifiers, but the ordering of the identifiers will have no implication on the temporal order in which the jobs must be scheduled.

Each job created by this module will have associated with it the most specific resource information contained in either $OBJECTIVES or $PROCESS. If resource alternatives are defined in $PROCESS, a separate job identifier will be assigned to each process-resource combination. Where process alternatives are indicated in $OBJECTIVES or $OPSEQ, all alternatives will also be assigned a unique job identifier. This module will write the job identifiers for all alternatives under each job. For example, if Jobs 1, 2, and 3 are mutually substitutable, a unique node will appear in the output $JOBSET for each of the three jobs. The node ALTERNATIVE under Job 1 would contain subnodes JOB 2 and JOB 3, the node ALTERNATIVE for Job 2 would contain sub-nodes JOB 1 and JOB 3, etc. It is recognized that if no implied jobs are rescheduled the number of jobs in the final schedule will always be less than or equal to the number of jobs in $JOBSET.

Temporal relations between elements of an operations sequence will be included in the output tree $JOBSET. The module will, however, replace a generic process name that appears as a value under a TEMPORAL_RELATION node with an appropriate specific job identifier. If the element name that appears under a TEMPORAL_RELATION node with an appropriate specific job identi-fier. If the element name that appears under a TEMPORAL_RELATIONS node is itself an operations sequence, a separate subnode will be written for each job in that operations sequence.

2.4.8.2 <u>Modules Called</u>

None

2.4.8.3 Module Input

The input to this module consists of the trees, $OBJECTIVES, $OPSEQ, and $PROCESS, defined previously, and the integer INITIAL_ID. The minimum required data structure from these standard structures is shown in Fig. 2.4.8-1. INITIAL_ID is the first integer to be used in constructing unique job identifiers within the module.

2.4.8.4 Module Output

This module will return an output tree $JOBSET to the calling program. It will contain the RESOURCE information from $PROCESS with any specific ASSOCIATED_RESOURCE information from $OBJECTIVES replacing the corresponding generic information in the RESOURCES. Since it is permissable to specify specific resources in both $PROCESS and $OBJECTIVES, this module will produce an error message when inconsistent data are specified. The structure of $JOBSET is shown in Fig. 2.4.8-2.

Note: Minimum (i.e., relevant) portion of required input Standard Data Structures is shown. Any additional structure will be preserved in all trees.

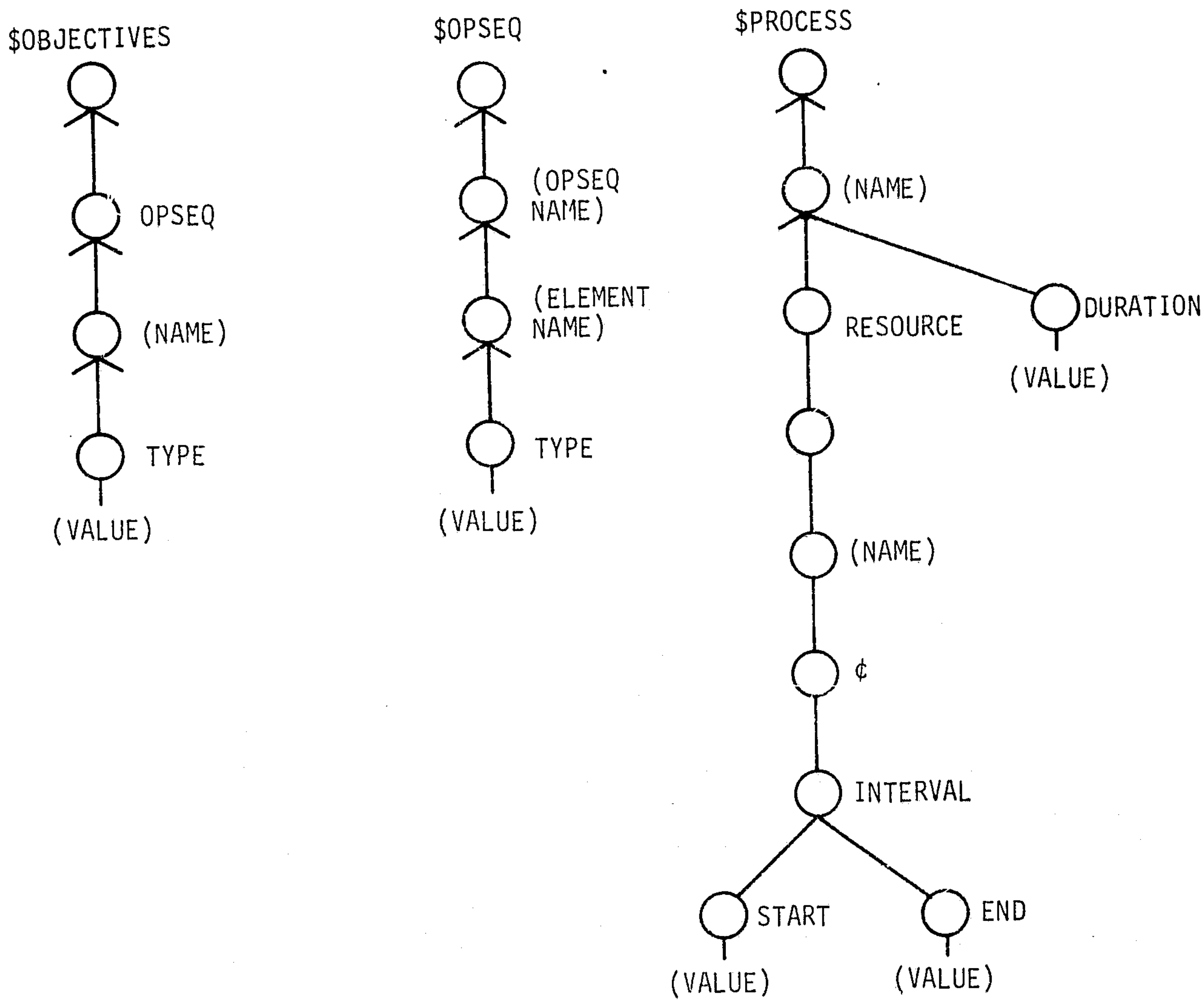


Fig. 2.4.8-1
Minimum Required Input Structures from Standard Data Structures for Module Generation

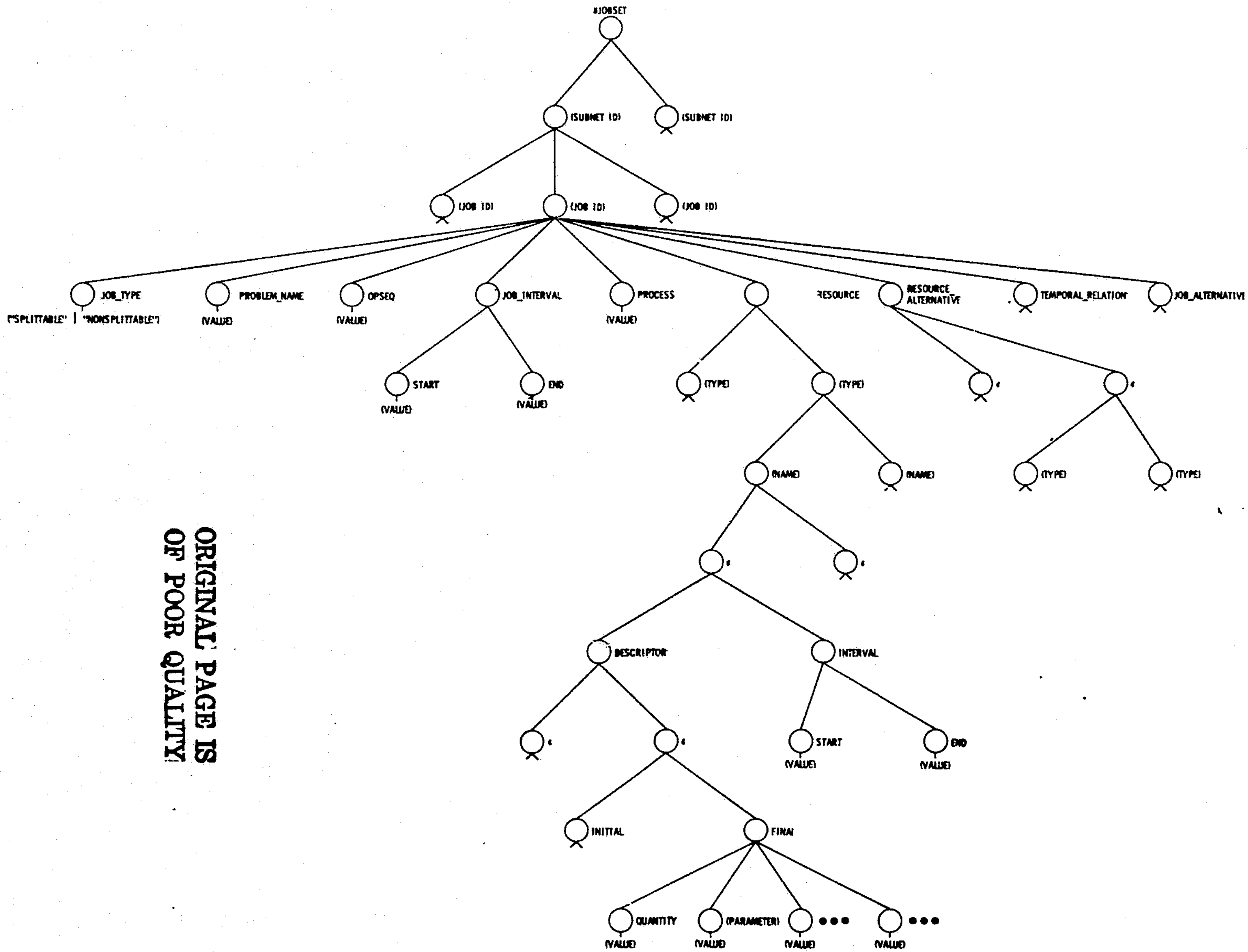
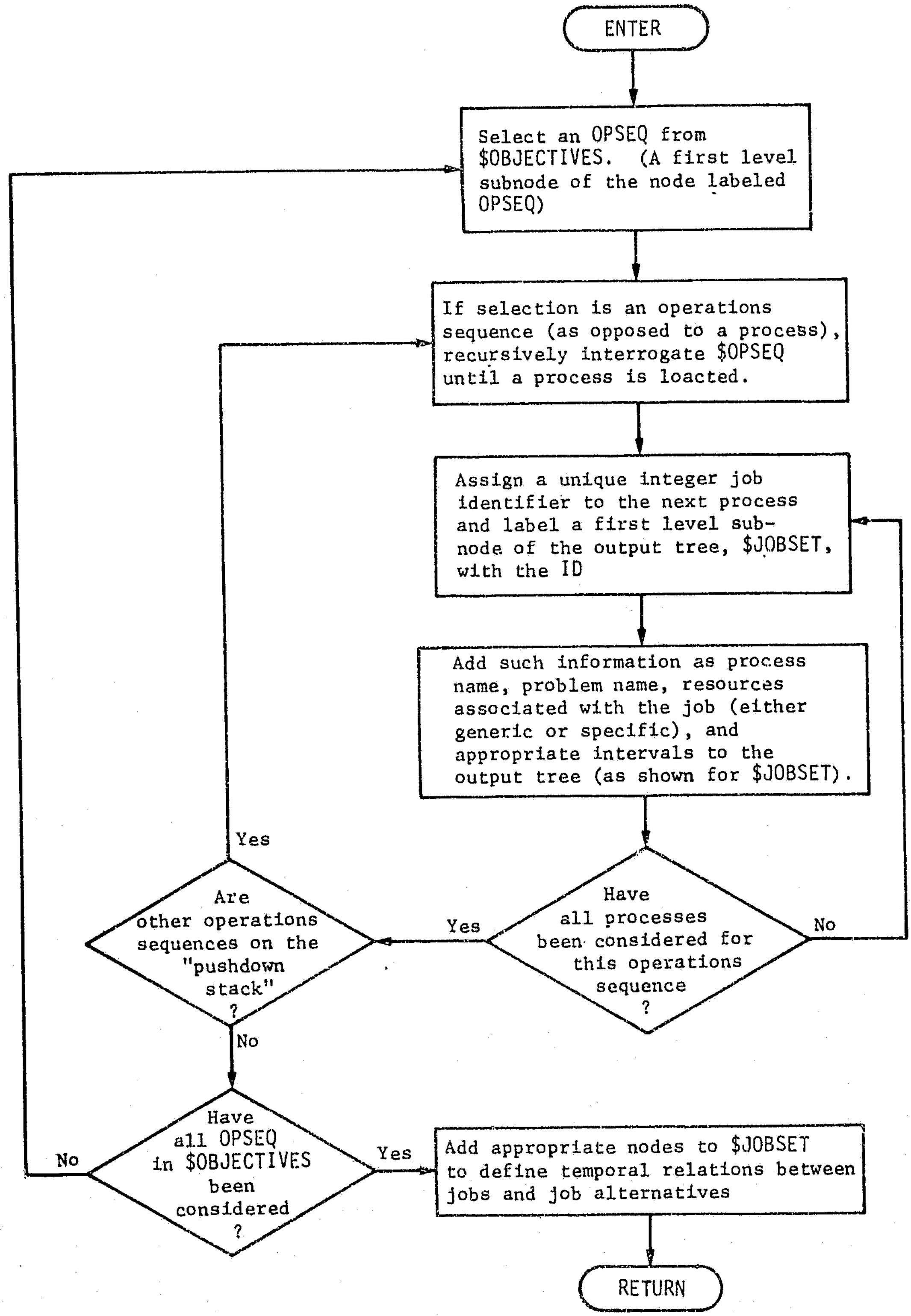


Fig. 2.4.8-2 GENERATE_JOBSET Standard Data Structure

2.4.8.5 Functional Block Diagram



2.4.8.6 Typical Applications

Since this module merges some of the information contained in $OBJECTIVES with all of the required information from $OPSEQ and $PROCESS, its usefulness is in eliminating numerous accesses of those structures that result from the information in one tree pointing to information in another. The creation of jobs is a logical first step in building a schedule. It will be recognized that if no alternatives or temporal relations appear in $JOBSET, then $JOBSET represents a generic schedule unit that may be given a specific time assignment. If, however, either resource alternatives, job alternatives, or temporal relations do appear in the problem specification, these alternatives and constraints are still represented in $JOBSET. Thus, the initial creation of $JOBSET permits the subsequent scheduling logic to deal with only $OBJECTIVES and $JOBSET without reference to $OPSEQ or $PROCESS.

2.4.8.7 DETAILED DESIGN

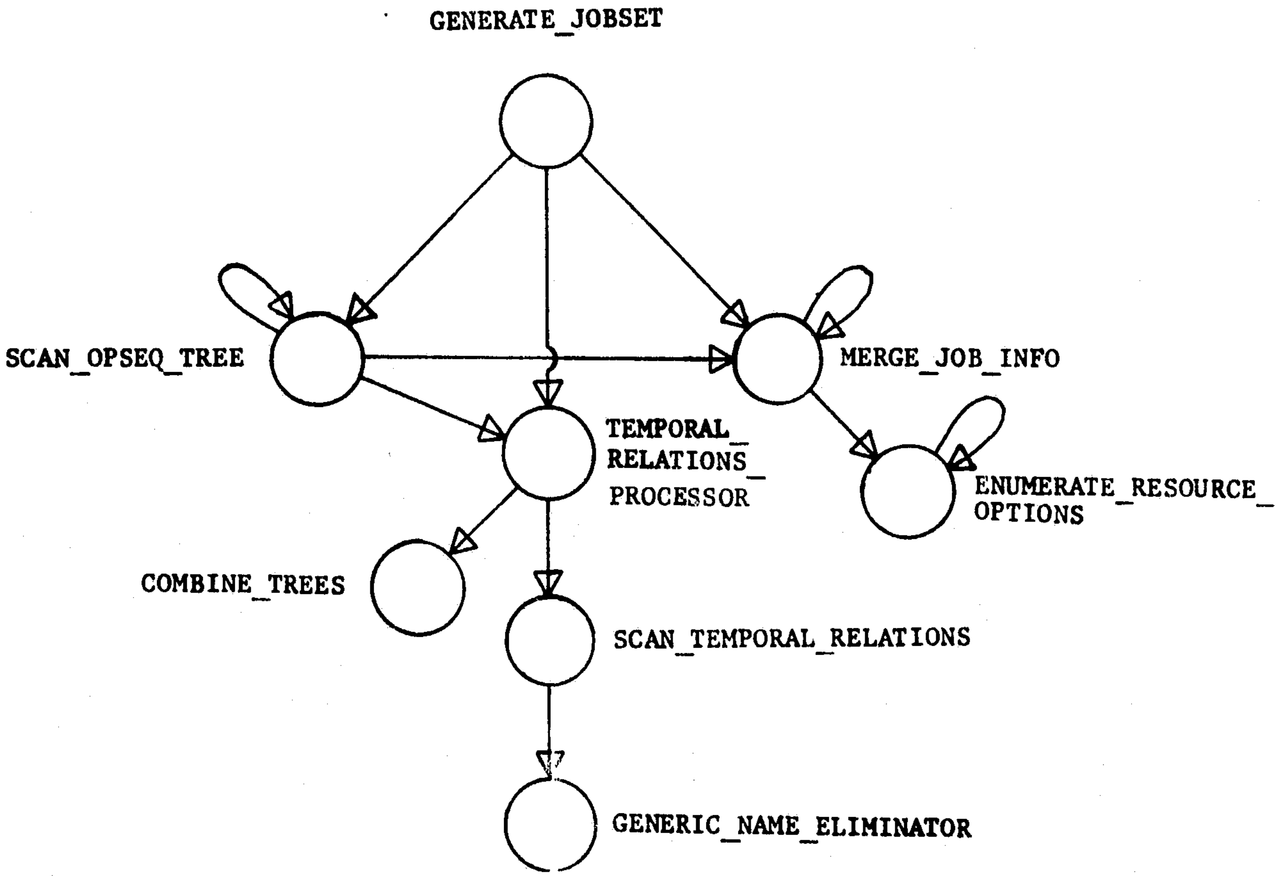
GENERATE_JOBSET builds the $JOBSET standard data structure by iteratively interrogating the input data trees until the problem has been reduced to that of building a single 'JOB ID' substructure. At each level in tracing down to this relatively simple problem, a different tree structure is built. The tree structures at each level are used to make up the tree at the next higher level. By using this "building block" approach the relatively complex tree structure that exists at the highest level ($JOBSET) is generated by dealing with its much simpler component parts.

The "building block" tree structures generated and maintained by the program are, in order of descending complexity, $JOBSET, $OPSEQ_SET, $JOBS, and $JOB. $JOBSET is, of course, the goal structure and will look exactly as it is shown in the Module Output section. $OPSEQ_SET and $JOBS will, in their final form, look exactly like one of the 'SUBNET ID' substructures of $JOBSET. Although these two trees eventually assume the same position in $JOBSET, there is a basic difference in what they represent. $OPSEQ_SET will contain all of the jobs contained in a given op-sequence, including those arising from any nested op-sequences. $JOBS, on the other hand, represents the occurrence of a single process. It may, however, contain several "job" subnodes representing a complete set of possible job alternatives, only one of which will need to be scheduled. $JOB is the most basic structure and is used to build all of the other trees previously mentioned. It looks exactly like one of the 'JOB ID' substructures of $JOBSET.

$JOBSET, $OPSEQ_SET, and $JOBS are each built by a separate PLANS procedure. They are: GENERATE_JOBSET (main procedure), SCAN_OPSEQ_TREE, and MERGE_JOB_INFO, respectively. The four other internal procedures operate at a lower level and provide special-purpose services to the three major procedures listed above. In several cases these small procedures were separated from the three major ones in order to make the code more efficient. This was accomplished by using a technique made possible by PLANS conventions. In several uses it was necessary to frequently reference a subnode deep within a tree. This becomes very expensive since each node reference requires a long list of node qualifiers. To eliminate this expense, a separate procedure was written which includes the node reference as an input parameter. The dummy tree name used in the PROCEDURE statement is then effectively overlayed at the subnode which was passed in the parameter list of the CALL statement. This technique reduces the length of node references, increases access efficiency, and has the added advantage of prompting the PLANS programmer to write modular programs.

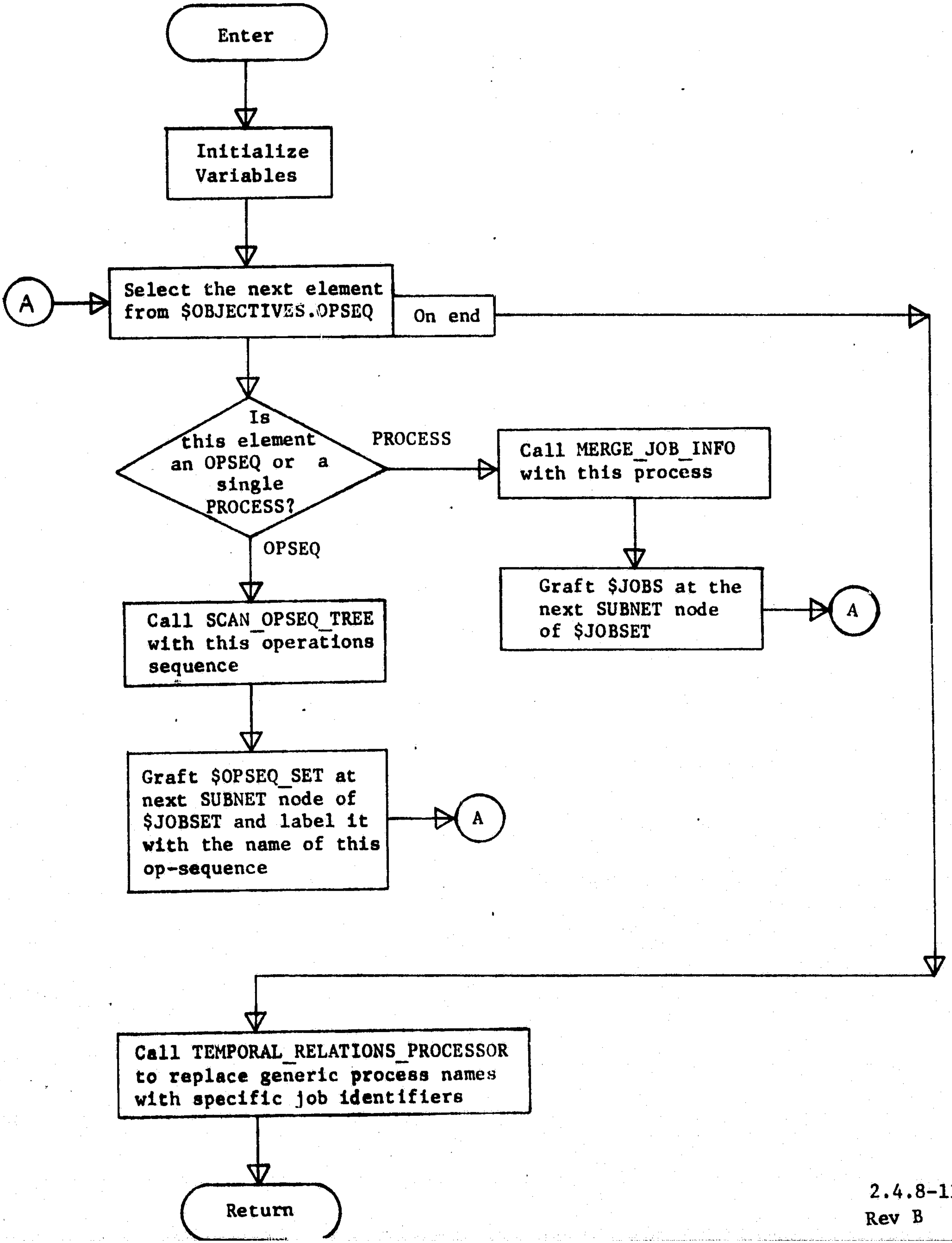
In some instances, program efficiency was sacrificed for readibility. This is justified by the fact that the module will undoubtedly need to be modified, since this code is not intended to be the final version.

The sketch below shows the calling hierarchy of GENERATE_JOBSET and its seven internal procedures. An arrow indicates a "calling" relationship between two procedures.

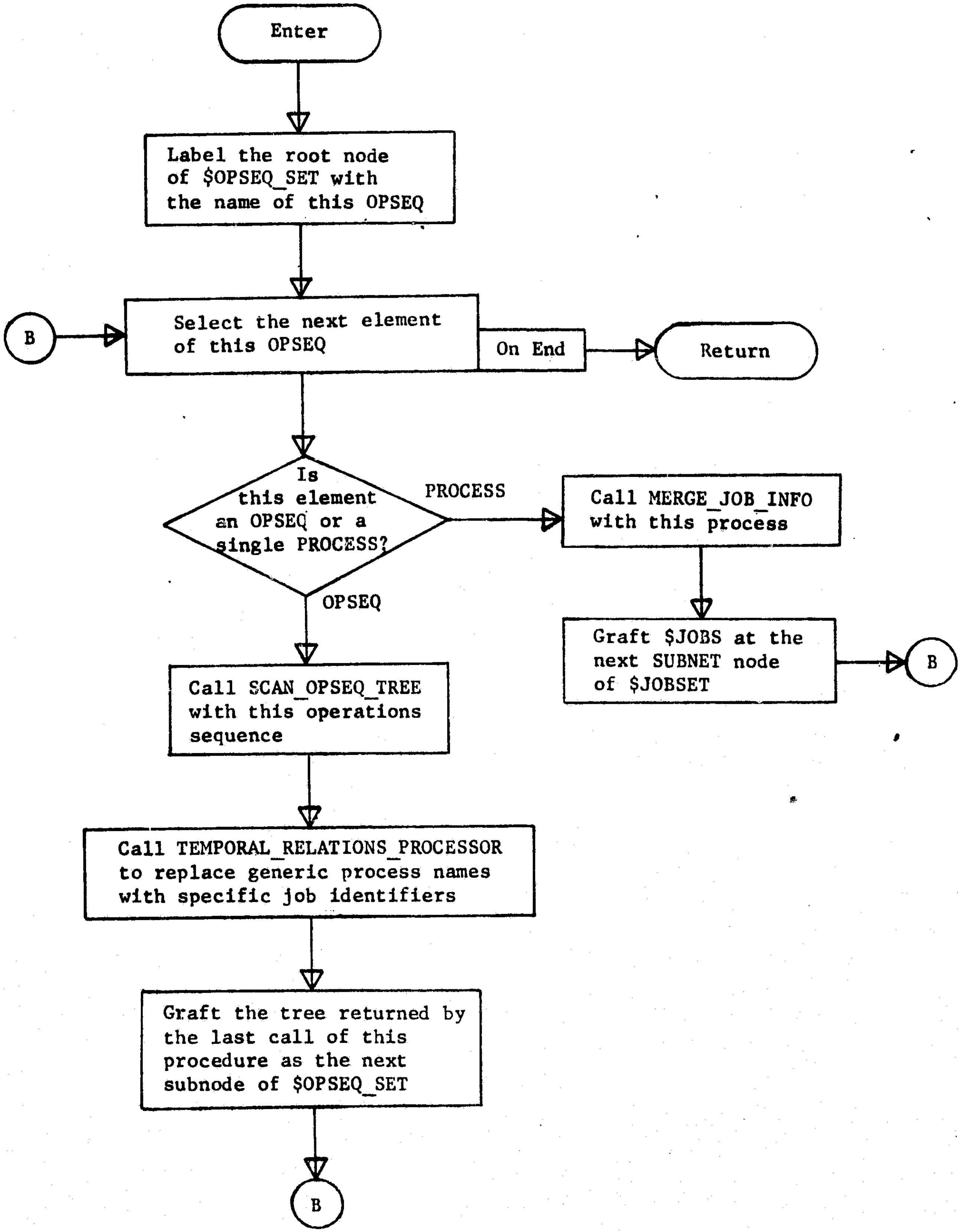


Calling relationships of GENERATE_JOBSET and its internal procedures

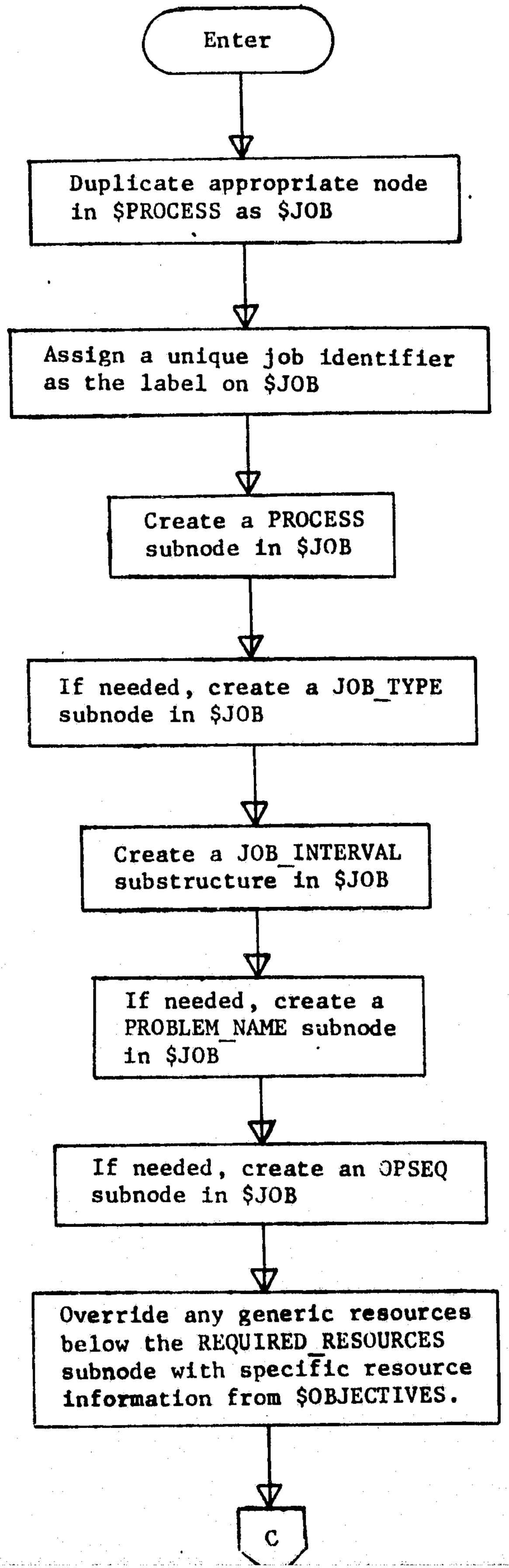
GENERATE_JOBSET (builds $JOBSET)

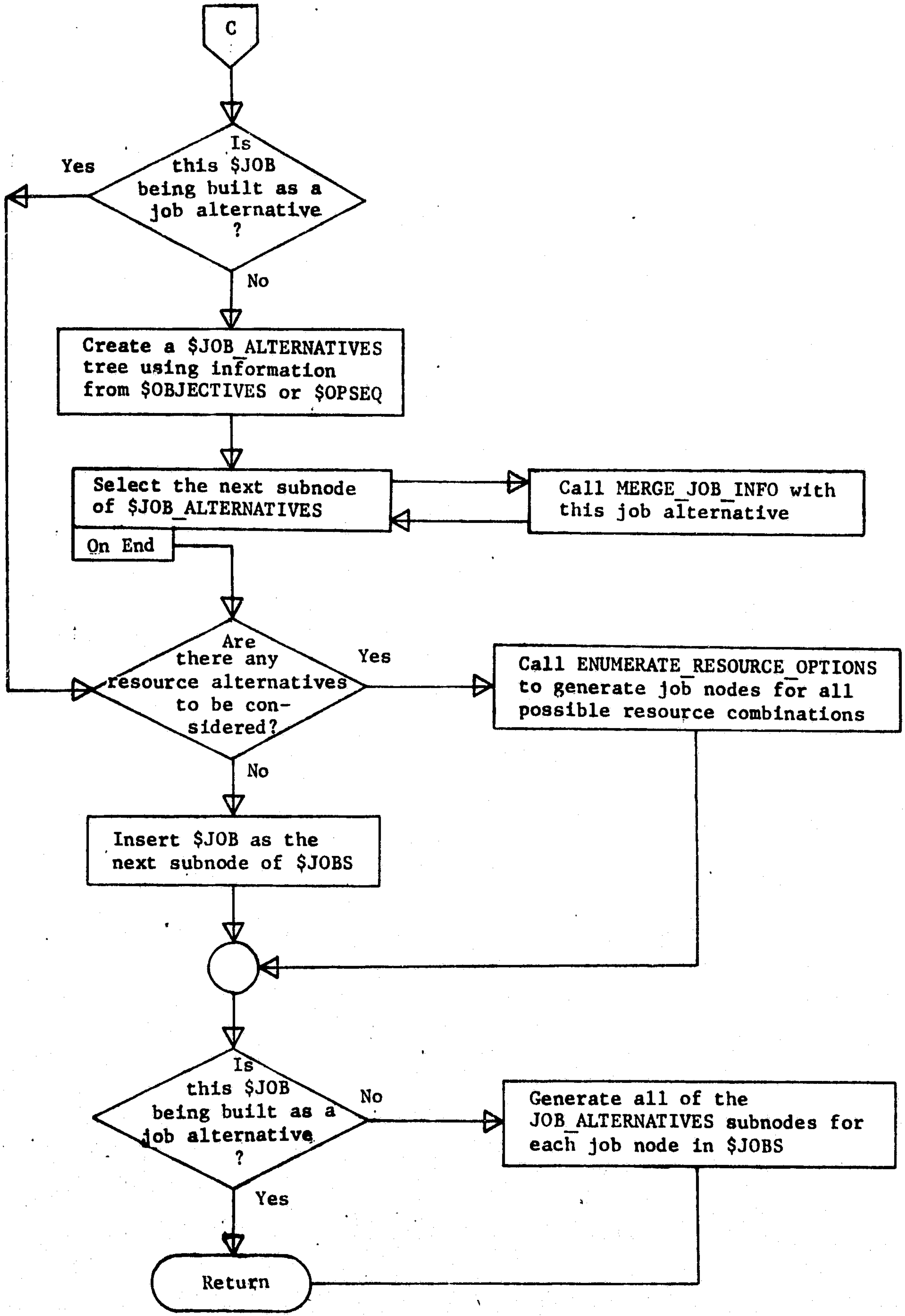


SCAN_OPSEQ_TREE (builds $OPSEQ_SET)

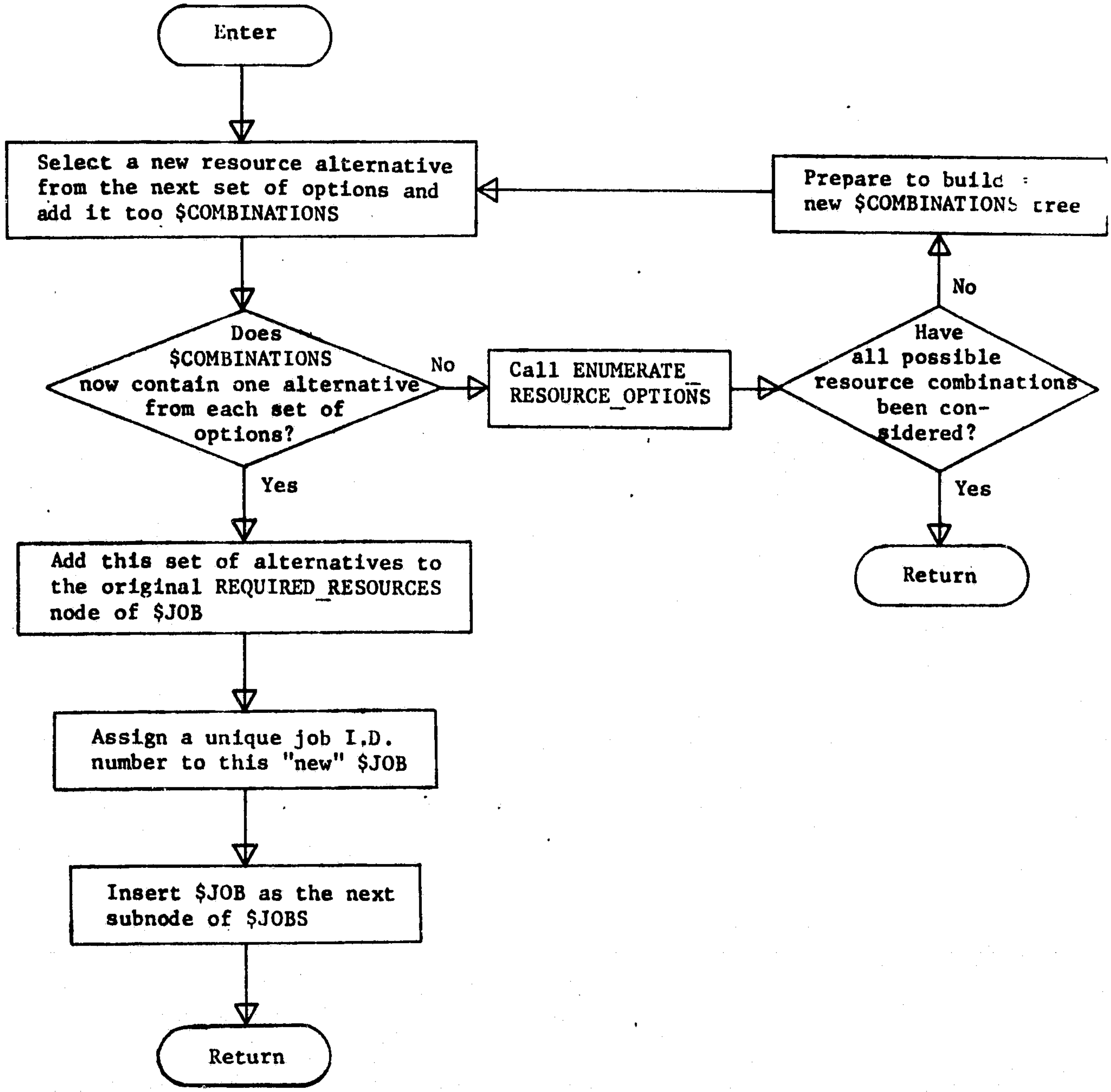


MERGE_JOB_INFO (builds $JOBS)

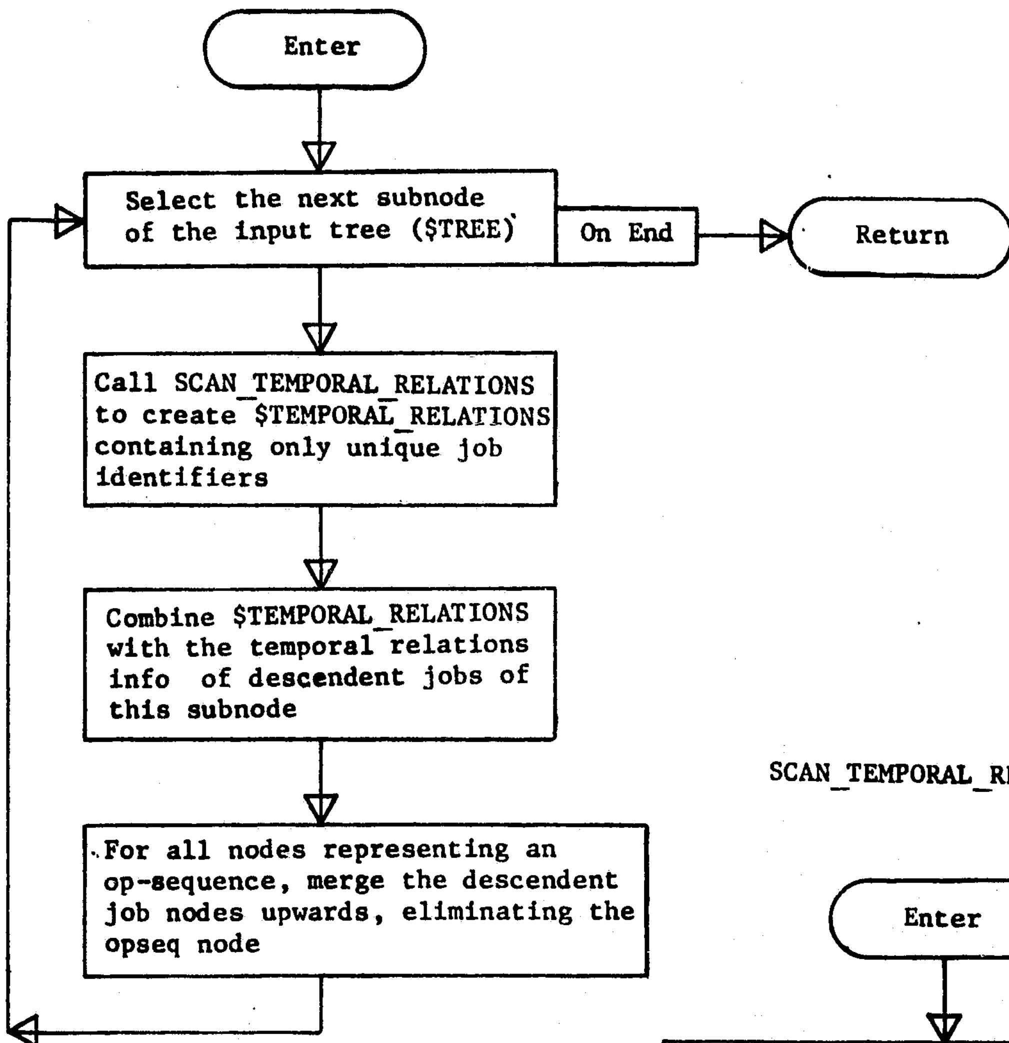



C
Is this $JOB being built as a job alternative ?
Yes
No
Create a $JOB_ALTERNATIVES tree using information from $OBJECTIVES or $OPSEQ
Select the next subnode of $JOB_ALTERNATIVES
On End
Call MERGE_JOB_INFO with this job alternative
Are there any resource alternatives to be considered?
Yes
No
Call ENUMERATE_RESOURCE_OPTIONS to generate job nodes for all possible resource combinations
Insert $JOB as the next subnode of $JOBS
Is this $JOB being built as a job alternative ?
No
Yes
Generate all of the JOB_ALTERNATIVES subnodes for each job node in $JOBS
Return

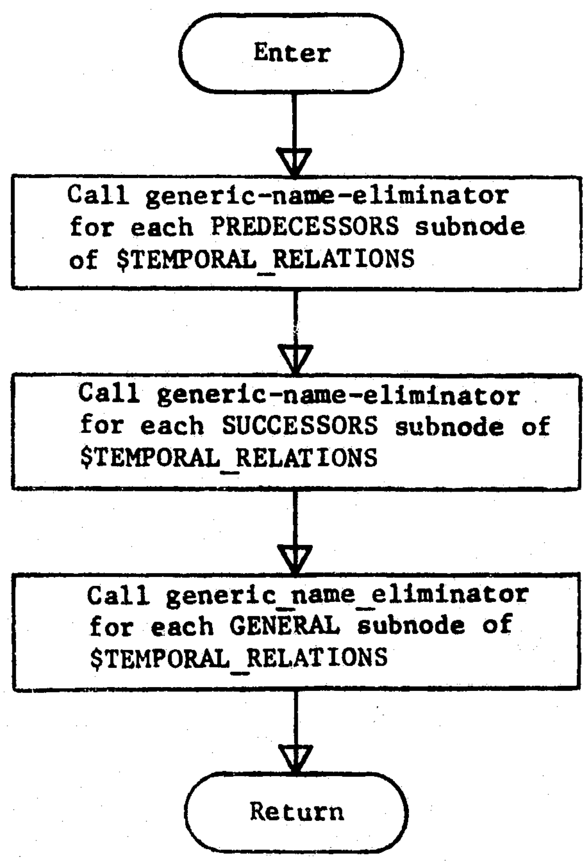
ENUMERATE_RESOURCE_OPTIONS (builds $COMBINATIONS)



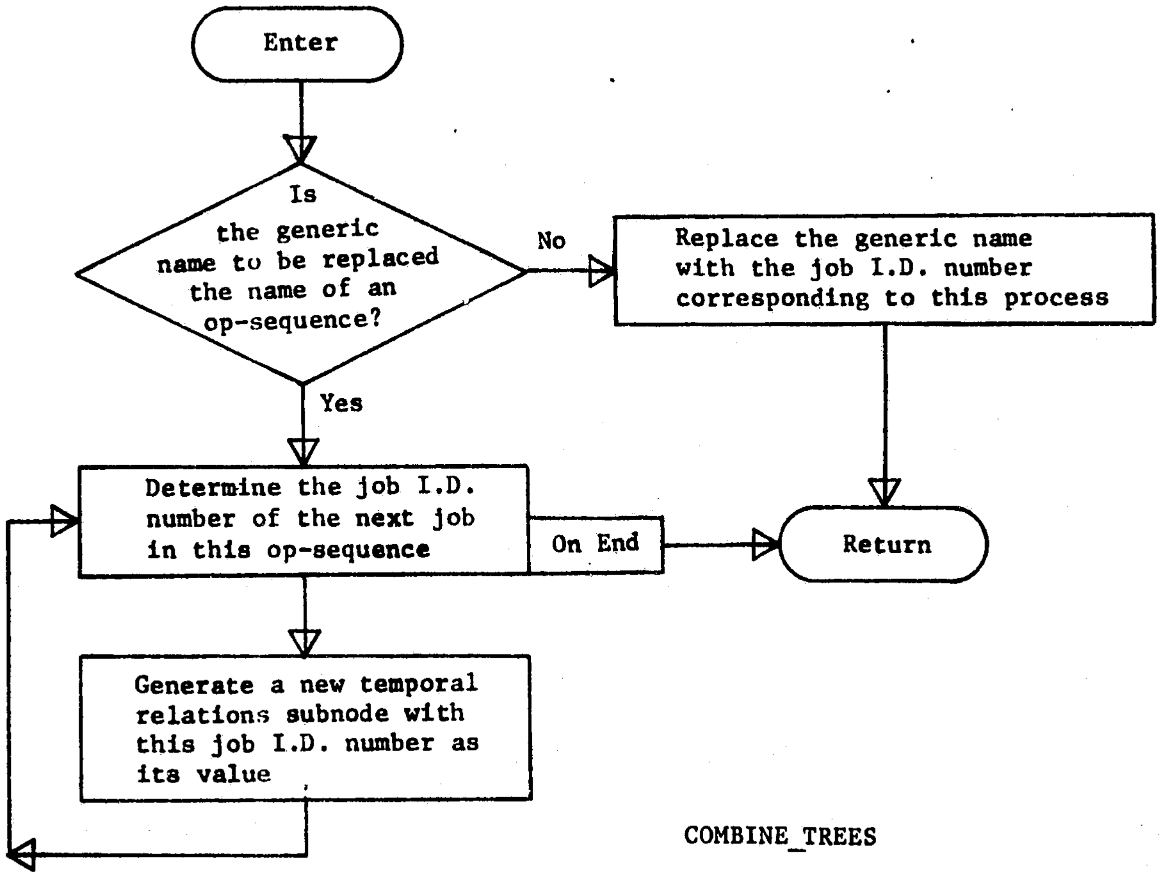
TEMPORAL_RELATIONS_PROCESSOR



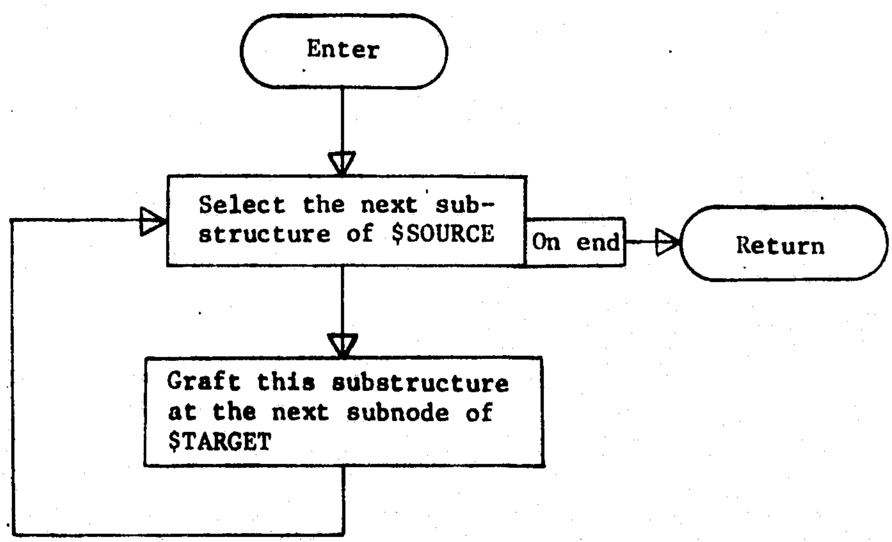
SCAN_TEMPORAL_RELATIONS



GENERIC_NAME_ELIMINATOR



COMBINE_TREES



2.4.8.8 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

$COMBINATIONS	-	used to store a set of resource options when generating all possible resource combinations
I	-	a global pointer used to indicate which subnode of $OBJECTIVES. OPSEQ is currently being processed.
I_RECURSIVE_CALL_FLAG	-	used to indicate whether or not the current invocation of MERGE JOB INFO was recursive
ISAVE	-	used to save the value of the job ID number of the first job node in $JOBS
J	-	a pointer used to indicate which op-sequence in $OPSEQ is the source of the jobs currently being generated.
$JOB	-	the basic structure build by GENERATE_JOB_SUBSTRUCTURE
$JOBS	-	the final product of a call to GENERATE_JOB_SUBSTRUCTURE, its subnodes formed by grafting on one or more $JOB trees.
$JOB_ALTERNATIVE_SET	-	used to store the specific job identifiers for a complete set of job alternatives.
$JOB_ALTERNATIVES	-	a duplication of an ALTERNATIVES subnode of $OBJECTIVES or $OPSEQ
$NAME	-	used to store the substructure of an op-sequence whose elements have been reduced to job nodes.
$OPSEQ_NAME	-	used to indicate the name of the op-sequence which gave rise to the job node currently being generated.
$OPSEQ_NODE	-	is the element of $OPSEQ. (OBJECT_ELEMENT) that is currently being processed.

$ACTIVITY	-	equal to $OPSEQ_SET in the case of an opsequence equal to $NULL when processing a process.
$SUB_ACTIVITY	-	points at the subnodes of $ACTIVITY.
$JOB_NODE	-	points at the subnodes of $SUB_ACTIVITY, i.e., the $JOB information.
$RESOURCE_RELATIONS	-	a duplication of the RESOURCE_RELATIONS subnode of $OBJECTIVES or $OPSEQ.
$RES_REL	-	a pointer, pointing at the subnodes of $RESOURCE_RELATIONS
$POINT	-	points at the second level down of $ACTIVITY, the job node.
K, M, L	-	index pointers.
$RELATION	-	points at a subnode of $TEMPORAL_NODE.
$SUBNET	-	duplicates the job node in $OPSEQ_SET if the generic name to be replaced is the name of an opsequence.
$DUMI	-	temporary storage area
$DUM2	-	temporary storage area
$DUM3	-	temporary storage area
KODE	-	used to indicate what kind of input error existed when the call to $JOBSET occurred (see section 2.4.8.9)
$OBJECT_ELEMENT	-	indicates the subnode of $OBJECTIVES. OPSEQ which is currently being processed.
$NILL	-	equivalent to $NULL.

$OPSEQ_SET	-	tree structure built by GENERATE_OPSEQ_SET.
$OPTIONS	-	is the RESOURCE_ALTERNATIVES subnode of the job node which is currently being built.
$PROC_NODE	-	is the subnode of $PROCESS corresponding to the process for which a job node is being built.
$PROCESS_ID	-	is the subnode of $TEMPORAL_RELATIONS whose value is being replaced with a specific job identifier.
$REF	-	is the tree from which $TEMPORAL_RELATIONS will be taken (either $OBJECTIVES or $OPSEQ)
$REQUIRED	-	is the REQUIRED_RESOURCES subnode of the job node which is currently being built.
$SPECIFICS	-	a duplication of an ASSOCIATED_RESOURCES subnode of $OBJECTIVES
$TEMP	-	used in various places as a temporary storage area.
$TEMP_NODE	-	used to temporarily store a single subnode of $JOB_ALTERNATIVE_SET.
$TEMPORAL_NODE	-	is the subnode of $TEMPORAL_RELATIONS which is currently being processed.
$TEMPORAL_RELATIONS	-	a duplication of a TEMPORAL_RELATIONS subnode of $OBJECTIVES OR $OPSEQ.
$TYPE	-	used to store the name of a resource type
$WORKSPACE	-	is used as a storage area where resource alternatives from $COMBINATIONS can be added to the resources in $REQUIRED.
$SOURCE	-	the tree whose substructure will be grafted into $TARGET.
$TARGET	-	the tree onto which additional first-level subnodes will be added.

$JOBID	-	a pointer, pointing to the job in $JOBSET currently being processed.
$R_R	-	a pointer, pointing at subnodes of RESOURCE_RELATION substructure.
$OP_ELEMENT	-	pointer, describing the substructure of $OPSEQ_NODE
START_TIME	-	the user specified start time for a given job.
$DUMMY1	-	temporary storage
$DUMMY2	-	temporary storage
$DUMMY3	-	temporary storage
N	-	number of subnodes of $OPTIONS

2.4.8.9 COMMENTED CODE

```
GENERATE_JOBSET:
/*************************************************************************/
/*                                                                       */
/*  THIS MODULE CREATES INDIVIDUAL JOBS FOR EACH OCCURRENCE OF A         */
/*  PROCESS SPECIFIED EXPLICITLY OR VIA AN OPERATIONAL SEQUENCE IN       */
/*  $OBJECTIVES.  IT MERGES INFORMATION CONTAINED IN $OBJECTIVES,        */
/*  $OPSEQ, AND $PROCESS INTO A TREE CALLED $JOBSET.  THE INTEGER        */
/*  VALUE OF 'INITIAL_ID' IS USED TO CONSTRUCT UNIQUE JOB IDENTIFI-      */
/*  ERS ON EACH 'JOB ID' SUBNODE OF $JOBSET.  ON RETURN, $JOBSET         */
/*  WILL CONTAIN THE ENTIRE SET OF JOBS (INCLUDING ALTERNATIVES)         */
/*  WHICH ARE TO BE SCHEDULED.  THIS IS THE FINAL STEP BEFORE THE        */
/*  DECISION ALGORITHMS CAN MAKE EXPLICIT TIME AND RESOURCE ASSIGN-      */
/*  MENTS.  IT SHOULD BE NOTED THAT 'GENERATE_JOBSET' ALSO CHECKS        */
/*  FOR SEVERAL ERROR CONDITIONS THAT MAY BE PRESENT IN THE INPUT        */
/*  DATA TREES.  AS A TEMPORARY WARNING MECHANISM, THE MODULE RE-        */
/*  TURNS AN ERROR CODE TO THE CALLING PROGRAM VIA THE OUTPUT VARI-      */
/*  ABLE 'KODE'.  IN THE FUTURE, COMPLETE ERROR MESSAGES WILL BE         */
/*  WRITTEN OUT DESCRIBING THE PROBLEM AND ITS PROBABLE CAUSE.           */
/*                                                                       */
/*************************************************************************/
PROCEDURE ($OBJECTIVES, $OPSEQ, $PROCESS, INITIAL_ID,  $JOBSET, KODE)
             OPTIONS(EXTERNAL);
   DECLARE I,I_RECURSIVE_CALL_FLAG,ISAVE,$JOB,$JOBS,K,KODE,L,M,N,$NAME,
                    $OBJECT_ELEMENT,$OPSEQ_SET,$RELATION,$SPECIFICS,
                    START_TIME,$TEMP_NODE,$TYPE,$WORKSPACE,
                    $NILL, $OPS_ELE, $JOBID, $R_R  LOCAL;
/*  INITIALIZE VARIABLES.                                                */
   KODE = 0 ;  LABEL($JOBSET) = 'OPSEQ' ;  I_RECURSIVE_CALL_FLAG = 0 ;
/*  SELECT THE NEXT ELEMENT OF $OBJECTIVES.OPSEQ AND CHECK TO SEE        */
/*  IF IT IS AN OP-SEQUENCE OR A SINGLE PROCESS.                         */
  /*  THIS LOOP ITERATES ACROSS THE FIRST-LEVEL SUBNODES OF $OBJEC-      */
  /*  TIVES.OPSEQ .  ON EACH SUCCESSIVE ITERATION IT GENERATES A         */
  /*  'SUBNET ID' SUBNODE IN $JOBSET WHICH BECOMES A FATHER NODE TO      */
  /*  ALL OF ITS DESCENDENT JOB NODES.                                   */
   DO FOR ALL SUBNODES OF $OBJECTIVES.OPSEQ USING $OBJECT_ELEMENT ;
      IF $OBJECT_ELEMENT.TYPE = 'OPSEQ'
         THEN IF $OPSEQ.#LABEL($OBJECT_ELEMENT)(1)  IDENTICAL TO  $NULL
                 THEN DO ;  KODE = 2 ;  RETURN ;  END ;
                 ELSE DO ;
/*  CALL 'GENERATE_OPSEQ_SET' WITH THIS OP-SEQUENCE AND THEN GRAFT       */
/*  $OPSEQ_SET AT THE NEXT 'SUBNET ID' NODE OF $JOBSET.                  */
                      CALL GENERATE_OPSEQ_SET
                           ($OPSEQ.#LABEL($OBJECT_ELEMENT),$OPSEQ_SET) ;
                      CALL TEMPORAL_RELATIONS_PROCESSOR
                                              ($OPSEQ_SET,$OPSEQ) ;
                      GRAFT $OPSEQ_SET AT $JOBSET(NEXT) ;
                      LABEL($JOBSET(LAST)) = LABEL($OBJECT_ELEMENT) ;
                      END ;
         ELSE IF $OBJECT_ELEMENT.TYPE ¬= 'PROCESS'
                 THEN DO ;  KODE = 3 ;  RETURN ;  END ;
                 ELSE IF $PROCESS.#LABEL($OBJECT_ELEMENT)
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
                                                  IDENTICAL TO  $NULL
                        THEN DO ;  KODE = 1 ;  RETURN ;  END ;
                        ELSE DO ;
/*  CALL 'GENERATE_JOB_SUBSTRUCTURE' WITH THIS PROCESS                    */
/*  AND THEN GRAFT $JOBS AT THE NEXT 'SUBNET ID' NODE OF $JOBSET.        */
                                $NILL = $NULL;
                                CALL GENERATE_JOB_SUBSTRUCTURE($NILL,
                                     $PROCESS,#LABEL($OBJECT_ELEMENT),
                                    $NILL);
                                GRAFT $JOBS AT $JOBSET(NEXT) ;
                                END ;
      END ;
/*  CALL 'TEMPORAL_RELATIONS_PROCESSOR' TO REPLACE GENERIC PROCESS        */
/*  NAMES WITH SPECIFIC JOB IDENTIFIERS.                                  */
   CALL TEMPORAL_RELATIONS_PROCESSOR($JOBSET,$OBJECTIVES) ;
   I = 0 ;
   DO FOR ALL SUBNODES OF $OBJECTIVES.OPSEQ USING $OBJECT_ELEMENT ;
      I = I+1 ;
      IF $OBJECT_ELEMENT.SUBNET_ID  IDENTICAL TO  $NULL
         THEN LABEL($JOBSET(I)) = I + 0.1111 ;
         ELSE LABEL($JOBSET(I)) = $OBJECT_ELEMENT.SUBNET_ID ;
      IF $OBJECT_ELEMENT.RESOURCE_RELATION(FIRST) NOT IDENTICAL TO
        $NULL
        THEN DO;
          DO FOR ALL SUBNODES OF $JOBSET(I) USING $JOBID;
           DO FOR ALL SUBNODES OF $OBJECT_ELEMENT.RESOURCE_RELATION
                   USING $R_R;
            INSERT $R_R BEFORE $JOBID.RESOURCE_RELATION(FIRST);
           END;
          END;
        END;
      END ;
 RETURN_ERROR_CODE:  RETURN ;


 GENERATE_OPSEQ_SET:  PROCEDURE ($OPSEQ_NODE, $OPSEQ_SET) RECURSIVE ;
 /*  THIS PROCEDURE BUILDS THE $OPSEQ_SET TREE STRUCTURE WHICH CON-      */
 /*  TAINS ALL OF THE JOBS CONTAINED IN A GIVEN OP-SEQUENCE INCLUDING    */
 /*  THOSE ARISING FROM ANY NESTED OP-SEQUENCES.  THE ONLY INPUT,        */
 /*  $OPSEQ_NODE, IS THE FIRST-LEVEL SUBSTRUCTURE OF $OPSEQ CORRES-      */
 /*  PONDING TO THE OPSEQ FOR WHICH $OPSEQ_SET IS TO BE BUILT.           */
    DECLARE $TEMP, $OP_ELEMENT  LOCAL;
 /*  LABEL THE ROOT NODE OF $OPSEQ_SET WITH THE NAME OF THIS OPSEQ.      */
    LABEL($OPSEQ_SET) = LABEL($OPSEQ_NODE) ;
 /*  SELECT THE NEXT ELEMENT OF THIS OP-SEQUENCE AND CHECK TO SEE IF    */
 /*  IT IS ANOTHER OP-SEQUENCE OR A SINGLE PROCESS.                     */
    DO FOR ALL SUBNODES OF $OPSEQ_NODE USING $OP_ELEMENT ;
       IF $OP_ELEMENT.TYPE = 'PROCESS'
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        THEN IF $PROCESS.#LABEL($OP_ELEMENT)  IDENTICAL TO  $NULL
                THEN DO ;  KODE = 4 ;  GO TO RETURN_ERROR_CODE ;  END;
/*  CALL 'GENERATE_JOB_SUBSTRUCTURE' WITH THIS PROCESS                        */
/*  AND THEN GRAFT $JOBS AT THE NEXT 'SUBNET ID' NODE OF $OPSEQ_SET.          */
                ELSE DO ;
                  $TEMP= LABEL($OPSEQ_NODE);
                  CALL GENERATE_JOB_SUBSTRUCTURE($TEMP,
                                    $PROCESS.#LABEL($OP_ELEMENT),
                                    $OP_ELEMENT);
                        PRUNE $TEMP;
                        GRAFT $JOBS AT $OPSEQ_SET(NEXT) ;
                  END ;
        ELSE IF $OP_ELEMENT.TYPE ¬= 'OPSEQ'
                THEN DO ;  KODE = 6 ;  GO TO RETURN_ERROR_CODE ;  END;
                ELSE IF $OPSEQ.#LABEL($OP_ELEMENT)(FIRST)
                                                   IDENTICAL TO  $NULL
                     THEN DO ;  KODE = 5 ;
                                GO TO RETURN_ERROR_CODE ;
                          END ;
                     ELSE DO ;
/*  CALL 'GENERATE_OPSEQ_SET' WITH THIS OP-SEQUENCE AND THEN CALL            */
/*  'TEMPORAL_RELATIONS_PROCESSOR' TO REPLACE THE GENERIC PROCESS            */
/*  NAMES WITH SPECIFIC JOB IDENTIFIERS.                                     */
                          CALL GENERATE_OPSEQ_SET
                               ($OPSEQ.#LABEL($OP_ELEMENT),$TEMP) ;
                          CALL TEMPORAL_RELATIONS_PROCESSOR
                                                ($TEMP,$OPSEQ) ;
/*  GRAFT THE TREE RETURNED BY THE LAST CALL OF THIS PROCEDURE AS            */
/*  THE NEXT SUBNODE OF $OPSEQ_SET.                                          */
                          GRAFT $TEMP AT $OPSEQ_SET(NEXT) ;
                          END ;
      END ;
END GENERATE_OPSEQ_SET ;




GENERATE_JOB_SUBSTRUCTURE:
/*  THIS PROCEDURE BUILDS THE $JOBS TREE STRUCTURE REPRESENTING THE          */
/*  OCCURRENCE OF A SINGLE PROCESS OR ONE OF ITS ALTERNATIVES.               */
/*  $OPSEQ_NAME IS AN INPUT TREE WHOSE VALUE IS THE NAME OF THE OP-          */
/*  SEQUENCE TO WHICH THE CURRENT JOB BELONGS.  $PROC_NODE IS THE            */
/*  FIRST-LEVEL SUBSTRUCTURE OF $PROCESS CORRESPONDING TO THE                */
/*  PROCESS FOR WHICH $JOBS IS TO BE BUILT.                                  */
PROCEDURE ($OPSEQ_NAME, $PROC_NODE, $OPS_ELE) RECURSIVE ;
   DECLARE $ALTERNATIVE,$COMBINATIONS,$JOB_ALTERNATIVES,
              $JOB_ALTERNATIVE_SET,$OPTION,$OPTIONS,$REQUIRED,
              $DUMMY1, $DUMMY2    LOCAL;
/*  DUPLICATE THE APPROPRIATE NODE IN $PROCESS AS $JOB AND ASSIGN A          */
/*  UNIQUE JOB ID. NUMBER TO THIS JOB.                                       */
```

```
   $JOB = $PROC_NODE ;
   $JOB.PROCESS = LABEL($JOB) ;
   LABEL($JOB) = INITIAL_ID ;  INITIAL_ID = INITIAL_ID + 1 ;
/* IF NEEDED, CREATE 'JOB_TYPE', 'JOB_INTERVAL', 'PROBLEM_NAME', AND */
/*  'OPSEQ' NODES IN $JOB WITH APPROPRIATE SUBSTRUCTURE AND VALUES.  */
   IF $JOB.PROCESS_TYPE  NOT IDENTICAL TO  $NULL
      THEN LABEL($JOB.PROCESS_TYPE) = 'JOB_TYPE' ;
   IF $JOB.DURATION  IDENTICAL TO  $NULL
      THEN DO ;  KODE = 7 ;  GO TO RETURN_ERROR_CODE ;  END ;
   START_TIME = INFINITY ;
   IF $OBJECT_ELEMENT.SCHEDULE_TIME.START  NOT IDENTICAL TO  $NULL
      THEN IF $OPSEQ_NAME  IDENTICAL TO  $NULL
              THEN START_TIME = $OBJECT_ELEMENT.SCHEDULE_TIME.START ;
      ELSE DO;
       LABEL($DUMMY1) = $OBJECT_ELEMENT.SCHEDULE_TIME.PROCESS_NAME;
       LABEL($DUMMY2) = $JOB.PROCESS;
       IF LABEL($DUMMY1)= LABEL($DUMMY2)
              THEN START_TIME = $OBJECT_ELEMENT.SCHEDULE_TIME.START ;
      END;
              ELSE ;
   IF START_TIME ¬= INFINITY
      THEN DO ;  $JOB.JOB_INTERVAL.START = START_TIME ;
                 $JOB.JOB_INTERVAL.END = START_TIME + $JOB.DURATION ;
           END ;
   IF $OBJECTIVES.PROBLEM_NAME  NOT IDENTICAL TO  $NULL
      THEN $JOB.PROBLEM_NAME = $OBJECTIVES.PROBLEM_NAME ;
   IF $OPSEQ_NAME  NOT IDENTICAL TO  $NULL
      THEN $JOB.OPSEQ = $OPSEQ_NAME ;
/*  IF THERE ARE RESOURCE ALTERNATIVES TO BE CONSIDERED, CALL          */
/*  'ENUMERATE_RESOURCE_OPTIONS' TO GENERATE JOB NODES FOR ALL         */
/*  POSSIBLE RESOURCE COMBINATIONS.  OTHERWISE, INSERT $JOB AS THE     */
/*  NEXT SUBNODE OF $JOBS.                                             */
   IF $JOB.RESOURCE_ALTERNATIVE(FIRST) IDENTICAL TO  $NULL
      THEN IF I_RECURSIVE_CALL_FLAG = 0
              THEN GRAFT INSERT $JOB BEFORE $JOBS(FIRST);
              ELSE GRAFT $JOB AT $JOBS(NEXT) ;
      ELSE DO ;  K = 1 ;  INITIAL_ID = INITIAL_ID - 1 ;
           GRAFT $JOB.RESOURCE AT $REQUIRED ;
           GRAFT $JOB.RESOURCE_ALTERNATIVE AT $OPTIONS ;
           N = NUMBER($OPTIONS) ;
           CALL ENUMERATE_RESOURCE_OPTIONS ;
           PRUNE $JOB,$REQUIRED,$OPTIONS,$WORKSPACE,$COMBINATIONS ;
           END ;
/*  IF THIS $JOB IS BEING BUILT AS A JOB ALTERNATIVE, RETURN.          */
/*  OTHERWISE, CREATE A $JOB_ALTERNATIVES TREE USING INFORMATION       */
/*  FROM $OBJECTIVES OR $OPSEQ.                                        */
   IF I_RECURSIVE_CALL_FLAG ¬= 0  THEN DO;
              I_RECURSIVE_CALL_FLAG = 0 ;
      RETURN;
      END;
   IF $OPSEQ_NAME  IDENTICAL TO  $NULL
```

```
      THEN IF $OBJECT_ELEMENT.ALTERNATIVE(FIRST) NOT IDENTICAL TO $NULL
              THEN $JOB_ALTERNATIVES = $OBJECT_ELEMENT.ALTERNATIVE ;
              ELSE ;
      ELSE IF $OPS_ELE.   ALTERNATIVE(FIRST) NOT IDENTICAL TO  $NULL
              THEN $JOB_ALTERNATIVES = $OPS_ELE.   ALTERNATIVE ;
   IF $JOB_ALTERNATIVES  NOT IDENTICAL  TO  $NULL
      THEN DO;
/*  SELECT THE NEXT SUBNODE OF $JOB_ALTERNATIVES.                            */
           DO FOR ALL SUBNODES OF $JOB_ALTERNATIVES USING $ALTERNATIVE;
              I_RECURSIVE_CALL_FLAG=1;
              IF $PROCESS.#($ALTERNATIVE)  IDENTICAL TO  $NULL
                 THEN DO ;  KODE = 9 ;  GO TO RETURN_ERROR_CODE ;  END;
                 ELSE CALL GENERATE_JOB_SUBSTRUCTURE($OPSEQ_NAME,
                                              $PROCESS.#($ALTERNATIVE),
                                              $OPS_ELE);
              END ;
           PRUNE $JOB_ALTERNATIVES ;
/*  $JOBS NOW CONTAINS A COMPLETE SET OF JOB ALTERNATIVES.  GENERATE */
/*  ALL 'JOB_ALTERNATIVES' SUBNODES FOR EACH JOB NODE IN $JOBS.      */
   PRUNE $JOB_ALTERNATIVE_SET ;  ISAVE = LABEL($JOBS(FIRST));
   DO K=ISAVE TO INITIAL_ID-1 ;
      $JOB_ALTERNATIVE_SET(NEXT) = K ;
      END ;
   DO K=1 TO INITIAL_ID-ISAVE ;
      GRAFT $JOB_ALTERNATIVE_SET(FIRST) AT $TEMP_NODE ;
      $JOBS(K).JOB_ALTERNATIVE = $JOB_ALTERNATIVE_SET ;
      GRAFT $TEMP_NODE AT $JOB_ALTERNATIVE_SET(NEXT) ;
      END ;
           END ;



ENUMERATE_RESOURCE_OPTIONS:  PROCEDURE RECURSIVE ;
/*  THIS PROCEDURE GENERATES A $JOB TREE STRUCTURE FOR EVERY POS-    */
/*  SIBLE COMBINATION OF RESOURCE ALTERNATIVES.  IT CALLS ITSELF RE- */
/*  CURSIVELY, ONCE FOR EACH FIRST-LEVEL SUBNODE OF $OPTIONS.        */
/*                                                                   */
/*  SELECT A NEW RESOURCE ALTERNATIVE FROM THE NEXT SET OF OPTIONS,  */
/*  ADD IT TO $COMBINATIONS AND CHECK TO SEE IF $COMBINATIONS NOW    */
/*  CONTAINS ONE ALTERNATIVE FROM EACH SET OF OPTIONS.               */
 DECLARE $OPTION  LOCAL;
   DO FOR ALL SUBNODES OF $OPTIONS(K) USING $OPTION ;
      $COMBINATIONS(K) = $OPTION ;
      IF K ¬= N
         THEN DO ;  K = K+1 ;  CALL ENUMERATE_RESOURCE_OPTIONS ;
  K=K-1;  END;
/*  ADD THIS SET OF ALTERNATIVES TO THE ORIGINAL                     */
/*  'REQUIRED_RESOURCES' NODE OF $JOB.                               */
         ELSE DO ;  $WORKSPACE = $REQUIRED ;
```

```
            DO FOR ALL SUBNODES OF $COMBINATIONS USING $SELECTION ;
               DO FOR ALL SUBNODES OF $SELECTION USING $TYPE ;
                  IF $WORKSPACE.#LABEL($TYPE)  IDENTICAL TO  $NULL
                     THEN $WORKSPACE(NEXT) = $TYPE ;
                     ELSE DO FOR ALL SUBNODES OF $TYPE USING $NAME ;
                             $WORKSPACE.#LABEL($TYPE)(NEXT) = $NAME ;
                             END ;
                  END ;
               END ;
            GRAFT $WORKSPACE AT $JOB.RESOURCE ;
/*  ASSIGN A UNIQUE JOB ID. NUMBER TO THIS "NEW" $JOB AND DUPLICATE   */
/*  IT AS THE NEXT SUBNODE OF $JOBS.                                   */
            LABEL($JOB) = INITIAL_ID ;
            $JOBS(NEXT) = $JOB ;
            INITIAL_ID = INITIAL_ID + 1 ;
            END ;
      END ;
END ENUMERATE_RESOURCE_OPTIONS ;
END GENERATE_JOB_SUBSTRUCTURE ;




TEMPORAL_RELATIONS_PROCESSOR:  PROCEDURE ($ACTIVITY, $REF) ;
/*  THIS PROCEDURE CONTAINS THE EXECUTIVE LOGIC WHICH REPEATATIVELY  */
/*  BUILDS AND SCANS $TEMPORAL_RELATIONS, ONCE FOR EACH FIRST-LEVEL  */
/*  SUBNODE OF THE INPUT PARAMETER, $ACTIVITY.  $REF IDENTIFIES THE  */
/*  REFERENCE TREE FROM WHICH THE 'TEMPORAL_RELATIONS' SUBNODE IS TO */
/*  BE TAKEN WHEN CREATING $TEMPORAL_RELATIONS.                      */
   DECLARE $JOB_NODE,$SUB_ACTIVITY,$TEMP,$TEMPORAL_RELATIONS LOCAL ;
DECLARE $RESOURCE_RELATIONS, $RES_REL, $POINT  LOCAL;
   K = 0 ;
/*  SELECT THE NEXT SUBNODE OF THE INPUT TREE ($ACTIVITY) AND BUILD  */
/*  A $TEMPORAL_RELATIONS TREE CONTAINING UNIQUE JOB IDENTIFIERS.    */
   DO FOR ALL SUBNODES OF $ACTIVITY USING $SUB_ACTIVITY ;
      K = K+1 ;
      $TEMPORAL_RELATIONS=$REF.#LABEL($ACTIVITY)(K).TEMPORAL_RELATION;
/*  THESE THREE STATEMENTS CALL 'GENERIC_NAME_ELIMINATOR' FOR EACH   */
/*  'PREDECESSOR', 'SUCCESSOR', AND 'GENERAL' SUBNODE OF $TEMPORAL_  */
/*  RELATIONS.  ANY OTHER SUBNODES OF THIS TREE ARE IGNORED.         */
         CALL GENERIC_NAME_ELIMINATOR($TEMPORAL_RELATIONS.PREDECESSOR);
         CALL GENERIC_NAME_ELIMINATOR($TEMPORAL_RELATIONS.SUCCESSOR) ;
         CALL GENERIC_NAME_ELIMINATOR($TEMPORAL_RELATIONS.GENERAL) ;
      IF LABEL($SUB_ACTIVITY) = ''
         THEN DO FOR ALL SUBNODES OF $SUB_ACTIVITY USING $JOB_NODE ;
           IF $TEMPORAL_RELATIONS(FIRST) ¬ IDENTICAL TO $NULL
            THEN $JOB_NODE.TEMPORAL_RELATION = $TEMPORAL_RELATIONS ;
                END ;
/*  COMBINE $TEMPORAL_RELATIONS WITH THE TEMPORAL RELATIONS INFO OF  */
/*  THE DESCENDENT JOBS OF THIS SUBNODE.                              */
```

```
      ELSE DO FOR ALL SUBNODES OF $SUB_ACTIVITY USING $JOB_NODE ;
        IF $TEMPORAL_RELATIONS(FIRST) ¬ IDENTICAL TO $NULL
           THEN DO;
              CALL GENERATE_SUBNODES($TEMPORAL_RELATIONS.
               PREDECESSOR,$JOB_NODE.TEMPORAL_RELATION.PREDECESSOR);
              CALL GENERATE_SUBNODES($TEMPORAL_RELATIONS.SUCCESSOR,
                            $JOB_NODE.TEMPORAL_RELATION.SUCCESSOR) ;
              CALL GENERATE_SUBNODES($TEMPORAL_RELATIONS.GENERAL,
                              $JOB_NODE.TEMPORAL_RELATION.GENERAL) ;
           END;
              END ;
    END ;
/*  MERGE ALL OF THE DESCENDENT JOB NODES UPWARDS, ELIMINATING ALL     */
/*  OF THE FIRST-LEVEL SUBNODES OF $ACTIVITY.                          */
   IF $REF  NOT IDENTICAL TO  $OBJECTIVES
      THEN DO ;
          LABEL($TEMP) = LABEL($ACTIVITY);
          DO L=NUMBER($ACTIVITY) TO 1 BY -1 ;
   PRUNE $RESOURCE_RELATIONS;
   $RESOURCE_RELATIONS = $REF.#LABEL($ACTIVITY)(L).RESOURCE_RELATION;
   IF $RESOURCE_RELATIONS(FIRST) NOT IDENTICAL TO $NULL
     THEN DO;
       DO FOR ALL SUBNODES OF $RESOURCE_RELATIONS USING $RES_REL;
         DO FOR ALL SUBNODES OF $ACTIVITY(L) USING $POINT;
           INSERT $RES_REL BEFORE $POINT.RESOURCE_RELATION(FIRST);
         END;
       END;
     END;
           IF LABEL($ACTIVITY(L)) ¬= ''
             THEN DO;
              DO M=NUMBER($ACTIVITY(L)) TO 1 BY -1;
               GRAFT INSERT $ACTIVITY(L)(M) BEFORE $TEMP(FIRST);
              END;
          END;
             ELSE DO;
              DO M=NUMBER($ACTIVITY(L)) TO 1 BY -1 ;
                 GRAFT INSERT $ACTIVITY(L)(M) BEFORE $TEMP(FIRST);
                 END ;
             END;
              END ;
           GRAFT $TEMP AT $ACTIVITY ;
           END ;


GENERATE_SUBNODES:  PROCEDURE ($SOURCE, $TARGET) ;
/*  THIS PROCEDURE DUPLICATES THE SUBNODES OF $SOURCE AS SUBNODES OF */
/*  $TARGET BY 'INSERT'-ING EACH ONE (WITH ITS SUBSTRUCTURE) BEFORE  */
/*  THE FIRST SUBNODE OF $TARGET, ONE LEVEL BELOW THE ROOT NODE.     */
```

```
IF $SOURCE(FIRST) IDENTICAL TO $NULL & $TARGET(FIRST) IDENTICAL TO
        $NULL
   THEN DO;
     PRUNE $SOURCE;
     PRUNE $TARGET;
     RETURN;
   END;
   DO M=NUMBER($SOURCE) TO 1 BY -1 ;
      INSERT $SOURCE(M) BEFORE $TARGET(FIRST);
      END ;
END GENERATE_SUBNODES ;




GENERIC_NAME_ELIMINATOR:  PROCEDURE ($TEMPORAL_NODE) ;
/*  THIS PROCEDURE IS RESPONSIBLE FOR ELIMINATING THE GENERIC          */
/*  PROCESS OR OPSEQ NAMES THAT APPEAR BELOW $TEMPORAL_NODE BY         */
/*  REPLACING THEM WITH SPECIFIC JOB IDENTIFIERS.  $TEMPORAL_NODE      */
/*  WILL ALWAYS BE ONE OF THE THREE STANDARD SUBNODES OF $TEMPORAL_    */
/*  RELATIONS:  'PREDECESSOR', 'SUCCESSOR', OR 'GENERAL'.              */
   DECLARE $SUBNET, $DUM1, $DUM2, $SUB, L LOCAL;
   IF $TEMPORAL_NODE(FIRST) IDENTICAL TO $NULL
      THEN DO;
         PRUNE $TEMPORAL_NODE;
         RETURN;
         END;
   DO L = NUMBER($TEMPORAL_NODE) TO 1 BY -1;
    DEFINE $RELATION AS $TEMPORAL_NODE(L);
      IF LABEL($TEMPORAL_NODE) = 'GENERAL'
         THEN DEFINE $PROCESS_ID AS $RELATION(4) ;
         ELSE DEFINE $PROCESS_ID AS $RELATION ;
    $SUBNET = $ACTIVITY(FIRST: (LABEL($ELEMENT) = $PROCESS_ID));
/*  IS THE GENERIC NAME TO BE REPLACED THE NAME OF A PROCESS?          */
      IF $SUBNET  IDENTICAL TO  $NULL
         THEN DO ;
              DO FOR ALL SUBNODES OF $ACTIVITY USING $SUB;
/*  REPLACE THE GENERIC NAME WITH THE JOB ID. NUMBER CORRESPONDING     */
/*  TO THIS PROCESS.                                                   */
                 LABEL($DUM1) = $PROCESS_ID;
                 LABEL($DUM2) = $SUB(FIRST).PROCESS;
                 IF LABEL($DUM1) = LABEL($DUM2)
                    THEN DO ;
                         $PROCESS_ID = LABEL($SUB(FIRST));
                         GO TO NEXT_NODE;
                         END ;
                 END ;
              KODE = 10 ;  GO TO RETURN_ERROR_CODE ;
              END ;
/*  DETERMINE THE JOB ID. NUMBER OF EACH JOB IN THIS OP-SEQUENCE       */
```

```
/*   AND GENERATE NEW TEMPORAL RELATIONS SUBNODES WITH THE                  */
/*   CORRESPONDING JOB ID. NUMBERS AS THEIR VALUES.                         */
           ELSE DO ;
                IF LABEL($TEMPORAL_NODE) ¬= 'GENERAL'
                   THEN DO FOR ALL SUBNODES OF $SUBNET USING $JOB_NODE ;
                           $TEMPORAL_NODE(NEXT) = LABEL($JOB_NODE) ;
                           END ;
                   ELSE DO FOR ALL SUBNODES OF $SUBNET USING $JOB_NODE ;
                           $PROCESS_ID = LABEL($JOB_NODE) ;
                           $TEMPORAL_NODE(NEXT) = $RELATION ;
                           END ;
                PRUNE $RELATION ;
                END ;
   NEXT_NODE:
       END ;
END GENERIC_NAME_ELIMINATOR ;
END TEMPORAL_RELATIONS_PROCESSOR ;
END GENERATE_JOBSET ;
```

2.4.8.11 COMMENTS ON FUTURE MODIFICATIONS

In its present form GENERATE_JOBSET provides many useful services to the PLANS programmer. Below is a discussion of some needed changes and possible extensions of its scope that will make it an even more powerful module.

It would be desirable to compact the code of this module by combining the main procedure (GENERATE_JOBSET) with the internal procedure called SCAN_OPSEQ_TREE. Since the logic of these two modules is functionally similar, it appears that this change could be accomplished without too much difficulty. The major problem in combining the two procedures arises from the fact that one of them is recursive.

Another needed change to this module is to include processing of the information found below the RESOURCES_GENERATED and RESOURCES_DELETED nodes of $PROCESS. All of the resource data for a given job should be available beneath the RESOURCE node of $JOBSET. This allows for easier access of information by NEXTSET and several other modules.

Also of major importance is a problem with the "brute-force" method that has been used to resolve temporal relations with an op-sequence. Presently, if a job is encountered that has an op-sequence as its predecessor, GENERATE_JOBSET simply lists all of the jobs contained in the op-sequence as its predecessors. Although this method is functionally sufficient, it is unsatisfactory since it results in the generation of many unneeded nodes. It may be necessary to call REDUNDANT_PREDECESSOR_CHECKER, PREDECESSOR_SET_INVERTER, and/or ORDER_BY_PREDECESSORS in order to alleviate this problem.

Another much-needed change is to provide GENERATE_JOBSET with some additional logic to recognize a process or opseq name for which it has already built appropriate structures in $JOBSET. When such a reoccurring activity is encountered its JOB ID nodes can be built simply by duplicating the previously generated nodes and making some minor changes. This change would greatly increase the efficiency of the module, despite the fact that it will lengthen the code.

Currently, GENERATE_JOBSET creates $JOBSET so that its first-level substructure is identical to that of $OBJECTIVES.OPSEQ. That is, each SUBNET ID node corresponds directly to a first-level subnode of $OBJECTIVES.OPSEQ. All of the descendent job nodes will be located one level below the SUBNET ID node. This approach was used mainly because it greatly simplifies the logic of the TEMPORAL_RELATIONS_PROCESSOR procedure. A possible alternative approach is to create a new SUBNET ID node for each op-sequence encountered. This would require some extra logic to process nested op-sequences, but it would insure that each job would appear in $JOBSET as a direct descendent of its "father" op-sequence. Again, the major drawback of this approach is the difficulty it creates in resolving temporal relations. However, it allows for a greater degree of problem decomposition by the project scheduling modules. This would facilitate a more efficient solution of the scheduling problem.

Some possible changes of questionable desirability are listed below. At this point, it has been determined that more time and data are needed to properly assess their usefulness and feasibility.

(1) allow the user to specify the number of JOB ID nodes to be generated

(2) (contingent on #1) provide the restart capability needed in order to allow the user to input a partially built $JOBSET.

(3) write out a warning message if a generic resource is encountered in $OBJECTIVES.ASSOCIATED_RESOURCES

(4) allow the user to specify alternative op-sequences in $OPSEQ and/or $OBJECTIVES.

2.4.9 EXTERNAL_TEMP_RELATIONS

2.4.9 EXTERNAL_TEMP_RELATIONS

2.4.9.1 Purpose and Scope

This module will determine the temporal relations specified for the jobs under a single subnode of $JOBSET which will be violated by merging two partial schedules each of which consists of one or more schedule units (jobs). This single subnode and its substructure will be passed through the argument list as $SUBNET.

This module will identify violations of temporal relations that result from the association of the two partial schedules. It is not necessary that either or both partial schedules be free of internal temporal constraint violations; if such violations are present however, they will not be detected by this module unless a violation results from the association of a job in the first partial schedule with one or more jobs in the second partial schedule. It is thus possible to call the module with partial schedules that are temporally infeasible.

The module will build an output tree containing a first-level node for each identified violation of a temporal relation. If redundant temporal relations are specified in $JOBSET then redundant nodes will appear in the output tree. (e.g., A < B, B > A is a redundant specification). For each such node the identifiers of the conflicting jobs, the respective job intervals, and the violated temporal relation will be recorded.

2.4.9.2 Modules Called

ELEMENTARY_TEMP_RELATIONS

2.4.9.3 Module Input

This module will be called with four arguments. There are three input arguments: $SUBNET, $SCHED1, and $SCHED2. The structure of $SUBNET is identical to a single, first-level subnode of the structure output from the module GENERATE_JOBSET.

The structure of the two partial schedules to be examined for temporal constraint consistency have the standard schedule unit (job) structure, and are equivalent to first-level subnodes of $SCHEDULE.

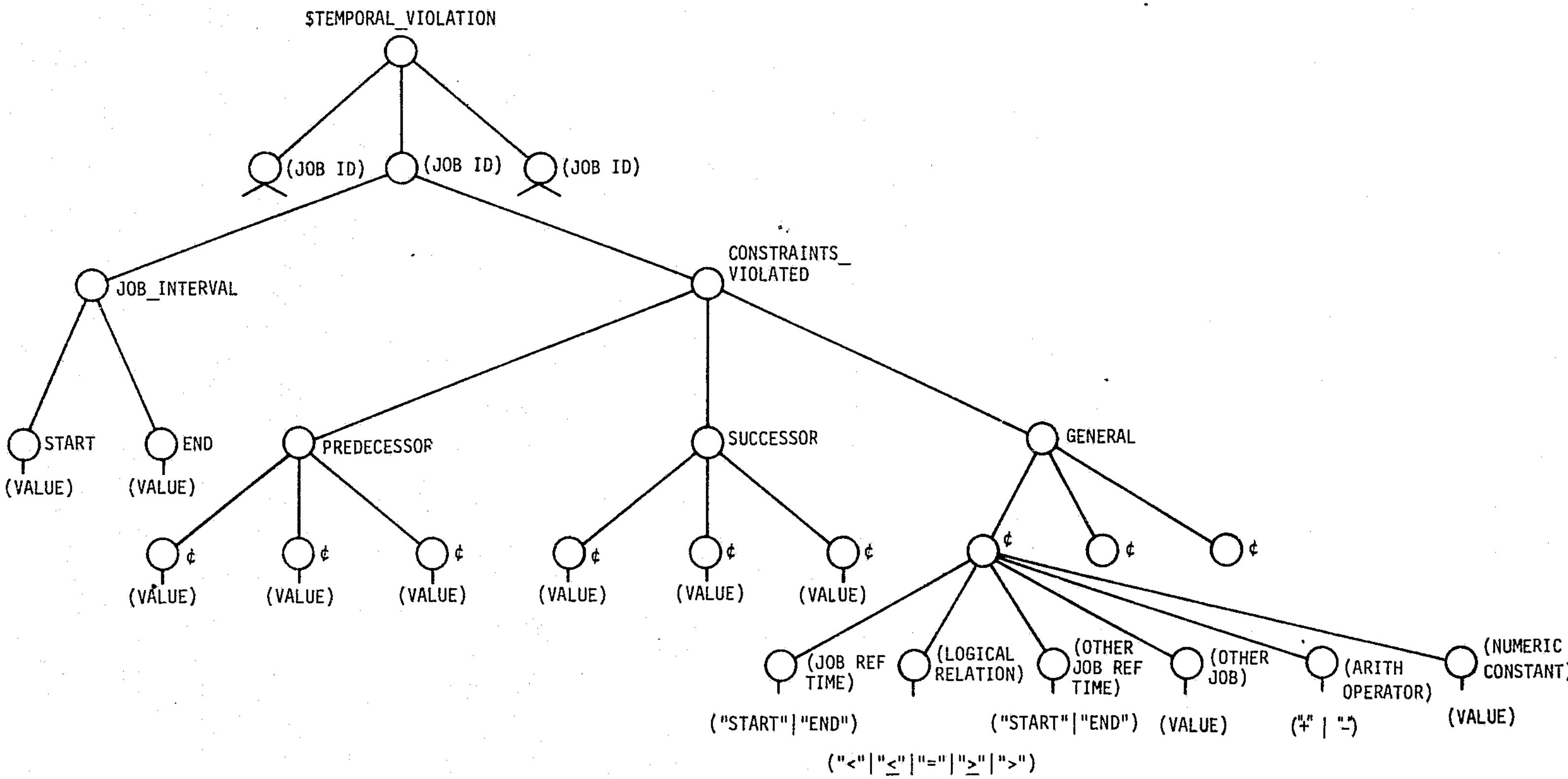
The minimum data structures required from the standard structures $SUBNET, $SCHED1, and $SCHED2 are shown in Fig. 2.4.9-1. Note that in the minimum structure the fifth and sixth subnodes of a relation in the TEMPORAL_RELATION substructure are not mandatory in every case.

2.4.9.4 Module Output

This module will build and return an output tree with the structure shown on the following page.

Each node of $TEMPORAL_VIOLATION will correspond to a violation of a temporal relation in $SUBNET (input) that results only from the association of $SCHED1 and $SCHED2.

OUTPUT DATA STRUCTURE



Note: Minimum (i.e., relevant) portion of the required input standard data structures is shown. In all trees, any additional structure will be preserved.

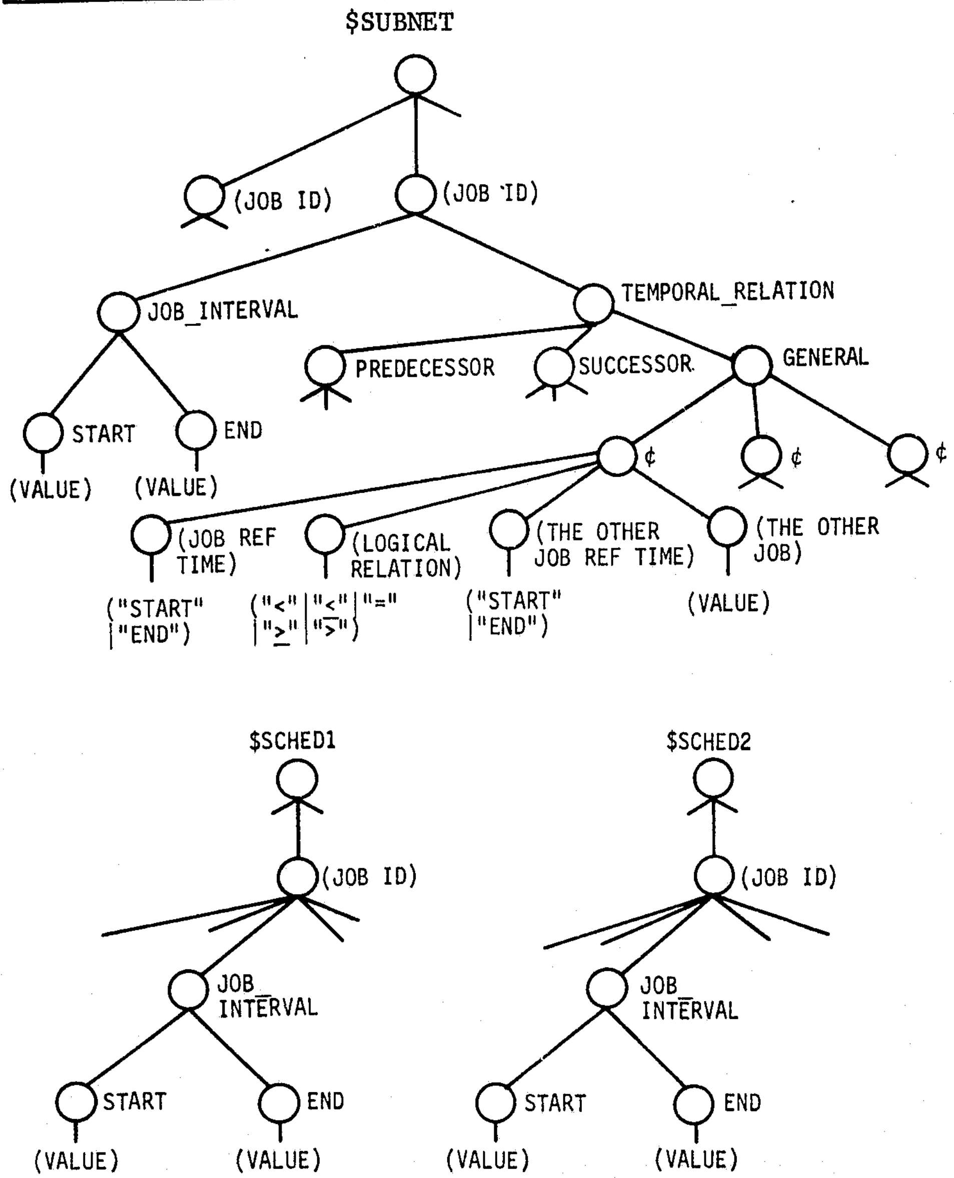
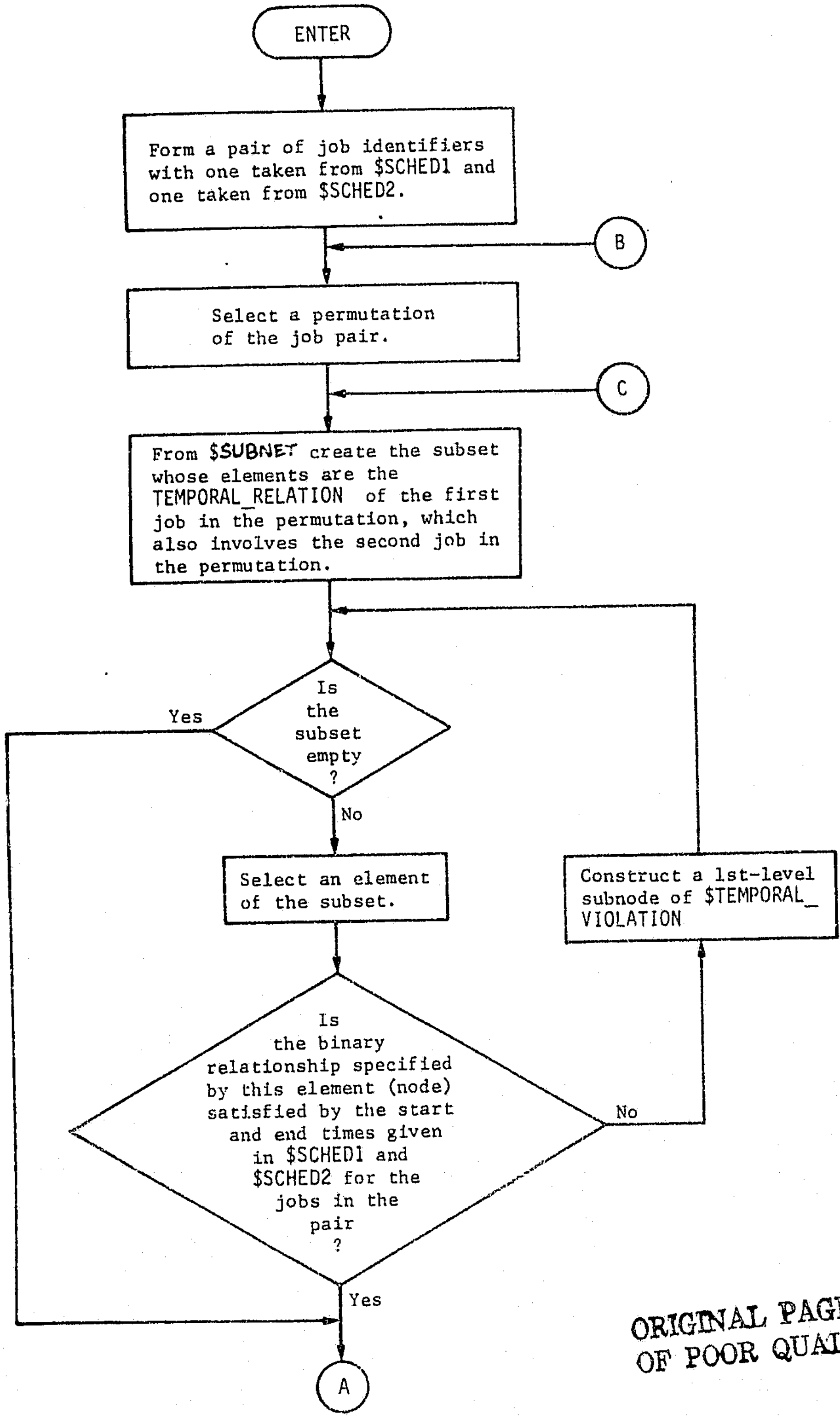
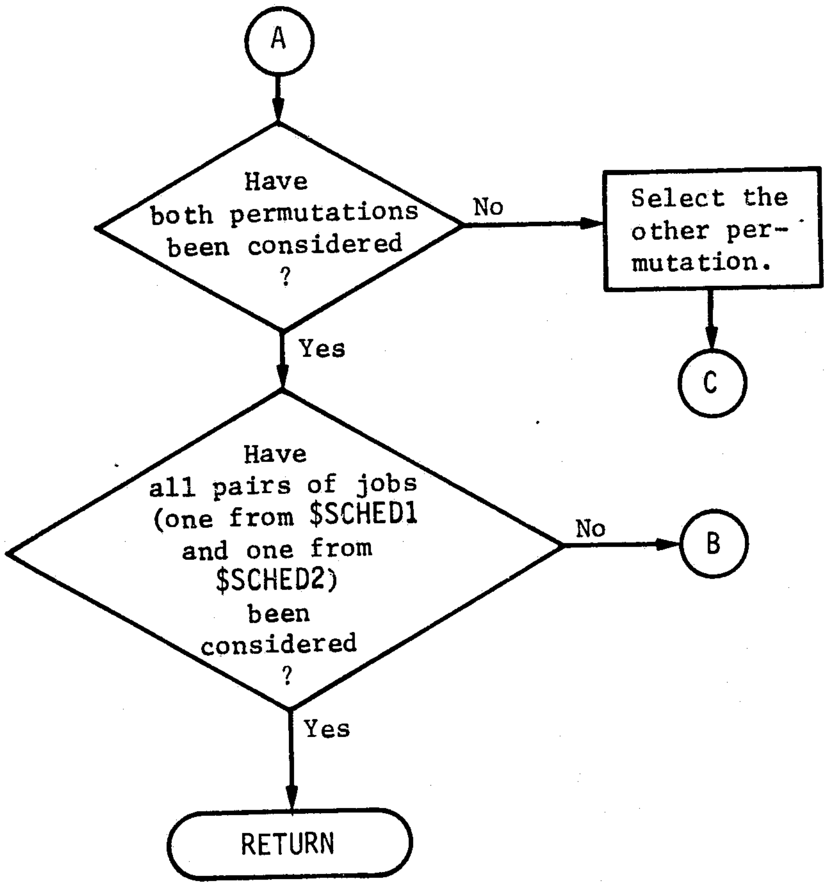


Fig. 2.4.9-1
Minimum Required Input Structures from Standard Data Structures for Module:
EXTERNAL_TEMP_RELATIONS

2.4.9.5 Functional Block Diagram



ORIGINAL PAGE IS OF POOR QUALITY


A
Have both permutations been considered ?
No
Select the other permutation.
C
Yes
Have all pairs of jobs (one from $SCHED1 and one from $SCHED2) been considered ?
No
B
Yes
RETURN

2.4.9.6 Detailed Design

From each of the two partial schedules a single job is taken. One job is selected and examined to see if the other job is a predecessor, successor, or generally related temporally to the first. If not the other job is selected and examined. When a temporal relationship is identified, the module ELEMENTARY_TEMP_RELATION is called to determine the satisfaction of the relationship. If the relationship is not satisfied a node is built to identify the temporal violation. The other job is then selected or examined, or another job pair is identified and the procedure repeated until all job pairs have been checked.

2.4.9.7 Internal Variable and Tree Names

I, J, K - index pointers

$JOB_1 - job taken from one of the partial schedules

$JOB_2 - job taken from the other partial schedule

$RELATION - temporal relationship between the two jobs

$RESULT - returns from ELEMENTARY_TEMP_RELATIONS as a flag to indicate the existance of temporal relations between JOB_1 and JOB_2

$SUB_JOB - points to all subnodes of JOB_1 and JOB_2

2.4.9.8 Modifications to Functional Specifications and/or Standard Data Structures Assumed

$SCHED1 and $SCHED2 are equivalent to first-level subnodes of $SCHEDULE.

2.4.9.9 COMMENTED CODE

```
EXTERNAL_TEMP_RELATIONS: PROCEDURE($SUBNET,$SCHED_1,$SCHED_2,
   $TEMPORAL_VIOLATION) OPTIONS(EXTERNAL);

DECLARE                      $SUB_JOB LOCAL;
DECLARE I,J,K,$JOB_1,$JOB_2,$RELATION LOCAL;
/*  FORM A PAIR OF JOB IDENTIFIERS WITH ONE TAKEN FROM $SCHED1 AND    */
/*   ONE TAKEN FROM $SCHED2                                           */
DO I = 1 TO NUMBER($SCHED_1);
LABEL($JOB_1) = LABEL($SCHED_1(I));
$JOB_1.JOB_INTERVAL = $SCHED_1(I).JOB_INTERVAL;
   DO J = 1 TO NUMBER($SCHED_2);
   LABEL($JOB_2) = LABEL($SCHED_2(J));
   $JOB_2.JOB_INTERVAL = $SCHED_2(J).JOB_INTERVAL;
/*  SELECT A PERMUTATION OF THE JOB PAIR                              */
/*  FROM $JOBSET, CREATE THE SUBSET WHOSE ELEMENTS ARE THE            */
/*   TEMPORAL_RELATION OF THE FIRST JOB IN THE PERMUTATION, WHICH     */
/*   ALSO INVOLVES THE SECOND JOB OF THE PERMUTATION                  */
   PRUNE $RELATION;
   IF LABEL($SCHED_2(J)) ELEMENT OF $SUBNET.#LABEL($JOB_1).
      TEMPORAL_RELATION.PREDECESSOR
      THEN DO;
         $RELATION(FIRST) = LABEL($SCHED_2(J));
         LABEL($RELATION) = 'PREDECESSOR';
         END;
      ELSE IF LABEL($SCHED_2(J)) ELEMENT OF $SUBNET.#LABEL($JOB_1).
         TEMPORAL_RELATION.SUCCESSOR
         THEN DO;
            $RELATION(FIRST) = LABEL($SCHED_2(J));
            LABEL($RELATION) = 'SUCCESSOR';
            END;
         ELSE DO K = 1 TO NUMBER($SUBNET.#LABEL($JOB_1).
            TEMPORAL_RELATION.GENERAL);
            IF LABEL($SCHED_2(J)) = $SUBNET.#LABEL($JOB_1).
               TEMPORAL_RELATION.GENERAL(K)(4)
               THEN $RELATION = $SUBNET.#LABEL($JOB_1).
                  TEMPORAL_RELATION.GENERAL(K);
               ELSE;
            END;
/*  IS THE SUBSET EMPTY?                                              */
   IF $RELATION IDENTICAL TO $NULL THEN GO TO NEXT_PERM;
/*  SELECT AN ELEMENT OF THE SUBSET                                   */
   CALL ELEMENTARY_TEMP_RELATIONS($JOB_1,$JOB_2,$RELATION,$RESULT);
/*  IS THE BINARY RELATIONSHIP SPECIFIED BY THIS ELEMENT (NODE)       */
/*   SATISFIED BY THE START AND END TIMES GIVEN IN $SCHED1 AND        */
/*   $SCHED2 FOR THE JOBS IN THE PAIR?                                */
   IF $RESULT.SATISFIED = 'NO'
/*  CONSTRUCT A FIRST-LEVEL SUBNODE OF $TEMPORAL_VIOLATIONS           */
      THEN DO;
         DO FOR ALL SUBNODES OF $JOB_1 USING $SUB_JOB;
         IF $SUB_JOB SUBSET OF $TEMPORAL_VIOLATION.#LABEL($JOB_1)
            THEN;
```

```
            ELSE $TEMPORAL_VIOLATION.#LABEL($JOB_1)(NEXT) = $SUB_JOB;
         END;
         IF (LABEL($RELATION) = 'PREDECESSOR' | LABEL($RELATION) =
            'SUCCESSOR')
            THEN $TEMPORAL_VIOLATION.#LABEL($JOB_1).CONSTRAINT_VIOLATED
                  .#LABEL($RELATION)(NEXT) = LABEL($JOB_2);
            ELSE $TEMPORAL_VIOLATION.#LABEL($JOB_1).CONSTRAINT_VIOLATED
                  .GENERAL(NEXT) = $RELATION;
         END;
       ELSE;
/*  SELECT THE OTHER PERMUTATION                                          */
    NEXT_PERM:
    PRUNE $RELATION;
    IF LABEL($SCHED_1(I)) ELEMENT OF $SUBNET.#LABEL($JOB_2).
       TEMPORAL_RELATION.PREDECESSOR
       THEN DO;
          $RELATION(FIRST) = LABEL($SCHED_1(I));
          LABEL($RELATION) = 'PREDECESSOR';
          END;
       ELSE IF LABEL($SCHED_1(I)) ELEMENT OF $SUBNET.#LABEL($JOB_2).
          TEMPORAL_VIOLATION.SUCCESSOR
          THEN DO;
             $RELATION(FIRST) = LABEL($SCHED_1(I));
             LABEL($RELATION) = 'SUCCESSOR';
             END;
          ELSE DO K = 1 TO NUMBER($SUBNET.#LABEL($JOB_2).
             TEMPORAL_RELATION.GENERAL);
             IF LABEL($SCHED_1(I)) = $SUBNET.#LABEL($JOB_2).
                TEMPORAL_RELATION.GENERAL(K)(4)
                THEN $RELATION = $SUBNET.#LABEL($JOB_2).
                   TEMPORAL_RELATION.GENERAL(K);
                ELSE;
             END;
     IF $RELATION IDENTICAL TO $NULL THEN GO TO NEXT_PAIR;
     CALL ELEMENTARY_TEMP_RELATIONS($JOB_2,$JOB_1,$RELATION,$RESULT);
     IF $RESULT.SATISFIED ='NO'
        THEN DO;
           DO FOR ALL SUBNODES OF $JOB_2 USING $SUB_JOB;
           IF $SUB_JOB SUBSET OF $TEMPORAL_VIOLATION.#LABEL($JOB_2)
              THEN;
              ELSE $TEMPORAL_VIOLATION.#LABEL($JOB_2)(NEXT) = $SUB_JOB;
           END;
           IF (LABEL($RELATION) = 'PREDECESSOR' | LABEL($RELATION) =
              'SUCCESSOR')
              THEN $TEMPORAL_VIOLATION.#LABEL($JOB_2).CONSTRAINT_VIOLATED
                    .#LABEL($RELATION)(NEXT) = LABEL($JOB_1);
              ELSE $TEMPORAL_VIOLATION.#LABEL($JOB_2).CONSTRAINT_VIOLATED
                    .GENERAL(NEXT) = $RELATION;
           END;
        ELSE;
 /*  HAVE ALL PAIRS OF JOBS BEEN CONSIDERED?                              */
     NEXT_PAIR:
     END;
   END;

   END EXTERNAL_TEMP_RELATIONS;
```

2.4.10 INTERNAL_TEMP_RELATIONS

2.4.10 INTERNAL_TEMP_RELATIONS

2.4.10 INTERNAL_TEMP_RELATIONS

2.4.10.1 Purpose and Scope

This module will determine the temporal relations specified for jobs under a single subnode of $JOBSET (i.e. $SUBNET) that are violated within a single partial schedule that has two or more jobs.

Unlike EXTERNAL_TEMP_RELATIONS, this module will identify all violations of temporal relations that exist within a single tree containing several schedule units. The module will build an output tree containing a first-level node for each identified violation of a temporal relation. Identifiers of the conflicting jobs, the identifiers of the violated temporal relations and the interval of the violation will be recorded for each such node.

2.4.10.2 Modules Called

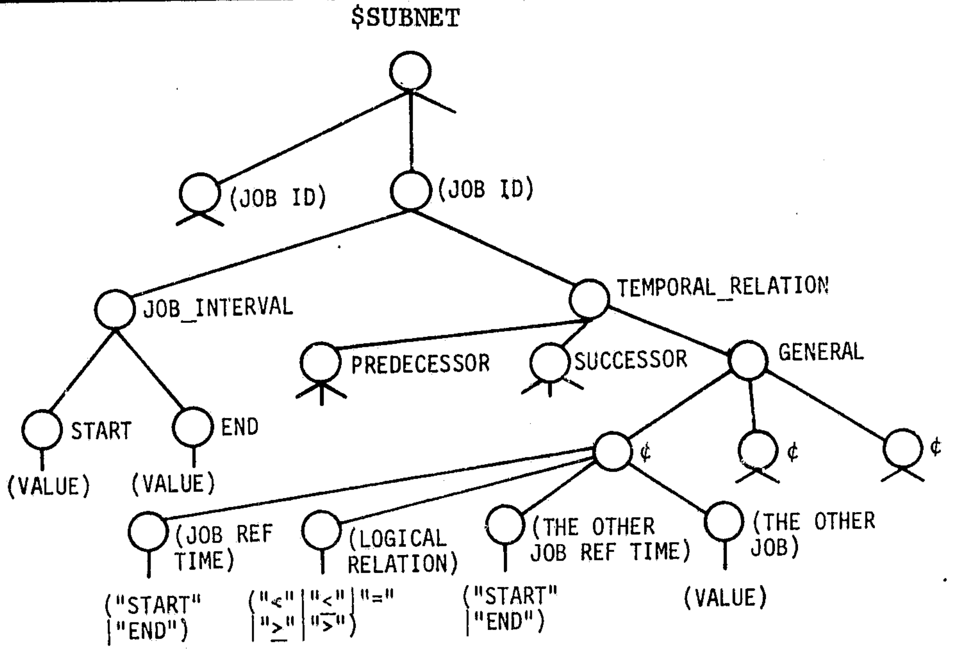
ELEMENTARY_TEMP_RELATIONS

2.4.10.3 Module Input

This module will be called with three arguments. There are two input arguments: $SUBNET and $SCHED. The structure of $SUBNET is identical to a single, first-level subnode of the structure output from the module GENERATE_JOBSET. The structure of $SCHED is that of the standard schedule unit (job), and is equivalent to a first-level subnode of $SCHEDULE.

The minimum data structures required from the standard structures $SUBNET and $SCHED1 are shown in Fig. 2.4.10.1. Note that in the minimum structure the fifth and sixth subnodes of a relation in the TEMPORAL_RELATION substructure are not mandatory in every case.

Note:	Minimum (i.e., relevant) portion of the required input standard data structures is shown. In all trees, any additional structure will be preserved.



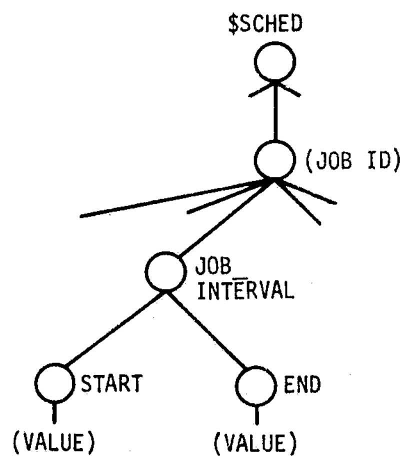


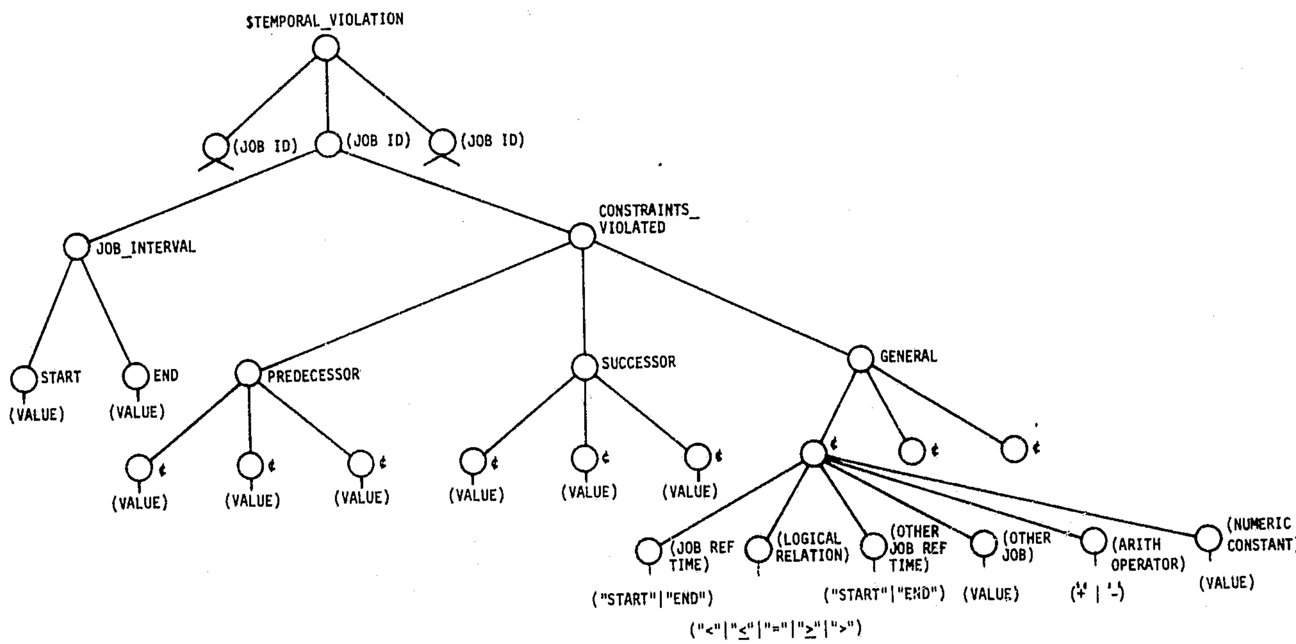
Fig. 2.4.10-1
Minimum Required Input Structures from Standard Data Structures for Module: CHECK_INTERNAL_TEMP_RELATIONS

ORIGINAL PAGE IS OF POOR QUALITY

2.4.10.4 Module Output

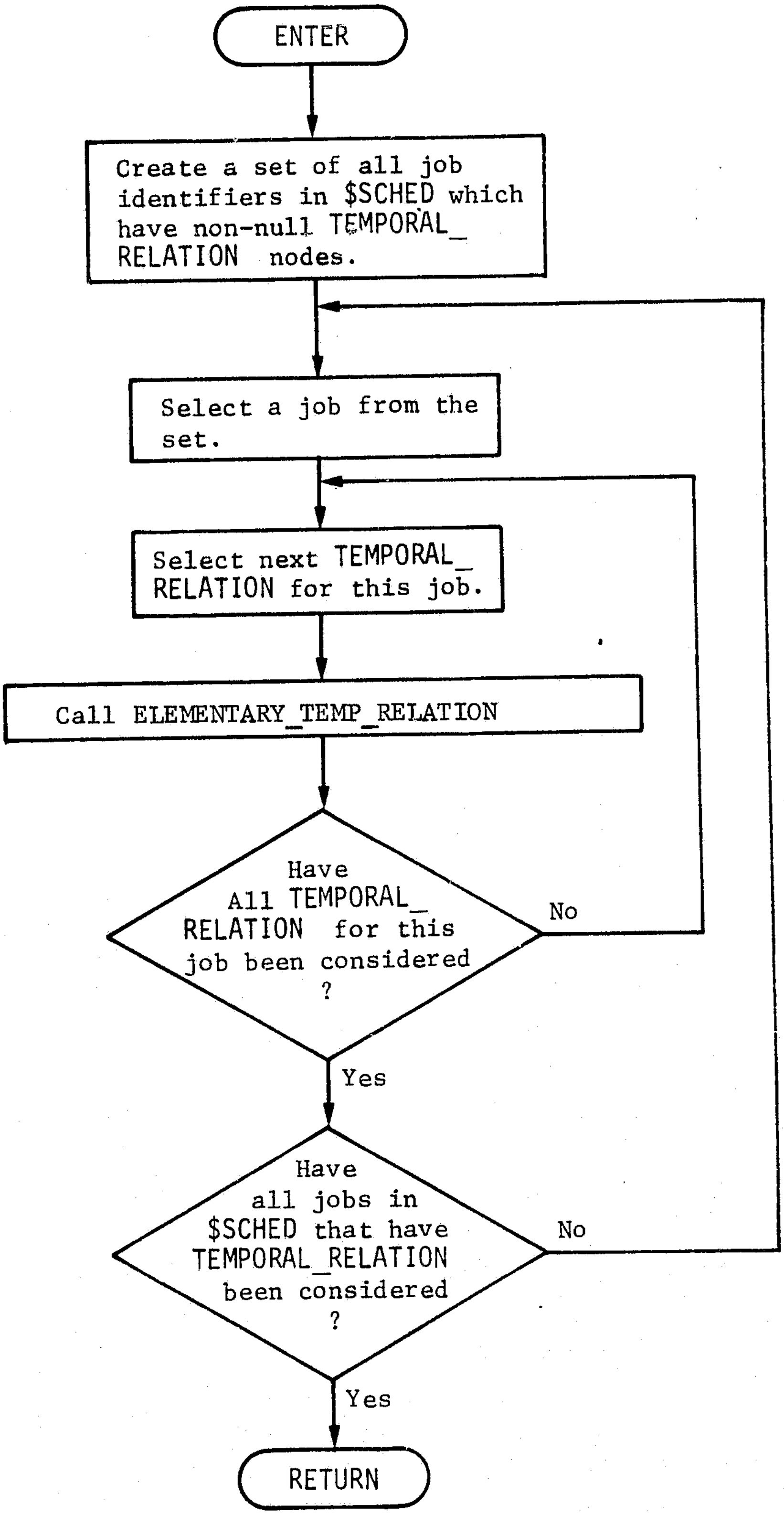
This module will build and return an output tree with the structure shown below:

OUTPUT DATA STRUCTURE



Each node of $TEMPORAL_VIOLATION will correspond to a violation of a temporal relation in $SUBNET (input) that appears internally in $SCHED (input).

2.4.10.5 Functional Block Diagram



2.4.10.6 Detailed Design

The functional block diagram provides the flow chart for this module. The module selects each job in turn from an input schedule unit. Each temporal relationship of the job, whether predecessor, successor, or general, is utilized in a call to the module ELEMENTARY_TEMP_RELATION to determine the satisfaction of the relationship. If the relationship is not satisfied, a node is built to identify the temporal violation. The next temporal relation, or the next job is the selected and the procedure repeated.

2.4.10.7 Internal Variable and Tree Names

$JOB_1 - job taken from the partial schedule

$JOB_2 - job related temporally to $JOB_1

$RELATION - temporal relationship between the two jobs

2.4.10.8 Modifications to Functional Specifications and/or Standard Data Structures Assumed

Only a single subnode of $JOBSET is examined with each call to this module, so this subnode is identified as $SUBNET through the parameter list. $SCHED is equivalent to a first level subnode of $SCHEDULE.

2.4.10.9 Commented Code

```
INTERNAL_TEMP_RELATIONS:  PROCEDURE($SUBNET,$SCHED,
   $TEMPORAL_VIOLATION) OPTIONS(EXTERNAL);
/*  CREATE A SET OF ALL JOB IDENTIFIERS IN $SCHED WHICH HAVE NON-      */
/*   NULL TEMPORAL_RELATION NODES                                       */
DO I = 1 TO NUMBER($SCHED);
LABEL($JOB1) = LABEL($SCHED(I));
$JOB1.JOB_INTERVAL = $SCHED(I).JOB_INTERVAL;
/*  SELECT A JOB FROM THE SET                                           */
/*  SELECT NEXT TEMPORAL_RELATION FOR THIS JOB                          */
   DO J = 1 TO NUMBER($SUBNET.#LABEL($JOB1).TEMPORAL_RELATION.
      PREDECESSOR);
      PRUNE $RELATION;
      LABEL($JOB2) = $SUBNET.#LABEL($JOB1).TEMPORAL_RELATION.
         PREDECESSOR(J);
      $JOB2.JOB_INTERVAL = $SCHED.#LABEL($JOB2).JOB_INTERVAL;
      $RELATION(FIRST) = LABEL($JOB2);
      LABEL($RELATION) = 'PREDECESSOR';
      CALL ELEMENTARY_TEMP_RELATIONS($JOB1,$JOB2,$RELATION,$RESULT);
      IF $RESULT.SATISFIED = 'NO'
         THEN DO;
            $TEMPORAL_VIOLATION(NEXT) = $JOB1;
            $TEMPORAL_VIOLATION(LAST).CONSTRAINT_VIOLATED.PREDECESSOR
               (NEXT) = LABEL($JOB2);
            END;
   END;
   DO J = 1 TO NUMBER($SUBNET.#LABEL($JOB1).TEMPORAL_RELATION.
      SUCCESSOR);
      PRUNE $RELATION;
      LABEL($JOB2) = $SUBNET.#LABEL($JOB1).TEMPORAL_RELATION.
         SUCCESSOR(J);
      $JOB2.JOB_INTERVAL = $SCHED.#LABEL($JOB2).JOB_INTERVAL;
      $RELATION(FIRST) = LABEL($JOB2);
      LABEL($RELATION) = 'SUCCESSOR';
      CALL ELEMENTARY_TEMP_RELATIONS($JOB1,$JOB2,$RELATION,$RESULT);
      IF $RESULT.SATISFIED = 'NO'
         THEN DO;
            $TEMPORAL_VIOLATION(NEXT) = $JOB1;
            $TEMPORAL_VIOLATION(LAST).CONSTRAINT_VIOLATED.SUCCESSOR
               (NEXT) = LABEL($JOB2);
            END;
   END;
```

```
   DO J = 1 TO NUMBER($SUBNET.#LABEL($JOB1).TEMPORAL_RELATION.
      GENERAL);
      PRUNE $RELATION;
      LABEL($JOB2) = $SUBNET.#LABEL($JOB1).TEMPORAL_RELATION.
         GENERAL(J)(4);
      $JOB2.JOB_INTERVAL = $SCHED.#LABEL($JOB2).JOB_INTERVAL;
      $RELATION = $SUBNET.#LABEL($JOB1).TEMPORAL_RELATION.GENERAL(J);
      CALL ELEMENTARY_TEMP_RELATIONS($JOB1,$JOB2,$RELATION,$RESULT);
      IF $RESULT.SATISFIED = 'NO'
         THEN DO;
            $TEMPORAL_VIOLATION(NEXT) = $JOB1;
            $TEMPORAL_VIOLATION(LAST).CONSTRAINT_VIOLATED.GENERAL
               (NEXT) = $RELATION;
         END;
/*  HAVE ALL TEMPORAL_RELATION FOR THIS JOB BEEN CONSIDERED?              */
   END;
/*  HAVE ALL JOBS IN $SCHED BEEN CONSIDERED?                              */
END;


END INTERNAL_TEMP_RELATIONS;
```

2.4.11 ELEMENTARY_TEMP_RELATIONS

2.4.11 ELEMENTARY_TEMP_RELATIONS

2.4.11.1 Purpose and Scope

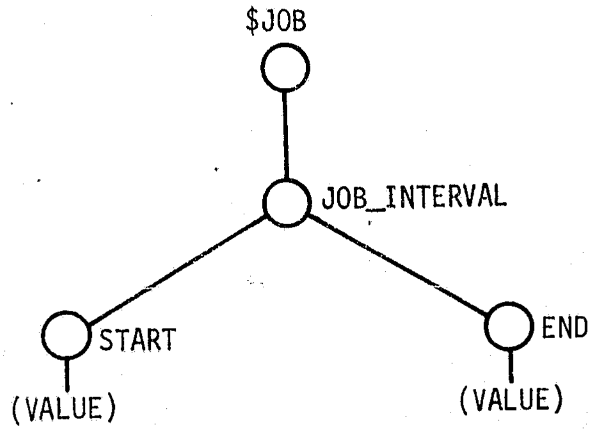
This module is elementary in the sense that it determines satisfaction or nonsatisfaction of a single input relationship involving the start or end times of two jobs for which specific start and end times have been assigned. The principal use of this module is to service higher level logic that is checking multiple temporal relations between or within sets of jobs.

2.4.11.2 Modules Called

None

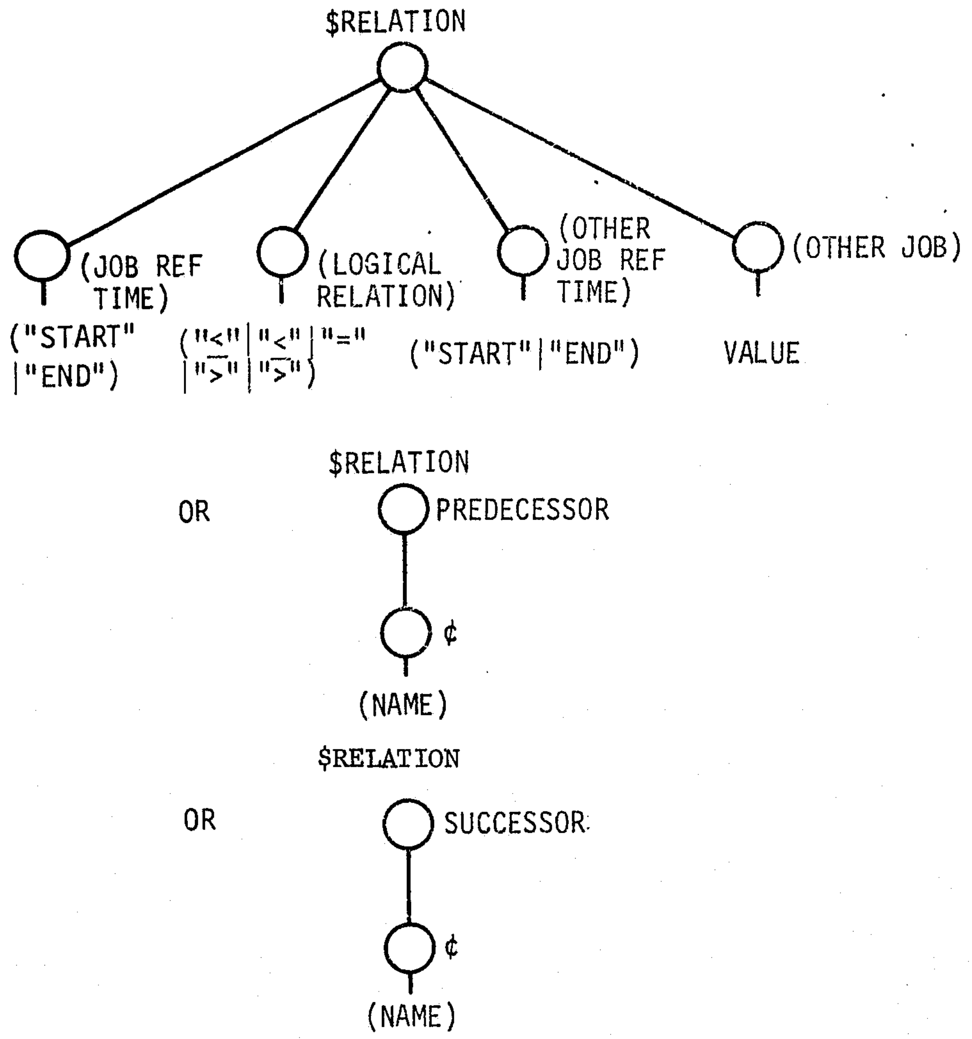
2.4.11.3 Module Input

There are three input arguments to this module. These are $JOB1, $JOB2, and $RELATION. The structure of $JOB1 and $JOB2 is shown below:



The structure of $RELATION is the structure of one of the subnodes of TEMPORAL_RELATION shown in the section on standard data structures. This module assumes that $JOB1 is the same job for which the structure TEMPORAL_RELATION is written and that $JOB2 is the other job that is referred to in the fourth subnode of the special structure of $RELATION. Note that in illustrating the minimum required data structure for this information that the fifth and sixth subnodes for the structure $RELATION are not mandatory to specify temporal relationships in every case.

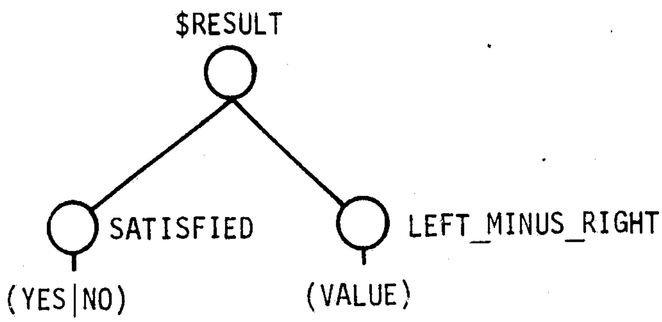
Note: The minimum (i.e., relevant) portion of the required input standard data structures is shown. In all trees, any additional structure will be preserved.



Minimum Required Input Structures from Standard Data Structures for Module:
ELEMENTARY_TEMP_RELATIONS

2.4.11.4 Module Output

This module returns a tree $RESULT with two first level subnodes as shown below:



The value returned for the LEFT_MINUS_RIGHT node is simply the algebraic result of subtracting the quantity on the right of the binary operator (≤, <, =, ≥, >) of the input TEMPORAL_RELATION from the quantity on the left. If the module is called with a PREDECESSOR or SUCCESSOR, this module assumes the following equivalent relations to compute the LEFT_MINUS_RIGHT value:

GENERAL RELATION

$$\underbrace{\left\{\begin{matrix}\text{START}\\ \text{END}\end{matrix}\right\}}_{\text{LEFT\_SIDE}} \left\{\begin{matrix}<\\ \leq\\ =\\ \geq\\ >\end{matrix}\right\} \underbrace{\left\{\begin{matrix}\text{START}\\ \text{END}\end{matrix}\right\} \text{ OF } \left|(\text{JOB\_ID})\right| \left\{\begin{matrix}+\\ -\end{matrix}\right\} \left|(\text{CONSTANT})\right|}_{\text{RIGHT\_SIDE}}$$

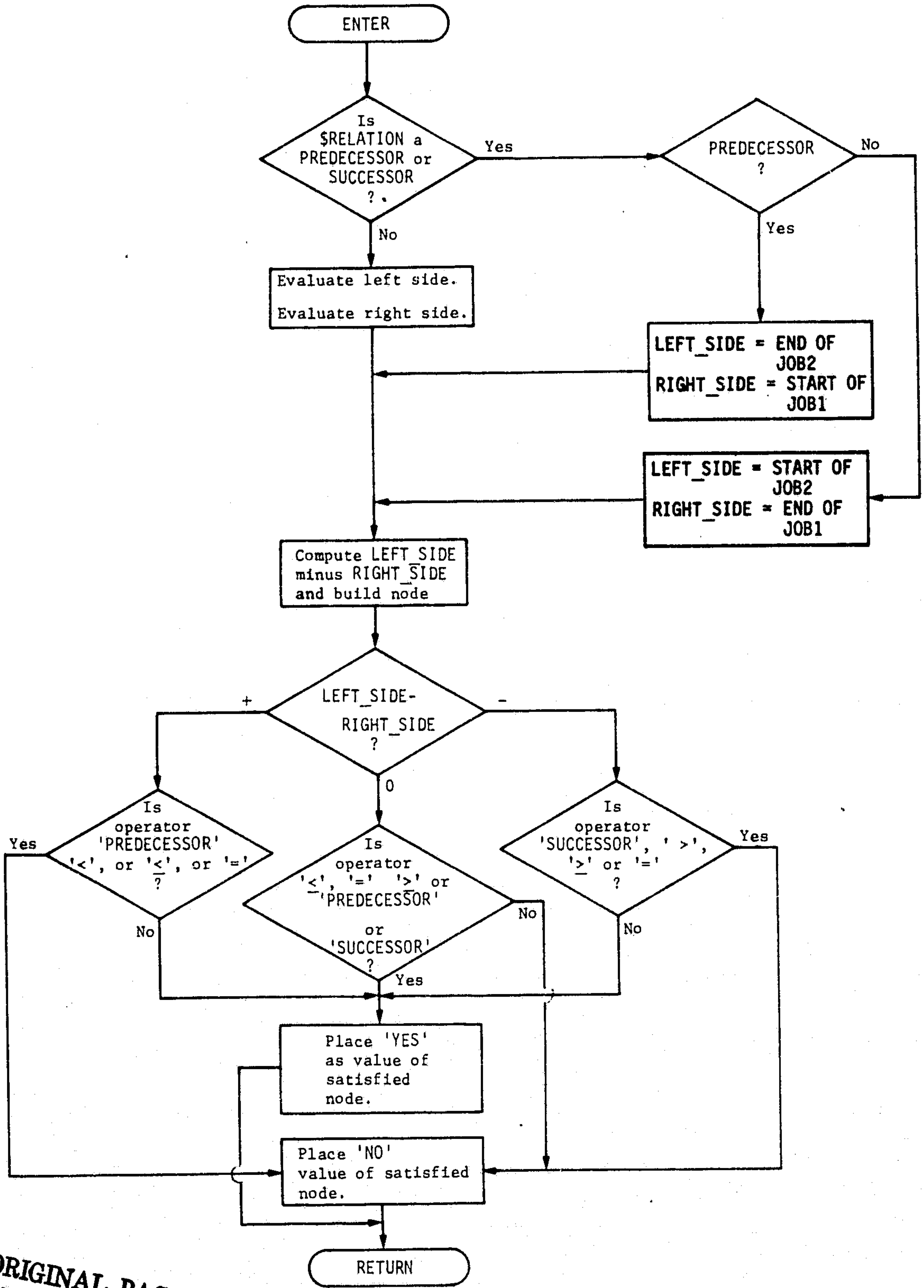
PREDECESSOR

$$\underbrace{\text{END OF JOB2}}_{\text{LEFT\_SIDE}} \leq \underbrace{\text{START OF JOB1}}_{\text{RIGHT\_SIDE}}$$

SUCCESSOR

$$\underbrace{\text{START OF JOB2}}_{\text{LEFT\_SIDE}} \geq \underbrace{\text{END OF JOB 1}}_{\text{RIGHT\_SIDE}}$$

2.4.11.5 Functional Block Diagram



2.4.11.6 Detailed Design

The functional block diagram provides the flow chart for this module. This module takes the two input jobs and constructs values called left side and right side dependent upon the relationship between the two jobs. For the general temporal relationship; left side is the value of the first subnode, right side is the evaluation of subnodes three through six. If JOB_2 is a predecessor of JOB_1, left side is the end of JOB_2, right side is the end of JOB_1. If JOB_2 is a successor of JOB_1, left side is the start of JOB_2, right side is the end of JOB_1. The difference in value between left side and right side id determined and satisfaction of the temporal relation is checked. If the difference is positive (greater than zero) and the temporal relation is "SUCCESSOR" or the value of the second node of the general relation is "≥" or ">", the relation is satisfied. If the difference between left side and right side is negative and the temporal relation is "PREDECESSOR" or the second node is "≤" or "<", the relation is satisfied. If the difference is zero, the relation is satisfied by temporal relations "PREDECESSOR" and "SUCCESSOR" and by second nodes values "≤", "=", and "≥".

2.4.11.7 Internal Variable and Tree Names

LEFT_SIDE - reference to the value in the general temporal relation to the left of the logical relation, i.e. the first subnode.

RIGHT_SIDE - reference to the evaluation of the general temporal relation to the right of the logical relation, i.e., nodes three through six.

LEFT_RIGHT - the difference between LEFT_SIDE and RIGHT_SIDE

2.4.11.8 Modifications to Functional Specifications and/or Standard Data Structures Assumed

None

2.4.11.9 Commented Code

```
ELEMENTARY_TEMP_RELATIONS:  PROCEDURE($JOB1,$JOB2,$RELATION,
   $RESULT) OPTIONS(EXTERNAL);
DECLARE LEFT_SIDE,RIGHT_SIDE,LEFT_RIGHT LOCAL;
/*  IS $RELATION A PREDECESSOR?                                              */
IF LABEL($RELATION) = 'PREDECESSOR'
   THEN DO;
      LEFT_SIDE = $JOB2.JOB_INTERVAL.END;
      RIGHT_SIDE = $JOB1.JOB_INTERVAL.START;
      END;
/*  IS $RELATION A SUCCESSOR?                                                */
   ELSE IF LABEL($RELATION) = 'SUCCESSOR'
      THEN DO;
         LEFT_SIDE = $JOB2.JOB_INTERVAL.START;
         RIGHT_SIDE = $JOB1.JOB_INTERVAL.END;
         END;
      ELSE DO;
         LEFT_SIDE = $JOB1.JOB_INTERVAL.#($RELATION(1));
         IF $RELATION(5) = '-'
            THEN RIGHT_SIDE = $JOB2.JOB_INTERVAL.#($RELATION(3)) -
               $RELATION(6);
            ELSE RIGHT_SIDE = $JOB2.JOB_INTERVAL.#($RELATION(3)) +
               $RELATION(6);
            END;
/*  COMPUTE LEFT_SIDE MINUS RIGHT_SIDE                                       */
LEFT_RIGHT = LEFT_SIDE - RIGHT_SIDE;
$RESULT.LEFT_MINUS_RIGHT = LEFT_RIGHT;
IF LEFT_RIGHT > 0
   THEN IF (LABEL($RELATION) = 'PREDECESSOR' | $RELATION(2) = '<' |
      $RELATION(2) = '<=' | $RELATION(2) = '=')
      THEN DO;
         $RESULT.SATISFIED = 'NO';
         RETURN;
         END;
      ELSE DO;
         $RESULT.SATISFIED = 'YES';
         RETURN;
         END;
   ELSE IF LEFT_RIGHT < 0
      THEN IF (LABEL($RELATION) = 'SUCCESSOR' | $RELATION(2) = '>' |
         $RELATION(2) = '>=' | $RELATION(2) = '=')
         THEN DO;
            $RESULT.SATISFIED = 'NO';
            RETURN;
            END;
         ELSE DO;
            $RESULT.SATISFIED = 'YES';
            RETURN;
            END;
```

```
      ELSE IF ($RELATION(2) = '<' | $RELATION(2) = '>')
         THEN DO;
            $RESULT.SATISFIED = 'NO';
            RETURN;
            END;
         ELSE DO;
            $RESULT.SATISFIED = 'YES';
            RETURN;
            END;
END;     /*      ELEMENTARY_TEMP_RELATIONS          */
```

ORIGINAL PAGE IS
OF POOR QUALITY

2.4.12 NEXTSET

2.4.12 NEXTSET

2.4.12.1 Purpose and Scope

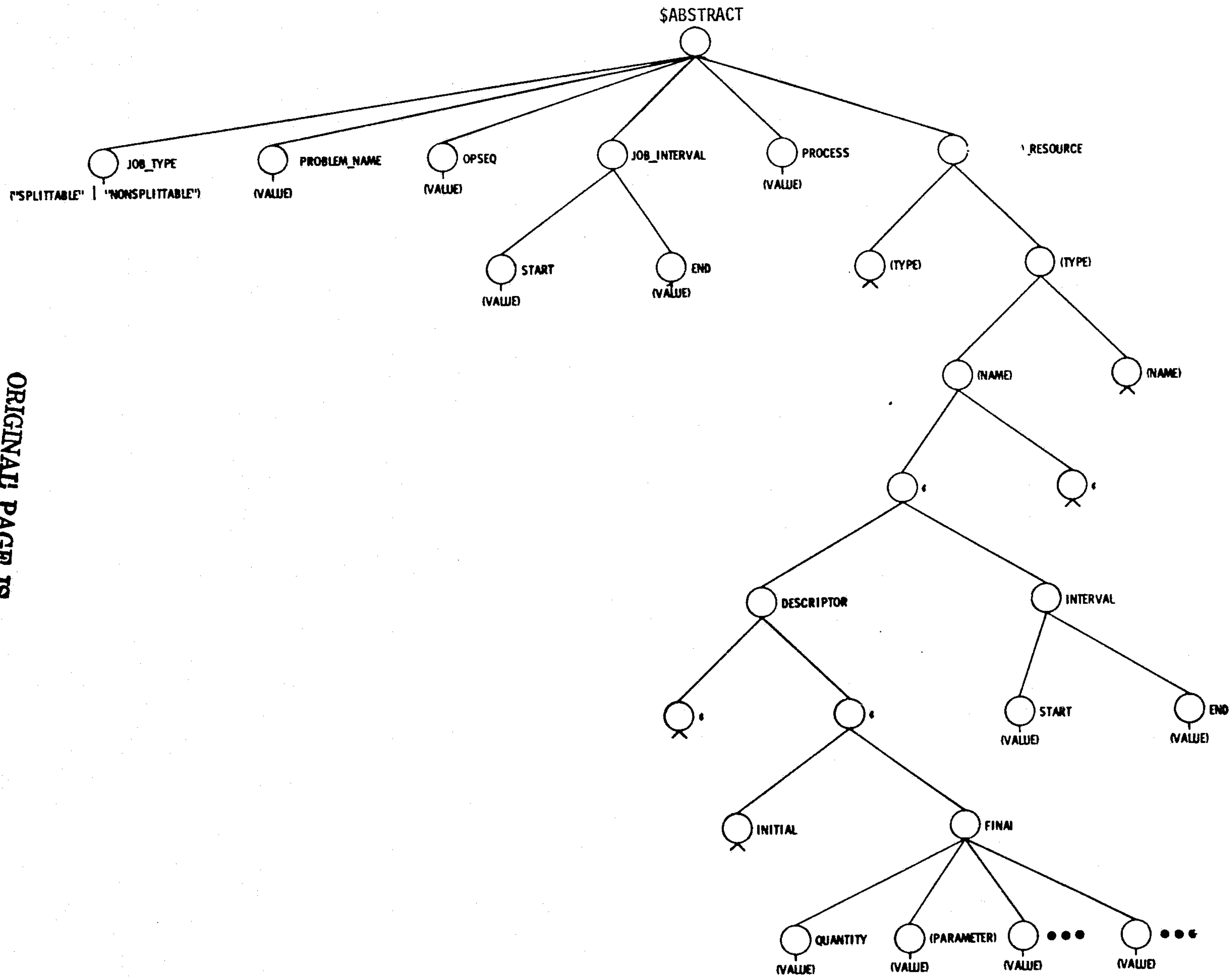
This module accepts an abstract description of item specific resource requirements associated with a specific job and, by referring to information about the assignments already scheduled for the resources, determines the earliest possible time (within a designated interval) at which the resource requirements can be fulfilled. It generates all information required to actually place the job on the schedule but does not cause resource assignments to be written. The module also determines the time intervals during which the resource requirements are met using the same permutation of resources and time intervals for which any permutation of available resources meets the requirements.

2.4.12.2 Modules Called

DURATION
INTERVAL_UNION
INTERVAL_INTERSECT
FIND_MIN

2.4.12.3 Module Input

$ABSTRACT is a tree structure that describes the job in terms of its general characteristics, resource requirements, and, if applicable, in terms of any user-designated specific resource allocations. Its structure is shown on the following page.


$ABSTRACT
JOB_TYPE
("SPLITTABLE" | "NONSPLITTABLE")
PROBLEM_NAME
(VALUE)
OPSEQ
(VALUE)
JOB_INTERVAL
PROCESS
(VALUE)
RESOURCE
START
(VALUE)
END
(VALUE)
(TYPE)
(TYPE)
(NAME)
(NAME)
DESCRIPTOR
INTERVAL
START
(VALUE)
END
(VALUE)
INITIAL
FINAL
QUANTITY
(VALUE)
(PARAMETER)
(VALUE)
(VALUE)
(VALUE)

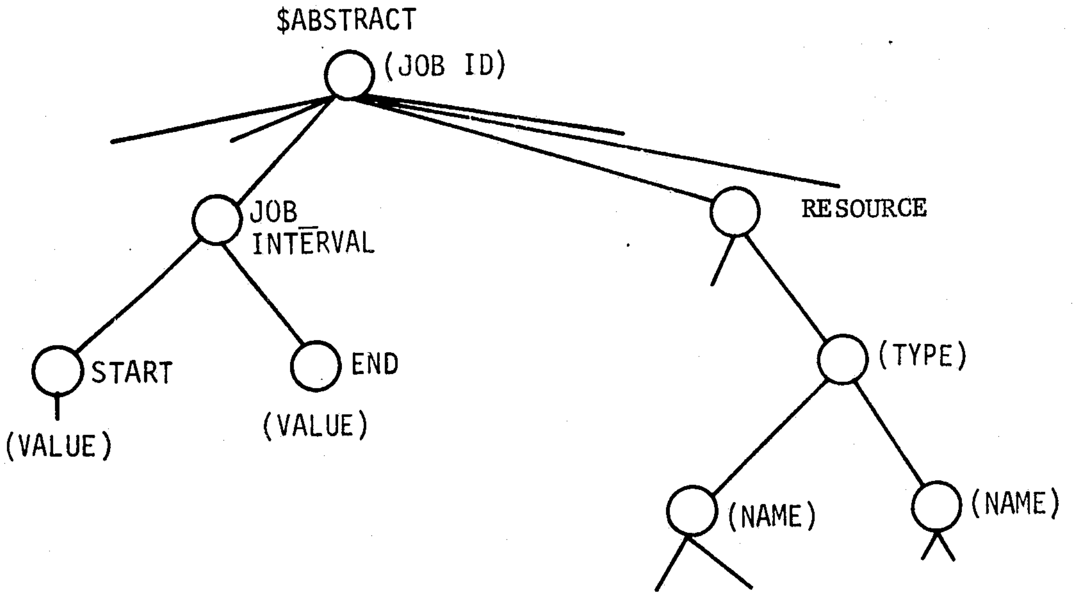
Except for the job, process, and resource intervals, the information is exactly as used elsewhere for abstract process and job description. Specifically, the information is in the form generated by the module GENERATE_JOBSET.

Since the absolute start and end times of the jobs, processes, and resource allocations are an output of this (and other) modules, rather than an input, the intervals in this structure are relative. The resource interval represents the start and end times (relative to the start of the process) of a single resource allocation. These relative times may be positive, zero, or (very rarely) negative.

The absolute start and end times of interest are specified in the argument list as subnodes of $REQUESTED_INTERVAL to limit the scope of assignments considered, and $RESOURCE is referenced to allow access to the resource assignments.

If for a given resource unit, the resource unit name is specified (i.e., LABEL($ABSTRACT.RESOURCE(J)(K)) is not null, then it is assumed that the named resource unit is to be used. Regardless of the specification or nonspecification of the resource unit, the requirements (descriptors, quantity, etc.) still apply and must be satisfied, if possible, by NEXTSET.

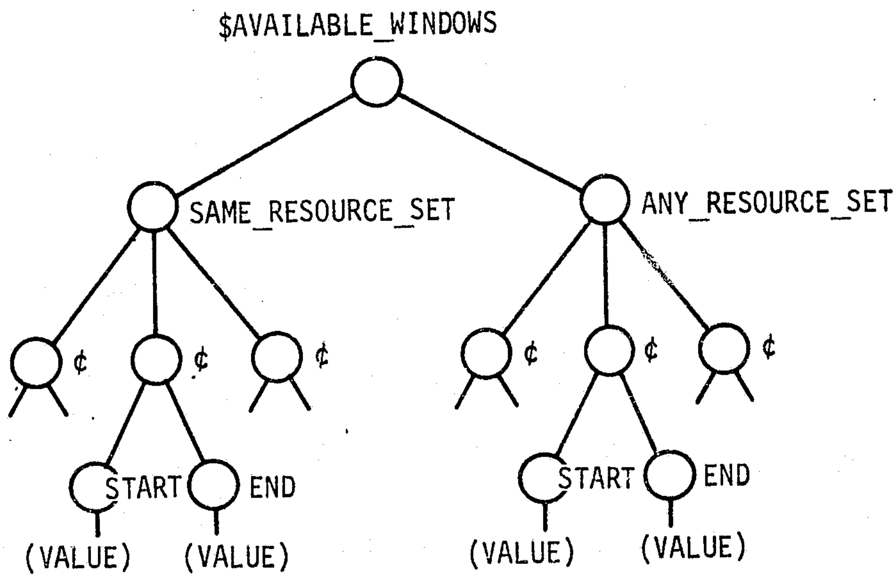
Note: The minimum (i.e., relevant) portion of the required input standard data structures is shown. In all trees, any additional structure will be preserved.



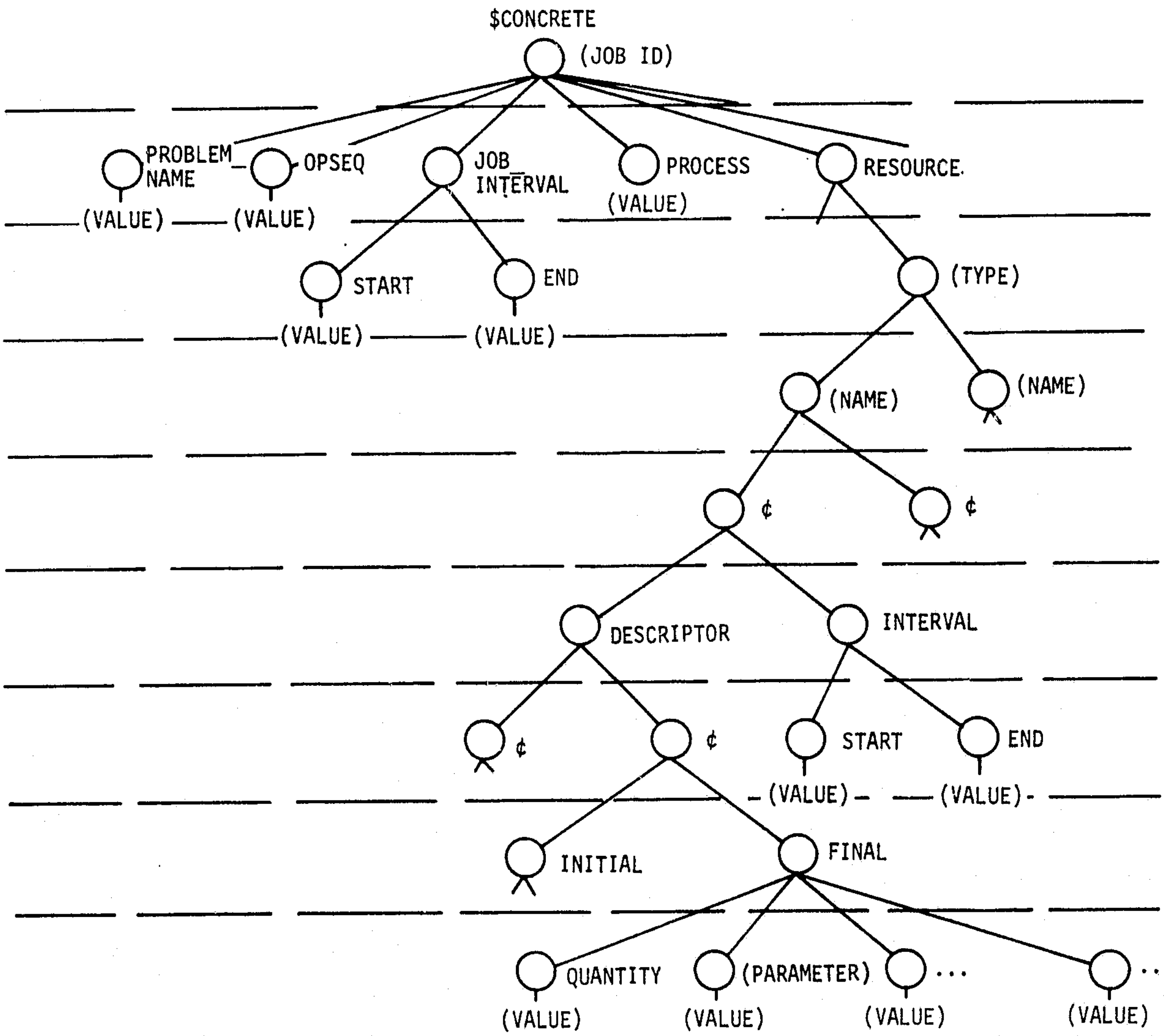
Minimum Required Input Structures from Standard Data Structures for Module: NEXTSET

2.4.12.4 Module Output

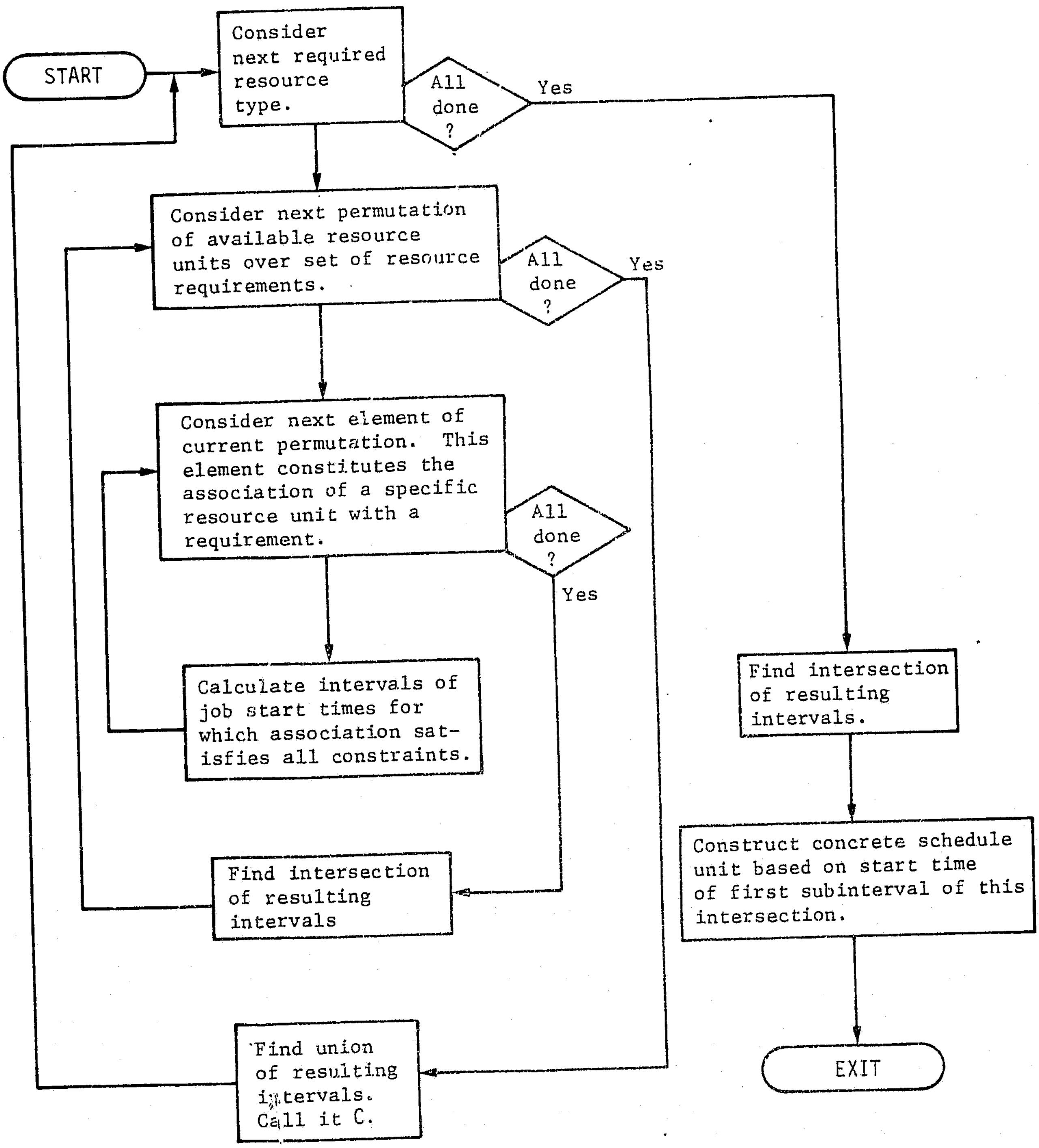
The output of NEXTSET consists of two output trees, $CONCRETE and $AVAILABLE_WINDOWS. $CONCERETE, as shown on the following page, describes a specific execution of a job, with all times and resource allocations fully specified in absolute terms at the earliest available opportunity within the specified window. $AVAILABLE_WINDOWS, also shown below, defines all of the available time intervals, within the specified window, for the set of resources corresponding to the set representing the earliest available time. It also defines the available time intervals if any permutation of acceptable resources is considered.



OUTPUT DATA STRUCTURE



2.4.12.5 Functional Block Diagram



2.4.12.6 Typical Applications

This module can be applied to both time progressive and time transcendent scheduling procedures. The module identifies the earliest time within a given interval at which the resource requirements of a single job are fulfilled by some permutation of item-specific resource elements. The time intervals within the given interval for which the resource requirements are met with the selected permutation of resources are identified permitting scheduling based on criteria other than earliest start time. Time intervals for which any permutation of resources meet the requirements allow the same flexibility of scheduling criteria; however, permutations of resources other than the earliest are not identified.

2.4.12.7 DETAILED DESIGN

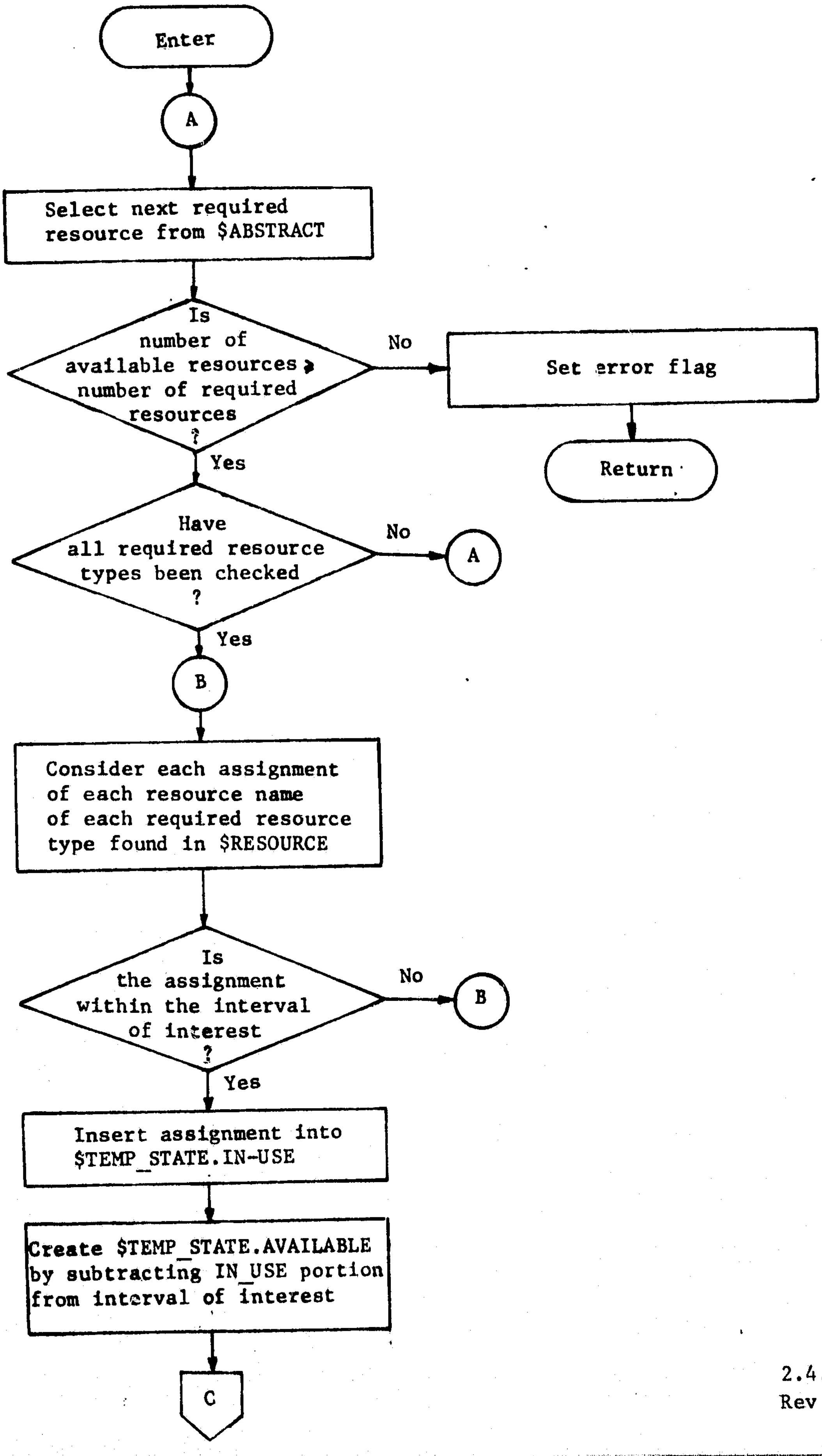
The logical path for the module NEXTSET illustrated in the Functional Block Diagram is developed in greater detail in the flowchart presented in the sketch below. The module starts by determining that the standard data structure, $RESOURCE, contains at least as many resource elements of each type as is required by the input structure, $ABSTRACT. Given that enough resources are named, the module develops a usage profile for each resource element of all the resource types requested by $ABSTRACT over the period of interest. From the usage profile the availability profile is readily developed for the same time period.

Since permutations of available resource units over the set of requirements will be formed, the required resource units specified by name must be included in each permutation. To accomplish this, a tree is formed containing only the labels of the names of available resource units. From this tree are pruned the names of the specified resource units and the remaining tree structure is utilized for the formation of the permutations. The intervals of the start times for the specified resource units is determined and the intersection of these intervals is maintained for combination with acceptable permutations.

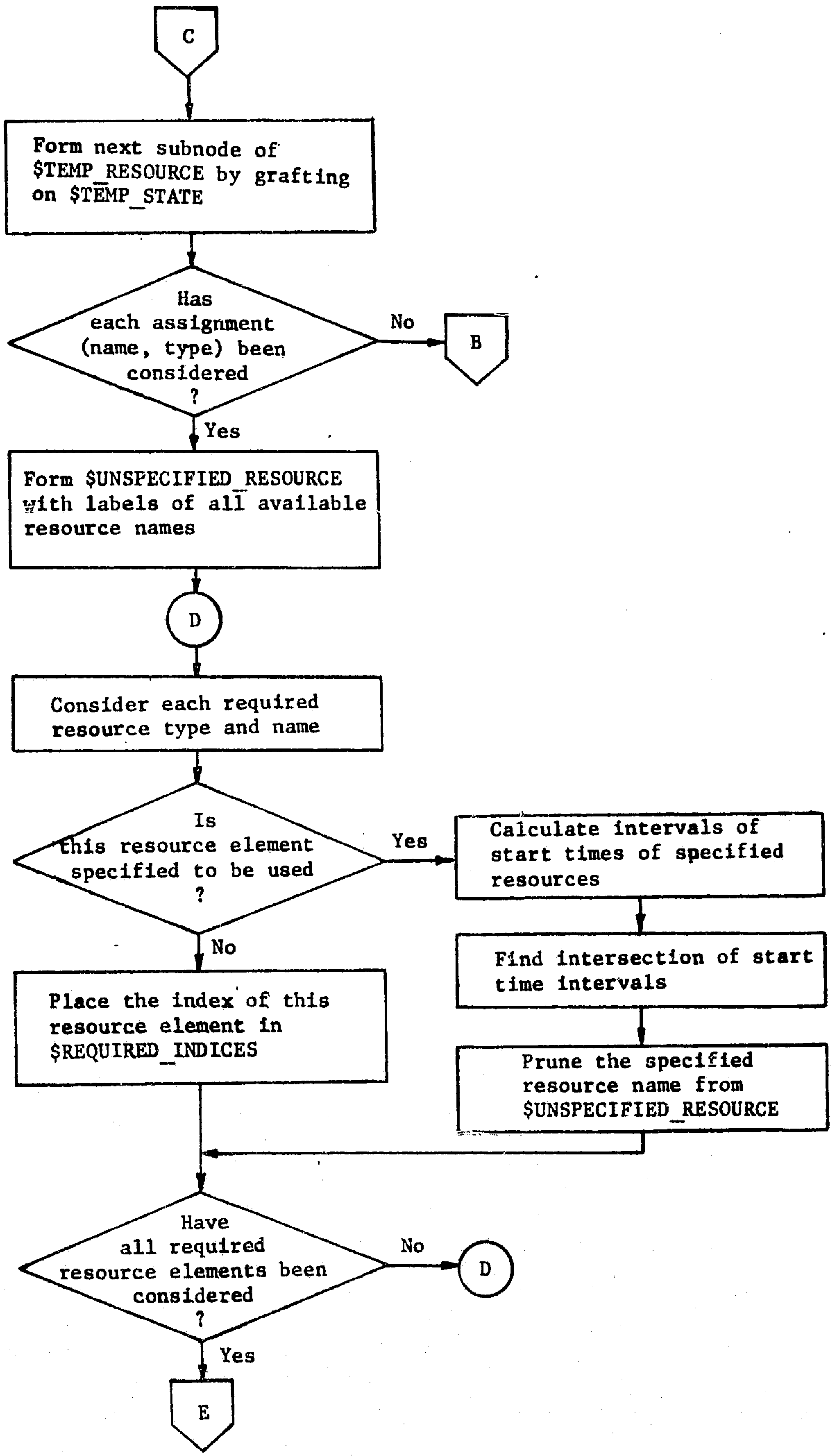
For each resource type all permutations of available, unspecified resources are taken over the set of required resources. Then each element of each permutation is checked for a match of descriptors between required and available. If an element of the permutation does not provide this match, no further elements are checked and the next permutation is tested. When all elements of a permutation match descriptors, the interval of start times for each element is calculated. The intersection of these intervals forms an interval of start times for one permutation of one resource type. When all permutations for a given resource type have been formed, and at least one interval of start times has been identified, the union of intervals of the various permutations provides the interval of feasible start times for the current resource type. If no feasible intervals are determined, an error message is printed, and control is returned to the calling program.

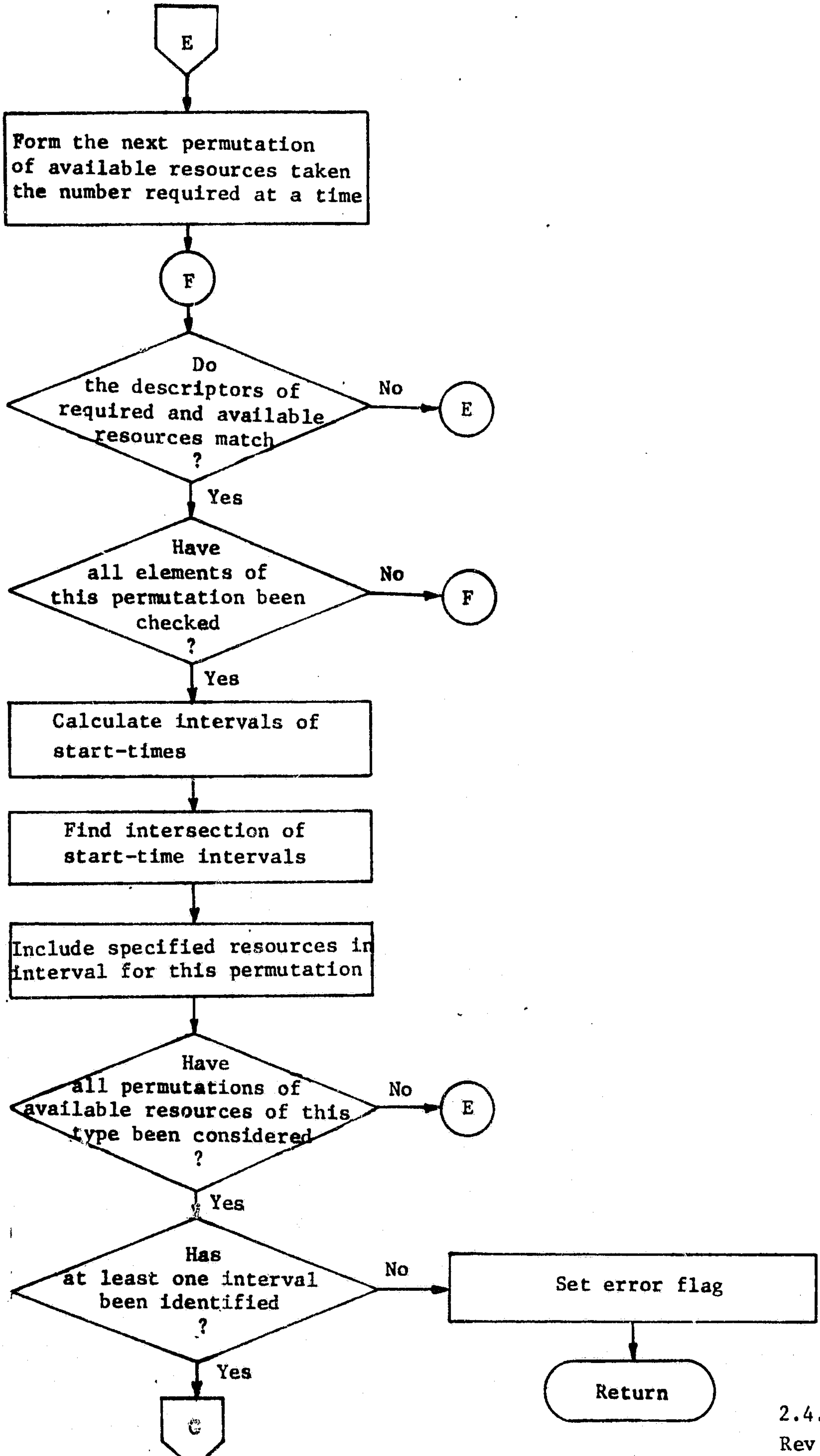
After the intervals of feasible start times for all resource types have been determined, the intersection of these intervals is the desired interval of times at which the process may be started using some set of resources. The resource set yielding the earliest start time is then identified and the interval over which the process may be started using this resource set is determined. The output data structures are then constructed and control returned to the calling program.

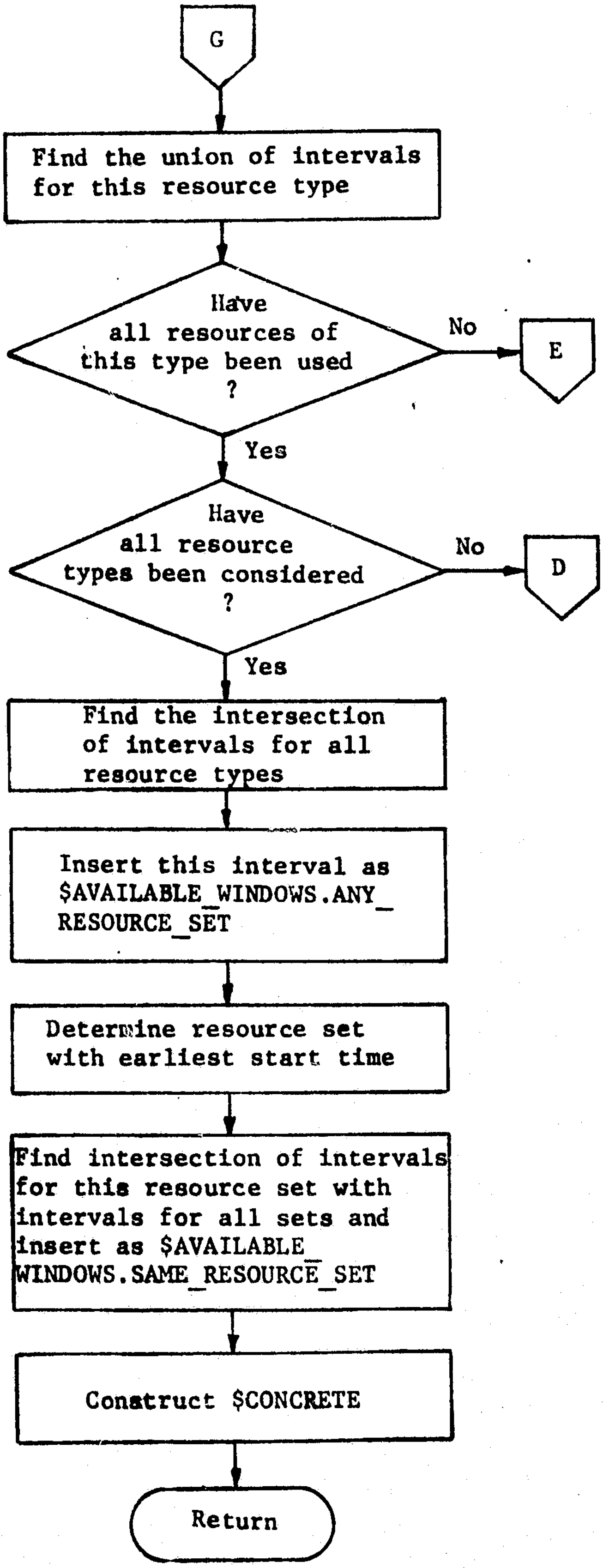
NEXTSET FLOWCHART



NEXTSET FLOWCHART (cont)




E
Form the next permutation of available resources taken the number required at a time
F
Do the descriptors of required and available resources match ?
No
E
Yes
Have all elements of this permutation been checked ?
No
F
Yes
Calculate intervals of start-times
Find intersection of start-time intervals
Include specified resources in interval for this permutation
Have all permutations of available resources of this type been considered ?
No
E
Yes
Has at least one interval been identified ?
No
Set error flag
Return
Yes


G
Find the union of intervals for this resource type
Have all resources of this type been used ?
No
E
Yes
Have all resource types been considered ?
No
D
Yes
Find the intersection of intervals for all resource types
Insert this interval as $AVAILABLE_WINDOWS.ANY_RESOURCE_SET
Determine resource set with earliest start time
Find intersection of intervals for this resource set with intervals for all sets and insert as $AVAILABLE_WINDOWS.SAME_RESOURCE_SET
Construct $CONCRETE
Return

2.4.12.8 INTERVAL VARIABLE AND TREE NAME DEFINITIONS

AVAILABLE_DURATION	-	the length of time a given resource element is available
BEGIN_TIME	-	earliest start time meeting all resource constraints
END_TIME	-	earliest time following BEGIN_TIME for which the resources to be used are no longer available
INDEX	-	tree subnode indicator of one of the required unspecified resource names
WINDOW	-	single subnode of $WINDOW_SET, an interval denoting one interval of start times for a resource element required more than once during a process
REQUIRED_DURATION	-	the length of time a given resource element is required
$ASSIGN_INT	-	a standard interval sturucture denoting the time periods a given resource element is in use
$AVAIL_INT	-	A standard interval structure denoting the time periods a given resource element is available
$AVAIL_PROFILE	-	a standard interval structure denoting feasible start times for a single resource element
$INTER_ASSIGN	-	a tree structure containing descriptors and intervals of the resource set having earliest start time, used to construct $CONCRETE
$INTERSECT	-	a standard interval structure used to determine feasibility of a resource element required more than once during a process

$INTERSECTION_FINAL - the (multiple) interval of feasible start times with any resource set

$INTERSECTION_INITIAL - the interval of feasible start times for one resource set

$INTERSECTION_SPECIFIED - the interval of feasible start times for the specified resource names only

$NAME - a single node tree whose value is the label of one resource element

$REQ_INT - a standard interval structure denoting the time periods a given resource element is required

$REQUIRED_INDICES - a tree whose subnode values are the position indicators (in $ABSTRACT) of the required, unspecified resources

$SAME_SET - the (multiple) interval of all feasible start times using the resource set that provides the earliest possible start time

$SPECIFIED_PROFILE - a standard interval denoting feasible start times of all the specified resources

$TEMP_ASSIGN - a tree structure providing the intervals of start times and resource names for each feasible permutation group.

$TEMP_INT - a standard interval structure used in determining multiple interval requirements

$TEMP_RESOURCE - tree structure containing usage and availability descriptions for all required resources, built from $TEMP_STATE

$TEMP_STATE	-	tree providing usage and availability descriptions of a single required resource
$TYPE	-	a single node tree whose value is the label of a resource type
$UNSPECIFIED_RESOURCE	-	a tree whose values are the names of the item specific resources of a single type; and which are not specifically required
$WINDOW_SET	-	a standard interval set defining feasible start times for a resource element which is required more than once during a process

2.4.12.9 Commented Code

```
NEXTSET: PROCEDURE($ABSTRACT,$REQUESTED_INTERVAL,$RESOURCE,$CONCRETE,
  $AVAILABLE_WINDOWS) OPTIONS(EXTERNAL);
DECLA E $SPECIFIED_RESOURCE,NS LOCAL;
 DECLARE $INT_UNION,NUM,$SAVE_INTERSECTION LOCAL;
  DECLARE $TEMP_ASSIGN, $INTER_ASSIGN, $TEMP_RESOURCE, $TEMP_STATE,
     $UNION, $INTERSECTION_INITIAL, $INTERSECTION_SPECIFIED,
     $INTERSECTION_FINAL LOCAL;
DECLARE I,J,K,L,KK,LL,START,    NUMBER_PERMUTATION,AVAILABLE_DURATION,
      SPECIFIED_DURATION,REQUIRED_DURATION,INDEX,DELTA,WINDOW LOCAL;
DECLARE VAL_MIN,BEGIN_TIME,END_TIME,FINISH LOCAL;
DECLARE $TYPE,$UNSPECIFIED_RESOURCE,$REQUIRED_INDICES,$NAME LOCAL;
DECLARE $SPECIFIED_PROFILE,$INTERSECTION_SPECIFIED,$PERMUTATION LOCAL;
DECLARE $AVAIL_PROFILE,$REQ_INT,$ASSIGN_INT,$AVAIL_INT,$TEMP_INT,
      $WINDOW_SET,$INDICES,$INTERSECTION_INITIAL,$INTERSECTION_FINAL,
      $SAME_SET,$INTERSECT,$FOUND_UNION LOCAL;
SET_INITIAL_VALUES:
  PRUNE $AVAILABLE_WINDOWS;
  PRUNE $CONCRETE;
  PRUNE $UNION;
  START = $REQUESTED_INTERVAL.START;
  FINISH = $REQUESTED_INTERVAL.END;
  IF FINISH < START
     THEN DO;
        WRITE 'REQUESTED INTERVAL OF SHORTER DURATION THEN JOB INTERVAL
 - SCHEDULING IMPOSSIBLE';
        RETURN;
        END;
  $INTERSECTION_INITIAL.START = START;
  $INTERSECTION_INITIAL.END = FINISH;
  $INTERSECTION_FINAL.START = START;
  $INTERSECTION_FINAL.END = FINISH;
  $INTERSECTION_SPECIFIED.START = START;
  $INTERSECTION_SPECIFIED.END = FINISH;
  PRUNE $TEMP_ASSIGN;
  PRUNE $TEMP_RESOURCE;
/*  SELECT NEXT REQUIRED RESOURCE FROM $ABSTRACT                        */
  DO I = 1 TO NUMBER($ABSTRACT.RESOURCE );
/* IS NUMBER OF AVAILABLE RESOURCES >= NUMBER OF REQUIRED RESOURCES? */
  IF NUMBER($ABSTRACT.RESOURCE (I)) > NUMBER($RESOURCE.#
     LABEL($ABSTRACT.RESOURCE (I)))
     THEN DO;
        WRITE 'NOT_ENOUGH_RESOURCES_AVAILABLE';
        RETURN;
/*  HAVE ALL REQUIRED RESOURCE TYPES BEEN CHECKED?                       */
        END;
  END;
/*  CONSIDER EACH ASSIGNMENT OF EACH RESOURCE NAME OF EACH REQUIRED    */
/*  RESOURCE TYPE FOUND IN $RESOURCE                                    */
     DO I = 1 TO NUMBER($ABSTRACT.RESOURCE);
        LABEL($TYPE) = LABEL($ABSTRACT.RESOURCE (I));
        LABEL($TEMP_RESOURCE(I)) = LABEL($TYPE);
```

```
         DO J = 1 TO NUMBER($RESOURCE.#LABEL($TYPE));
         LABEL($NAME) = LABEL($RESOURCE.#LABEL($TYPE)(J));
         LABEL($TEMP_RESOURCE(I)(J)) = LABEL($NAME);
            DO K = 1 TO NUMBER($RESOURCE.#LABEL($TYPE)(J).ASSIGNMENT);
/*   IS THE ASSIGNMENT WITHIN THE INTERVAL OF INTEREST?                  */
            IF (START >= $RESOURCE.#LABEL($TYPE)(J).ASSIGNMENT(K).
               INTERVAL.END | FINISH <= $RESOURCE.#LABEL($TYPE)(J).
               ASSIGNMENT(K).INTERVAL.START) THEN GO TO END_LOOP_K;
               ELSE $TEMP_STATE.IN_USE(NEXT) = $RESOURCE.#LABEL($TYPE)
                  (J).ASSIGNMENT(K);
            END_LOOP_K: END;
/*   INSERT ASSIGNMENT INTO $TEMP_STATE.IN_USE                           */
         IF $TEMP_STATE.IN_USE(1).INTERVAL.START < START
               & $TEMP_STATE.IN_USE(1) ¬ IDENTICAL TO $NULL
            THEN $TEMP_STATE.IN_USE(1).INTERVAL.START = START;
         IF $TEMP_STATE.IN_USE(LAST).INTERVAL.END > FINISH
               & $TEMP_STATE.IN_USE(LAST) ¬ IDENTICAL TO $NULL
            THEN $TEMP_STATE.IN_USE(LAST).INTERVAL.END = FINISH;
/*   CREATE $TEMP_STATE.AVAILABLE BY SUBTRACTING IN_USE PORTION FROM     */
/*   INTERVAL OF INTEREST                                                */
         $TEMP_STATE.AVAILABLE(1).INTERVAL.START = START;
         $TEMP_STATE.AVAILABLE(1).DESCRIPTOR (1).INITIAL =
            $TEMP_STATE.IN_USE(1).DESCRIPTOR (1).INITIAL;
            DO K = 1 TO NUMBER($TEMP_STATE.IN_USE);
            $TEMP_STATE.AVAILABLE(K).INTERVAL.END =
               $TEMP_STATE.IN_USE(K).INTERVAL.START;
            $TEMP_STATE.AVAILABLE(K+1).INTERVAL.START =
               $TEMP_STATE.IN_USE(K).INTERVAL.END;
            $TEMP_STATE.AVAILABLE(K+1).DESCRIPTOR (1).INITIAL =
               $TEMP_STATE.IN_USE(K).DESCRIPTOR (1).FINAL;
            END;
         IF $TEMP_STATE.IN_USE(LAST).INTERVAL.END = FINISH
               & $TEMP_STATE.IN_USE(LAST) ¬ IDENTICAL TO $NULL
            THEN PRUNE $TEMP_STATE.AVAILABLE(LAST);
            ELSE $TEMP_STATE.AVAILABLE(LAST).INTERVAL.END = FINISH;
         IF $TEMP_STATE.IN_USE(1).INTERVAL.START = START
               & $TEMP_STATE.IN_USE(1) ¬ IDENTICAL TO $NULL
            THEN PRUNE $TEMP_STATE.AVAILABLE(1);
/*   FORM NEXT SUBNODE OF $TEMP_RESOURCE BY GRAFTING ON $TEMP_STATE      */
        GRAFT $TEMP_STATE AT $TEMP_RESOURCE.#LABEL($TYPE).#LABEL($NAME);
         END;
/*   HAS EACH ASSIGNMENT (NAME, TYPE) BEEN CONSIDERED?                   */
      END;
CONSIDER_NEXT_RESOURCE_TYPE:
   DO I = 1 TO NUMBER($ABSTRACT.RESOURCE);
   PRUNE $REQUIRED_INDICES;
   PRUNE $SPECIFIED_PROFILE;
   PRUNE $UNSPECIFIED_RESOURCE;
   NUMBER_PERMUTATION = 0;
/*   SEPARATE SPECIFIED FROM UNSPECIFIED RESOURCES                       */
      SEPARATE_SPECIFIC_RESOURCES:
```

```
/* FORM $UNSPECIFIED_RESOURCE WITH LABELS OF AVAILABLE RESOURCE NAMES*/
      DO J = 1 TO NUMBER($TEMP_RESOURCE(I))$
      LABEL($UNSPECIFIED_RESOURCE(NEXT)) =
            LABEL($TEMP_RESOURCE(I)(J))$
      END$
/*  CONSIDER EACH REQUIRED RESOURCE TYPE AND NAME                      */
      DO J = 1 TO NUMBER($ABSTRACT.RESOURCE (I))$
/*  IS THIS RESOURCE ELEMENT SPECIFIED TO BE USED?                     */
      IF LABEL($ABSTRACT.RESOURCE (I)(J)) = ''
         THEN DO$
/*  PLACE THE INDEX OF THIS REQUIRED RESOURCE ELEMENT IN               */
/*  $REQUIRED_INDICES                                                  */
            $REQUIRED_INDICES(NEXT) = J$
            GO TO END_SEPARATE_SPECIFIED$
            END$
/*  CALCULATE INTERVALS OF START TIMES OF SPECIFIED RESOURCES          */
         ELSE DO$
            PRUNE $SPECIFIED_PROFILE$
            IF $ABSTRACT.RESOURCE (I)(J)(1).INTERVAL IDENTICAL TO
                                                              $NULL
               THEN CALL DURATION($ABSTRACT.JOB_INTERVAL,
                  SPECIFIED_DURATION)$
               ELSE SPECIFIED_DURATION = $ABSTRACT.RESOURCE(I)
                  (J)(LAST).INTERVAL.END - $ABSTRACT.RESOURCE(I)(J)
                  (1).INTERVAL.START$
            DO K = 1 TO NUMBER($TEMP_RESOURCE(I)(J).AVAILABLE)$
               CALL DURATION($TEMP_RESOURCE(I)(J).AVAILABLE(K).
                  INTERVAL,AVAILABLE_DURATION)$
               IF AVAILABLE_DURATION < SPECIFIED_DURATION
                  THEN GO TO END_SPEC_INTERVAL$
                  ELSE DO$
                     DELAY = $ABSTRACT.RESOURCE(I)(J)(1).INTERVAL.
                        START - $ABSTRACT.JOB_INTERVAL.START$
                     $SPECIFIED_PROFILE(NEXT).START = $TEMP_RESOURCE
                        (I)(J).AVAILABLE(K).INTERVAL.START - DELAY$
                     $SPECIFIED_PROFILE(LAST).END = $TEMP_RESOURCE
                        (I)(J).AVAILABLE(K).INTERVAL.END -
                        SPECIFIED_DURATION - DELAY$
                     IF $SPECIFIED_PROFILE(LAST).END < START
                        THEN PRUNE $SPECIFIED_PROFILE(LAST)$
                        ELSE IF $SPECIFIED_PROFILE(LAST).START <
                                                              START
                           THEN $SPECIFIED_PROFILE(LAST).START =
                                                              START$
                           ELSE$
                     END$
            END_SPEC_INTERVAL:END$
/*  FIND INTERSECTION OF START-TIME INTERVALS                          */
            CALL INTERVAL_INTERSECT($INTERSECTION_SPECIFIED,
               $SPECIFIED_PROFILE,$INTERSECT)$
            GRAFT $INTERSECT AT $INTERSECTION_SPECIFIED$
```

```
/*  PRUNE THE SPECIFIED RESOURCE NAME FROM $UNSPECIFIED_RESOURCE        */
              PRUNE $UNSPECIFIED_RESOURCE
                   .#LABEL($ABSTRACT.RESOURCE(I)(J));
              $SPECIFIED_RESOURCE(I)(NEXT) = LABEL($ABSTRACT.RESOURCE
                   (I)(J));
              END;
/*  HAVE ALL REQUIRED RESOURCE ELEMENTS BEEN CONSIDERED?               */
        END_SEPARATE_SPECIFIED:END;
/*  FORM THE NEXT PERMUTATION OF AVAILABLE RESOURCES TAKEN THE         */
/*  NUMBER REQUIRED AT A TIME                                           */
     IF $REQUIRED_INDICES IDENTICAL TO $NULL
        THEN DO;
           CALL INTERVAL_INTERSECT($INTERSECTION_INITIAL,
              $INTERSECTION_SPECIFIED,$INTERSECT);
           GRAFT $INTERSECT AT $INTERSECTION_INITIAL;
           END;
     FIND_PERMUTATION:
        DO FOR ALL PERMUTATIONS OF $UNSPECIFIED_RESOURCE
           TAKEN NUMBER($REQUIRED_INDICES) AT A TIME;
        $INTERSECTION_INITIAL = $REQUESTED_INTERVAL;
        NUMBER_PERMUTATION = NUMBER_PERMUTATION + 1;
/*  DO THE DESCRIPTORS OF REQUIRED AND AVAILABLE RESOURCES MATCH?      */
        ELEMENT_CHECK:
           DO J = 1 TO NUMBER($REQUIRED_INDICES);
              INDEX = $REQUIRED_INDICES(J);
              LABEL($NAME) = LABEL($PERMUTATION(J));
           IF LABEL($ABSTRACT.RESOURCE(I)(INDEX)) ¬= ''
              THEN GO TO END_ELEMENT_CHECK;
              ELSE DO K = 1 TO NUMBER($ABSTRACT.RESOURCE (I)(INDEX));
                 DO L = 1 TO NUMBER($TEMP_RESOURCE(I).#LABEL($NAME).
                    AVAILABLE);
                 IF ( $ABSTRACT.RESOURCE(I)(INDEX)(K).DESCRIPTOR(1).
                       INITIAL SUBSET OF $TEMP_RESOURCE(I).#LABEL($NAME
                       ).AVAILABLE(L).DESCRIPTOR(1).INITIAL |
                       $TEMP_RESOURCE(I).#LABEL($NAME).AVAILABLE(L).
                       DESCRIPTOR(1).INITIAL SUBSET OF $ABSTRACT.
                       RESOURCE(I)(INDEX)(K).DESCRIPTOR(1).INITIAL )
                    THEN GO TO END_ELEMENT_CHECK;
                    ELSE;
                 END;
/*  HAVE ALL ELEMENTS OF THIS PERMUTATION BEEN CHECKED?                 */
                 END;
              GO TO NEXT_PERMUTATION;
           END_ELEMENT_CHECK:END;
/*  CALCULATE INTERVALS OF START TIME                                   */
        DETERMINE_INTERVALS:
           DO J = 1 TO NUMBER($REQUIRED_INDICES);
              PRUNE $AVAIL_PROFILE;
              INDEX = $REQUIRED_INDICES(J);
              LABEL($NAME) = LABEL($PERMUTATION(J));
              IF $ABSTRACT.RESOURCE (I)(INDEX)(1).INTERVAL IDENTICAL
```

```
      TO $NULL
   THEN DO;
      CALL DURATION($ABSTRACT.JOB_INTERVAL,
         REQUIRED_DURATION);
      GO TO START_K_LOOP;
      END;
IF NUMBER($ABSTRACT.RESOURCE (I)(INDEX)) > 1
   THEN DO;
      DO L = 1 TO NUMBER($ABSTRACT.RESOURCE (I)(INDEX));
      $REQ_INT(L) = $ABSTRACT.RESOURCE (I)(INDEX)(L).
         INTERVAL;
      END;
      DO L = 1 TO NUMBER($TEMP_RESOURCE(I)(J).IN_USE);
      $ASSIGN_INT(L) = $TEMP_RESOURCE(I)(J).IN_USE(L).
         INTERVAL;
      END;
      DO L = 1 TO NUMBER($TEMP_RESOURCE(I)(J).AVAILABLE);
      $AVAIL_INT(L) = $TEMP_RESOURCE(I)(J).AVAILABLE(L).
         INTERVAL;
      END;
      DO L = 1 TO NUMBER($REQ_INT);
         DO K = L TO NUMBER($AVAIL_INT);
         DELTA = $AVAIL_INT(K).START - $REQ_INT(L).START;
            DO LL = 1 TO NUMBER($REQ_INT);
            $TEMP_INT(LL).START = $REQ_INT(LL).START +
               DELTA;
            IF ($TEMP_INT(LL).START < START | $TEMP_INT
               (LL).START > END) THEN GO TO NEXT_TRIAL;
            $TEMP_INT(LL).END = $REQ_INT(LL).END + DELTA;
            IF $TEMP_INT(LL).END > END
               THEN GO TO NEXT_TRIAL;
            END;
         CALL INTERVAL_INTERSECT($TEMP_INT,$ASSIGN_INT
            ,$INTERSECT);
         IF $INTERSECT IDENTICAL TO $NULL
            THEN DO;
               DO KK = 1 TO NUMBER($AVAIL_INT);
                  DO LL = 1 TO NUMBER($TEMP_INT);
                  WINDOW = $AVAIL_INT(KK).END -
                     $TEMP_INT(LL).END;
                  IF WINDOW >= 0
                     THEN $WINDOW_SET(NEXT) = WINDOW;
                  END;
               END;
               CALL FIND_MIN($WINDOW_SET,$INDICES,
                  VAL_MIN);
               $AVAIL_PROFILE(NEXT).START = $TEMP_INT(1).
                  START;
               $AVAIL_PROFILE(LAST).END = VAL_MIN +
                  $TEMP_INT(1).START;
               GO TO NEXT_TRIAL;
```

```
                        END;
                    ELSE;
                NEXT_TRIAL: END;
            END;
        END;
        ELSE CALL DURATION($ABSTRACT.RESOURCE (I)(INDEX)(1).
            INTERVAL,REQUIRED_DURATION);
/*  DELETE INTERVALS WHICH ARE TOO SHORT    */
        START_K_LOOP:
            DO K = 1 TO NUMBER($TEMP_RESOURCE(I).#LABEL($NAME).
                AVAILABLE);
            CALL DURATION($TEMP_RESOURCE(I).#LABEL($NAME).
                    AVAILABLE(K).INTERVAL,AVAILABLE_DURATION);
            IF AVAILABLE_DURATION < REQUIRED_DURATION
                THEN GO TO END_INTERVAL;
                ELSE DO;
                    DELAY = $ABSTRACT.RESOURCE(I)(INDEX)(1).
                            INTERVAL.START - $ABSTRACT.JOB_INTERVAL.
                            START;
                    $AVAIL_PROFILE(NEXT).START = $TEMP_RESOURCE
                        (I).#LABEL($NAME).AVAILABLE(K).INTERVAL.
                        START - DELAY;
                    $AVAIL_PROFILE(LAST).END = $TEMP_RESOURCE(I)
                        .#LABEL($NAME).AVAILABLE(K).INTERVAL.END -
                        REQUIRED_DURATION - DELAY;
                    IF $AVAIL_PROFILE(LAST).END < START
                        THEN PRUNE $AVAIL_PROFILE(LAST);
                        ELSE IF $AVAIL_PROFILE(LAST).START < START
                            THEN $AVAIL_PROFILE(LAST).START = START;
                            ELSE;
                    END;
            END_INTERVAL:END;
/*  FIND INTERSECTION OF START-TIME INTERVALS                          */
                CALL INTERVAL_INTERSECT($INTERSECTION_INITIAL,
                    $AVAIL_PROFILE,$INTERSECT);
                GRAFT $INTERSECT AT $INTERSECTION_INITIAL;
                $TEMP_ASSIGN(I)(NUMBER_PERMUTATION).RESOURCE_NAME(J) =
                    LABEL($PERMUTATION(J));
        END;
/*  INCLUDE SPECIFIED RESOURCES IN INTERVAL FOR THIS PERMUTATION      */
        CALL INTERVAL_INTERSECT($INTERSECTION_INITIAL,
            $INTERSECTION_SPECIFIED,$INTERSECT);
        GRAFT $INTERSECT AT $INTERSECTION_INITIAL;
                LABEL($TEMP_ASSIGN(I)) = LABEL($ABSTRACT.RESOURCE(I));
                $TEMP_ASSIGN(I)(NUMBER_PERMUTATION).INTERVAL =
                    $INTERSECTION_INITIAL;
/*  HAVE ALL PERMUTATIONS OF AVAILABLE RESOURCES OF THIS TYPE BEEN    */
/*  CONSIDERED?                                                        */
        NEXT_PERMUTATION:END;
/*  HAS AT LEAST ONE INTERVAL BEEN IDENTIFIED?                         */
        IF NUMBER_PERMUTATION = 0 THEN DO;
```

```
              CALL INTERVAL_INTERSECT($INTERSECTION_INITIAL,
                   $INTERSECTION_SPECIFIED,$INTERSECT);
              GRAFT $INTERSECT AT $UNION(I);
              GO TO NEXT_RESOURCE_TYPE;
              END;
              ELSE;
           DO NP = 1 TO NUMBER($TEMP_ASSIGN(I));
           IF $TEMP_ASSIGN(I)(NP).INTERVAL IDENTICAL TO $NULL
              THEN GO TO NEXT_TEMP_ASSIGN;
              ELSE DO;
/*  FIND THE UNION OF INTERVALS FOR THIS RESOURCE TYPE                        */
                 $DUMMY_INTERVAL = $TEMP_ASSIGN(I)(NP).INTERVAL;
           CALL INTERVAL_UNION($UNION(I),$DUMMY_INTERVAL,
              $INT_UNION);
           GRAFT $INT_UNION AT $UNION(I);
              END;
           NEXT_TEMP_ASSIGN: END;
/*  HAVE ALL RESOURCES OF THIS TYPE BEEN CONSIDERED?                          */
/*  HAVE ALL RESOURCE TYPES BEEN CONSIDERED?                                  */
  NEXT_RESOURCE_TYPE:END;
/*  FIND THE INTERSECTION OF INTERVALS FOR ALL RESOURCE TYPES                 */
  DO I = 1 TO NUMBER($UNION);
     CALL INTERVAL_INTERSECT($INTERSECTION_FINAL,$UNION(I),
        $INTERSECT);
     GRAFT $INTERSECT AT $INTERSECTION_FINAL;
  END;
/*  INSERT THIS INTERVAL AS $AVAILABLE_WINDOWS.ANY_RESOURCE_SET               */
  $AVAILABLE_WINDOWS.ANY_RESOURCE_SET = $INTERSECTION_FINAL;
/*  DETERMINE RESOURCE SET WITH EARLIEST START TIME                           */
  FIND_RESOURCE_TYPE:
     BEGIN_TIME = $INTERSECTION_FINAL(1).START;
     END_TIME = $INTERSECTION_FINAL(1).END;
     ASSIGN_TYPE:
        DO I = 1 TO NUMBER($TEMP_ASSIGN);
        ASSIGN_PERMUTATION_NUMBER:
           DO J = 1 TO NUMBER($TEMP_ASSIGN(I));
           ASSIGN_INTERVAL:
              DO K = 1 TO NUMBER($TEMP_ASSIGN(I)(J).INTERVAL );
/*  FIND INTERSECTION OF INTERVALS FOR THIS RESOURCE SET WITH                 */
/*  INTERVALS FOR ALL SETS AND INSERT AS $AVAILABLE_WINDOWS.SAME_SET          */
                 IF BEGIN_TIME >= $TEMP_ASSIGN(I)(J).INTERVAL (K).START
                    & BEGIN_TIME <= $TEMP_ASSIGN(I)(J).INTERVAL (K).END
                    THEN IF END_TIME > $TEMP_ASSIGN(I)(J).INTERVAL (K).
                       END
                       THEN DO;
                          END_TIME = $TEMP_ASSIGN(I)(J).INTERVAL (K).
                             END;
                          $INTER_ASSIGN(I).RESOURCE  =
                             $TEMP_ASSIGN(I)(J).RESOURCE_NAME;
                          LABEL($INTER_ASSIGN(I)) = LABEL($TEMP_ASSIGN
                                (I));
```

```
                    $INTER_ASSIGN(I).INTERVAL =
                       $TEMP_ASSIGN(I)(J).INTERVAL (1);
                    DO NS = 1 TO NUMBER($SPECIFIED_RESOURCE(I));
                       $INTER_ASSIGN(I).RESOURCE(NEXT) =
                             $SPECIFIED_RESOURCE(I)(NS);
                       END;
                    END;
                 ELSE IF $INTER_ASSIGN(I).RESOURCE IDENTICAL TO
                       $NULL
                    THEN DO;
                       $INTER_ASSIGN(I).RESOURCE =
                          $TEMP_ASSIGN(I)(J).RESOURCE_NAME;
                       LABEL($INTER_ASSIGN(I)) =
                             LABEL($TEMP_ASSIGN(I));
                       $INTER_ASSIGN(I).INTERVAL =
                          $TEMP_ASSIGN(I)(J).INTERVAL (1);
                       DO NS = 1 TO NUMBER($SPECIFIED_RESOURCE);
                          $INTER_ASSIGN(I).RESOURCE(NEXT) =
                                $SPECIFIED_RESOURCE(I)(NS);
                          END;
                       END;
                    ELSE;
              ELSE;
           END;
        END;
     END;
  COMPLETE_SAME_SET:
     $SAME_SET = $INTERSECTION_FINAL;
     DO I = 1 TO NUMBER($TEMP_ASSIGN);
        CALL INTERVAL_INTERSECT($SAME_SET,$INTER_ASSIGN(I).INTERVAL,
              $INTERSECT);
        GRAFT $INTERSECT AT $SAME_SET;
     END;
     $AVAILABLE_WINDOWS.SAME_RESOURCE_SET = $SAME_SET;
/*   CONSTRUCT $CONCRETE                                              */
     GRAFT $ABSTRACT AT $CONCRETE;
     DO I = 1 TO NUMBER($INTER_ASSIGN);
        DO J = 1 TO NUMBER($REQUIRED_INDICES);
           INDEX = $REQUIRED_INDICES(J);
           LABEL($CONCRETE.RESOURCE(I)(INDEX)) = $INTER_ASSIGN(I).
                 RESOURCE(J);
           END;
        DO FOR ALL SUBNODES OF $CONCRETE.RESOURCE(I) USING $CON_NAME;
           DO FOR ALL SUBNODES OF $CON_NAME USING $SUB_NAME;
              $SUB_NAME.INTERVAL.START = $SUB_NAME.INTERVAL.START +
                    $INTER_ASSIGN(I).INTERVAL.START;
              $SUB_NAME.INTERVAL.END = $SUB_NAME.INTERVAL.END +
                    $INTER_ASSIGN(I).INTERVAL.START;
              END;
           END;
        END;
```

```
  $CONCRETE.JOB_INTERVAL.START = $CONCRETE.JOB_INTERVAL.START +
        $SAME_SET(FIRST).START;
  $CONCRETE.JOB_INTERVAL.END = $CONCRETE.JOB_INTERVAL.END +
        $SAME_SET(FIRST).START;
  $CONCRETE.UNAVAIL_TIME = $AVAILABLE_WINDOWS.SAME_RESOURCE_SET(1).END;
  LABEL($CONCRETE) = '$CONCRETE';
  LABEL($AVAILABLE_WINDOWS) = '$AVAILABLE_WINDOWS';
END_NEXTSET; END;
```

2.4.13 RESOURCE_PROFILE

C.3

2.4.13 RESOURCE_PROFILE

2.4.13.1 Purpose and Scope

In project scheduling the resources are assigned from a pool and, upon completion of the job, are returned to the pool of available resources. Thus, the quantity of a given resource, available in the pool for a given time interval, is required to determine the advisability of scheduling a given job at a given time. Further, if sufficient resources are not available at the desired time, a contingency level of resources may be considered. This module determines the profile of available resources over a given time interval for both a "normal" and "contingency" level of resource. If contingency levels are not to be considered, they are set equal to the normal level. Certain functional characteristics of project scheduling also create the need to determine the usage of a pool assigned over a given interval (such as in attempts to level resource usage). Therefore, this module also determines the profile of the assigned portion of the pool and defines the association of jobs that make up the usage profile.

2.4.13.2 Modules Called

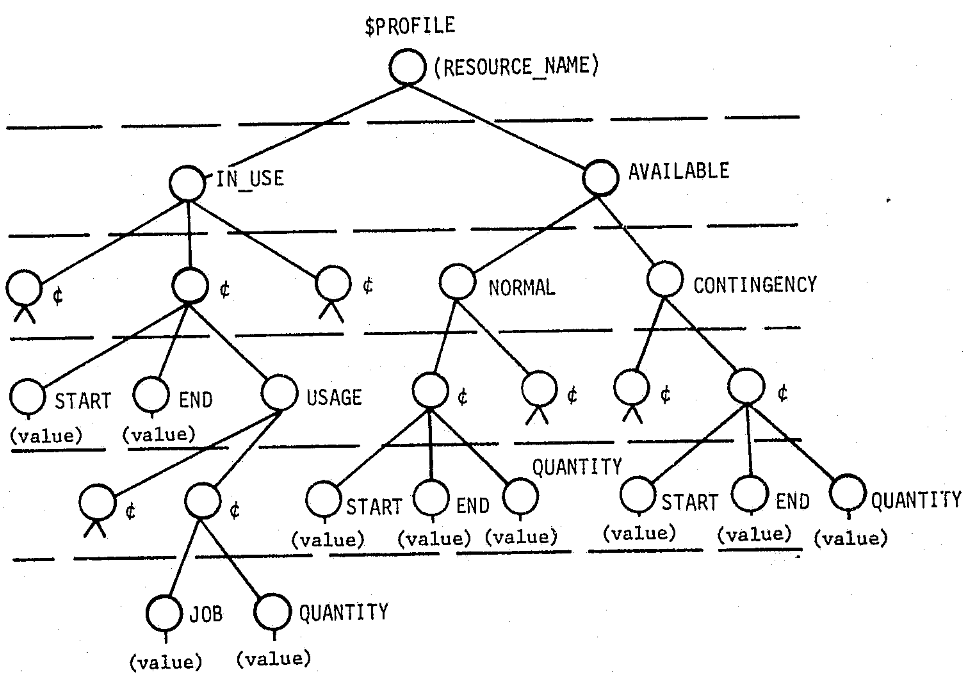
None.

2.4.13.3 Module Input

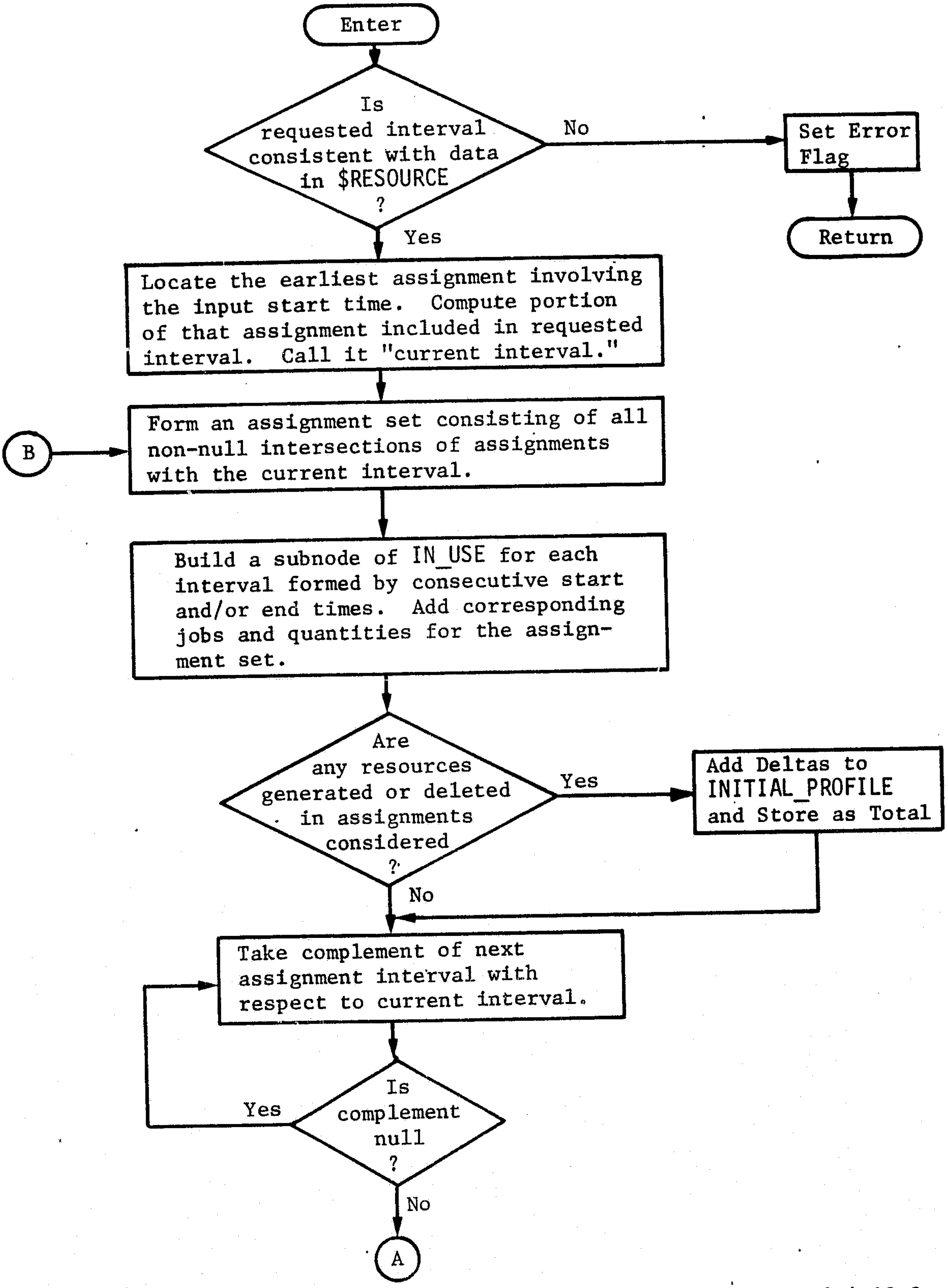
The input to this module will consist of that portion of the $RESOURCE tree for the pooled resource type and name whose profile is to be generated. This is the substructure of a second-level subnode of $RESOURCE and is referred to as $RESOURCE_NAME. Further input will consist of the time interval for which the profile is to be generated, $REQUIRED_INTERVAL.

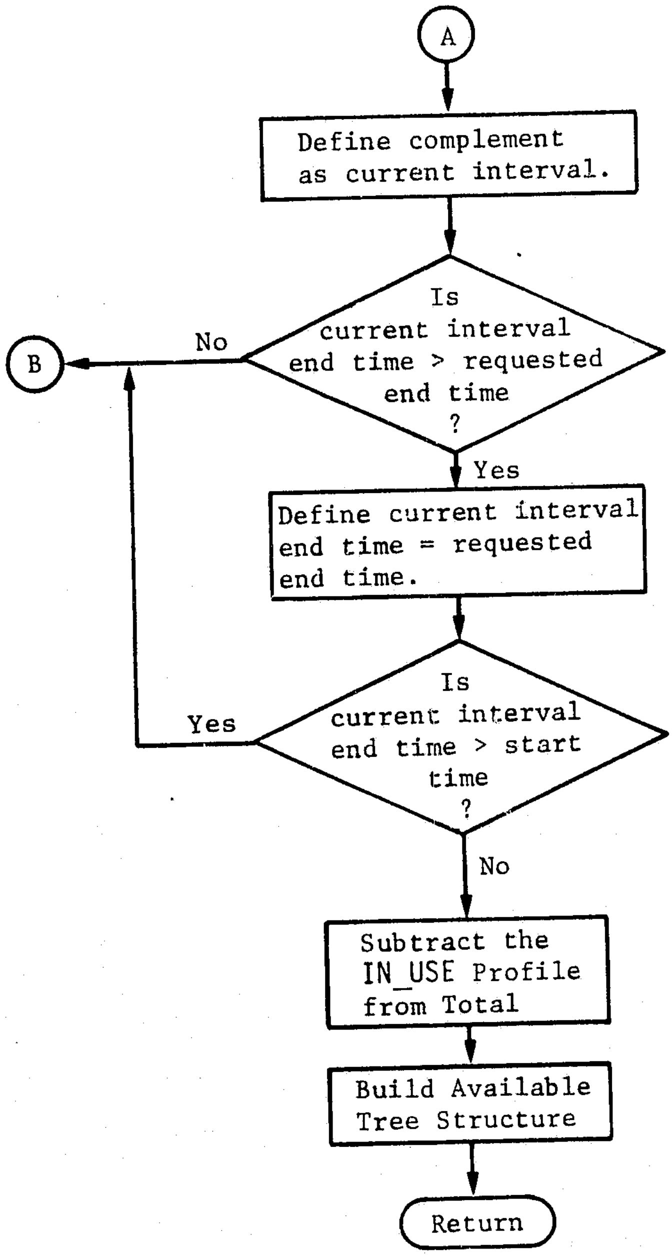
2.4.13.4 Module Output

The output of this module will consist of a tree structure as shown in the sketch. The IN_USE portion of the tree defines the quantity of the pooled resource assigned to a job for a given time interval. Therefore, the sum of the quantities for a given interval define the total IN_USE resources for that interval. The span of intervals listed will be consistent with the input interval requested. The available portion of the tree defines the quantity of resource pool that is unassigned for both a normal and contingency mode of operation. These quantities are determined from the initial levels defined in $RESOURCE, the allocations recorded in the ASSIGNMENT portion of $RESOURCE, and the resources DELETED or GENERATED recorded in the ASSIGNMENT portion of $RESOURCE.



2.4.13.5 Functional Block Diagram




A
Define complement as current interval.
Is current interval end time > requested end time ?
No
B
Yes
Define current interval end time = requested end time.
Is current interval end time > start time ?
Yes
No
Subtract the IN_USE Profile from Total
Build Available Tree Structure
Return

2.4.13.6 Typical Applications

This module would be used to determine availability of resources for project scheduling, or as a potential output to a user or scheduling heuristic that was attempting to allocate resources in a predetermined manner.

2.4.13.7 DETAILED DESIGN

This module first checks the standard data structure, $RESOURCE, to determine any assignments of the requested resource element during the time period of interest. If there are none, a message is written and control is returned to the calling program. The earliest assignment about the input start time is located and defined as the "current interval." The start of the current interval will never be earlier than the input start time.

All other assignment intervals are compared with the current interval, and non-null intersections are placed in $ASSIGN_SET. The start and end times of the intersections are also placed in $TEMP_SET. The times in $TEMP_SET are then ordered and non-equal pairs are used to construct the subnodes of $PROFILE.IN_USE. Job usage and quantities are then added to the IN_USE subnode.

If any resources are generated or deleted during the assignment under consideration, as noted by a change in quantity from initial to final, the deltas are added to the INITIAL_PROFILE subnode for both normal and contingency usage.

The complement of the next assignment with respect to the current interval is developed. If this complement is not null, it is defined as a new "current interval." If this new current interval is within the time period of interest the process is repeated. Otherwise, the available tree structure is developed by subtracting the in use portion from the initial profile.

The functional block diagram can be used as a module flowchart.

2.4.13.8 Internal Variable and Tree Name Definitions

DELTA	-	The number of a given resource elements either generated or deleted during an assignment.
QUAN_USE	-	The total number of a resource element in use during a given IN_USE interval.
$ASSIGN_SET	-	A data structure having the form of the ASSIGNMENT node in $RESOURCE but with the interval set equal to the intersection with "current interval".
$ASSIGNMENT	-	A pointer to each subnode of the ASSIGNMENT node in $RESOURCE.
$BUILD_PROFILE	-	A single node tree used as a flag to specify whether or not to construct the initial profile.
$COMPLEMENT	-	Output interval from internal procedure INTERVAL_COMPLEMENT.
$CURRENT_INTERVAL	-	Output interval from procedure INTERVAL_INTERSECTION.
$TEMP_RESOURCE	-	A temporary data structure equivalent to $RESOURCE but for only the single resource element being considered.
$TEMP_SET	-	A set of start and end times of non-null intersections of assignments with "current-interval"

2.4.13.9 Commented Code

```
RESOURCE_PROFILE: PROCEDURE($RESOURCE_NAME,$REQUIRED_INTERVAL,
      $PROFILE) OPTIONS(EXTERNAL);
DECLARE $BUILD_PROFILE LOCAL;
DECLARE MM,$ASSIGNMENT,$ELEMENT LOCAL;
DECLARE I,J,K,L,KK,LL,DELTA,QUAN_USE,$ASSIGN_SET,$COMPLEMENT LOCAL;
DECLARE $CURRENT_INTERVAL,$INTERSECT,$TEMP_RESOURCE,$TEMP_SET LOCAL;
LABEL($PROFILE) = LABEL($RESOURCE_NAME);
$BUILD_PROFILE = 'YES';
/*  IS REQUESTED INTERVAL CONSISTENT WITH DATA IN $RESOURCE?          */
IF $REQUIRED_INTERVAL.END < $RESOURCE_NAME.INITIAL_TIME
   THEN DO;
      WRITE 'INTERVAL REQUESTED PRIOR TO RESOURCE AVAILABILITY';
      RETURN;
      END;
   ELSE IF $REQUIRED_INTERVAL.END < $RESOURCE_NAME.ASSIGNMENT(FIRST).
         INTERVAL.START
      THEN DO;
         WRITE 'INTERVAL REQUESTED PRIOR TO FIRST ASSIGNMENT';
         $PROFILE.AVAILABLE = $RESOURCE_NAME.INITIAL_PROFILE;
         RETURN;
         END;
      ELSE;
IF $REQUIRED_INTERVAL.START > $RESOURCE_NAME.ASSIGNMENT(LAST).INTERVAL
      .END
   THEN DO;
      WRITE 'INTERVAL REQUESTED IS LATER THAN LAST ASSIGNMENT';
      RETURN;
      END;
   ELSE $TEMP_RESOURCE = $RESOURCE_NAME;
/*  LOCATE THE EARLIEST ASSIGNMENT INVOLVING THE INPUT START TIME.    */
/* COMPUTE PORTION OF THAT ASSIGNMENT INCLUDED IN REQUESTED INTERVAL.*/
/*    CALL IT 'CURRENT INTERVAL'.                                     */
DO I = 1 TO NUMBER($TEMP_RESOURCE.ASSIGNMENT);
IF $TEMP_RESOURCE.ASSIGNMENT(I).INTERVAL.END > $REQUIRED_INTERVAL.START
   THEN IF $TEMP_RESOURCE.ASSIGNMENT(I).INTERVAL.END <=
         $REQUIRED_INTERVAL.END
      THEN IF $TEMP_RESOURCE.ASSIGNMENT(I).INTERVAL.START <=
            $REQUIRED_INTERVAL.START
         THEN DO;
            $CURRENT_INTERVAL.START = $REQUIRED_INTERVAL.START;
            $CURRENT_INTERVAL.END  = $TEMP_RESOURCE.ASSIGNMENT(I).
                  INTERVAL.END;
            GO TO POINT_B;
            END;
         ELSE DO;
            $CURRENT_INTERVAL = $TEMP_RESOURCE.ASSIGNMENT(I).INTERVAL;
            GO TO POINT_B;
            END;
      ELSE IF $TEMP_RESOURCE.ASSIGNMENT(I).INTERVAL.START <
            $REQUIRED_INTERVAL.END
         THEN DO;
```

```
            $CURRENT_INTERVAL.START = $TEMP_RESOURCE.ASSIGNMENT(I).
                INTERVAL.START;
            $CURRENT_INTERVAL.END = $REQUIRED_INTERVAL.END;
            GO TO POINT_B;
            END;
          ELSE;
    ELSE;
END;
I = I - 1;
/*  FORM AN ASSIGNMENT SET CONSISTING OF ALL NON-NULL INTERSECTIONS   */
/*    OF ASSIGNMENTS WITH THE 'CURRENT INTERVAL'.                     */
POINT_B:
PRUNE $ASSIGN_SET;
PRUNE $TEMP_SET;
DO J = 1 TO NUMBER($TEMP_RESOURCE.ASSIGNMENT);
CALL INTERVAL_INTERSECT($TEMP_RESOURCE.ASSIGNMENT(J).INTERVAL,
    $CURRENT_INTERVAL,$INTERSECT);
IF $INTERSECT IDENTICAL TO $NULL
    THEN GO TO END_LOOP_J;
    ELSE DO K = 1 TO NUMBER($INTERSECT);
        IF $INTERSECT(K).START = $INTERSECT(K).END THEN GO TO END_K_LOOP;
        $ASSIGN_SET(NEXT) = $TEMP_RESOURCE.ASSIGNMENT(J);
        $ASSIGN_SET(LAST).INTERVAL = $INTERSECT(K);
        IF $ASSIGN_SET(LAST).INTERVAL.END < $TEMP_RESOURCE.ASSIGNMENT(J).
            INTERVAL.END
          THEN $ASSIGN_SET(LAST).DESCRIPTOR(LAST).FINAL.QUANTITY =
               $TEMP_RESOURCE.ASSIGNMENT(J).DESCRIPTOR(LAST).INITIAL.
               QUANTITY;
          $TEMP_SET(NEXT) = $INTERSECT(K).START;
          $TEMP_SET(NEXT) = $INTERSECT(K).END;

          $TEMP_SET(NEXT) = $INTERSECT(K).END;
      END_K_LOOP:  END;
END_LOOP_J: END;
/*  BUILD A SUBNODE OF IN_USE FOR EACH INTERVAL FORMED BY CONSECUTIVE*/
/*  START AND/OR END TIMES.  ADD CORRESPONDING JOBS AND QUANTITIES   */
/*  FOR THE ASSIGNMENT SET.                                           */
ORDER $TEMP_SET BY -$ELEMENT;
DO J = 2 TO NUMBER($TEMP_SET);
IF $TEMP_SET(J-1) = $TEMP_SET(J)
    THEN GO TO END_LOOP_J2;
    ELSE DO;
        $PROFILE.IN_USE(NEXT).START = $TEMP_SET(J-1);
        $PROFILE.IN_USE(LAST).END = $TEMP_SET(J);
        DO K = 1 TO NUMBER($ASSIGN_SET);
          IF $PROFILE.IN_USE(LAST).START >= $ASSIGN_SET(K).INTERVAL.START
             THEN IF $PROFILE.IN_USE(LAST).END <= $ASSIGN_SET(K).
                  INTERVAL.END THEN DO;
                     $PROFILE.IN_USE(LAST).USAGE(NEXT).JOB = $ASSIGN_SET
                        (K).JOB_ID;
                     DO L = 1 TO NUMBER($ASSIGN_SET(K).DESCRIPTOR );
```

```
                  $PROFILE.IN_USE(LAST).USAGE(LAST).QUANTITY =
                     $PROFILE.IN_USE(LAST).USAGE(LAST).QUANTITY +
                     $ASSIGN_SET(K).DESCRIPTOR(L).INITIAL.QUANTITY;
                  END;
                  END;
         END;
         END;
   END_LOOP_J2: END;
   /* ARE ANY RESOURCES GENERATED OR DELETED IN ASSIGNMENTS CONSIDERED? */
   IF $BUILD_PROFILE = 'YES' THEN
   DO FOR ALL SUBNODES OF $TEMP_RESOURCE.ASSIGNMENT USING $ASSIGNMENT;
   $BUILD_PROFILE = 'NO';
      DELTA = 0;
      DO L = 1 TO NUMBER($ASSIGNMENT   .DESCRIPTOR);
      IF $ASSIGNMENT   .DESCRIPTOR(L).FINAL.QUANTITY
           IDENTICAL TO $NULL
         THEN GO TO END_LOOP_L;
         ELSE DELTA = DELTA + $ASSIGNMENT   .DESCRIPTOR(L).FINAL.
           QUANTITY - $ASSIGNMENT   .DESCRIPTOR(L).INITIAL.QUANTITY;
      END_LOOP_L: END;
   IF DELTA = 0 THEN GO TO END_LOOP_K;
   /*  ADD DELTAS TO INITIAL_PROFILE AND STORE AS TOTAL.                  */
      DO L = 1 TO NUMBER($TEMP_RESOURCE.INITIAL_PROFILE.NORMAL);
      IF $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L).START >= $ASSIGNMENT.
           INTERVAL.END
         THEN $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L).QUANTITY = DELTA +
            $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L).QUANTITY;
         ELSE IF $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L).END >
            $ASSIGNMENT.INTERVAL.END
               THEN DO;
                  $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L+1).START =
                     $ASSIGNMENT.INTERVAL.END;
                  $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L+1).END =
                     $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L).END;
                  $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L+1).QUANTITY =
                     $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L).QUANTITY;
                  $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L+1).QUANTITY =
                     $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L+1).QUANTITY +
                     DELTA;
                  $TEMP_RESOURCE.INITIAL_PROFILE.NORMAL(L).END =
                     $ASSIGNMENT.INTERVAL.END;
                  END;
               ELSE;
      END;
      DO L = 1 TO NUMBER($TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY);
      IF $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L).START >=
           $ASSIGNMENT.INTERVAL.END
         THEN $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L).QUANTITY =
           $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L).QUANTITY+ DELTA;
         ELSE IF $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L).END >
           $ASSIGNMENT.INTERVAL.END
```

```
         THEN DO;
            $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L+1).START =
               $ASSIGNMENT.INTERVAL.END;
            $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L+1).END =
               $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L).END;
            $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L+1).
               QUANTITY = $TEMP_RESOURCE.INITIAL_PROFILE.
               CONTINGENCY(L).QUANTITY;
            $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L+1).QUANTITY
               = $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L+1).
               QUANTITY + DELTA;
            $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(L).END =
               $ASSIGNMENT.INTERVAL.END;
            END;
         ELSE;
   END;
END_LOOP_K: END;
/*  TAKE COMPLEMENT OF NEXT ASSIGNMENT WITH RESPECT TO 'CURRENT        */
/*  INTERVAL'.                                                         */
TAKE_COMPLEMENT:
I = I + 1;
IF $TEMP_RESOURCE.ASSIGNMENT(I) IDENTICAL TO $NULL
      THEN GO TO BUILD_TREE;
CALL INTERVAL_COMPLEMENT($CURRENT_INTERVAL,$TEMP_RESOURCE.ASSIGNMENT(I)
   .INTERVAL,$COMPLEMENT);
/*  IS COMPLEMENT NULL?                                                */
IF $COMPLEMENT IDENTICAL TO $NULL
   THEN GO TO TAKE_COMPLEMENT;
/*  DEFINE COMPLEMENT AS 'CURRENT INTERVAL'.                           */
$CURRENT_INTERVAL = $COMPLEMENT;
/*  IS 'CURRENT INTERVAL' END TIME > REQUESTED END TIME?               */
IF $CURRENT_INTERVAL.END <= $REQUIRED_INTERVAL.END
   THEN GO TO POINT_B;
/*  DEFINE 'CURRENT INTERVAL' END TIME = REQUESTED END TIME.           */
$CURRENT_INTERVAL.END = $REQUIRED_INTERVAL.END;
/*  IS 'CURRENT INTERVAL' END TIME > START TIME?                       */
IF $CURRENT_INTERVAL.END > $REQUIRED_INTERVAL.START
   THEN GO TO POINT_B;
/*  SUBTRACT THE IN_USE PROFILE FROM TOTAL.                            */
/*  BUILD AVAILABLE TREE STRUCTURE.                                    */
BUILD_TREE:
DO KK = 1 TO NUMBER($PROFILE.IN_USE);
   DO LL = 1 TO NUMBER($TEMP_RESOURCE.INITIAL_PROFILE.NORMAL);
   IF $PROFILE.IN_USE(KK).START >= $TEMP_RESOURCE.INITIAL_PROFILE.
      NORMAL(LL).START
      THEN IF $PROFILE.IN_USE(KK).END <= $TEMP_RESOURCE.INITIAL_PROFILE
         .NORMAL(LL).END
         THEN DO;
            $PROFILE.AVAILABLE.NORMAL(KK) = $PROFILE.IN_USE(KK);
            PRUNE $PROFILE.AVAILABLE.NORMAL(KK).USAGE;
            QUAN_USE = 0;
```

```
            DO MM = 1 TO NUMBER($PROFILE.IN_USE(KK).USAGE);
            QUAN_USE = QUAN_USE + $PROFILE.IN_USE(KK).USAGE(MM).
               QUANTITY;
            END;
            $PROFILE.AVAILABLE.NORMAL(KK).QUANTITY = $TEMP_RESOURCE.
               INITIAL_PROFILE.NORMAL(LL).QUANTITY - QUAN_USE;
            GO TO BUILD_CONTINGENCY;
            END;
         ELSE GO TO END_LL_LOOP;
      ELSE;
   END_LL_LOOP: END;
   BUILD_CONTINGENCY:
   DO LL = 1 TO NUMBER($TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY);
   IF $PROFILE.IN_USE(KK).START >= $TEMP_RESOURCE.INITIAL_PROFILE.
      CONTINGENCY(LL).START
      THEN IF $PROFILE.IN_USE(KK).END <= $TEMP_RESOURCE.INITIAL_PROFILE
         .CONTINGENCY(LL).END
         THEN DO;
            $PROFILE.AVAILABLE.CONTINGENCY(KK) = $PROFILE.IN_USE(KK);
            PRUNE $PROFILE.AVAILABLE.CONTINGENCY(KK).USAGE;
            QUAN_USE = 0;
            DO MM = 1 TO NUMBER($PROFILE.IN_USE(KK).USAGE);
            QUAN_USE = QUAN_USE + $PROFILE.IN_USE(KK).USAGE(MM).
               QUANTITY;
            END;
            $PROFILE.AVAILABLE.CONTINGENCY(KK).QUANTITY =
            $TEMP_RESOURCE.INITIAL_PROFILE.CONTINGENCY(LL).QUANTITY
               - QUAN_USE;
            GO TO END_KK_LOOP;
            END;
         ELSE GO TO END_LOOP_LL;
      ELSE;
   END_LOOP_LL: END;
END_KK_LOOP: END;
IF $PROFILE.AVAILABLE.NORMAL(FIRST).START > $REQUIRED_INTERVAL.START
   THEN DO;
      INSERT $RESOURCE_NAME.INITIAL_PROFILE.NORMAL(FIRST) BEFORE
            $PROFILE.AVAILABLE.NORMAL(FIRST);
     $PROFILE.AVAILABLE.NORMAL(FIRST).START = $REQUIRED_INTERVAL.START;
      $PROFILE.AVAILABLE.NORMAL(FIRST).END = $PROFILE.AVAILABLE.NORMAL
            (2).START;
      END;
IF $PROFILE.AVAILABLE.CONTINGENCY(FIRST).START > $REQUIRED_INTERVAL
      .START
   THEN DO;
      INSERT $RESOURCE_NAME.INITIAL_PROFILE.CONTINGENCY(FIRST) BEFORE
            $PROFILE.AVAILABLE.CONTINGENCY(FIRST);
      $PROFILE.AVAILABLE.CONTINGENCY(FIRST).START = $REQUIRED_INTERVAL
            .START;
      $PROFILE.AVAILABLE.CONTINGENCY(FIRST).END = $PROFILE.AVAILABLE.
            CONTINGENCY(2).START;
```

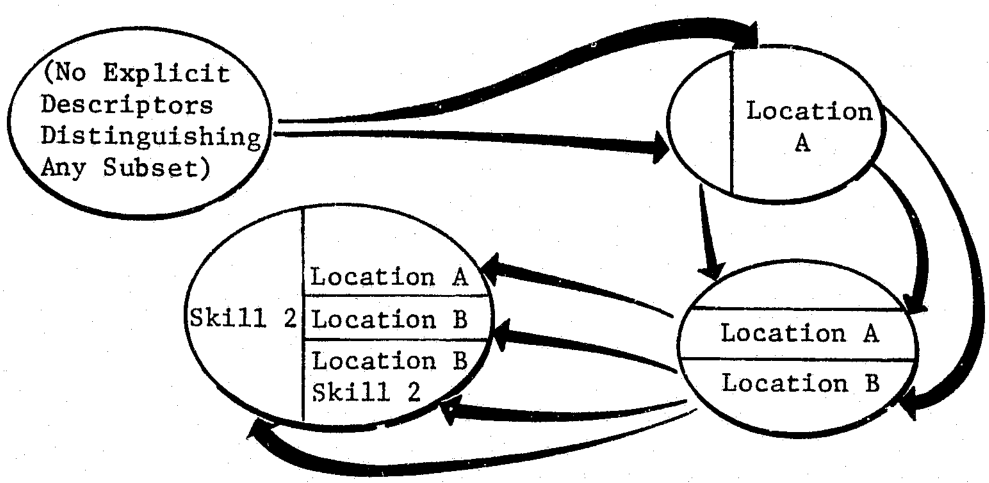
```
      END;
IF $PROFILE.AVAILABLE.NORMAL(LAST).END < $REQUIRED_INTERVAL.END
   THEN DO;
      $TEMP = $PROFILE.AVAILABLE.NORMAL(LAST).END;
      $PROFILE.AVAILABLE.NORMAL(NEXT) = $RESOURCE_NAME.INITIAL_PROFILE.
            NORMAL(LAST);
      $PROFILE.AVAILABLE.NORMAL(LAST).START = $TEMP;
      $PROFILE.AVAILABLE.NORMAL(LAST).END = $REQUIRED_INTERVAL.END;
      END;
IF $PROFILE.AVAILABLE.CONTINGENCY(LAST).END < $REQUIRED_INTERVAL.END
   THEN DO;
      $TEMP = $PROFILE.AVAILABLE.CONTINGENCY(LAST).END;
      $PROFILE.AVAILABLE.CONTINGENCY(NEXT) = $RESOURCE_NAME.
            INITIAL_PROFILE.CONTINGENCY(LAST);
      $PROFILE.AVAILABLE.CONTINGENCY(LAST).START = $TEMP;
     $PROFILE.AVAILABLE.CONTINGENCY(LAST).END = $REQUIRED_INTERVAL.END;
      END;
RETURN;

END;     /*   RESOURCE_PROFILE   */
```

2.4.14 POOLED_DESCRIPTOR_
COMPATIBILITY

2.4.14 POOLED_DESCRIPTOR_COMPATIBILITY

2.4.14.1 Purpose and Scope

This module identifies occurrences of resource description incompatibilities that arise when a single job is to be added to a schedule at a time equal to or later than a set of jobs that have already been assigned (i.e., resources assignments have been made in $RESOURCE tree). It applies to jobs that require resources from pools or from partitions of pools distinguished from other partitions by a separate set of explicit descriptors. For example, if an activity is described as requiring 13 laborors taken from location A and the laborors are not described (in $RESOURCE) as individuals, but rather as a collection of originally undistinguishable resources, then the laborors that are in location A represent a partition of the pool distinguished from other partitions only by the descriptor 'LOCATION' with value A. Since different jobs may not only change the value of 'LOCATION' but may also add new descriptors such as 'SKILL' the partitions of the original pool may proliferate as the schedule is developed. An illustration of this is shown in the sketch.

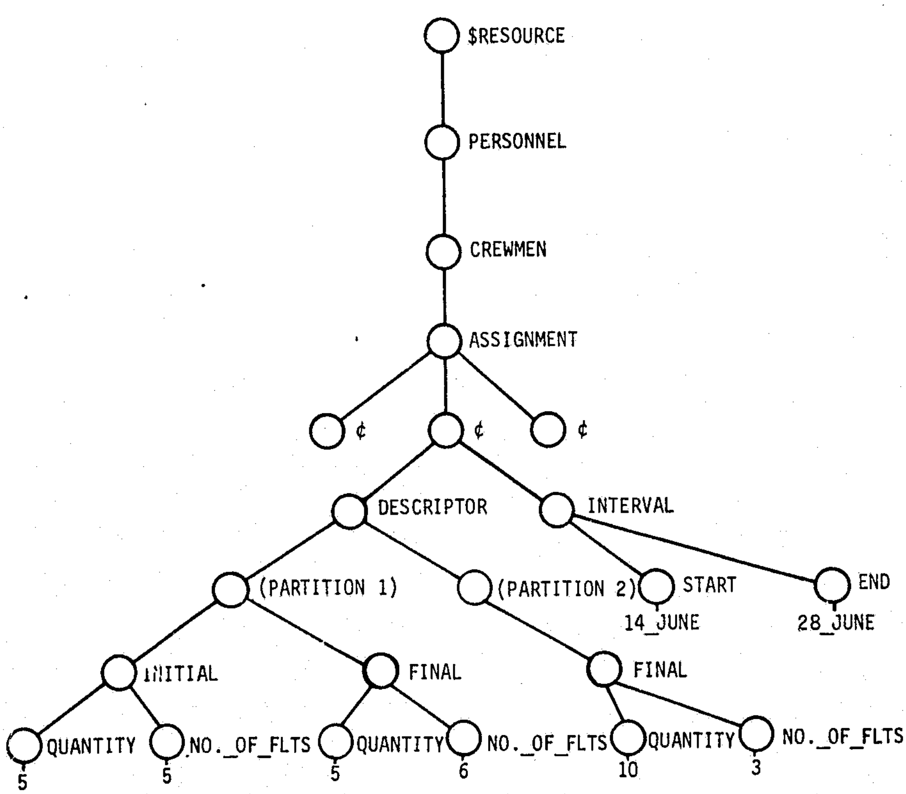


It is important to emphasize that this module applies to pooled resources where different subsets of the pools may acquire distinguishing descriptors. The module applies directly to time progressive scheduling strategies.

A time transcendent strategy to schedule jobs whose required resources are described as pools with varying explicit descriptors must necessarily be a very complex algorithm. An algorithm that places such a job on a timeline between two scheduled jobs would have to resolve any descriptor conflicts that occurred after the assignment time of the new job; it is likely that this conflict resolution would be done by working progressively from the assignment time of the inserted job to the last assignment time in the schedule. Thus, the conflict resolution strategy is likely to be time progressive even within a time transcendent assignment strategy. Therefore, through repetitive application, this module will have applicability to time transcendent algorithms. The problem classes to which this module applies are illustrated.

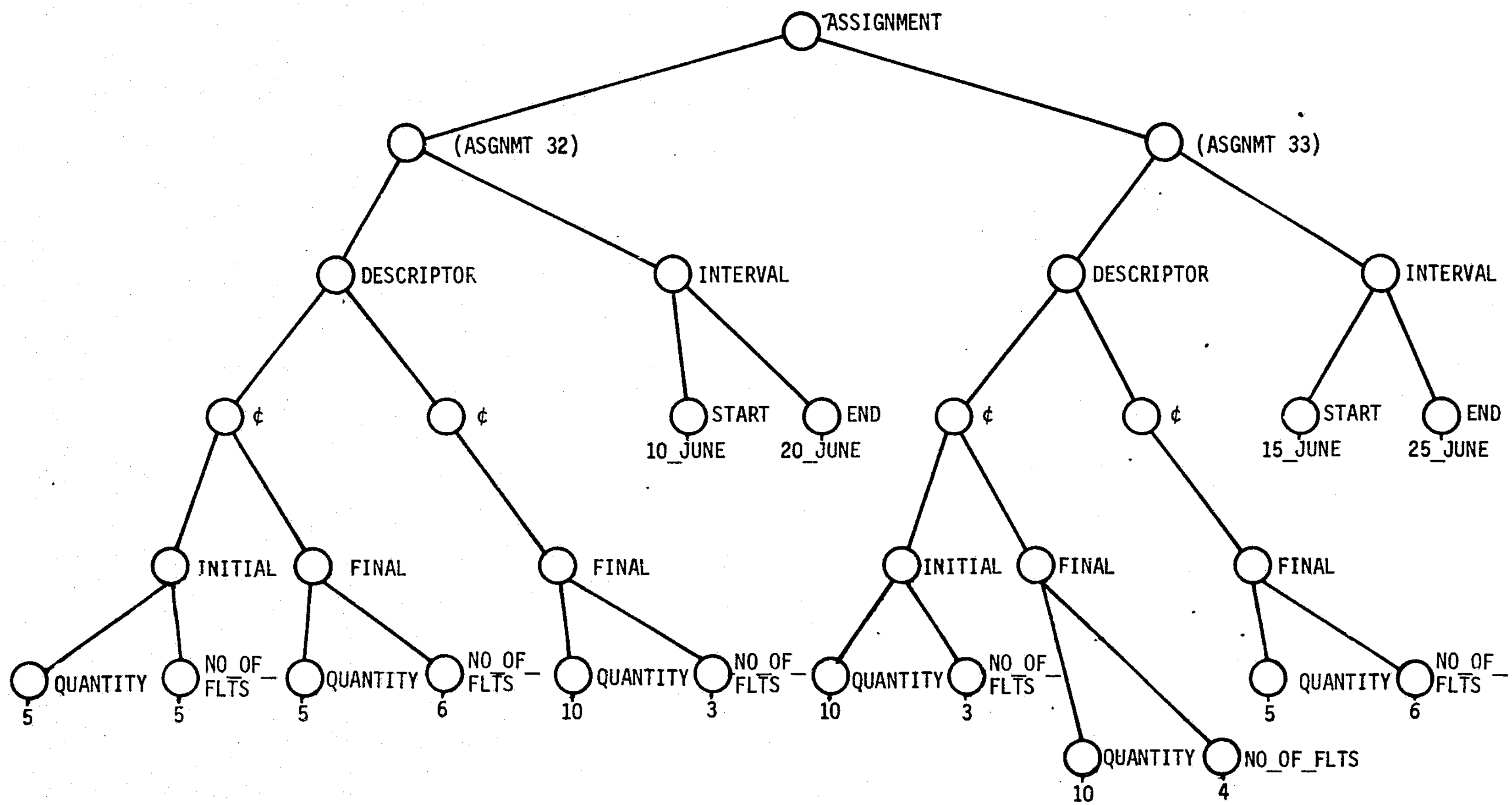
	Time Progressive Assignment Algorithms		Time Transcendent Assignment Algorithms	
	Implicit Descriptors Only	Explicit Descriptors	Implicit Descriptors Only	Explicit Descriptors
Item Specific Resources				
Pooled Resources		This module applies		Poor assignment strategy for this type of modeling: this algorithm applies with repeated calls.

This module assumes some conventions about the structure of the ASSIGNMENT node of any resource that is a pooled resource (i.e., for which the node CLASS has a value 'POOLED'). A pooled resource that has explicit descriptors must contain a subnode of DESCRIPTOR for each partition of the pool. Those partitions that are being used in the assignment interval are distinguished from those not used in the interval by the appearance of the 'INITIAL' and the 'FINAL' nodes. Thus, the availability of a particular partition of a pool is precluded during the assignment interval only if that partition has a subnode of the 'DESCRIPTOR' node labeled 'INITIAL'. This convention is illustrated in the following structure:



The structure illustrates one assignment for the pooled resource named CREWMEN and indicates that between 14 June and 28 June five crewmen were assigned (indicated by the appearance of the INITIAL node) and 10 crewmen were not assigned.

A slight generalization of the convention is required for pools that have overlapping assignments. The sketch illustrates the assumed structure of a portion of the ASSIGNMENT substructure for a pool of CREWMEN that has been separated into two partitions by previous assignments. Two assignments whose intervals overlap are shown.


ASSIGNMENT
(ASGNMT 32)
(ASGNMT 33)
DESCRIPTOR
INTERVAL
DESCRIPTOR
INTERVAL
¢
¢
START
END
¢
¢
START
END
10_JUNE
20_JUNE
15_JUNE
25_JUNE
INITIAL
FINAL
FINAL
INITIAL
FINAL
FINAL
QUANTITY
NO_OF_FLTS
QUANTITY
NO_OF_FLTS
QUANTITY
NO_OF_FLTS
QUANTITY
NO_OF_FLTS
QUANTITY
NO_OF_FLTS
5
5
5
6
10
3
10
3
5
6
QUANTITY
NO_OF_FLTS
10
4

Note in the illustration that the availability of the crewmen in the 10-man partition during the overlap of the assignment intervals (15 June through 20 June) cannot be determined correctly by merely noting the absence of the 'INITIAL' node in the first assignment. This is because that partition is used in the second assignment. Therefore, the convention adopted requires that all assignments whose intervals include the availability time in question be considered in determineing the pool condition at that time. Note also that the ASSIGNMENT conventions for pooled resources permit the determination of descriptors by considering only the assignments whose intervals include the time in question; unlike the case for item-specific resources, there is no need to work progressively through all the descriptor changes from a set of initial descriptors to correctly determine the descriptors of pooled resource. (See the discussions in volume II on pooled and item-specific resources and the implication the corresponding conventions have on scheduling and unscheduling using time progressive and time transcendent strategies).

This module builds a tree that displays for each conflict the set of resource pool descriptors that exist because of jobs already scheduled and those required to be added to the schedule. No information on which previously assigned jobs caused the conflicts is included because the description of any pool is a result of the composite of all decisions on resource and job alternatives that have been made throughout development of the schedule. The most basic information needed to resolve the conflicts is simply what descriptors exist and what descriptors are required. This information is provided by the output tree from this module.

This module does not write or remove any assignments in $RESOURCE, i.e., $RESOURCE is returned unaltered. $RESOURCE is required by the module to assess the complete set of descriptors describing the pooled resources.

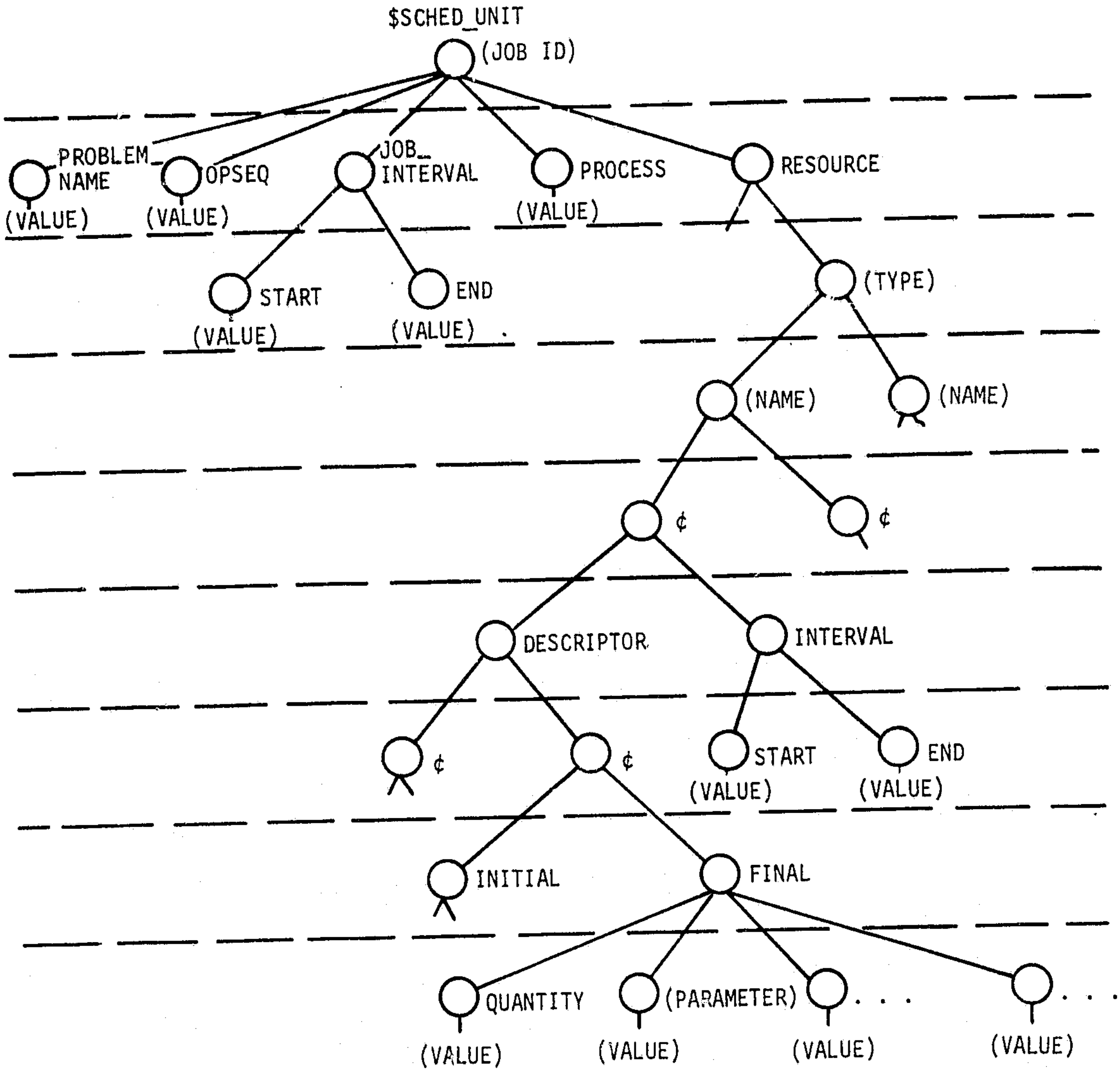
2.4.14.2 Modules Called

None

2.4.14.3 Module Input

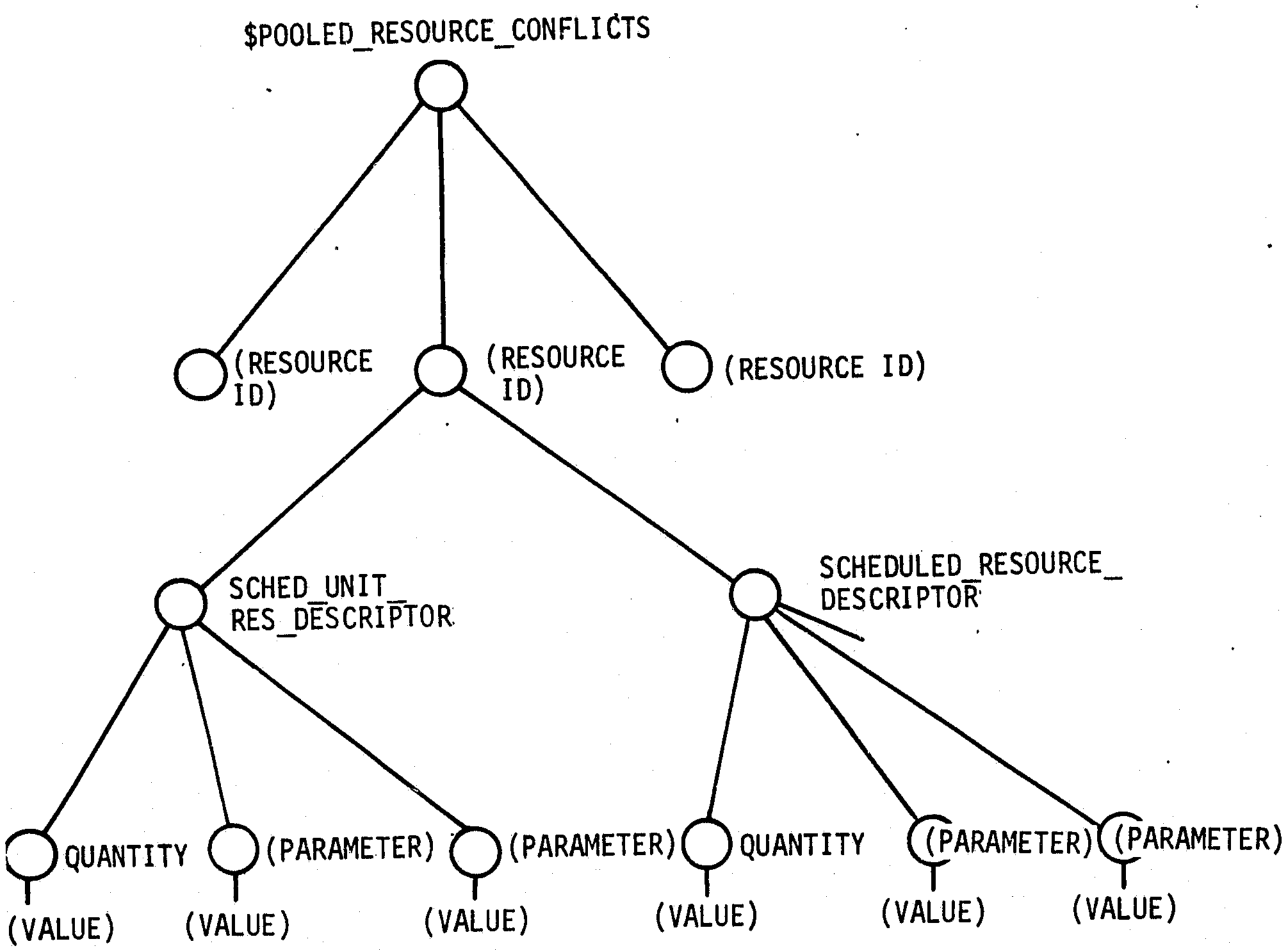
This module is called with two arguments: $RESOURCE and $SCHED_UNIT. $RESOURCE has the general structure given in paragraph 2.4.14.1; $SCHED_UNIT has the general structure of a schedule unit shown in the following illustration, and is equivalent of a second-level subnode of the stand data structure, $SCHEDULE.

Note that in $SCHED_UNIT the node labeled JOB_INTERVAL.START must contain the value of the assignment time for the job to be inserted.

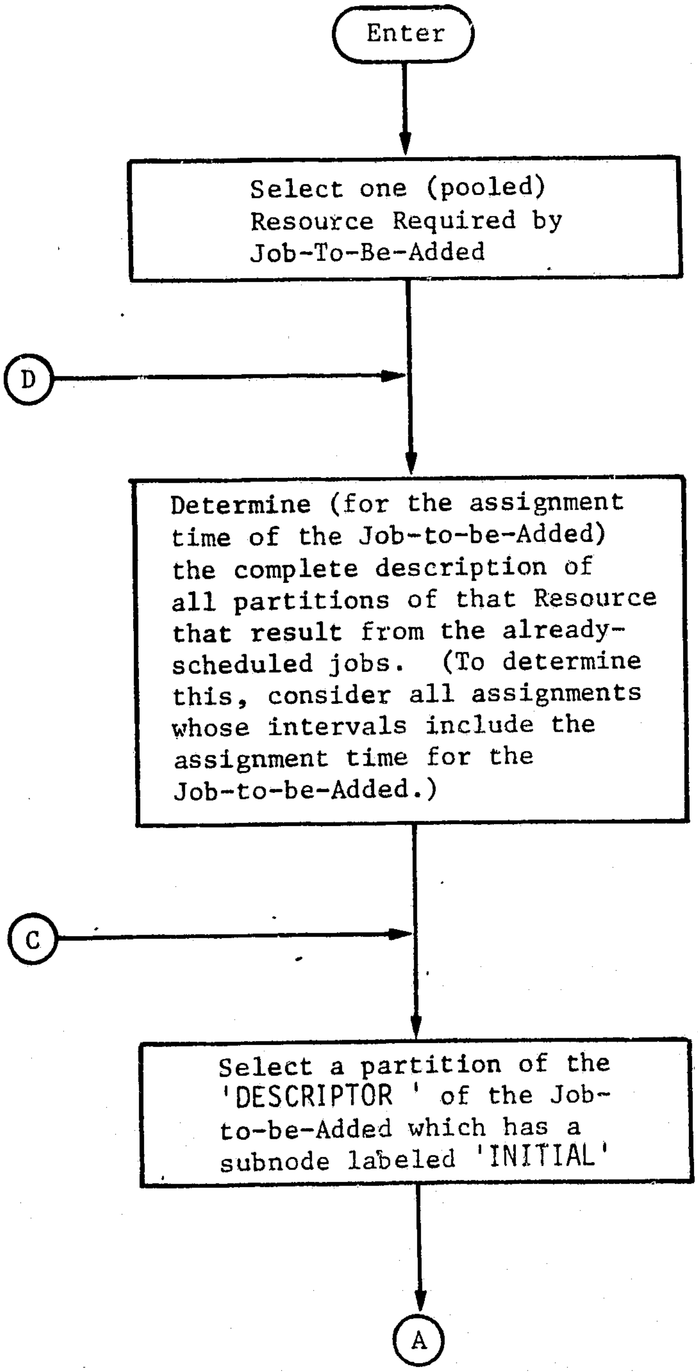

$SCHED_UNIT
(JOB ID)
PROBLEM_
NAME
(VALUE)
OPSEQ
(VALUE)
JOB_
INTERVAL
PROCESS
(VALUE)
RESOURCE
START
(VALUE)
END
(VALUE)
(TYPE)
(NAME)
(NAME)
¢
¢
DESCRIPTOR
INTERVAL
¢
¢
START
(VALUE)
END
(VALUE)
INITIAL
FINAL
QUANTITY
(VALUE)
(PARAMETER)
(VALUE)
. . .
(VALUE)
. . .
(VALUE)

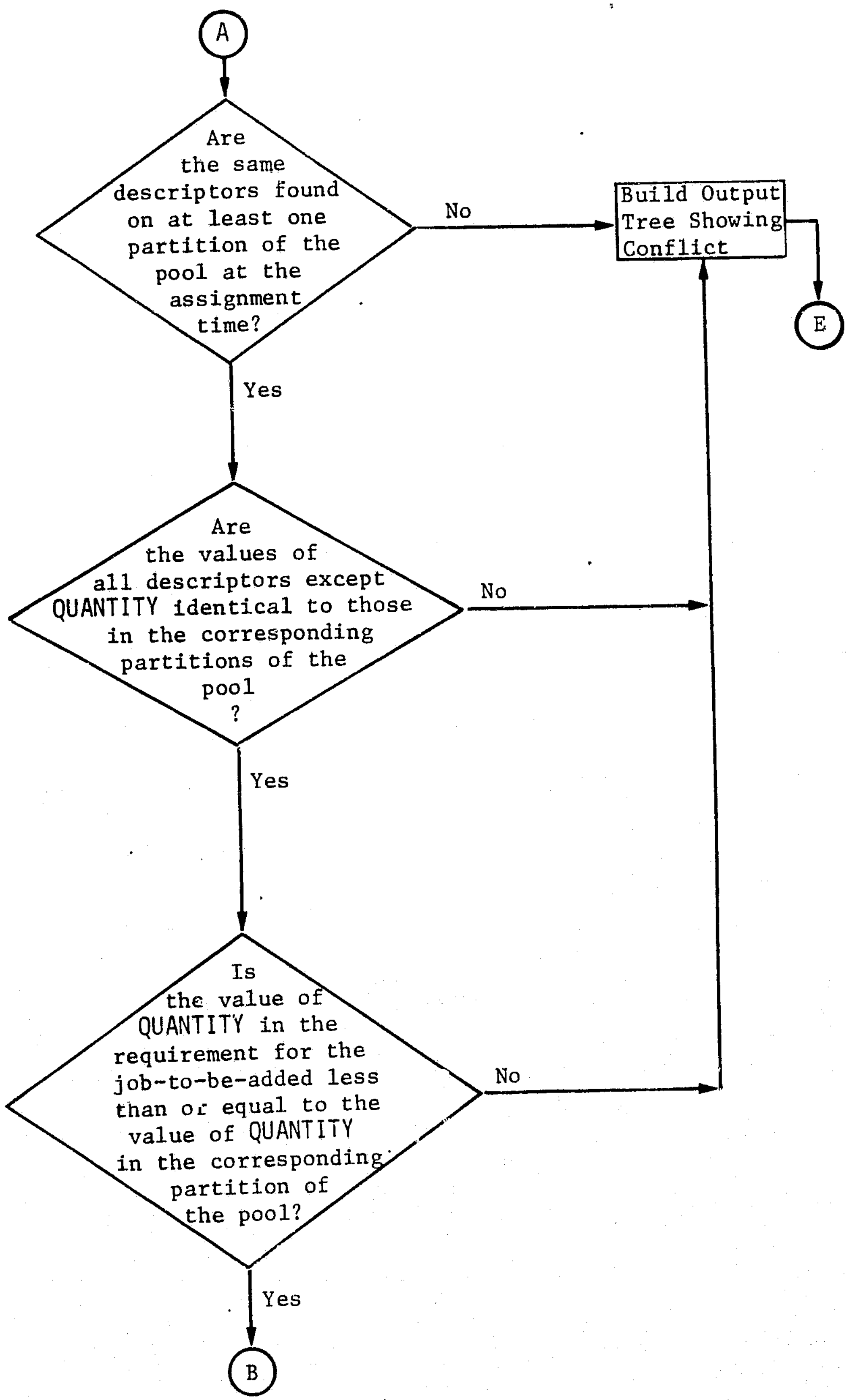
2.4.14.4 Module Output

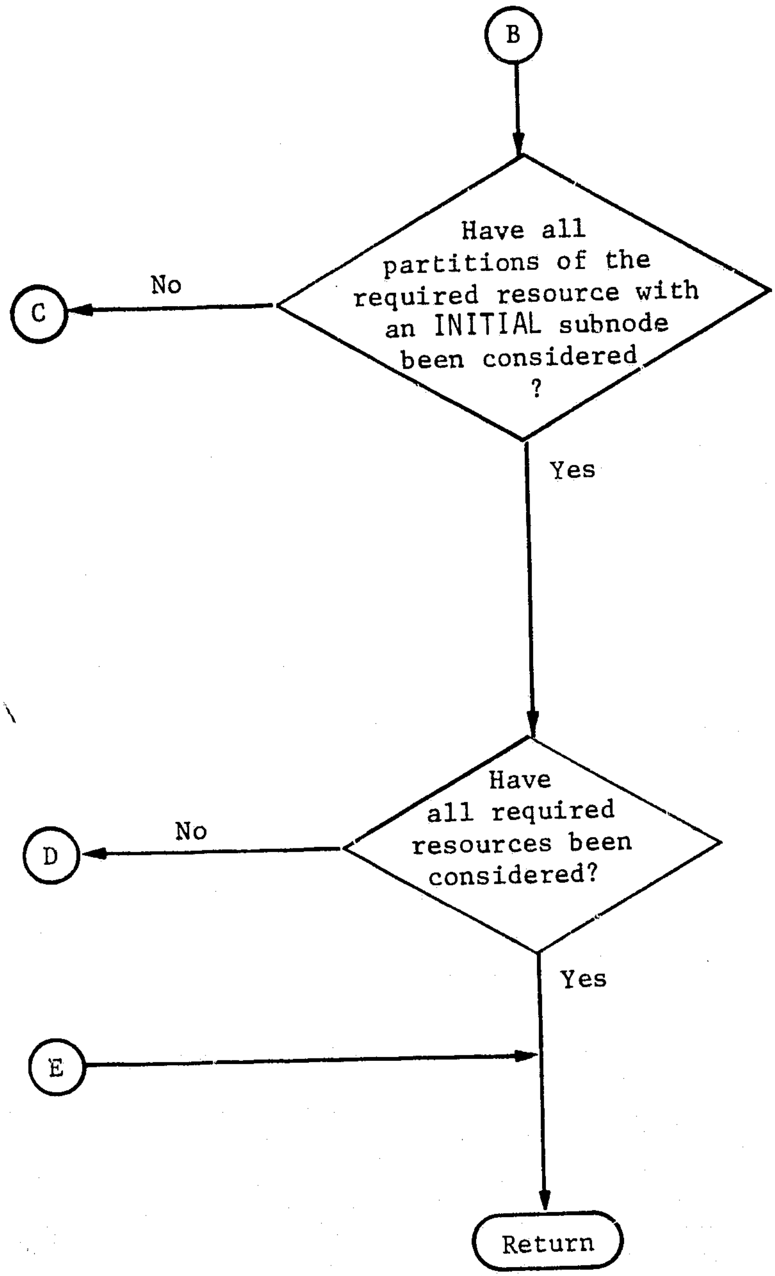
This module returns a structure called $POOLED_RESOURCE_CONFLICTS which contains information about conflicts that would result if $SCHED_UNIT were assigned at its specified time. The general structure of $POOLED_RESOURCE_CONFLICTS is illustrated.



2.4.14.5 Functional Block Diagram




A
Are the same descriptors found on at least one partition of the pool at the assignment time?
No
Build Output Tree Showing Conflict
E
Yes
Are the values of all descriptors except QUANTITY identical to those in the corresponding partitions of the pool ?
No
Yes
Is the value of QUANTITY in the requirement for the job-to-be-added less than or equal to the value of QUANTITY in the corresponding partition of the pool?
No
Yes
B


B
Have all partitions of the required resource with an INITIAL subnode been considered ?
No
C
Yes
Have all required resources been considered?
No
D
Yes
E
Return

2.4.14.6 Detailed Design

The functional block diagram provides a flow chart for this module. Since the module is developed for pooled resources with explicit descriptors, a check is made initially to determine that each resource required by the candidate job is, in fact, a pooled resource. If an item specific resource is required, a message is written identifying the resource and control is returned to the calling program. Each previously scheduled assignment of the resource which includes the assignment time of the candidate job is identified and placed in the tree, $ASSIGN. Each partition of the required resource is compared with the available resource to determine: first, whether a sufficient quantity is available, and second, whether the descriptors and values required are included in the available resources. If either comparison fails a resource conflict has been identified, and a node is constructed on the output tree, $POOLED_RESOURCE_CONFLICTS. The comparisons are repeated for all partitions and all required resources.

2.4.14.7 Internal Variable and Tree Name Definitions

$ASSIGN	-	tree built from $AVAILABILITY whose intervals contain assignment time of candidate job
$AVAIL	-	identifier for each subnode of $ASSIGN
$AVAILABILITY	-	identifier for each subnode of $RESOURCE.(TYPE.(NAME).ASSIGNMENT
$CONFLICT_FLAG	-	flag set to indicate resource conflict
$DESCRIP	-	identifier for each subnode of $AVAIL.DESCRIPTOR
$NAME	-	identifier for each subnode of $TYPE
$SUBDESC	-	identifier for each subnode of $SUBNAME.DESCRIPTOR
$SUBNAME	-	identifier for each subnode of $NAME
$TEMP_QUAN	-	temporary location for $SUBDESC.INITIAL.QUANTITY
$TYPE	-	identifier for each subnode of $SCHED_UNIT.RESOURCE

2.4.14.8 Modifications to Functional Specifications and/or Standard Data Structures Assumed

For pooled resources with explicit descriptors, each partition must be included in the assignment portion of $RESOURCE. Partitions that have been assigned will be indicated by INITIAL descriptors, as well as FINAL descriptors. Partitions that are available for assignment will have only FINAL descriptors.

2.4.14.9 Commented Code

```
POOLED_DESCRIPTOR_COMPATIBILITY: PROCEDURE($RESOURCE,$SCHED_UNIT,
     $POOLED_RESOURCE_CONFLICTS) OPTIONS(EXTERNAL);
DECLARE $ASSIGN,$TYPE,$NAME,$AVAILABILITY,$SUBNAME,$SUBDESC LOCAL;
DECLARE $CONFLICT_FLAG,$AVAIL,$DESCRIP,$TEMP_QUAN LOCAL;
DO FOR ALL SUBNODES OF $SCHED_UNIT.RESOURCE USING $TYPE;
   DO FOR ALL SUBNODES OF $TYPE USING $NAME;
   IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).CLASS = 'SPECIFIC'
      THEN DO;
         WRITE 'THIS_IS_A_SPECIFIC_RESOURCE',LABEL($TYPE),LABEL($NAME);
         RETURN;
         END;
      DO FOR ALL SUBNODES OF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
         ASSIGNMENT USING $AVAILABILITY;
      IF ($SCHED_UNIT.JOB_INTERVAL.END >= $AVAILABILITY.INTERVAL.START
         & $SCHED_UNIT.JOB_INTERVAL.START <= $AVAILABILITY.INTERVAL.END
         ) THEN $ASSIGN(NEXT) = $AVAILABILITY;
      END;
      DO FOR ALL SUBNODES OF $NAME USING $SUBNAME;
         DO FOR ALL SUBNODES OF $SUBNAME.DESCRIPTOR USING $SUBDESC;
            IF $SUBDESC.INITIAL IDENTICAL TO $NULL THEN;
               ELSE DO FOR ALL SUBNODES OF $ASSIGN USING $AVAIL;
 /*  PICK ONE PARTITION OF THE POOL AT THE REQUIRED TIME                    */
                     DO FOR ALL SUBNODES OF $AVAIL.DESCRIPTOR USING
                        $DESCRIP;
                     $CONFLICT_FLAG = 'YES';
 /*  IS THE REQUIRED QUANTITY GREATER THAN QUANTITY AVAILABLE IN THIS */
 /*      PARTITION? IF NOT CHECK OTHER DESCRIPTORS OTHERWISE BUILD      */
 /*      CONFLICT NODE AND FIND NEXT PARTITION                          */
                     IF $SUBDESC.INITIAL.QUANTITY > $DESCRIP.FINAL.
                        QUANTITY & $DESCRIP.INITIAL IDENTICAL TO $NULL
                        THEN;
                        ELSE DO;
 /*  ARE OTHER DESCRIPTOR AND VALUE IDENTICAL ?                           */
                           GRAFT $SUBDESC.INITIAL.QUANTITY AT
                              $TEMP_QUAN;
                           IF $SUBDESC.INITIAL SUBSET OF $DESCRIP.FINAL
                              & $DESCRIP.INITIAL IDENTICAL TO $NULL
                              THEN DO;
                                 $CONFLICT_FLAG = 'NO';
                                 GRAFT $TEMP_QUAN AT $SUBDESC.INITIAL.
                                        QUANTITY;
                                 END;
                              ELSE GRAFT $TEMP_QUAN AT $SUBDESC.
                                 INITIAL.QUANTITY;
                        END;
                     IF $CONFLICT_FLAG = 'YES'
                        THEN DO;
                          LABEL($POOLED_RESOURCE_CONFLICTS(NEXT)) =
                             LABEL($TYPE);
                          $POOLED_RESOURCE_CONFLICTS(LAST).
                             SCHED_UNIT_RES_DESCRIPTOR = $SUBDESC.
```

```
                                                INITIAL;
                                            $POOLED_RESOURCE_CONFLICTS(LAST).
                                                SCHED_UNIT_RES_DESCRIPTOR(NEXT) =
                                                $SCHED_UNIT.JOB_INTERVAL;
                                            $POOLED_RESOURCE_CONFLICTS(LAST).
                                                SCHEDULED_RESOURCE_DESCRIPTOR =
                                                $DESCRIP.FINAL;
                                            $POOLED_RESOURCE_CONFLICTS(LAST).
                                                SCHEDULED_RESOURCE_DESCRIPTOR(NEXT) =
                                                $AVAIL.INTERVAL;
                                            END;
/*  PICK NEXT PARTITION OF ASSIGNED RESOURCES FOR THIS ONE REQUIRED  */
                                      END;
                                 END;
/*  PICK NEXT DESCRIPTORS FOR THIS REQUIRED RESOURCE                  */
          END;
        END;
/*  PICK NEXT REQUIRED RESOURCES                                      */
    END;
END;

END;   /*    POOLED_DESCRIPTOR_COMPATIBIILTY    */
```

2.4.15 DESCRIPTOR_PROFILE

2.4.15 DESCRIPTOR_PROFILE

2.4.15.1 Purpose and Scope

This module is used to update the set of descriptors that apply to an item-specific resource, i.e., an individual, identifiable resource that would correspond to the first subnode level of the resource "type" in the $RESOURCE tree. The update of descriptors will consist of an assignment or set of assignments that define initial and final descriptors for each assignment. The original set of descriptors to be updated and their corresponding values will be supplied by the calling program. This could consist of reference to the resource descriptors in the $RESOURCE tree, a derived tree that has been maintaining the descriptors of that resource as a function of time, or a tree built by the calling program with specific (possibly artificial) descriptors.

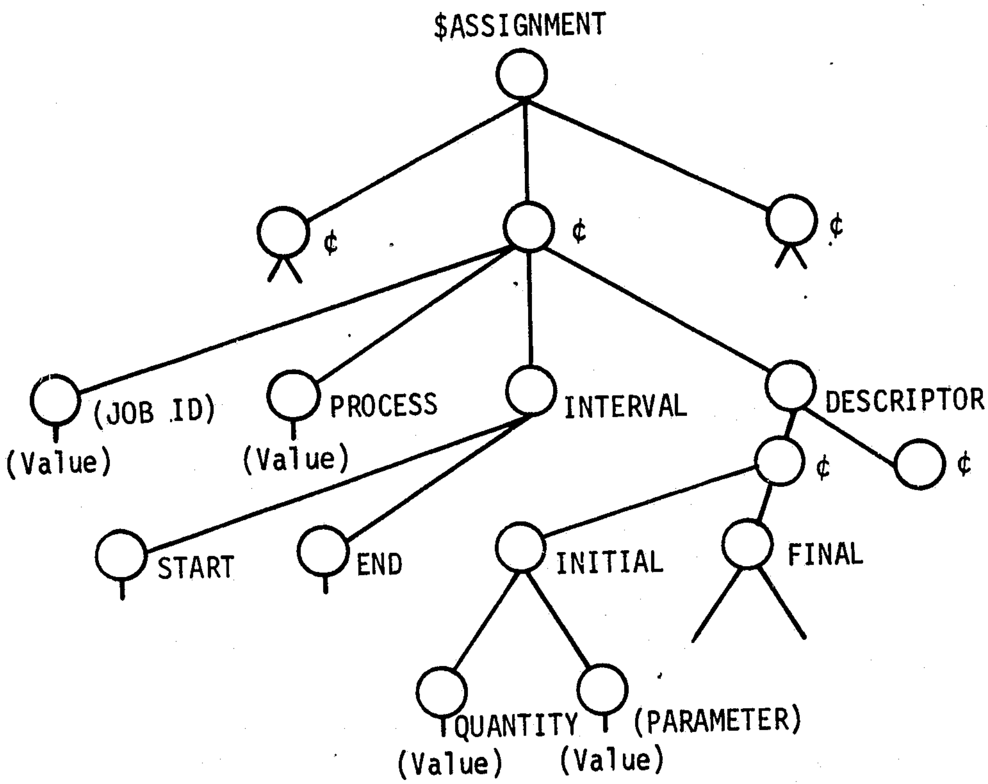
Any number of descriptive parameters may have been used in the resource assignments, but any one parameter will be assumed to contain only mutually exclusive values. For example, if the descriptive parameter, LOCATION is specified, values of DENVER, DALLAS, or DETROIT are obviously mutually exclusive. If, however, the location were specified as DENVER and a process moved the resource to WAREHOUSE 3, this module would retain only the location WAREHOUSE 3 whether or not Warehouse 3 was located in Denver.

2.4.15.2 Modules Called

None.

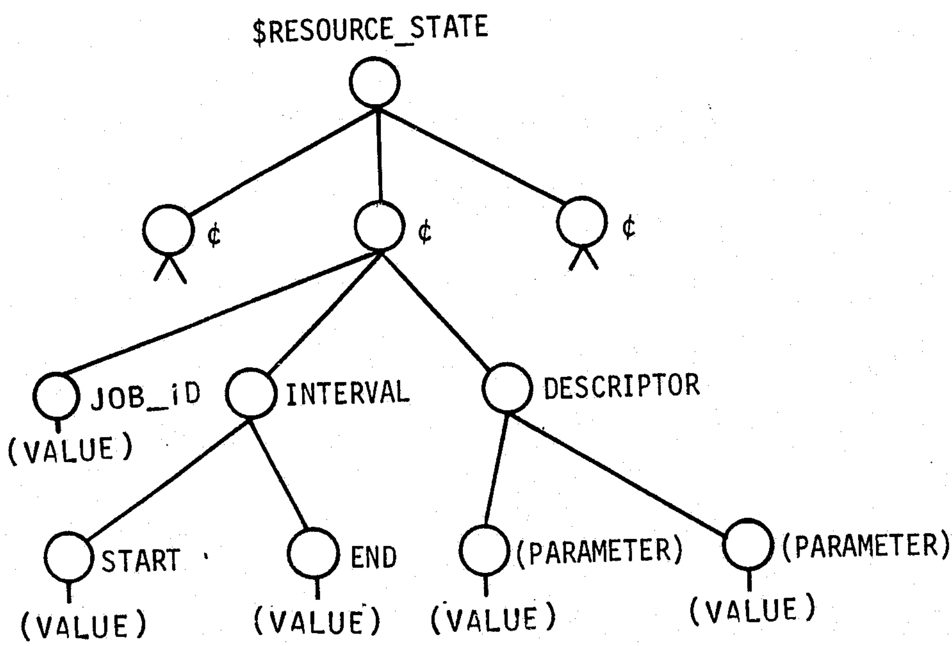
2.4.15.3 <u>Module Input</u>

Input consists of the item-specific resource to be considered, the original values of the descriptors to be updated and the corresponding time, the assignments to be considered, and the final time that assignments are to be considered. The resource is identified by a pointer, $RESOURCE_NAME, to a second level subnode of the $RESOURCE tree. The original descriptors and their values are defined under the INITIAL_DESCRIPTOR node of $RESOURCE-NAME. The corresponding time is defined under the INITIAL_TIME node. The assignments to be considered have a format corresponding to the subnode levels of the ASSIGNMENT node in $RESOURCE_NAME as illustrated in the sketch. Any nodes, other than the time interval and descriptors (which are required), will be retained for added traceability. The final time will be defined by a single valued tree $MAX_TIME, which will represent the final time of the profile.

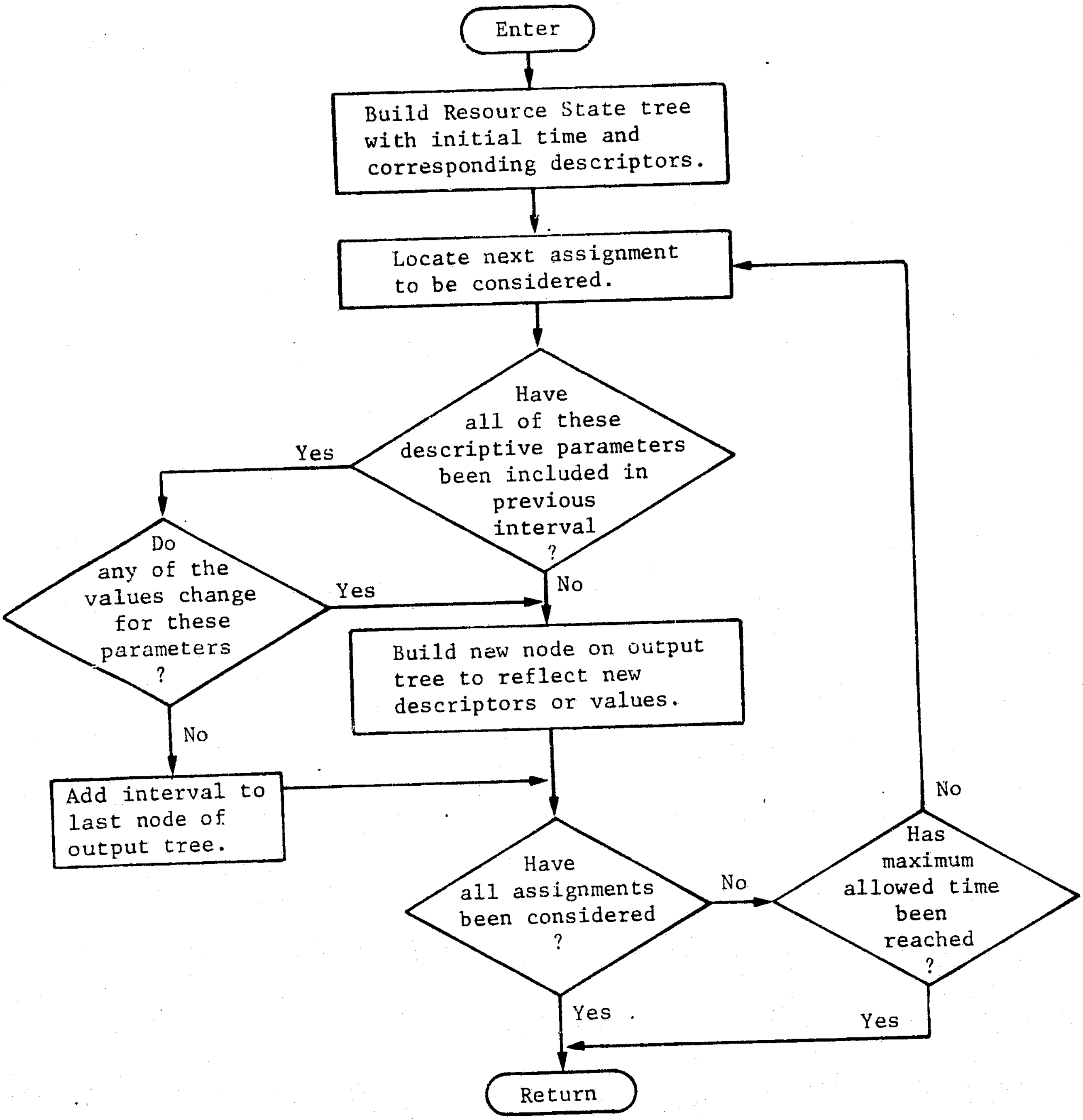


2.4.15.4 Module Output

The output consists of a "resource state" tree (shown) that lists the resource descriptors as a function of time.



2.4.15.5 Functional Block Diagram



2.4.15.6 Detailed Design

The functional block diagram may be used as the flowchart for this module. The module first checks to determine if a 'pooled' resource has inadvertently been transferred through the parameter list. If this happens, an error message is written and control is returned to the calling program. Otherwise, a tree, $RESOURCE_STATE, is constructed with the initial descriptors of the resource. The next assignment of the resource is selected, and the resource descriptors are compared with the last subnode of $RESOURCE_STATE. If any of the labels or the values change a new subnode of $RESOURCE_STATE is added with the new label and value. If the descriptors have not changed, the interval in $RESOURCE_STATE is extended to include the current interval. When all assignments have been considered, or the maximum input time has been reached, control is returned to the calling program.

2.4.15.7 Internal Variable and Tree Name Definitions

$ASSIGN	-	Identifier for each subnode of ASSIGNMENT
$DUMMY	-	Temporary tree made up of descriptors identified in $ASSIGNMENT which are not in $RESOURCE_STATE
$LAST	-	Pointer at the descriptor subnode of the last subnode of $RESOURCE_STATE
$NEW_DESCRIP	-	Identifier for each subnode of $DUMMY when new descriptors are recognized.
$NEW_VALUE	-	Identifier for each subnode of $DUMMY when values of descriptors have changed
$PARAMETER	-	Identifier for each subnode of initial descriptors in $ASSIGN

$SUBNAME - Identifier for each subnode of INITIAL_DESCRIPTOR node

2.4.15.8 Modifications to Functional Specification and/or Standard Data Structures Assumed

Since only a single resource element (second level subnode of $RESOURCE) is considered, this element is passed through the parameter list as $RESOURCE_NAME.

2.4.15.9 Commented Code

```
DESCRIPTOR_PROFILE: PROCEDURE($RESOURCE_NAME,$ASSIGNMENT,
   $RESOURCE_STATE,$MAX_TIME) OPTIONS(EXTERNAL);
DECLARE $ASSIGN,$DUMMY,$LAST,$NEW_DESCRIP,$NEW_VALUE,$PARAMETER,
      $SUBNAME LOCAL;
PRUNE $RESOURCE_STATE;
IF $RESOURCE_NAME.CLASS = 'POOLED'
   THEN DO;
      WRITE 'THIS_IS_A_POOLED_RESOURCE';
      WRITE LABEL($RESOURCE_NAME);
      RETURN;
      END;
/*  BUILD RESOURCE STATE TREE WITH INITIAL TIME AND CORRESPONDING    */
/*  DESCRIPTORS                                                       */
DO FOR ALL SUBNODES OF $RESOURCE_NAME.INITIAL_DESCRIPTOR USING
          $SUBNAME;
           $RESOURCE_STATE(FIRST).DESCRIPTOR(NEXT) = $SUBNAME;
END;
$RESOURCE_STATE(FIRST).DESCRIPTOR.QUANTITY=1;
$RESOURCE_STATE(FIRST).INTERVAL.START = $RESOURCE_NAME.INITIAL_TIME;
$RESOURCE_STATE(FIRST).INTERVAL.END = $RESOURCE_STATE(FIRST).INTERVAL.
   START;
/*  LOCATE NEXT ASSIGNMENT TO BE CONSIDERED                           */
IF $ASSIGNMENT(FIRST) IDENTICAL TO $NULL
   THEN DO;
      PRUNE $ASSIGNMENT;
      $RESOURCE_STATE(FIRST).INTERVAL.END = $MAX_TIME;
      RETURN;
   END;
DO FOR ALL SUBNODES OF $ASSIGNMENT USING $ASSIGN;
PRUNE $DUMMY;
/*  HAS MAXIMUM ALLOWED TIME BEEN REACHED?                            */
   IF $ASSIGN.INTERVAL.START >= $MAX_TIME
      THEN DO;
         $RESOURCE_STATE(LAST).INTERVAL.END = $MAX_TIME;
         PRUNE $RESOURCE_STATE(LAST).JOB_ID;
         RETURN;
         END;
      ELSE $RESOURCE_STATE(LAST).INTERVAL.END = $ASSIGN.INTERVAL.END;
/*  HAVE ALL OF THESE DESCRIPTIVE PARAMETERS BEEN INCLUDED IN         */
/*  PREVIOUS INTERVAL?                                                */
   IF $ASSIGN.DESCRIPTOR(FIRST).INITIAL ¬SUBSET OF
      $RESOURCE_STATE(LAST).DESCRIPTOR
/*  BUILD NEW NODE ON OUTPUT TREE TO REFLECT NEW VALUES               */
      THEN DO;
         DO FOR ALL SUBNODES OF $ASSIGN.DESCRIPTOR(FIRST).INITIAL
               USING $PARAMETER;
         IF $PARAMETER ¬ELEMENT OF $RESOURCE_STATE(LAST).DESCRIPTOR
            THEN $DUMMY(NEXT) = $PARAMETER;
         END;
         DO FOR ALL SUBNODES OF $DUMMY USING $NEW_DESCRIP;
            IF $RESOURCE_STATE(LAST).DESCRIPTOR.#LABEL($NEW_DESCRIP)
```

```
                 IDENTICAL TO $NULL THEN;
              ELSE WRITE 'THE_VALUE_OF_THIS_DESCRIPTOR_CHANGES_AT_THE_
START_OF_THE_ASSIGNMENT',LABEL($NEW_DESCRIP),$NEW_DESCRIP;
              $RESOURCE_STATE(LAST).DESCRIPTOR.#LABEL($NEW_DESCRIP) =
                   $NEW_DESCRIP;
         END;
         END;
      ELSE;
/*  DO ANY OF THE VALUES CHANGE FOR THESE PARAMETERS?                   */
      IF ($ASSIGN.DESCRIPTOR(FIRST).FINAL IDENTICAL TO $NULL | $ASSIGN.
         DESCRIPTOR(FIRST).FINAL SUBSET OF $ASSIGN.DESCRIPTOR(FIRST)
         .INITIAL)
/*  ADD INTERVAL TO LAST NODE OF OUTPUT TREE                            */
         THEN IF $ASSIGN.INTERVAL.END >= $MAX_TIME
            THEN DO;
               $RESOURCE_STATE(LAST).JOB_ID = $ASSIGN.JOB_ID;
               RETURN;
               END;
            ELSE;
         ELSE IF $ASSIGN.INTERVAL.END >= $MAX_TIME
            THEN DO;
               DO FOR ALL SUBNODES OF $ASSIGN.DESCRIPTOR(FIRST).FINAL
                    USING $PARAMETER;
             IF $PARAMETER ¬ELEMENT OF $RESOURCE_STATE(LAST).DESCRIPTOR
                  THEN $DUMMY(NEXT) = $PARAMETER;
               END;
               $RESOURCE_STATE(LAST).JOB_ID = $ASSIGN.JOB_ID;
               DEFINE $LAST AS $RESOURCE_STATE(LAST).DESCRIPTOR;
               $RESOURCE_STATE(NEXT).DESCRIPTOR = $LAST;
               $RESOURCE_STATE(LAST).INTERVAL.START = $ASSIGN.INTERVAL.
                    END;
               $RESOURCE_STATE(LAST).INTERVAL.END = $ASSIGN.INTERVAL.
                    END;
               DO FOR ALL SUBNODES OF $DUMMY USING $NEW_VALUE;
                  $RESOURCE_STATE(LAST).DESCRIPTOR.#LABEL($NEW_VALUE)
                       = $NEW_VALUE;
                  END;
               RETURN;
               END;
            ELSE DO;
/*  BUILD NEW NODE ON OUTPUT TREE TO REFLECT NEW DESCRIPTORS            */
               DO FOR ALL SUBNODES OF $ASSIGN.DESCRIPTOR(FIRST).FINAL
                    USING $PARAMETER;
             IF $PARAMETER ¬ELEMENT OF $RESOURCE_STATE(LAST).DESCRIPTOR
                  THEN $DUMMY(NEXT) = $PARAMETER;
               END;
               $RESOURCE_STATE(LAST).JOB_ID = $ASSIGN.JOB_ID;
               DEFINE $LAST AS $RESOURCE_STATE(LAST).DESCRIPTOR;
               $RESOURCE_STATE(NEXT).DESCRIPTOR = $LAST;
               $RESOURCE_STATE(LAST).INTERVAL.START = $ASSIGN.INTERVAL.
                    END;
```

```
                    DO FOR ALL SUBNODES OF $DUMMY USING $NEW_VALUE;
                       $RESOURCE_STATE(LAST).DESCRIPTOR.#LABEL($NEW_VALUE)
                            = $NEW_VALUE;
                       END;
                    END;
/*  HAVE ALL ASSIGNMENTS BEEN CONSIDERED?                                 */
END;
END;    /*   DESCRIPTOR_PROFILE    */
```

2.4.16 UPDATE_RESOURCE

2.4.16 UPDATE_RESOURCE

2.4.16 UPDATE_RESOURCE

2.4.16.1 Purpose and Scope

This module will update information in the data tree $RESOURCE for each resource assigned to a specific JOB_ID in the structure $SCHEDULE. It provides a standard method of reflecting in $RESOURCE, the results of a scheduling decision. It creates a data structure $NEXTUNIT that contains element(s) to be added to the chronologically ordered assignments of a specific $RESOURCE.(TYPE).(NAME) by calling the module WRITE_ASSIGNMENT.

2.4.16.2 Modules Called

WRITE_ASSIGNMENT

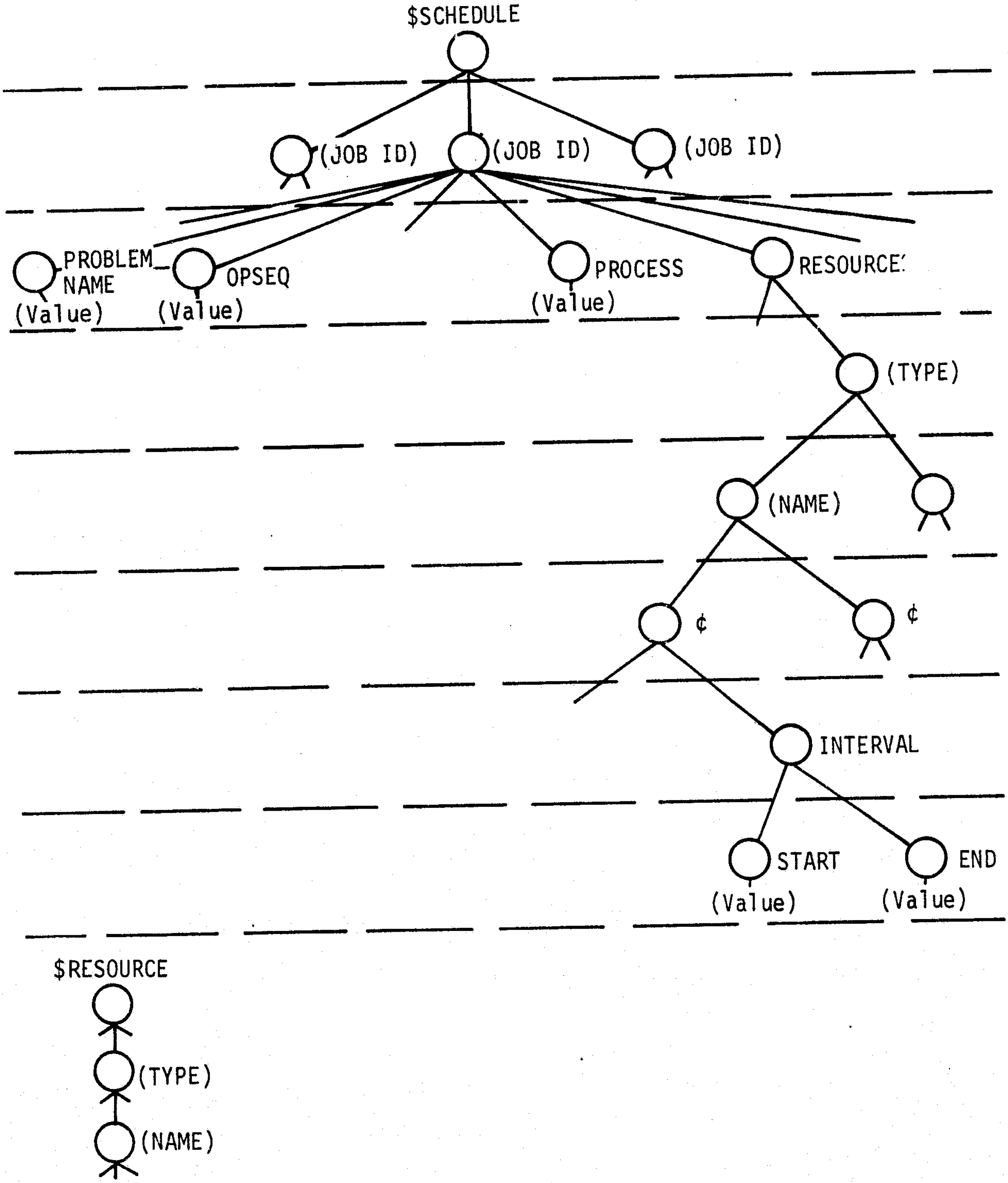
2.4.16.3 Module Input

Inputs consist of the standard data structures $SCHEDULE and $RESOURCE, that are shown in standard form on the following pages. The minimum relevant portions of the required input structures are shown on subsequent pages.

2.4.16.4 Module Output

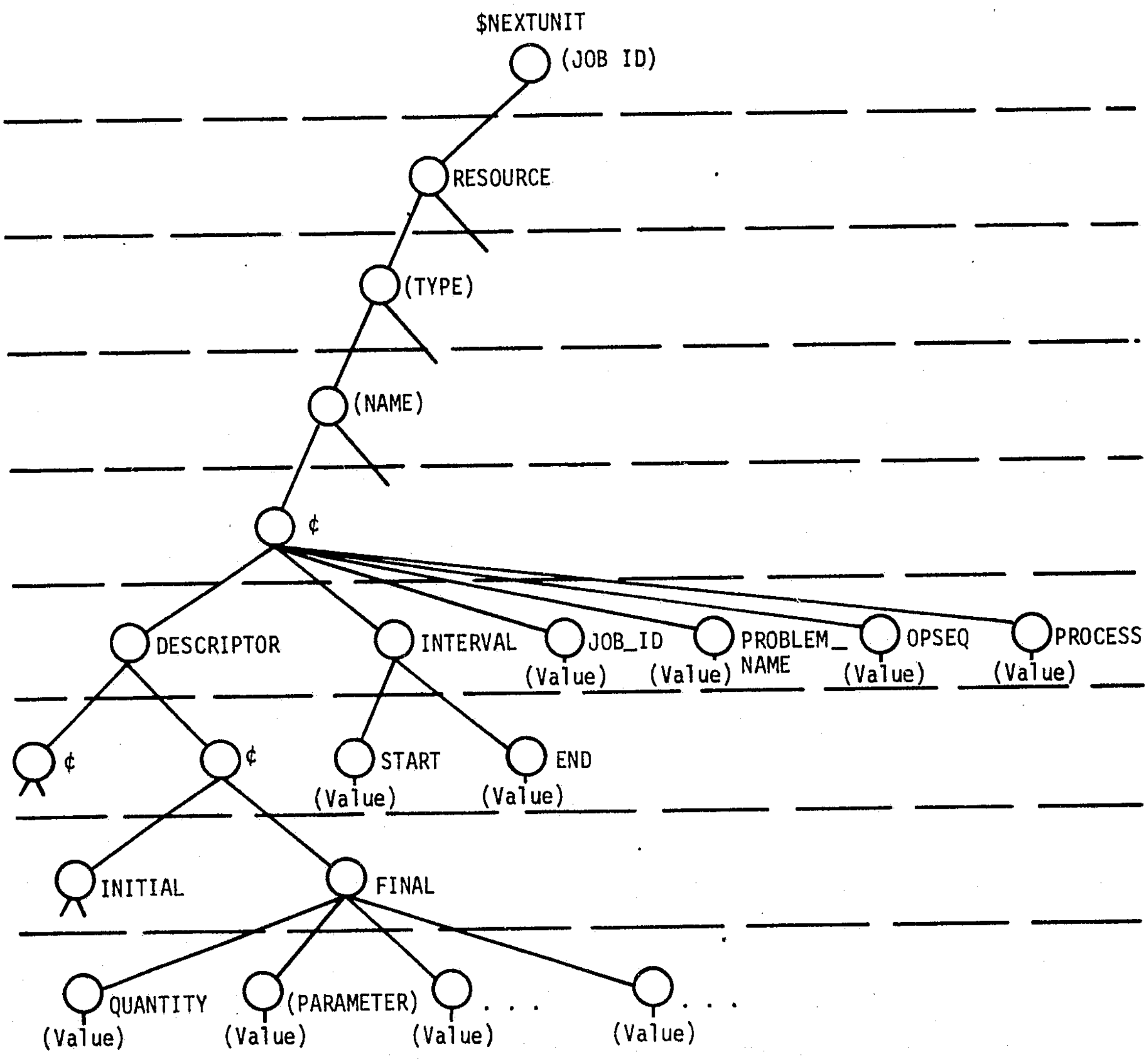
During execution the module creates the data structure $NEXTUNIT. (See the following illustrations) After execution, the $RESOURCE tree will reflect the changes in assignments that result from the scheduling of all jobs in $SCHEDULE.

Note: Minimum (i.e., relevant) portion of required input Standard Data Structures is shown. In all trees, any additional structure will be preserved.

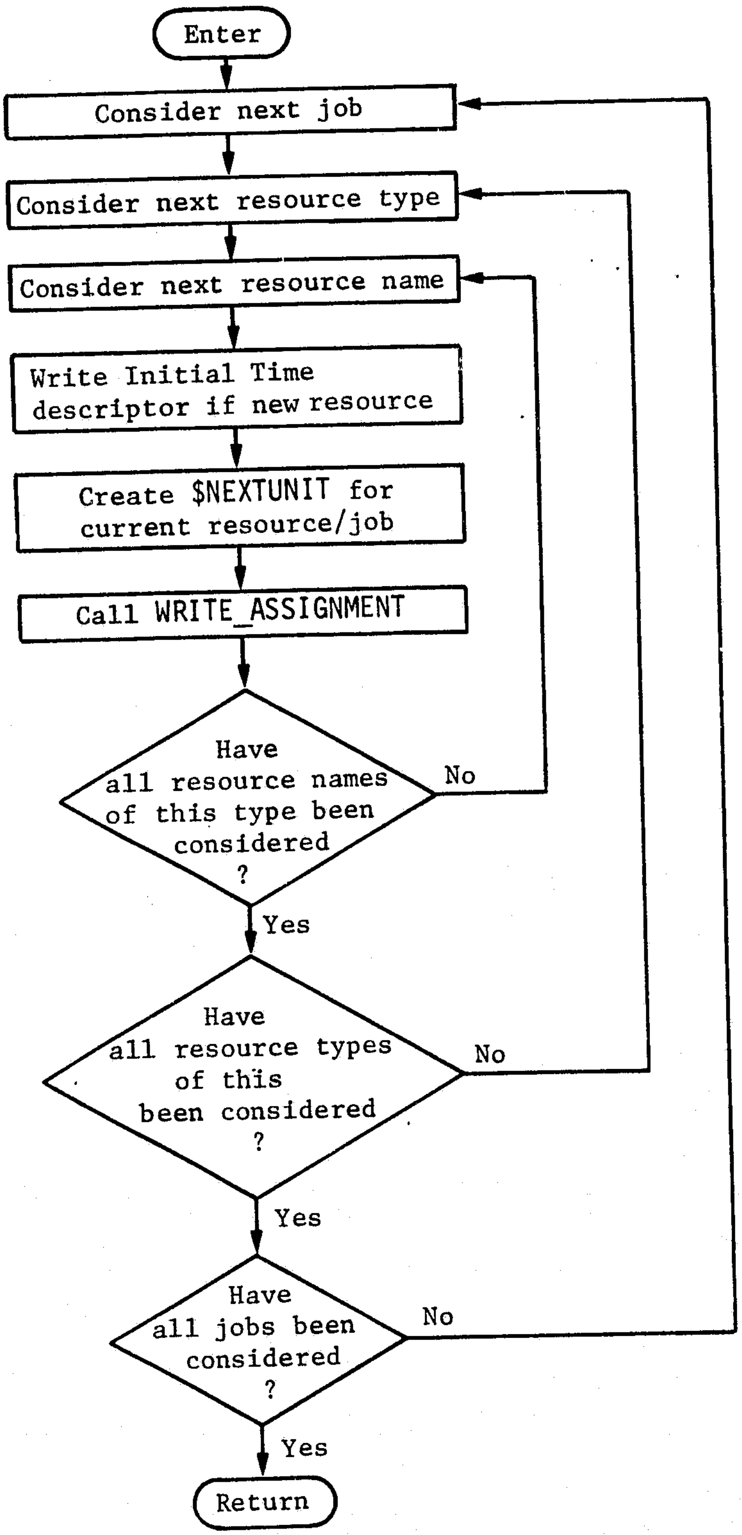


Minimum Required Input Structures from Standard Data Structures for Module: UPDATE_RESOURCE

INTERNALLY CREATED DATA STRUCTURE (NOT OUTPUT)



2.4.16.5 Functional Block Diagram



2.4.16.6 Typical Applications

The module would be used during or after the construction of a schedule to record the changes to the resource assignments resulting from temporary or permanent decision to schedule specific jobs.

2.4.16.7 DETAILED DESIGN

The functional block diagram provides the flow chart for this module. The module loops on job, resource type, and resource name as specified by the input data structure $SCHEDULE. For each specified resource element the data structure $NEXTUNIT is constructed. If this resource element is not contained within $RESOURCE, the normal initial profile is constructed in $RESOURCE. For each specified resource element, the module WRITE_ASSIGNMENT is called to add the element to the chronologically ordered assignments of $RESOURCE.

2.4.16.8 INTERNAL VARIABLE AND TREE NAMES

The internal variables used in this module are listed below, with definitions.

QUAN - Summation of quantities of partitions of a pooled resource.

$TYPE - A single node tree containing only the label of resource type provided in input $SCHEDULE.

$NAME - A single node tree containing only the resource name provided by the input $SCHEDULE.

$NEXTUNIT - A data structure derived from $RESOURCE, but containing only the assignment subnode of a single resource element.

2.4.16.9 COMMENTED CODE

```
UPDATE_RESOURCE: PROCEDURE($SCHEDULE,$RESOURCE) OPTIONS(EXTERNAL);
DECLARE I,J,K,L,M,QUAN,$NAME,$TYPE LOCAL;
  DECLARE $NEXTUNIT LOCAL;
/*  CONSIDER NEXT JOB.                                                    */
  DO I = 1 TO NUMBER($SCHEDULE);
/*  CONSIDER NEXT RESOURCE TYPE.                                          */
     DO J = 1 TO NUMBER($SCHEDULE(I).RESOURCES);
/*  CONSIDER NEXT RESOURCE NAME.                                          */
        DO K = 1 TO NUMBER($SCHEDULE(I).RESOURCES(J));
        $TYPE = LABEL($SCHEDULE(I).RESOURCES(J));
        $NAME = LABEL($SCHEDULE(I).RESOURCES(J)(K));
/*  WRITE 'INITIAL TIME' DESCRIPTOR IF NEW RESOURCE.                      */
        IF $RESOURCE.#($TYPE).#($NAME) IDENTICAL TO $NULL
           THEN DO;
              $RESOURCE.#($TYPE).#($NAME).INITIAL_TIME = $SCHEDULE(I).
                 RESOURCES(J)(K)(1).INTERVAL.END;
              DO L = 1 TO NUMBER($SCHEDULE(I).RESOURCES(J)(K));
              $RESOURCE.#($TYPE).#($NAME).INITIAL_PROFILE.NORMAL(I)
                 = $SCHEDULE(I).RESOURCES(J)(K)(L).INTERVAL;
              QUAN = 0;
                 DO M = 1 TO NUMBER($SCHEDULE(I).RESOURCES(J)(K)(I).
                    DESCRIPTORS);
                 QUAN = QUAN + $SCHEDULE(I).RESOURCES(J)(K)(L).
                    DESCRIPTORS(M).FINAL.QUANTITY;
                 END;
              $RESOURCE.#($TYPE).#($NAME).INITIAL_PROFILE.NORMAL(I)
                 .QUANTITY = QUAN;
              END;
              END;
/*  CREATE $NEXTUNIT FOR CURRENT RESOURCE/JOB.                            */
        LABEL ($NEXTUNIT.RESOURCES(J)) = $TYPE;
        LABEL ($NEXTUNIT.RESOURCES(J)(K)) = $NAME;
           DO L = 1 TO NUMBER($SCHEDULE(I).RESOURCES(J)(K));
           $NEXTUNIT.RESOURCES(J)(K)(L) = $SCHEDULE(I).RESOURCES
              (J)(K)(L);
           $NEXTUNIT.RESOURCES(J)(K)(L).JOB_ID = LABEL($SCHEDULE
              (I));
           IF $SCHEDULE(I).PROBLEM_NAME ¬IDENTICAL TO $NULL THEN
              $NEXTUNIT.RESOURCES(J)(K)(L).PROBLEM_NAME =
              $SCHEDULE(I).PROBLEM_NAME;
           IF $SCHEDULE(I).OPSEQ ¬IDENTICAL TO $NULL THEN
              $NEXTUNIT.RESOURCES(J)(K)(L).OPSEQ =
              $SCHEDULE(I).OPSEQ;
           $NEXTUNIT.RESOURCES(J)(K)(L).PROCESS =
              $SCHEDULE(I).PROCESS;
           $ASSIGNMENT_UNIT = $NEXTUNIT.RESOURCES(J)(K)(L);
/*  CALL WRITE_ASSIGNMENT.                                                */
           CALL WRITE_ASSIGNMENT($ASSIGNMENT_UNIT,$RESOURCE.#($TYPE).
              #($NAME).ASSIGNMENT);
           END_LOOP_L: END;
/*  HAVE ALL RESOURCE NAMES OF THIS TYPE BEEN CONSIDERED?                 */
```

```
        END_LOOP_K: END;
/*  HAVE ALL RESOURCE TYPES OF THIS JOB BEEN CONSIDERED?                    */
      END_LOOP_J: END;                                                      */
/*  HAVE ALL JOBS BEEN CONSIDERED?
    END_LOOP_I: END;
    RETURN;
    END ;  /*  UPDATE_RESOURCE   */
```

2.4.17 WRITE_ASSIGNMENT

2.4.17 WRITE_ASSIGNMENT

2.4.17 WRITE_ASSIGNMENT

This module will add an element to the **chronologically** ordered assignments of the **$ASSIGN** tree for a specified resource name and type. Basis for the order is the resource interval start time. If start times are equal, the assignment with an earlier end time is listed first. If start and end times are equal, no distinction is made in the order.

The specific data written for an assignment can vary with the calling module. That is, dummy assignments may be made as a means of constraining resources in which case processes, problem names, etc may be meaningless. However, selected resources for a given problem may contain many parameters and descriptors that define the usage and provide traceability for later retrieval.

2.4.17.2 Modules Called

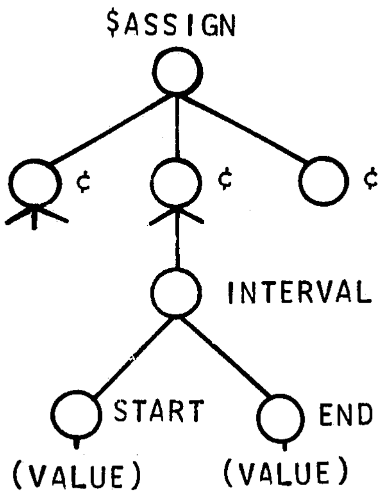
None

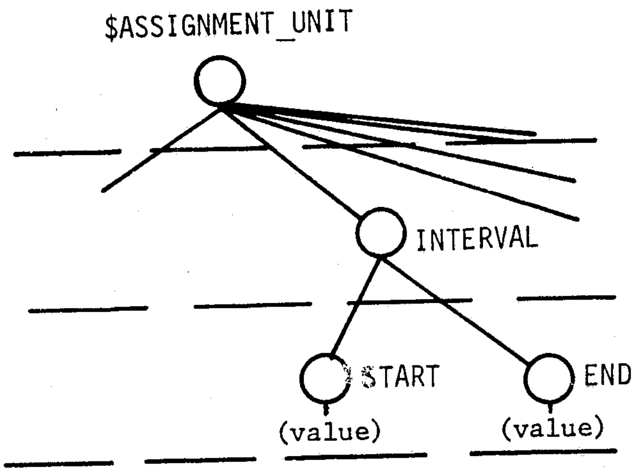
2.4.17.3 Module Input

Inputs to this module consist of $ASSIGNMENT_UNIT, the assignment node of $NEXTUNIT for which the assignment is to be written, and identificaiton of the $RESOURCE subnode where the assignment is made. In the standard case, the entire substructure of one of the third-level subnodes of $NEXTUNIT.RESOURCE becomes the substructure for one element of the standard data structure subnode $RESOURCE.(TYPE).(NAME).ASSIGNMENT that corresponds to the **subnode identified by $ASSIGN.**

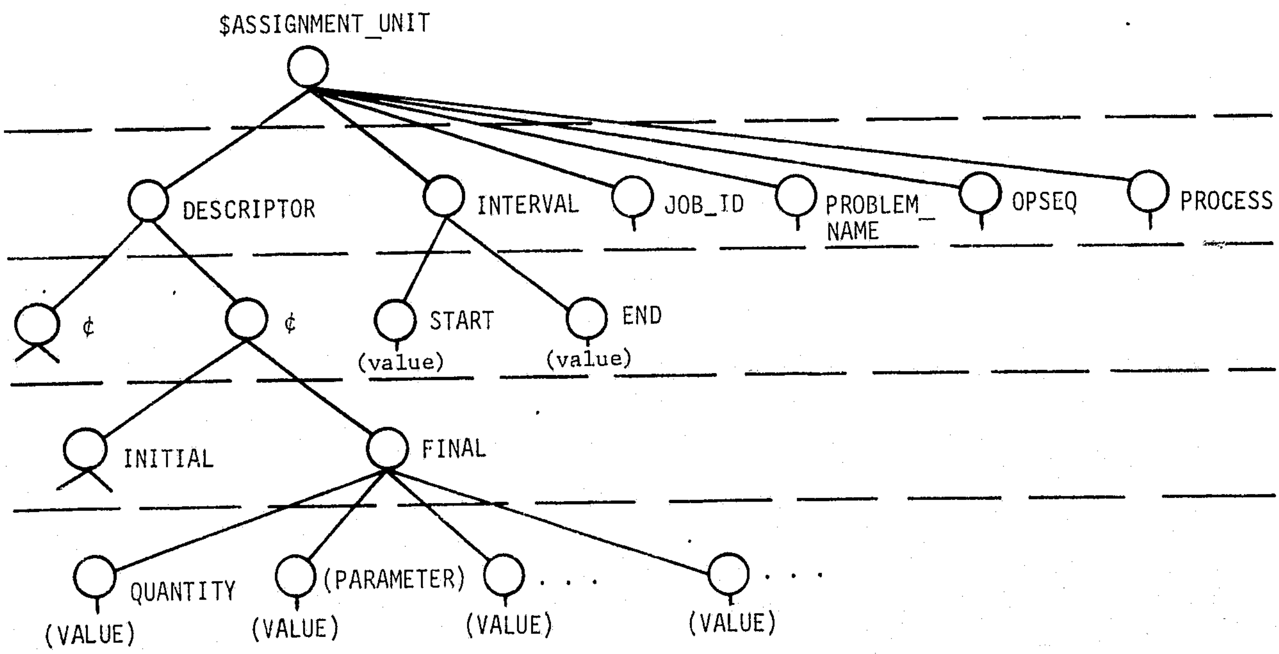
MINIMUM REQUIRED INPUT STRUCTURES FROM STANDARD DATA STRUCTURES
FOR MODULE: WRITE_ASSIGNMENT

> Note: Minimum (i.e., relevant) portion of required input standard Data Structures is shown. In all trees, any additional structure will be preserved.





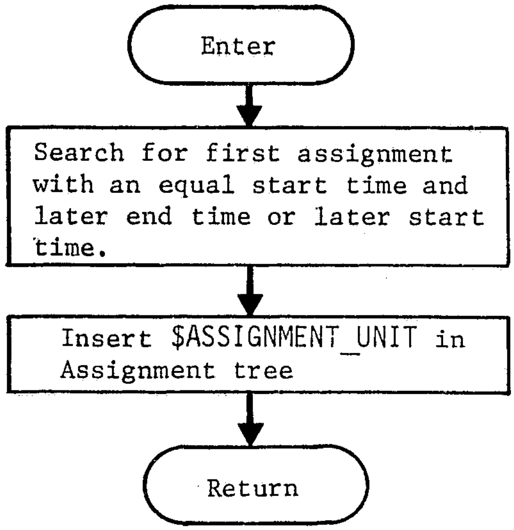
INPUT DATA STRUCTURE



2.4.17.4 Module Output

This module will modify **$ASSIGN** to include an additional assignment element for the specified resource and corresponding time interval.

2.4.17.5 Functional Block Diagram



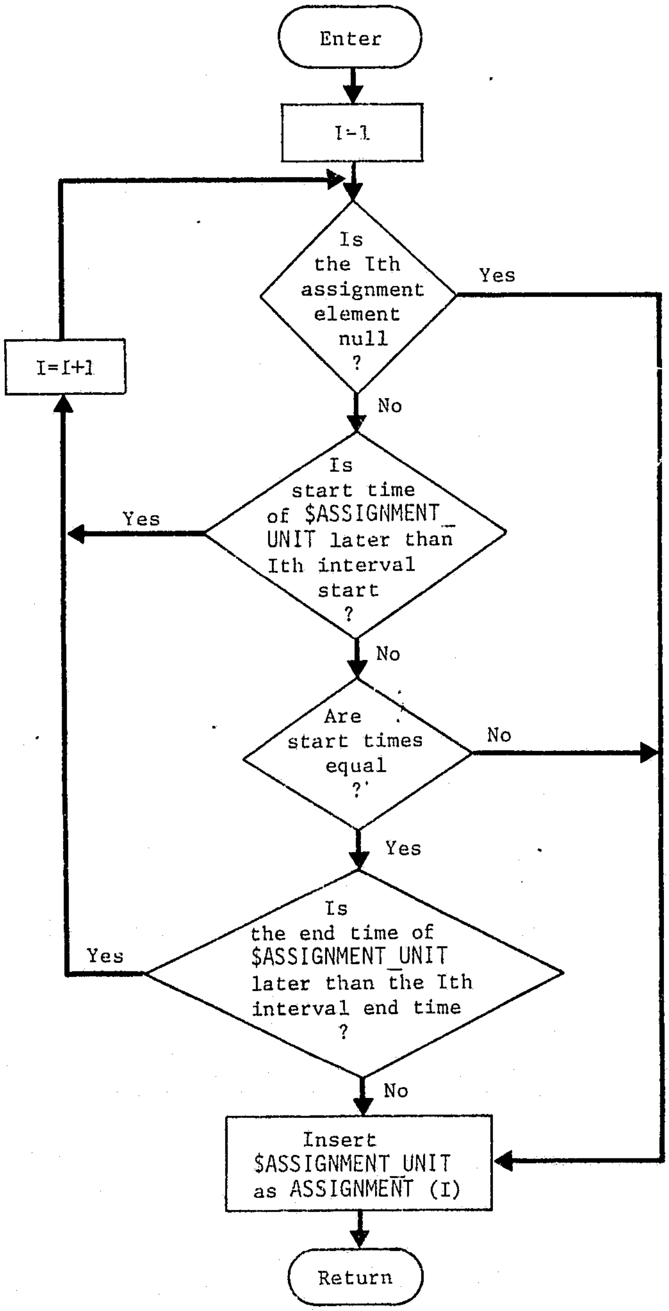
2.4.17.6 Typical Applications

This module is used to create a new assignment interval for a specific resource that has been selected for a given process. The selected interval may be a tentative selection used during the solution of a given problem or the "final" assignments decided upon. Nevertheless, an assignment for a specific resource must be made to indicate its "nonavailability" during the assignment interval.

2.4.17.7 Implementation Considerations

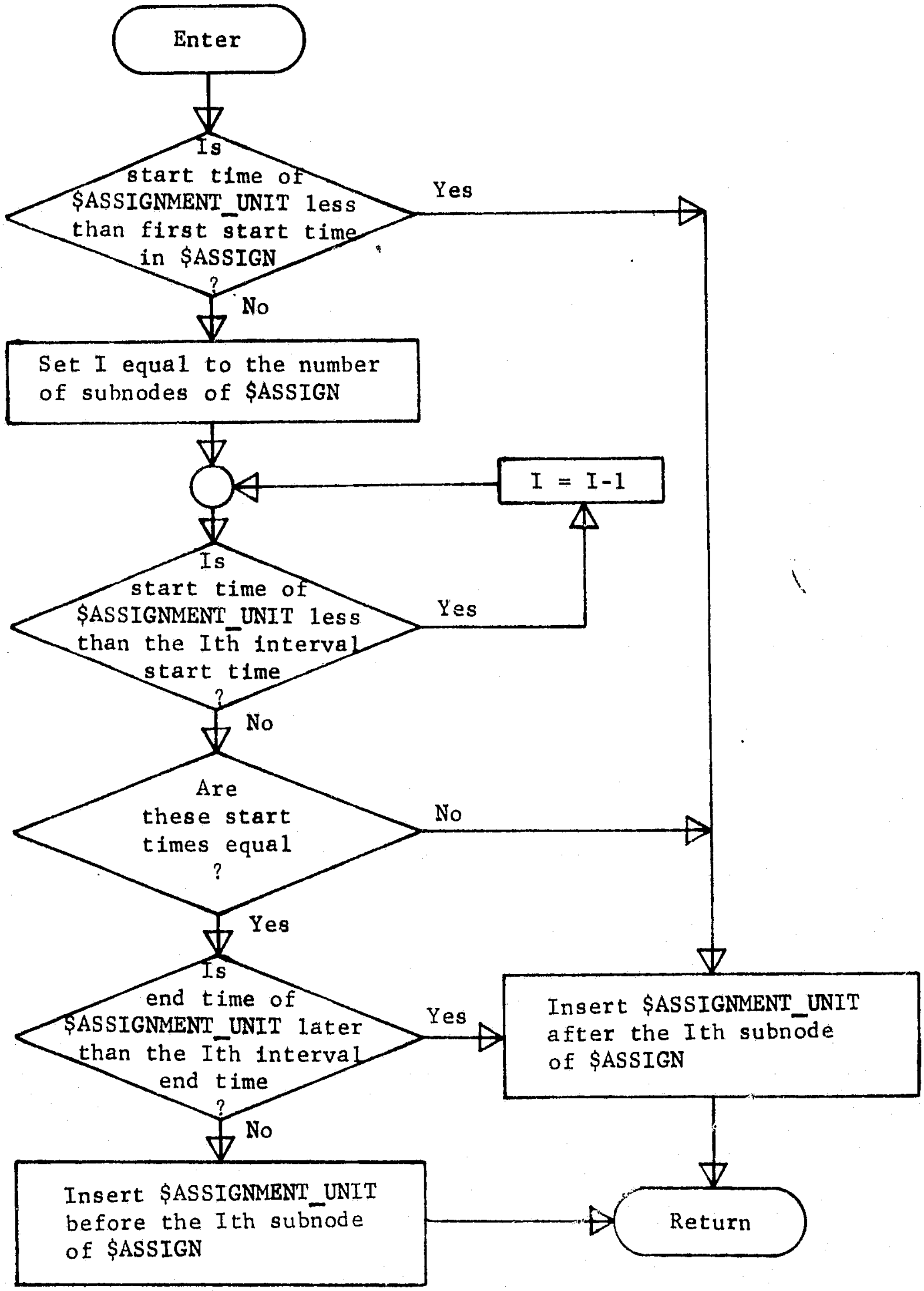
An example flow diagram of this module is shown in the sketch.

EXAMPLE FLOW DIAGRAM OF WRITE_ASSIGNMENT



ORIGINAL PAGE IS OF POOR QUALITY

2.4.17.8 WRITE_ASSIGNMENT FLOWCHART



2.4.17.9 DETAILED DESIGN

As this module was originally conceived, proper placement of the current $ASSIGNMENT_UNIT was determined by comparing it with the intervals in $ASSIGN in a chronological manner. However, it is expected that the usual placement will be as the last interval, so the ordering of comparisons was reversed. This is illustrated in the flowchart sketched below.

Since placing the interval as the first one in $ASSIGN is also expected to occur frequently, a check for this condition is made first. Then comparisons of the interval of $ASSIGNMENT_UNIT with each interval of $ASSIGN are made starting with the last and incrementing towards the first. If the start time of $ASSIGNMENT_UNIT is greater than that of the Ith interval of $ASSIGN, or if the start times are equal and the end time of $ASSIGNMENT_UNIT is greater than that of the Ith interval; $ASSIGNMENT_UNIT is inserted after the Ith interval.If the start times are equal and the end time of $ASSIGNMENT_UNIT is less than that of the Ith interval, $ASSIGNMENT_UNIT is inserted before the Ith interval of $ASSIGN.

```
WRITE_ASSIGNMENT: PROCEDURE($ASSIGNMENT_UNIT,$ASSIGN)
   OPTIONS(EXTERNAL);
   I = 0;
/*  IS START TIME OF $ASSIGNMENT_UNIT LESS THAN THAT OF                       */
/*  FIRST INTERVAL IN $ASSIGN?                                                */
   IF $ASSIGNMENT_UNIT.INTERVAL.START < $ASSIGN(1).INTERVAL.START
      THEN GO TO INSERT_A;
/*  SET I = NUMBER OF SUBNODES OF $ASSIGN                                     */
   DO I = NUMBER($ASSIGN) TO 1 BY -1;
/*  IS START TIME OF $ASSIGNMENT_UNIT LESS THAN                               */
/*  THE I'TH INTERVAL START TIME?                                             */
   IF $ASSIGNMENT_UNIT.INTERVAL.START < $ASSIGN(I).INTERVAL.START
      THEN GO TO END_LOOP;
/*  ARE START TIMES EQUAL?                                                    */
   IF $ASSIGNMENT_UNIT.INTERVAL.START ¬= $ASSIGN(I).INTERVAL.START
      THEN GO TO INSERT_A;
/*  IS END TIME OF $ASSIGNMENT_UNIT LATER THAN                                */
/*  THE I'TH INTERVAL END TIME?                                               */
   IF $ASSIGNMENT_UNIT.INTERVAL.END >= $ASSIGN(I).INTERVAL.END
      THEN GO TO INSERT_A;
      ELSE GO TO END_LOOP ;
/*  I = I - 1                                                                 */
   END_LOOP: END;
/*  INSERT $ASSIGNMENT_UNIT AFTER THE I'TH SUBNODE OF $ASSIGN                 */
   INSERT_A: INSERT $ASSIGNMENT_UNIT BEFORE $ASSIGN(I+1) ;
      RETURN;
/*  INSERT $ASSIGNMENT_UNIT BEFORE THE I'TH SUBNODE OF $ASSIGN                */

   END ;  /* WRITE_ASSIGNMENT  */
```

2.4.18 UNSCHEDULE

2.4.18 UNSCHEDULE

2.4.18.1 Purpose and Scope

This module deletes assignments from the $RESOURCE tree for a given resource or collection of resources. The deletion may be for a single assignment or a collection based on particular processes and/or jobs depending on the contents of the tree $UNSCHEDULE.

2.4.18.2 Modules Called

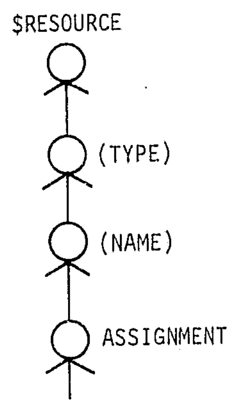
None

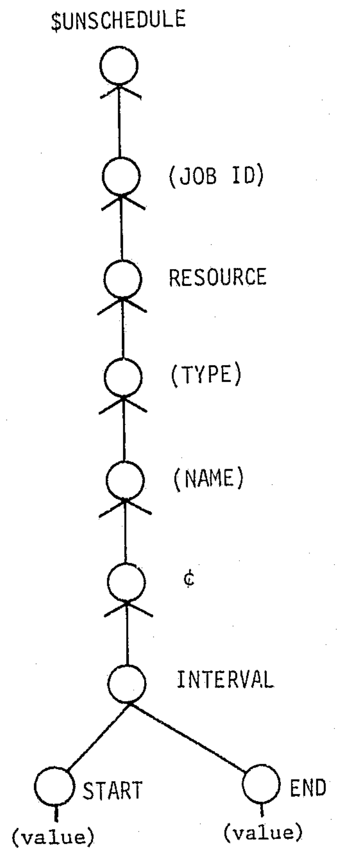
2.4.18.3 Module Input

Inputs to this module by the calling argument will be $RESOURCE and $UNSCHEDULE as shown.

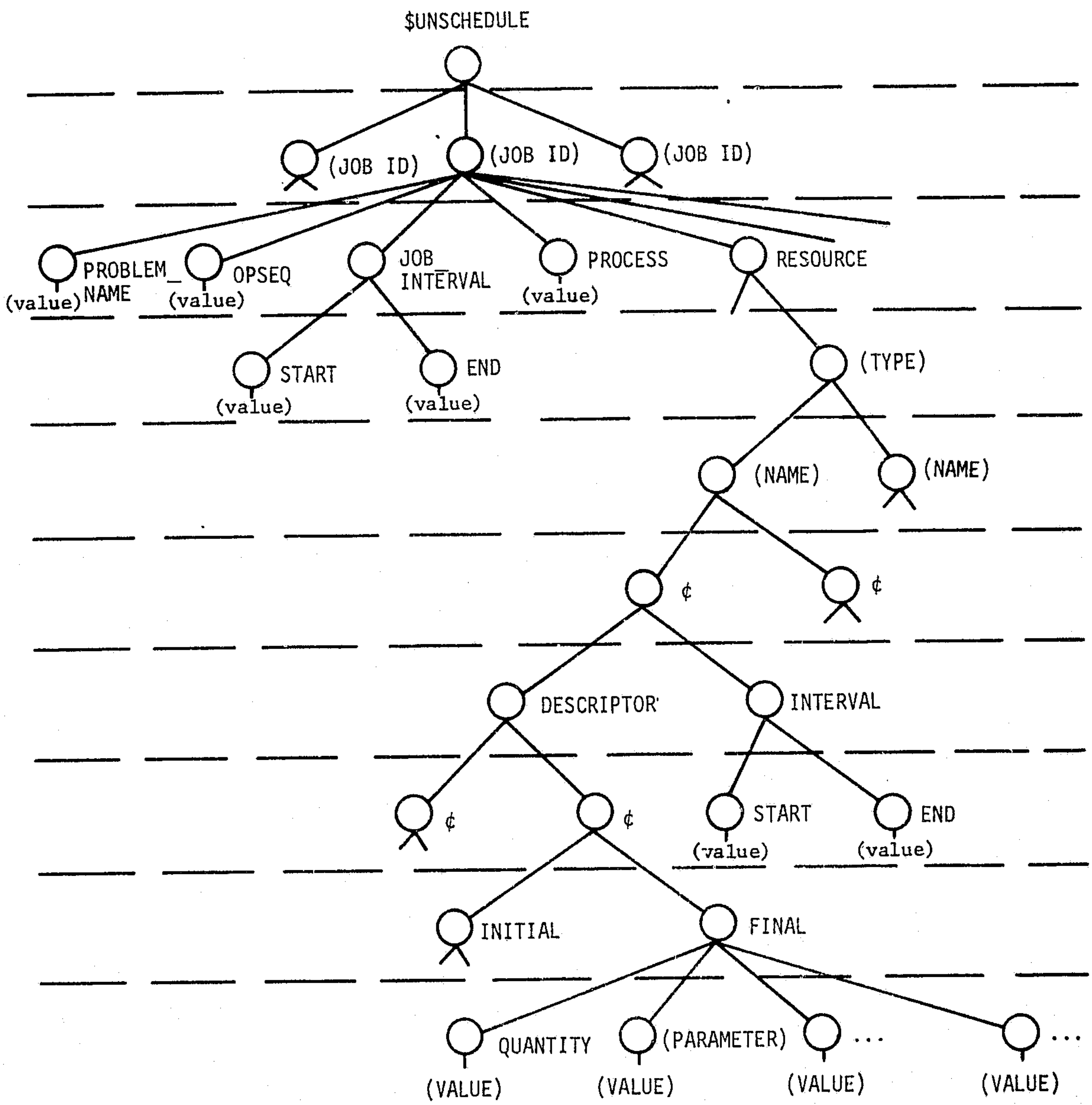
MINIMUM REQUIRED INPUT STRUCTURES FROM STANDARD DATA STRUCTURES
FOR MODULE: UNSCHEDULE

Note:	Minimum (i.e., relevant) portion of required input Standard Data Structures is shown. In all trees, any additional structure will be preserved.





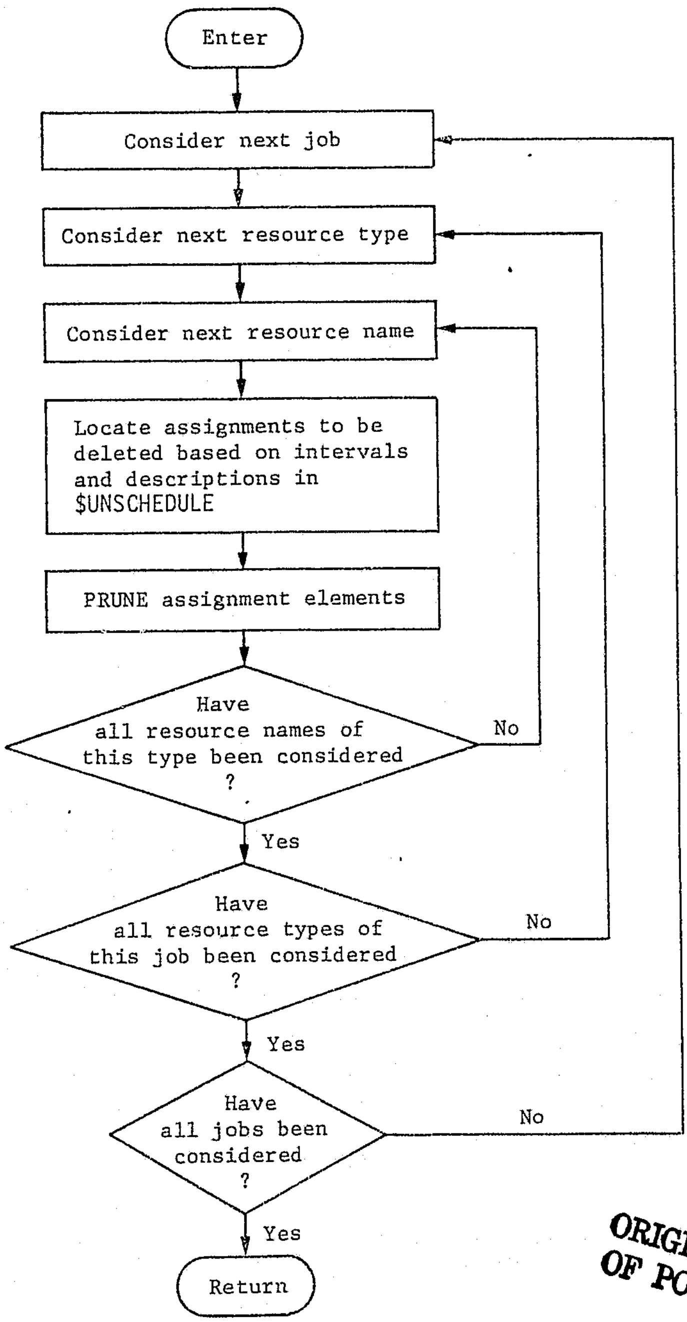
INPUT DATA STRUCTURE



2.4.18.4 Module Output

Upon completion of this module, the assignment portion of $RESOURCE will be altered based on the contents of $UNSCHEDULE.

2.4.18.5 Functional Block Diagram



ORIGINAL PAGE IS OF POOR QUALITY

2.4.18.6 Typical Applications

This module will be used to negate assignments that may have been tried in the problem solution sequence or to update $RESOURCE if previous problem solutions were to be altered.

2.4.18.6 Typical Applications

This module will be used to negate assignments that may have been tried in the problem solution sequence or to update $RESOURCE if previous problem solutions were to be altered.

2.4.18.7 Detailed Design

The functional block diagram for this module is sufficiently detailed for use as a program flowchart. The module loops on job, resource type, and resource name as specified by the input tree $UNSCHEDULE. For each specified resource element, the assignment is located in the $RESOURCE tree and is pruned. If the resource to be unscheduled is not found in $RESOURCE a message to this effect and the resource description and interval are output.

2.4.18.8 Internal Variable Definitions

$ASSIGN	-	Identifier for each subnode of ASSIGNMENT node
$DESCRIP	-	Identifier for each subnode of $NAME
$NAME	-	Identifier for each subnode of $TYPE
$TYPE	-	Identifier for each subnode of RESOURCE node
$UNSCHED_JOB	-	Identifier for each subnode of $UNSCHEDULE, this is the job to be unscheduled

2.4.18.9 COMMENTED CODE

```
UNSCHEDULE: PROCEDURE($UNSCHEDULE,$RESOURCE) OPTIONS(EXTERNAL);
DECLARE $UNSCHED_JOB,$TYPE,$NAME,$DESCRIP,$ASSIGN LOCAL;
/*  CONSIDER NEXT JOB.                                                    */
DO FOR ALL SUBNODES OF $UNSCHEDULE USING $UNSCHED_JOB;
/*  CONSIDER NEXT RESOURCE TYPE.                                          */
   DO FOR ALL SUBNODES OF $UNSCHED_JOB.RESOURCE USING $TYPE;
/*  CONSIDER NEXT RESOURCE NAME.                                          */
      DO FOR ALL SUBNODES OF $TYPE USING $NAME;
/*  LOCATE ASSIGNMENTS TO BE DELETED BASED                                */
/*  ON INTERVALS AND DESCRIPTIONS IN $UNSCHEDULE.                         */
         DO FOR ALL SUBNODES OF $NAME USING $DESCRIP;
            DO FOR ALL SUBNODES OF $RESOURCE.#(LABEL($TYPE)).
                  #(LABEL($NAME)).ASSIGNMENT USING $ASSIGN;
/*  PRUNE ASSIGNMENT ELEMENTS.                                            */
            IF $DESCRIP.INTERVAL IDENTICAL TO $ASSIGN.INTERVAL
               THEN IF $DESCRIP.DESCRIPTOR IDENTICAL TO $ASSIGN.
                  DESCRIPTOR THEN PRUNE $ASSIGN;
            END;
         END;
/*  HAVE ALL RESOURCE NAMES OF THIS TYPE BEEN CONSIDERED?                 */
      END;
/*  HAVE ALL RESOURCE TYPES OF THIS JOB BEEN CONSIDERED?                  */
   END;
/*  HAVE ALL JOBS BEEN CONSIDERED?                                        */
  END;
  RETURN;
END;     /*    UNSCHEDULE     */
```

2.4.19 COMPATIBILITY_SET_ GENERATOR

2.4.19 COMPATIBILITY_SET_GENERATOR

2.4.19.1 Purpose and Scope

The purpose of this module is to enumerate all compatible subsets of a given set when properties determining compatibility have the following frequently occurring structure.

1) Each element of the set has a common set of quantitative properties.

2) Each subset of the set has the same set of properties defined by certain composition rules on the corresponding properties of its elements.

3) For a subset to be compatible, each of its properties is constrained to be either less than or greater than some limiting value.

4) The composition rules are such that the constraint on any properties of any subset can only be tightened by adding another element to that subset.

For such sets and rules then, the compatibility set generator enumerates all of the ordered compatible subsets where the order is that of the input set. The enumeration is done in the lexicographic order based upon the original input ordering. By making use of the fact that under the above property of composition rules any subset of a compatible subset must in turn be compatible, a recursive enumeration scheme can be devised that is far more efficient than examination of all ordered subsets or even the back-tracking procedure used by Walker.

The algorithm should be able to handle sets of cardinality up to 100 with compatibility rules generating up to 10,000 compatible subsets of average cardinality 3. Execution time should be held to a minimum.

2.4.19.2 Algorithm Description

When it is known that any subset of a compatible subset is in turn compatible, the entire collection of ordered compatible subsets can be efficiently enumerated recursively. Let $\mathcal{L}$ denote a sequential set of indices denoting the respective elements of some set whose compatible subsets are to be generated. Let C_n denote an arbitrary ordered compatibility subset of $\mathcal{L}$ of cardinality k. Let E_n denote the set of elements from $\mathcal{L}$ which can be augmented to C_n to form a new ordered compatibility set C_p of cardinality k+1. Symbolically

[1] $$E_n = \left\{ e \in \mathcal{L} : C_p = C_n \cup \{e\} \right\}$$

is a compatible subset and e is greater than any element in C_n

Next, let P_ℓ be the set of elements from $\mathcal{L}$, which taken after ℓ, constitute an ordered compatible subset; that is

[2] $$P_\ell = \{p \in \mathcal{L} : \{\ell, p\} \text{ is a compatible subset and } p > \ell\}$$

Consider any element e in E_n. What elements ℓ from $\mathcal{L}$ are eligible for addition to the ordered set $C_p = C_n \cup \{e\}$ of cardinality k+1 to form the ordered set $C_m = C_p \cup \{\ell\}$ of cardinality k+2?

Clearly, ℓ must be both greater than e and compatible with it; that is ℓ is an element of P_e. Further, since any subset of a compatible subset must be compatible $C_n \cup \{\ell\}$ must be an ordered compatible subset for any ℓ greater than e. Thus, to determine all the ordered compatible subsets of cardinality k+2 that have the ordered compatible subset C_p of cardinality k+1 as a subset, it is only necessary to examine order subsets of the form $C_p \cup \{\ell\}$ where ℓ is an element of the ordered set

$$B_{ne} = P_e \cap E_n\big]_e$$

The notation $S]_s$ denotes the ordered subset of the ordered set S consisting of those elements that are strictly greater than s.

By building an enumeration tree in which each node corresponds to an ordered compatible subset all such subsets can be constructed. The decendants of each node are precisely those derived from it by the addition of one element. Each set of nodes in the tree representing ordered compatible subsets of the same cardinality constitutes a level. Thus, the tree can be built recursively level by level.

The efficiency of this recursive enumeration over the straightforward process of examining all 2^L subsets (L is the cardinality of $\mathcal{L}$) can be considered if the constraints are reasonably tight. In the extreme case of no compatible subsets, the recursive enumeration would terminate after the examination of only L subsets while the complete enumeration process will still try all 2^L subsets.

2.4.19.3 Module Input

1) Set $\mathcal{L}$ of sequential indices denoting set elements;

2) Procedure for determining whether or not a given subset is compatible.

2.4.19.4 Module Output

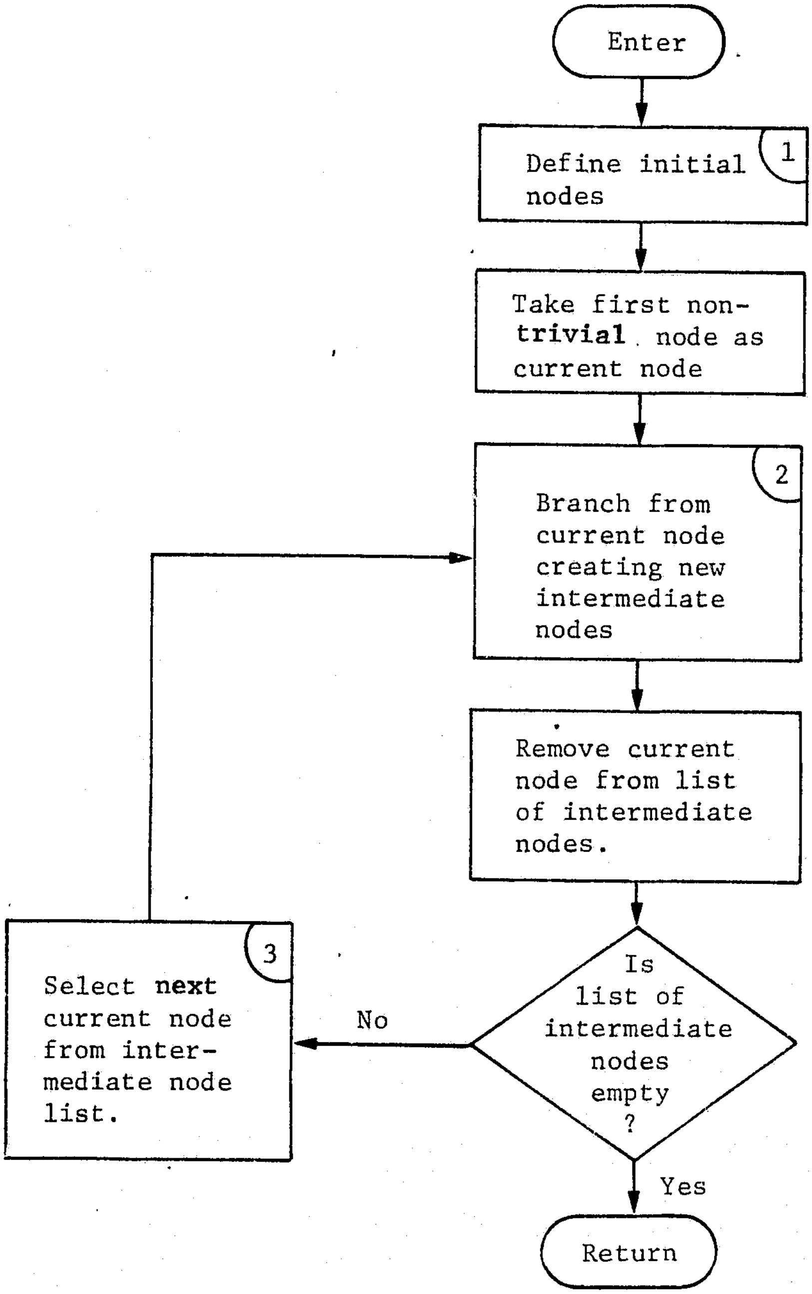
1) Number M of ordered compatible subsets;

2) Complete and nonredundant lexicographic list of ordered compatible subsets, $\left\{C_m\right\}_{m=1}^{M}$;

3) Cardinality K of largest compatible subset;

4) Complete set of starting indices for classes of compatible subsets with the same cardinality, $\left\{m_k\right\}_{k=1}^{K}$.

2.4.19.5 Functional Block Diagram



NOTES ON FUNCTIONAL BLOCKS

1) Initial node definition:

   a) The first node in the tree corresponds to the null subset (trivial node).

   b) Its descendants correspond to the compatible singleton subsets of $\mathcal{L}$.

2) Branching rule:

   Let the compatible subset corresponding to the current node be C_p, and the subset corresponding to its antecendent be C_n. Suppose {e} is augmented to C_n to form C_p; that is

   $C_p = C_n \cup \{e\}$.

   Consider the set

   $$B_{ne} = P_e \cap E_n \big]_e$$

   as previously defined. For each ℓ element of B_{ne} such that $C_m = C_p \cup \{e\}$ is compatible, create a new descendant node for the current node.

3) Current-node selection rule:

   Once branching is completed at a node, the next node selected is that corresponding to the next compatible subset of the same cardinality (the next node at the same level in the tree). If no such node exists, that node corresponding to the first compatibility set of one greater cardinality is selected; i.e., the first node in the next level of the tree is chosen.

2.4.19.6 Typical Applications

Given a set of payloads, determine all subsets whose elements can fly together on a single flight. Representative constraints would be

1) Cargo composite weight can not exceed Shuttle capability;

2) Cargo composite launch window must be at least 2 days wide;

3) Cargo composite volume cannot exceed that of Shuttle cargo bay.

2.4.19.7 Implementation Recommendations

Not all of the enumeration tree or the corresponding compatible subsets need be maintained in high-speed memory.

2.4.19.8 References

Walker, R. J., "An Enumerative Technique for a Class of Combinatorial Problems," Chapter 7 in R. Bellman and M. Hall, Jr. (editors), *Combinatorial Analysis, Proceedings of Symposium on Applied Mathematics*, Volume 10, Page 91-94, American Mathematical Society, Providence, Rhode Island, 1960.

2.4.19.9 Detailed Design

This module builds an enumeration tree of compatible subsets of the original input set of elements subject to criteria input by the user. The allowable criteria are: 1) sum of descriptor values must be less than or equal to a maximum; 2) descriptor values must be less than or equal to a maximum; 3) descriptor values must be greater than or equal to a minimum; and 4) descriptor values must be equal to a value. The module steps through the input $SET testing each element against the criteria and if acceptable, then building compatible subsets of these elements. Then subsets are generated of greater cardinality and tested for compatibility. The output contains the indices of the elements of $SET which are elements of compatible subsets. The indices are output as labels of the nodes of $COMPATIBLE_SUBSET_TREE. Each branch contains the indices of a compatible subset.

2.4.19.10 Internal Variable and Tree Name Definitions

$COMPATIBLE_SUBJECT_TREE	-	output, containing the compatible subsets
INDEX_OF_SET_ELEMENT	-	tracks, index of $SET_ELEMENT within $SET
$SET	-	input, describes the set of elements with descriptors which are to be examined for compatibility
$SET_ELEMENT	-	points at subnodes of $SET
COMPATIBLE_ELEMENT_FLAG	-	if equal to 0, elements are not compatible; if equal to 1, they are compatible
$INDICES_OF_SUBSET_ELEMENTS	-	contains a set of indices from INDEX_OF_SET_ELEMENT
COMPATIBLE_SUBJECT_FLAG	-	if equal to 0, the sum has exceeded upper limit
$FAMILY_HEAD	-	equivalent to $COMPATIBLE_SUBJECT_TREE
INDEX_OF_CHILD	-	the index of $CHILD
$CHILD	-	points at subnodes of $FAMILY_HEAD
INDEX_FIRST_POSSIBLE_GRANDCHILD	-	equal to the index of $CHILD plus 1
INDEX_OF_POSSIBLE_GRANDCHILD	-	temporary counter
INDEX_FIRST	-	the index of the $CHILD of $FAMILY_HEAD
COMPATIBLE_DOUBLET_FLAG	-	the index of the possible grandchild of $FAMILY_HEAD
DOUBLET_PREVIOUSLY_CHECKED_FLAG	-	equals 0 if doublet has been previously checked

$UNEXAMINED_FAMILY_HEADS	-	contains the indices of $CHILD as potential $FAMILY_HEADs.
$INDICES_OF_POSSIBLE_NEW_SUBJECT	-	contains indices of $FAMILY_HEAD
$FAMILY_HEAD_INDEX	-	defines the index of the node the current one came from
MEMBER_INDEX	-	value of $FAMILY_HEAD_INDEX subnode
$COMPATIBILITY_CRITERIA	-	input describing the criteria on which $SET are to be evaluated
$EQUALITY_CONSTRAINT	-	points to subnodes of $COMPATIBILITY_CRITERIA
$INDICES_OF_POSSIBLE_NEW_SUBJECT	-	contains the indices of possible family heads
INDEX_SECOND	-	equal to the value given by $FAMILY_HEAD sub INDEX_OF_POSSIBLE_GRANDCHILD
DESCRIPTOR_SUM	-	variable used to sum up the descriptor values
$INDEX_OF_SUBJECT_ELEMENT	-	points at subnodes of $INDICES_OF_SUBSET_ELEMENTS
$SUM_UPPER_LIMIT	-	points as subnodes of $COMPATIBILITY_CRITERIA equals the upper limit of the sum on descriptor values allowed to be compatible
INDEK	-	equal to the value of $INDEX_OF_SUBSET_ELEMENT, used as an index of $SET
$LOWER_LIMIT	-	a pointer, containing the value of the lower bound set for compatibility by the user
$UPPER_LIMIT	-	a pointer, containing the value of the upper bound set for compatibility by the user

$EQUALITY_CONSTRAINT - a pointer, containing the value of the equality constraint for compatibility as set by the user.

2.4.19.11 COMMENTED CODE

```
   COMPATIBILITY_SET_GENERATOR:    PROCEDURE
/*                                                                   */
/* PURPOSE:                                                          */
/*    GIVEN A SET OF ELEMENTS WITH DESCRIPTORS, ENUMERATE ALL        */
/*    COMPATIBLE SUBSETS OF THE ORIGINAL SET.  A SUBSET IS DETERMINED*/
/*    TO BE COMPATIBLE IF ITS ELEMENTS MEET THE FOLLOWING  CRITERIA. */
/*                                                                   */
/*         . SUM OF DESCRIPTOR VALUES LESS THAN OR EQUAL TO A MAXIMUM */
/*               EG.  SUM OF ELEMENT WEIGHTS    <  200               */
/*                                                                   */
/*         . DESCRIPTOR VALUES LESS THAN OR EQUAL TO A MAXIMUM       */
/*               EG.  ELEMENT.LENGTH     <  10                       */
/*                                                                   */
/*         . DESCRIPTOR VALUES GREATER THAN OR EQUAL TO A MINIMUM    */
/*               EG.  ELEMENT.POWER  <  2                            */
/*                                                                   */
/*         . DESCRIPTOR VALUES EQUAL TO A VALUE                      */
/*               EG.  ELEMENT.PREVIOUS_USAGE = 1                     */
/*                                                                   */
/*                    ELEMENT.COLOR = 'BLUE'                         */
/*                                                                   */
/* INPUT:                                                            */
/*                                                                   */
/*    $SET   -    THE FORM OF $SET IS                                */
/*                  $SET                       /  EXAMPLE:           */
/*                     ELEMENT_NAME            /    $SET             */
/*                        DESCRIPTOR - VALUE   /       EXPERIMENT_A  */
/*                        DESCRIPTOR - VALUE   /          WEIGHT - 50*/
/*                        ETC.                 /          LENGTH - 12*/
/*                     ELEMENT_NAME            /       EXPERIMENT_B  */
/*                     ETC.                    /          WEIGHT - 16*/
/*                                             /          LENGTH - 9 */
/*                                                                   */
/*    $COMPATIBILITY_CRITERIA  -  THE FORM OF $COMPATIBILITY_CRITERIA*/
/*                                IS                                 */
/*                                                                   */
/*                   $COMPATIBILITY_CRITERIA                         */
/*                      UPPER_LIMIT_ON_DESCRIPTOR_SUM                */
/*                         DESCRIPTOR - VALUE                        */
/*                         DESCRIPTOR - VALUE                        */
/*                         ETC.                                      */
/*                      DESCRIPTOR_VALUE_UPPER_LIMIT                 */
/*                         DESCRIPTOR - VALUE                        */
/*                         ETC.                                      */
/*                      DESCRIPTOR_VALUE_LOWER_LIMIT                 */
/*                         DESCRIPTOR - VALUE                        */
/*                         ETC.                                      */
/*                      DESCRIPTOR_VALUE_EQUALITY                    */
/*                         DESCRIPTOR - VALUE                        */
/*                         ETC.                                      */
/*                   ANY ELEMENT OF $SET WHICH DOES NOT HAVE A       */
```

```
/*                  DESCRIPTOR OF EACH TYPE INCLUDED IN                    */
/*                  $COMPATIBILITY_CRITERIA WILL NOT BE CONSIDERED         */
/*                  ELIGIBLE FOR A COMPATIBLE SUBSET. ELEMENTS MAY         */
/*                  HAVE OTHER DESCRIPTORS BUT WILL ONLY BE TESTED ON      */
/*                  THOSE INCLUDED IN $COMPATIBILITY CRITERIA.             */
/*                                                                         */
/*                                                                         */
/* OUTPUT:                                                                 */
/*    $COMPATIBLE_SUBSET_TREE                                              */
/*        THE FORM OF $COMPATIBLE_SUBSET_TREE IS                           */
/*                                                                         */
/*              $COMPATIBLE-SUBSET-TREE                                    */
/*                 INDEX                                                   */
/*                    INDEX                                                */
/*                       INDEX                                             */
/*                       INDEX                                             */
/*                    INDEX                                                */
/*                       INDEX                                             */
/*                          ETC.                                           */
/*        THE INDICES OF THE ELEMENTS OF $SET WHICH ARE ELEMENTS           */
/*        OF COMPATIBLE SUBSETS ARE OUTPUT AS LABELS OF THE NODES          */
/*        OF $COMPATIBLE_SUBSET_TREE. EACH BRANCH CONTAINS THE             */
/*        INDICES OF A COMPATIBLE SUBSET.                                  */
/*                                                                         */
/*     EXAMPLE:     $COMPATIBLE_SUBSET_TREE                                */
/*                       2                                                 */
/*                          5                                              */
/*                             7                                           */
/*                                9                                        */
/*                          7                                              */
/*                             9                                           */
/*                          9                                              */
/*                       5                                                 */
/*                          7                                              */
/*                             9                                           */
/*                          9                                              */
/*                       7                                                 */
/*                          9                                              */
/*                       9                                                 */
/*                                                                         */
/*       THE TOP MOST BRANCH REPRESENTS THE SUBSET                         */
/*                                                                         */
/*                                 ($SET(2),$SET(5),$SET(7),$SET(9) ).     */
/*                                                                         */
   ( $SET, $COMPATIBILITY_CRITERIA, $COMPATIBLE_SUBSET_TREE)
   OPTIONS(EXTERNAL);
/*                                                                         */
     DECLARE  INDEX_OF_SET_ELEMENT, COMPATIBLE_ELEMENT_FLAG,
              $INDICES_OF_SUBSET_ELEMENTS, COMPATIBLE_SUBSET_FLAG,
              INDEX_OF_CHILD,INDEX_FIRST_POSSIBLE_GRANDCHILD,
              DOUBLET_PREVIOUSLY_CHECKED_FLAG,
```

```
            INDEX_FIRST, INDEX_SECOND        LOCAL;
/*                                                                   */
/*    **** GENERATE FIRST LEVEL SUBNODES OF                       *** */
/*    ****      $COMPATIBLE_SUBSET_TREE AS INDICES OF COMPATIBLE  *** */
/*    ****        SUBSETS OF CARDINALITY 1                        *** */
/*    ****                                                        *** */
    PRUNE $COMPATIBLE_SUBSET_TREE ;
    INDEX_OF_SET_ELEMENT = 0 ;
    DO FOR ALL SUBNODES OF $SET USING $SET_ELEMENT ;
       INDEX_OF_SET_ELEMENT = INDEX_OF_SET_ELEMENT + 1 ;
       CALL CHECK_ELEMENT_COMPATIBILITY($SET_ELEMENT,
                                        $COMPATIBILITY_CRITERIA,
                                        COMPATIBLE_ELEMENT_FLAG) ;
       IF COMPATIBLE_ELEMENT_FLAG = 1
       THEN DO; $INDICES_OF_SUBSET_ELEMENTS(FIRST)=
                              INDEX_OF_SET_ELEMENT;
                CALL CHECK_DESCRIPTOR_VALUE_SUMS($SET,
                                        $INDICES_OF_SUBSET_ELEMENTS,
                                        $COMPATIBILITY_CRITERIA,
                                         COMPATIBLE_SUBSET_FLAG) ;
                IF COMPATIBLE_SUBSET_FLAG = 1
                THEN LABEL($COMPATIBLE_SUBSET_TREE(NEXT))
                                        = INDEX_OF_SET_ELEMENT ;
            END;
    END;   /**** GENERATION OF FIRST LEVEL SUBNODES ***/
/*                                                                   */
/* GENERATE SECOND LEVEL NODES OF $COMPATIBLE_SUBSET_TREE AS   ***/
/***     COMPATIBLE SUBSETS OF CARDINALITY 2  (DOUBLETS)            ***/
/*                                                                   */
    DOUBLET_PREVIOUSLY_CHECKED_FLAG = 0;
    DEFINE $FAMILY_HEAD AS $COMPATIBLE_SUBSET_TREE  ;
    INDEX_OF_CHILD = 0 ;
    DO FOR ALL SUBNODES OF $FAMILY_HEAD USING $CHILD ;
       INDEX_OF_CHILD = INDEX_OF_CHILD + 1 ;
       INDEX_FIRST_POSSIBLE_GRANDCHILD = INDEX_OF_CHILD + 1 ;
       DO  INDEX_OF_POSSIBLE_GRANDCHILD  =
           INDEX_FIRST_POSSIBLE_GRANDCHILD TO NUMBER($FAMILY_HEAD) ;
           INDEX_FIRST = LABEL($FAMILY_HEAD(INDEX_OF_CHILD));
           INDEX_SECOND =
              LABEL($FAMILY_HEAD(INDEX_OF_POSSIBLE_GRANDCHILD));
           CALL CHECK_DOUBLET_COMPATIBILITY
                (INDEX_FIRST, INDEX_SECOND,
                 DOUBLET_PREVIOUSLY_CHECKED_FLAG,
                 COMPATIBLE_DOUBLET_FLAG) ;
           IF COMPATIBLE_DOUBLET_FLAG = 1
           THEN DO; LABEL($CHILD(NEXT)) =
                    LABEL($FAMILY_HEAD(INDEX_OF_POSSIBLE_GRANDCHILD));
                    IF NUMBER($CHILD) = 1
                    THEN $UNEXAMINED_FAMILY_HEADS(NEXT)(FIRST)=
                           INDEX_OF_CHILD ;
                END;
```

```
         END;
      END;   /*** GENERATE SECOND LEVEL NODES  ***/
/*                                                                    */
/**   GENERATE ALL NODES BELOW THE SECOND LEVEL BY SUCCESSIVELY      **/
/**   EXAMINING FAMILY HEADS,GENERATING NEW ONES AS NECESSARY        **/
/*                                                                    */
      DOUBLET_PREVIOUSLY_CHECKED_FLAG = 1;
EXAMINE_NEXT_FAMILY_HEAD:
      CALL GENERATE_FAMILY_GRANDCHILDREN($UNEXAMINED_FAMILY_HEADS(FIRST)
                                                                      );
      PRUNE $UNEXAMINED_FAMILY_HEADS(FIRST);
      IF $UNEXAMINED_FAMILY_HEADS(FIRST) IDENTICAL TO $NULL
      THEN RETURN;
      ELSE GO TO EXAMINE_NEXT_FAMILY_HEAD ;
/*                                                                    */
    GENERATE_FAMILY_GRANDCHILDREN:  PROCEDURE  ($FAMILY_HEAD_INDEX);
/*                                                                    */
/***    GIVEN THE INDICES OF THE FAMILY HEAD, GENERATES GRANDCHILDREN */
/***    BY LOOKING FOR POSSIBLE DESCENDANTS OF EACH CHILD             */
/*                                                                    */
/***   FIRST, TRACE THROUGH $COMPATIBLE_SUBSET_TREE TO FIND THE       */
/***    FAMILY HEAD INDICATED BY $FAMILY_HEAD_INDEX                   */
/*                                                                    */
      PRUNE $INDICES_OF_POSSIBLE_NEW_SUBSET ;
      DEFINE $FAMILY_HEAD AS $COMPATIBLE_SUBSET_TREE  ;
      DO I = 1 TO NUMBER($FAMILY_HEAD_INDEX);
      MEMBER_INDEX = $FAMILY_HEAD_INDEX(I) ;
         DEFINE $FAMILY_HEAD AS
                     $FAMILY_HEAD(MEMBER_INDEX)   ;
         $INDICES_OF_POSSIBLE_NEW_SUBSET(I) = LABEL($FAMILY_HEAD);
      END;  /*** TRACE DOWN TO FAMILY HEAD ***/
/*                                                                    */
/**   SEARCH FOR DESCENDANTS OF EACH CHILD                          ***/
/*                                                                    */
      INDEX_OF_CHILD = 0;
      DO FOR ALL SUBNODES OF $FAMILY_HEAD USING $CHILD ;
         $INDICES_OF_POSSIBLE_NEW_SUBSET(NEXT) = LABEL($CHILD);
         INDEX_OF_CHILD = INDEX_OF_CHILD + 1 ;
         INDEX_FIRST_POSSIBLE_GRANDCHILD = INDEX_OF_CHILD + 1 ;
         DO INDEX_OF_POSSIBLE_GRANDCHILD =
            INDEX_FIRST_POSSIBLE_GRANDCHILD TO NUMBER($FAMILY_HEAD) ;
            INDEX_FIRST = LABEL($CHILD);
            INDEX_SECOND =
              LABEL($FAMILY_HEAD(INDEX_OF_POSSIBLE_GRANDCHILD));
            CALL CHECK_DOUBLET_COMPATIBILITY( INDEX_FIRST,INDEX_SECOND,
                  DOUBLET_PREVIOUSLY_CHECKED_FLAG,
                  COMPATIBLE_DOUBLET_FLAG    ) ;
/*                                                                    */
/***       IF DOUBLET PASSES TEST, CHECK THE WHOLE SUBSET           ***/
/*                                                                    */
            IF COMPATIBLE_DOUBLET_FLAG = 1
```

```
          THEN DO; $INDICES_OF_POSSIBLE_NEW_SUBSET(NEXT)
                   =LABEL($FAMILY_HEAD(INDEX_OF_POSSIBLE_GRANDCHILD));
                   CALL CHECK_DESCRIPTOR_VALUE_SUMS
                        ( $SET, $INDICES_OF_POSSIBLE_NEW_SUBSET,
                          $COMPATIBILITY_CRITERIA,
                          COMPATIBLE_SUBSET_FLAG) ;
/** IF POSSIBLE GRANDCHILD HAS PASSED ALL TESTS, ADD HIM TO THE TREE */
                   IF COMPATIBLE_SUBSET_FLAG = 1
                   THEN DO; LABEL($CHILD(NEXT))
                            =LABEL($FAMILY_HEAD
                                  (INDEX_OF_POSSIBLE_GRANDCHILD));
                            IF NUMBER($CHILD) = 1
                            THEN DO; $UNEXAMINED_FAMILY_HEADS(NEXT) =
                                     $UNEXAMINED_FAMILY_HEADS(FIRST);
                                     $UNEXAMINED_FAMILY_HEADS(LAST)
                                       (NEXT) =INDEX_OF_CHILD ;
                                 END;
                        END;
                   PRUNE $INDICES_OF_POSSIBLE_NEW_SUBSET(LAST) ;
                END;
        END;
        PRUNE $INDICES_OF_POSSIBLE_NEW_SUBSET(LAST);
      END;
    END; /*** GENERATE_FAMILY_GRANDCHILDREN  ***/
    CHECK_DOUBLET_COMPATIBILITY: PROCEDURE
      (INDEX_FIRST,INDEX_SECOND,DOUBLET_PREVIOUSLY_CHECKED_FLAG,
        COMPATIBLE_DOUBLET_FLAG)          ;
/*                                                                  */
/***  IF DOUBLET HAS PREVIOUSLY BEEN CHECKED, JUST LOOK UP THE    ***/
/*    RESULTS; OTHERWISE, TREAT DOUBLET AS UNCHECKED SUBSET OF      */
/*    CARDINALITY 2                                                 */
/*                                                                  */
      IF DOUBLET_PREVIOUSLY_CHECKED_FLAG = 0
      THEN DO;
              PRUNE $INDICES_OF_SUBSET_ELEMENTS ;
              $INDICES_OF_SUBSET_ELEMENTS(FIRST)= INDEX_FIRST;
              $INDICES_OF_SUBSET_ELEMENTS(2)     = INDEX_SECOND;
              CALL CHECK_DESCRIPTOR_VALUE_SUMS
                  ($SET,$INDICES_OF_SUBSET_ELEMENTS,
                   $COMPATIBILITY_CRITERIA,COMPATIBLE_DOUBLET_FLAG);
              RETURN;
           END;
      ELSE DO;
              COMPATIBLE_DOUBLET_FLAG = 0;
              DO FOR ALL SUBNODES OF $COMPATIBLE_SUBSET_TREE
                 USING  $CURRENT_ELEMENT;
                 IF LABEL($CURRENT_ELEMENT) > INDEX_FIRST
                 THEN RETURN;
                 IF LABEL($CURRENT_ELEMENT) = INDEX_FIRST
                 THEN DO; DO FOR ALL SUBNODES OF $CURRENT_ELEMENT USING
                                           $CURRENT_SECOND_ELEMENT;
```

```
                                IF LABEL($CURRENT_SECOND_ELEMENT)>INDEX_SECOND
                                THEN RETURN;
                                IF LABEL($CURRENT_SECOND_ELEMENT)=INDEX_SECOND
                                THEN DO; COMPATIBLE_DOUBLET_FLAG = 1;
                                         RETURN;
                                     END;
                                 END;
                            END;
                    END;
                  END;
       END;    /*** CHECK_DOUBLET_COMPATIBILITY ***/
/*                                                                         */
       CHECK_ELEMENT_COMPATIBILITY:    PROCEDURE
/*                                                                         */
/*       CHECKS THE DESCRIPTOR VALUES OF AN ELEMENT AGAINST  UPPER LIMIT,*/
/*       LOWER LIMIT, AND EQUALITY CONSTRAINTS                             */
/*                                                                         */
         ($SET_ELEMENT,$COMPATIBILITY_CRITERIA, COMPATIBLE_ELEMENT_FLAG)  ;
         COMPATIBLE_ELEMENT_FLAG = 1;
/*          ****  CHECK DESCRIPTORS CONSTRAINED TO BE EQUAL  ****          */
         DO FOR ALL SUBNODES OF
                $COMPATIBILITY_CRITERIA.DESCRIPTOR_VALUE_EQUALITY
                                USING $EQUALITY_CONSTRAINT ;
           IF $SET_ELEMENT.#LABEL($EQUALITY_CONSTRAINT) IDENTICAL TO $NULL
           |  $SET_ELEMENT.#LABEL($EQUALITY_CONSTRAINT)¬=
                                                        $EQUALITY_CONSTRAINT
           THEN DO; COMPATIBLE_ELEMENT_FLAG = 0 ;
                    RETURN;
                END;
         END;
/*                                                                         */
/**** CHECK DESCRIPTORS CONSTRAINED BY UPPER LIMITS  ***                   */
         DO FOR ALL SUBNODES OF
                $COMPATIBILITY_CRITERIA.DESCRIPTOR_VALUE_UPPER_LIMIT
                                USING  $UPPER_LIMIT;
           IF $SET_ELEMENT.#LABEL($UPPER_LIMIT)         IDENTICAL TO $NULL
           |   $SET_ELEMENT.#LABEL($UPPER_LIMIT)  >  $UPPER_LIMIT
            THEN DO;  COMPATIBLE_ELEMENT_FLAG = 0;
                      RETURN;
                 END;
         END;
/*                                                                         */
/*   ****     CHECK DESCRIPTORS CONSTRAINED BY LOWER LIMITS      ****    */
         DO FOR ALL SUBNODES OF
                $COMPATIBILITY_CRITERIA.DESCRIPTOR_VALUE_LOWER_LIMIT
                           USING $LOWER_LIMIT ;
           IF $SET_ELEMENT.#LABEL($LOWER_LIMIT)         IDENTICAL TO $NULL
            |  $SET_ELEMENT.#LABEL($LOWER_LIMIT)  <  $LOWER_LIMIT
            THEN DO;  COMPATIBLE_ELEMENT_FLAG = 0;
                      RETURN;
                 END;
```

```
    END;
    DO FOR ALL SUBNODES OF
             $COMPATIBILITY_CRITERIA.UPPER_LIMIT_ON_DESCRIPTOR_SUM
                               USING  $SUM_UPPER_LIMIT  ;
      IF $SET_ELEMENT.#LABEL($SUM_UPPER_LIMIT)
             IDENTICAL TO $NULL
       THEN DO;  COMPATIBLE_ELEMENT_FLAG = 0;
                 RETURN;
            END;
    END;
  END ;   /*    CHECK_ELEMENT_COMPATIBILITY   */
/*                                                                  */
/*                                                                  */
/*                                                                  */
 CHECK_DESCRIPTOR_VALUE_SUMS:  PROCEDURE
/*     CHECKS UPPER LIMIT CONSTRAINTS ON DESCRIPTOR VALUE SUMS      */
/*                                                                  */
    ( $SET, $INDICES_OF_SUBSET_ELEMENTS,$COMPATIBILITY_CRITERIA,
                  COMPATIBLE_SUBSET_FLAG ) ;
    COMPATIBLE_SUBSET_FLAG = 1 ;
    DO FOR ALL SUBNODES OF
             $COMPATIBILITY_CRITERIA.UPPER_LIMIT_ON_DESCRIPTOR_SUM
                               USING  $SUM_UPPER_LIMIT  ;
       DESCRIPTOR_SUM = 0;
       DO FOR ALL SUBNODES OF $INDICES_OF_SUBSET_ELEMENTS
                     USING $INDEX_OF_SUBSET_ELEMENT;
          INDEK = $INDEX_OF_SUBSET_ELEMENT ;
          DESCRIPTOR_SUM = DESCRIPTOR_SUM
           + $SET(INDEK).#LABEL($SUM_UPPER_LIMIT)  ;
       END;
       IF DESCRIPTOR_SUM > $SUM_UPPER_LIMIT
       THEN DO;  COMPATIBLE_SUBSET_FLAG = 0 ;
                RETURN;
            END;
    END;
  END;      /*  CHECK_DESCRIPTOR_VALUE_SUMS  */
  END;  /*** COMPATIBILITY_SET_GENERATOR  ***/
```

2.4.20 FEASIBLE_PARTITION_GENERATOR

2.4.20 FEASIBLE_PARTITION_GENERATOR

2.4.20.1 Purpose and Scope

Given a partition of the positive integer N containing precisely M feasible parts taken from the set $\mathcal{K}$ = {1,2,...K} each element k of which has a specified maximum numer of repetitions r_k, generate the next such feasible partition in decreasing lexicographic order. On the initial call to the module the highest lexicographically ranking partition is generated. On the subsequent call after the lowest lexicographically ranking feasible partition is generated, a flag is returned indicating that the set of feasible partitions has been exhausted.

The intended use of the module is to establish sets of feasible cardinalities for the partitioning subsets of the "Set Partitioning" problem. Since for scheduling it is desirable to have all parts of the partition approximately equal those of highest lexicographic rank are considered first.

For large N and small K, there are considerably fewer feasible partitions than unrestricted ones. For example, suppose N=25, M=10, and K=4. Suppose the repetition limits are 120, 40, 10, 3 for the parts 1, 2, 3, and 4 respectively. Then there are more than 80 unrestricted 10-part partitions of 25 whereas there are only 17 feasible partitions. Clearly, then, some procedure is necessary to eliminate the infeasible partitions if partition analysis is to expedite the solution of the set partitioning problem.

2.4.20.2 Algorithm Description

The generation of the next lower lexicographic-ranking partition of the positive integer N containing precisely M feasible parts taken from the first k positive integers, each with its respective repetition limit r_k, is best done recursively. Three key ideas are involved. First, the highest lexicographically ranking unrestricted partition of any positive integer involving M parts can be found using the "division algorithm" on the ring of integers. Thus, there exists a unique integer q called the quotient, and r called the remainder such that

[1] $$N = Mq + r \text{ and } 0 \leq r < q.$$

The highest lexicographically ranking unrestricted M-part partition of N is then

[2] $$P_N^M(1) = \underbrace{q,\ldots,q}_{M-r \text{ parts}},\underbrace{q+1,\ldots,q+1}_{r \text{ parts}}$$

Note that this partition has at most one break part. A part of a partition is called a break part when it is strictly larger than its preceding part. Partitions are arranged in increasing order of their parts.

The second basic concept is that the next lower ranking unrestricted M part partition N can always be obtained from its predecessor by the following four-step procedure:

1) **Select the least significant break part (ℓ), excluding the last break, in the preceding partition.**

2) If part ℓ has the value 1, no more partitions exist; therefore exit the procedure. (Note that only the first part can become a break part while it is one. When the first part is the least significant break part and it has the value unity, the set of M-part partitions of N have been exhausted.)

3) Decrement part ℓ by one.

4) Replace the final $M-\ell$ parts of the preceding partition by the highest lexicographically ranking $M-\ell$ part partition of N less the first ℓ parts.

The final fundamental idea is that upper bounds must be placed on the respective partition parts to eliminate infeasible partitions.

A reasonably tight bound on each part can be derived from the set of available parts and their respective repetition limits. Define the M-term sequence s_j inductively as

[3] $$s_m = \begin{cases} K \text{ for } M-r_K < m \leq M \\ \ell \text{ for } M-\sum_{k=\ell}^{K} r_k < m \leq M - \sum_{k=\ell+1}^{K} r_k \end{cases}$$

Clearly s_m is an upper bound on the m^{th} part of any feasible partition. Although the respective parts of a partition could fall below these upper bounds and the partition still be infeasible because a given part is repeated too many times, this situation is unlikely in practical set partitioning problems because r_k decreases exponentially with k so that only r_K is binding.

The adaptation of the procedure for recursively generating unrestricted M part partitions of N to a procedure for generating feasible partitions in terms of the above upper bounds is direct. Only the following three steps need be added to the procedure outlined earlier.

5) If none of the final $\ell+1$ parts exceeds its upper bound then exit the procedure with desired feasible partition.

6) If there is a break part more significant than ℓ, reset ℓ to the index of the next most significant part and return to step 2.

7) No more feasible partitions exist; therefore, exit the procedure.

2.4.20.3 Module Input

1) Positive integer, N, to be partitioned

2) Number, M, of parts (not necessarily distinct) required in each partition

3) Largest part, K, permissible

4) Set $\left\{r_k\right\}_{\ell=1}^{K}$ of maximum number of repetitions for each permissible

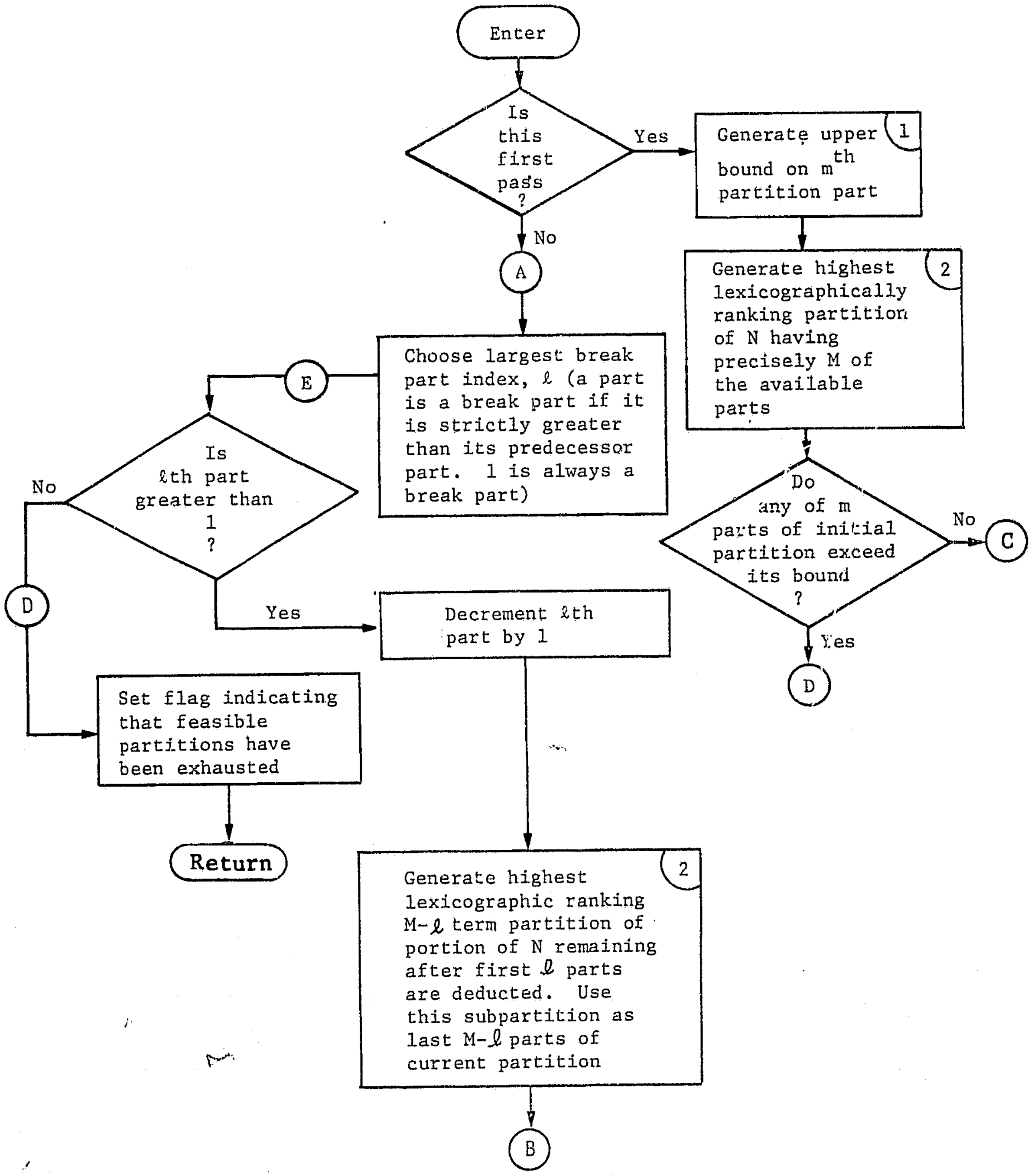
5) Flag indicating that current call is the initial one and that the first feasible partition must be generated

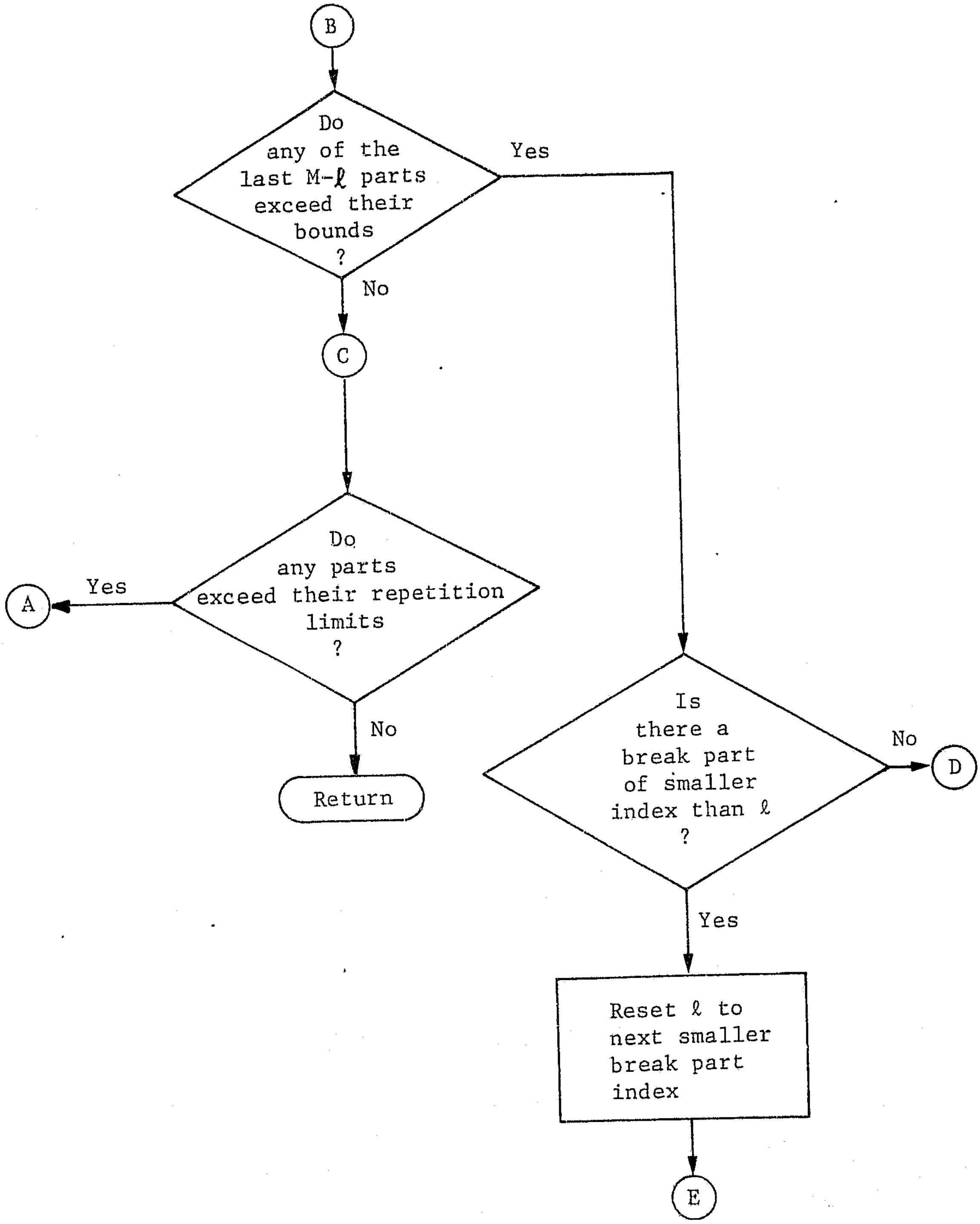
2.4.20.4 Module Output

1) Next partition, $\left\{q_m\right\}_{m+1}^{M}$, in decreasing lexicographic order. The parts of a given partition are ordered in the monotonic nondecreasing order of their indices.

2) Flag indicating that the set of feasible partitions has been exhausted

2.4.20.5 Functional Block Diagram





Notes:
1. Generation of the upper bound on m^{th} partition part (See Equation [3] of the Algorithm Description)
2. Generation of highest lexicographically ranking partition (See Equation [1] of the Algorithm Description)

2.4.20.6 Typical Application

Suppose N payloads are to be flown in M flights and that the compatible payload sets are given. Let the cardinality of the largest compatible payload set be K. Let r_k be the number of compatible payload sets of cardinality k. Then the FEASIBLE_PARTITION_GENERATOR module will generate one at a time and in decreasing lexicographic order all of the feasible ordered sequences of payload cardinalities. The most desirable ordered sequences or partititions from a scheduling point of view are those whose elements are approximately equal. This is true because statistically speaking each compatible payload set will be equally readily scheduled, that is some sets will not be made easy at the expense of making others difficult to schedule. The partitions resulting in the most nearly equal payload cardinalities are those of the highest lexicographic rank. Hence, the module begins with these so that a schedulable combination of compatible payload sets will be found as early as possible in the enumeration process.

2.4.20.7 References

Lehmer, Derrick H., "The Machine Tools of Combinatorics," Chapter 1, Edwin Beckenback (ed), *Applied Combinatorial Mathematics*, John Wiley and Sons, New York, 1964.

Riordan, John, *An Introduction to Combinatorial Analysis*, John Wiley and Sons, New York, 1958.

2.4.20.8 DETAILED DESIGN

This module partitions a positive integer into a specified number of parts which can take on the values of 1, 2, 3 . . . LARGEST_PART, obeying constraints on the maximum number of repetitions allowable for each value.

If a partition is input to the module, the next feasible partition of lower rank is returned. If no partition is input to the module and the INITIAL_PASS_FLAG is set, the feasible partition of highest rank is returned. If a partition is input and no feasible lower rank parition exists, the NO_LOWER_RANK_PARTITION_FLAG is set. Thus, the entire set of feasible partitions can be successively enumerated.

2.4.20.9 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

LARGEST_PART	-	input, largest value any part may take; defaults to INTEGER_FOR_PARTITIONING minus NUMBER_OF_PARTS plus 1
INTEGER_FOR_PARTITIONING	-	input, positive integer to be partitioned
NUMBER_OF_PARTS	-	input, number of parts into which the integer is to be partitioned
$MAXIMUM_REPETITIONS	-	$MAXIMUM_REPETITIONS(I) = the maximum number of times a part of value I may appear in a feasible partition. If $MAXIMUM_REPETITIONS does not have a value input for each I from 1 to LARGEST_PART, $MAXIMUM_REPETITIONS (I) is set to NUMBER_OF_PARTS.
NUMBER_OF_PARTS_MINUS_1	-	equal to NUMBER_OF_PARTS minus 1.
NO_LOWER_RANK_PARTITIONS_ FLAG	-	output, equal to 1 if no feasible partition of lower rank than the input partition exists
INITIAL_PASS_FLAG	-	if equal to 1 then the highest rank feasible partition will be generated if equal to 0, the next highest rank feasible partition counting down from input $INTEGER_ PARTITION will be generated
INDEX_OF_PART	-	an index which varies from 1 to NUMBER_OF_PARTS
$INTEGER_PARTITION	-	an output, $INTEGER_PARTITION(I) equal to the Ith part of the next lower rank feasible partition ordered from lowest valued part to highest. If there is no lower rank partition, a null tree is returned.

$LARGEST_FEASIBLE_PART - an output, $LARGEST_FEASIBLE_PART(I) is an upper bound on the value of $INTEGER_PARTITION (I). It is computed on the initial pass of FEASIBLE_PARTITION_GENERATOR based on LARGEST_PART and $MAXIMUM_REPETITIONS and used for all subsequent passes. It is only output so that it will not be destroyed when leaving the module and require recomputing on each subsequest pass

INDEX_OF_BREAK_PART - contains the index of $INTEGER_PARTITION which is the next break part.

INDEX_STARTER - equal to the starting point from which the module counts backward in $INTEGER_PARTITION to find the next break part

$NUMBER_OF_REPETITIONS - equal to 1 plus the magnitude of the partition contained in $INTEGER_PARTITION

$MAGNITUDE_OF_PART - points at subnodes of $INTEGER_PARTITION

INDEX_OF_REPETITION - equal to the value in $MAGNITUDE_OF_PART

IPORTION_ALREADY_PARTITIONED - the difference between INTEGER_FOR_PARTITIONING and the portion already partitioned

NUMBER_OF_REMAINING_PARTS - the difference between NUMBER_OF_PARTS and INDEX_OF_BREAK_PART

$PARTITION_AFTER_BREAK_PART - equivalent to $LOCAL_PARTITION, equal to the number of elements to be partitioned per part

INDEX_UPPER - equal to NUMBER_OF_PARTS

INDEX_LOWER - equal to the difference between the number of parts and the maximum number of repetitions allowed plus 1

IPART_VALUE	-	an index, decremented from LARGEST_PART to 1
INTEGER_QUOTIENT	-	ratio of number to be partitioned to the number of parts
LOCAL_INTEGER	-	equal to the number to be partitioned
LOCAL_NUMBER_OF_PARTS	-	equal to the number of parts
INTEGER_REMAINDER	-	the remainder as a result of calculating INTEGER_QUOTIENT
INTEGER_QUOTIENT_PLUS_1	-	equal to the value of INTEGER_QUOTIENT plus 1

```
   FEASIBLE_PARTITION_GENERATOR: PROCEDURE
/*                                                                      */
/* PURPOSE:                                                             */
/*   PARTITION A POSITIVE INTEGER INTO A SPECIFIED NUMBER OF PARTS      */
/*   WHICH CAN TAKE ON THE VALUES  1,2,3...,LARGEST_PART, OBEYING       */
/*   CONSTRAINTS ON THE MAXIMUM NUMBER OF REPETITIONS ALLOWABLE FOR     */
/*   EACH VALUE.                                                        */
/*                                                                      */
/*   IF A PARTITION IS INPUT TO THE MODULE, THE NEXT FEASIBLE           */
/*   PARTITION OF LOWER RANK WILL BE RETURNED .                         */
/*   IF NO PARTITION IS INPUT TO THE MODULE AND THE INITIAL_PASS_FLAG*/
/*   IS SET. THE FEASIBLE PARTITION OF HIGHEST RANK IS RETURNED.        */
/*   IF A PARTITION IS INPUT AND NO FEASIBLE LOWER RANK PARTITION       */
/*   EXISTS, THE NO_LOWER_RANK_PARTITION_FLAG IS SET.                   */
/*   THUS, THE ENTIRE SET OF FEASIBLE PARTITIONS CAN BE SUCCESSIVELY */
/*   ENUMERATED.                                                        */
/*                                                                      */
/*   NOTE: RANK IS DETERMINED BY THE MAGNITUDE OF THE LOWEST VALUED     */
/*         PART OF THE PARTITION, TIES BROKEN BY THE MAGNITUDE OF       */
/*         THE SECOND LOWEST VALUED PART ETC.                           */
/*                                                                      */
/*   EXAMPLE:  THE PARTITIONS OF THE INTEGER 6 INTO 3 PARTS RANKED      */
/*             FROM HIGH TO LOW ARE:  (2,2,2)                           */
/*                                    (1,2,3)                           */
/*                                    (1,1,4) .                         */
/*             (NO LIMIT ON PART SIZE OR NUMBER OF REPITITIONS)         */
/*                                                                      */
/* INPUT:                                                               */
/*                                                                      */
/*   INTEGER_FOR_PARTITIONING - POSITIVE INTEGER TO BE PARTITIONED      */
/*   NUMBER_OF_PARTS - NUMBER OF PARTS INTO WHICH THE INTEGER IS TO     */
/*                     BE PARTITIONED                                   */
/*   LARGEST_PART - LARGEST VALUE ANY PART MAY TAKE                     */
/*                  DEFAULTS TO INTEGER_FOR_PARTITIONING -              */
/*                                      (NUMBER_OF_PARTS - 1)           */
/*   $MAXIMUM_REPETITIONS  -  $MAXIMUM_REPITITIONS(I) = THE MAXIMUM     */
/*                            NUMBER OF TIMES A PART OF VALUE I MAY     */
/*                            APPEAR IN A FEASIBLE PARTITION.           */
/*                            IF $MAXIMUM_REPETITIONS DOES NOT HAVE A*/
/*                            VALUE INPUT FOR EACH I FROM 1 TO          */
/*                            LARGEST_PART, $MAXIMUM_REPETITIONS(I)     */
/*                            IS SET TO NUMBER_OF_PARTS.                */
/*   INITIAL_PASS_FLAG   -  =1 HIGHEST RANK FEASIBLE PARTITION WILL BE*/
/*                            GENERATED                                 */
/*                         =0 NEXT HIGHEST RANK FEASIBLE PARTITION      */
/*                            COUNTING DOWN FROM INPUT                  */
/*                            $INTEGER_PARTITION WILL BE GENERATED      */
/*   $INTEGER_PARTITION -  $INTEGER_PARTITION(I) IS THE I TH PART       */
/*                          OF A PARTITION FROM WHICH THE NEXT LOWEST*/
/*                          RANK FEASIBLE PARTITION WILL BE GENERATED*/
/*                          IF INITIAL_PASS_FLAG = 0.                   */
```

```
/*                            THE PARTS OF $INTEGER_PARTITION MUST BE   */
/*                            ORDERED FROM LOW TO HIGH, LOWEST VALUE    */
/*                            PART = $INTEGER_PARTITION(1).             */
/*                            THE INPUT PARTITION MUST BE FEASIBLE.     */
/*                                                                      */
/* OUTPUT:                                                              */
/*                                                                      */
/*   $INTEGER_PARTITION - $INTEGER_PARTITION(I) = I TH PART OF THE      */
/*                        NEXT LOWER RANK FEASIBLE PARTITION ORDERED    */
/*                        FROM LOWEST VALUED PART TO HIGHEST.           */
/*                        IF THERE IS NO LOWER RANK PARTITION, A        */
/*                       NULL TREE IS  RETURNED.                        */
/*   NO_LOWER_RANK_PARTITIONS_FLAG =1  NO FEASIBLE PARTITION OF LOWER*/
/*                              RANK THAN THE INPUT PARTITION           */
/*                              EXISTS                                  */
/*  $LARGEST_FEASIBLE_PART - $LARGEST_FEASIBLE_PART(I) IS AN UPPER      */
/*                           BOUND ON THE VALUE OF                      */
/*                           $INTEGER_PARTITION(I).                     */
/*                           IT IS COMPUTED ON THE INITIAL PASS OF      */
/*                           FEASIBLE_PARTITION_GENERATOR BASED ON      */
/*                           LARGEST_PART AND $MAXIMUM_REPETITIONS      */
/*                           AND USED FOR ALL SUBSEQUENT PASSES. IT     */
/*                           IS ONLY OUTPUT SO THAT IT WILL NOT BE      */
/*                           DESTROYED WHEN LEAVING THE MODULE AND      */
/*                           REQUIRE RECOMPUTING ON EACH SUBSEQUENT     */
/*                           PASS.                                      */
/*                                                                      */
/*                                                                      */
   (INTEGER_FOR_PARTITIONING,NUMBER_OF_PARTS,LARGEST_PART,
    $MAXIMUM_REPETITIONS,INITIAL_PASS_FLAG,$INTEGER_PARTITION,
    NO_LOWER_RANK_PARTITIONS_FLAG, $LARGEST_FEASIBLE_PART   )
   OPTIONS(EXTERNAL);
/*                                                                      */
   DECLARE    INDEX_OF_BREAK_PART,  I, NUMBER_OF_PARTS_MINUS_1,
              INDEX_STARTER , $NUMBER_OF_REPETITIONS, INDEX_OF_PART,
              $MAGNITUDE_OF_PART, INDEX_OF_REPETITION        LOCAL ;
/*                                                                      */
/*      FILL IN DEFAULTS ON LARGEST_PART AND $MAXIMUM_REPETITIONS       */
/*                                                                      */
   IF LARGEST_PART < 1
   THEN LARGEST_PART = INTEGER_FOR_PARTITIONING - (NUMBER_OF_PARTS-1);
   DO I = 1 TO LARGEST_PART ;
     IF  $MAXIMUM_REPETITIONS(I) = ''
     THEN $MAXIMUM_REPETITIONS(I) = NUMBER_OF_PARTS ;
   END;
   NUMBER_OF_PARTS_MINUS_1 = NUMBER_OF_PARTS - 1   ;
   NO_LOWER_RANK_PARTITIONS_FLAG = 0 ;
/*                                                                      */
/*   IF THIS IS NOT THE INITIAL PASS, BEGIN ORDINARY SEQUENCE,          */
/*   OTHERWISE GENERATE HIGHEST RANK PARTITION                          */
/*                                                                      */
```

```
    IF INITIAL_PASS_FLAG = 0
    THEN  CALL NEXT_BREAK_PART_STARTING_FROM(NUMBER_OF_PARTS_MINUS_1) ;
    ELSE DO;
              CALL FEASIBLE_PART_UPPER_BOUND ;
              IF NO_LOWER_RANK_PARTITIONS_FLAG = 1
              THEN RETURN;
              CALL HIGHEST_RANK_PARTITION ( INTEGER_FOR_PARTITIONING,
                                            NUMBER_OF_PARTS,
                                           $INTEGER_PARTITION) ;
              DO INDEX_OF_PART = 1 TO NUMBER_OF_PARTS ;

                IF $INTEGER_PARTITION(INDEX_OF_PART)
                              > $LARGEST_FEASIBLE_PART(INDEX_OF_PART)
                THEN DO;
                         NO_LOWER_RANK_PARTITIONS_FLAG = 1;
                         RETURN;
                     END;
              END;
              GO TO CHECK_REPETITION_LIMITS ;
          END;
/*                                                                      */
TRY_ANOTHER_PARTITION:
    IF $INTEGER_PARTITION(INDEX_OF_BREAK_PART) = 1
    THEN DO;
            PRUNE $INTEGER_PARTITION ;
            NO_LOWER_RANK_PARTITIONS_FLAG = 1 ;
            RETURN;
         END;
    CALL PARTITION_AT_KNOWN_BREAK_PART ;
/*                                                                      */
CHECK_PART_UPPER_BOUNDS:
    DO I = INDEX_OF_BREAK_PART + 1  TO  NUMBER_OF_PARTS ;
       IF $INTEGER_PARTITION(I) > $LARGEST_FEASIBLE_PART(I)
       THEN DO;
               INDEX_STARTER = INDEX_OF_BREAK_PART- 1 ;
               CALL NEXT_BREAK_PART_STARTING_FROM(INDEX_STARTER);
               GO TO TRY_ANOTHER_PARTITION ;
            END;
    END;

/*                                                                      */
CHECK_REPETITION_LIMITS:
    DO I = 1 TO LARGEST_PART ;
       $NUMBER_OF_REPETITIONS(I) = 0 ;
    END;
    DO FOR ALL SUBNODES OF $INTEGER_PARTITION USING $MAGNITUDE_OF_PART;
      INDEX_OF_REPETITION = $MAGNITUDE_OF_PART;
      $NUMBER_OF_REPETITIONS(INDEX_OF_REPETITION) =
      $NUMBER_OF_REPETITIONS(INDEX_OF_REPETITION) + 1 ;
    END;
```

```
   DO I = 1 TO LARGEST_PART ;
     IF  $NUMBER_OF_REPETITIONS(I)
                                  > $MAXIMUM_REPETITIONS(I)
     THEN DO;
              CALL NEXT_BREAK_PART_STARTING_FROM
                                         (NUMBER_OF_PARTS_MINUS_1) ;
              GO TO TRY_ANOTHER_PARTITION ;
          END;
   END ;
/*                                                                      */
/*                                                                      */
   NEXT_BREAK_PART_STARTING_FROM: PROCEDURE     (INDEX_STARTER) ;
/*      FINDS THE NEXT BREAK PART, STARTING AT INDEX_STARTER AND       */
/*      COUNTING BACKWARDS IN $INTEGER_PARTITION                       */
   INDEX_OF_BREAK_PART = 1 ;
   DO I = INDEX_STARTER TO 2 BY -1 ;
      IF  $INTEGER_PARTITION(I) >  $INTEGER_PARTITION(I-1)
      THEN DO;
               INDEX_OF_BREAK_PART = I ;
               RETURN;
           END;
   END;
   END;    /*   NEXT_BREAK_PART_STARTING_FROM    */
/*                                                                      */
/*                                                                      */
   PARTITION_AT_KNOWN_BREAK_PART:    PROCEDURE ;
/*                                                                      */
   DECLARE    IPORTION_ALREADY_PARTITIONED,NUMBER_OF_REMAINING_PARTS,
              IREMAINDER_TO_BE_PARTITIONED,$PARTITION_AFTER_BREAK_PART
   LOCAL ;
   $INTEGER_PARTITION(INDEX_OF_BREAK_PART) =
        $INTEGER_PARTITION(INDEX_OF_BREAK_PART) - 1 ;
   IPORTION_ALREADY_PARTITIONED = 0 ;
   DO I = 1 TO INDEX_OF_BREAK_PART ;
     IPORTION_ALREADY_PARTITIONED = IPORTION_ALREADY_PARTITIONED
                                  + $INTEGER_PARTITION(I);
   END;
   IREMAINDER_TO_BE_PARTITIONED = INTEGER_FOR_PARTITIONING
                                - IPORTION_ALREADY_PARTITIONED ;
   NUMBER_OF_REMAINING_PARTS = NUMBER_OF_PARTS - INDEX_OF_BREAK_PART ;
/*                                                                      */
   CALL HIGHEST_RANK_PARTITION ( IREMAINDER_TO_BE_PARTITIONED,
                                 NUMBER_OF_REMAINING_PARTS,
                                 $PARTITION_AFTER_BREAK_PART) ;
   DO I = 1 TO NUMBER_OF_REMAINING_PARTS ;
     $INTEGER_PARTITION(INDEX_OF_BREAK_PART + I)
                           = $PARTITION_AFTER_BREAK_PART(I);
   END;
   END ;   /*  PARTITION_AT_KNOWN_BREAK_PART    */
/*                                                                      */
/*                                                                      */
```

```
  FEASIBLE_PART_UPPER_BOUND: PROCEDURE  ;
/*                                                                          */
  DECLARE    INDEX_UPPER, INDEX_LOWER, IPART_VALUE   LOCAL  ;
  INDEX_UPPER = NUMBER_OF_PARTS ;
  INDEX_LOWER = NUMBER_OF_PARTS -$MAXIMUM_REPETITIONS(LARGEST_PART)+1;
  DO IPART_VALUE = LARGEST_PART TO 1 BY -1 ;

     IF INDEX_LOWER < 1  THEN  INDEX_LOWER = 1  ;
     DO I = INDEX_LOWER TO INDEX_UPPER ;
       $LARGEST_FEASIBLE_PART(I) = IPART_VALUE ;
     END;
     IF INDEX_LOWER = 1 THEN RETURN ;
     INDEX_UPPER = INDEX_LOWER - 1 ;
     INDEX_LOWER = INDEX_LOWER - $MAXIMUM_REPETITIONS(IPART_VALUE -1) ;
     IF IPART_VALUE = 1
     THEN IF INDEX_LOWER > 1
          THEN DO;  NO_LOWER_RANK_PARTITIONS_FLAG = 1 ;
                    RETURN;
               END;
   END;
   END;    /*   FEASIBLE_PART_UPPER_BOUND    */
/*                                                                          */
/*                                                                          */
   HIGHEST_RANK_PARTITION: PROCEDURE
     (LOCAL_INTEGER,LOCAL_NUMBER_OF_PARTS,$LOCAL_PARTITION)  ;
/*                                                                          */
   DECLARE   INTEGER_QUOTIENT, INTEGER_REMAINDER,  I,
             INTEGER_QUOTIENT_PLUS_1                        LOCAL;
/*                                                                          */
   INTEGER_QUOTIENT = LOCAL_INTEGER / LOCAL_NUMBER_OF_PARTS ;
   INTEGER_REMAINDER = LOCAL_INTEGER
                     - LOCAL_NUMBER_OF_PARTS * INTEGER_QUOTIENT;
   DO I=1 TO LOCAL_NUMBER_OF_PARTS - INTEGER_REMAINDER ;
     $LOCAL_PARTITION(I) = INTEGER_QUOTIENT ;
   END;
   IF INTEGER_REMAINDER > 0
   THEN DO;
           INTEGER_QUOTIENT_PLUS_1 = INTEGER_QUOTIENT + 1 ;
           DO I = LOCAL_NUMBER_OF_PARTS - INTEGER_REMAINDER + 1
                             TO LOCAL_NUMBER_OF_PARTS ;
             $LOCAL_PARTITION(I) = INTEGER_QUOTIENT_PLUS_1 ;
           END;
        END;
   END ;   /*   HIGHEST_RANK_PARTITION       */
   END ;     /*   FEASIBLE_PARTITION_GENERATOR    */
```

2.4.21 PROJECT_DECOMPOSER

2.4.21 PROJECT_DECOMPOSER

2.4.21.1 Purpose and Scope

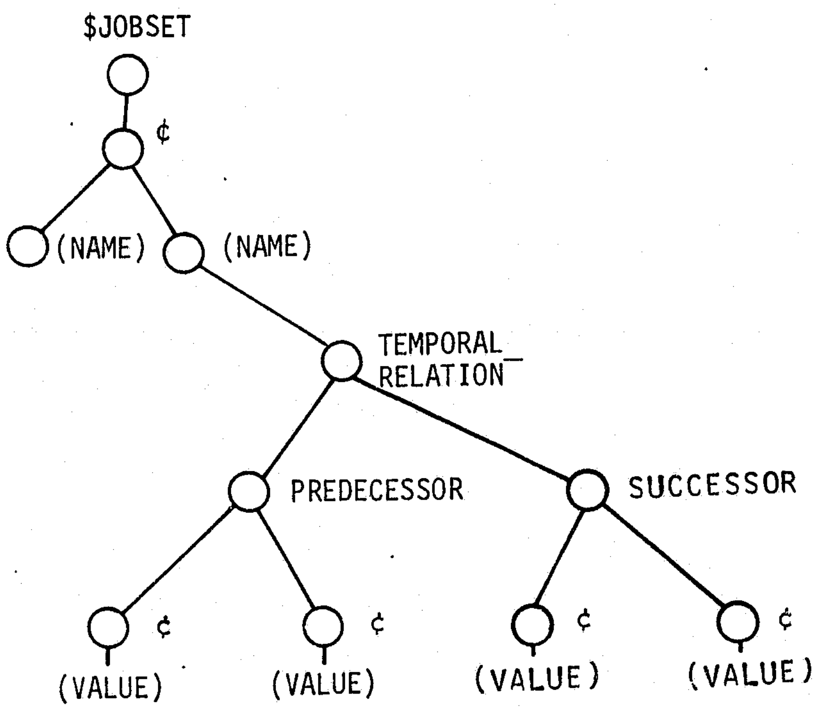
This module will identify all subprojects contained within a specified project. Frequently these subprojects, which are sometimes apparent to the scheduler, are difficult to recognize in the complete network. Identification of the subprojects can significantly reduce the computational effort required to schedule the entire project by enabling some of the scheduling analysis to be done separately for each subproject. For this reason the following analytical procedure is proposed for their detection.

2.4.21.2 Modules Called

None

2.4.21.3 Module Input

Critical path input data $JOBSET



2.4.21.4 Module Output

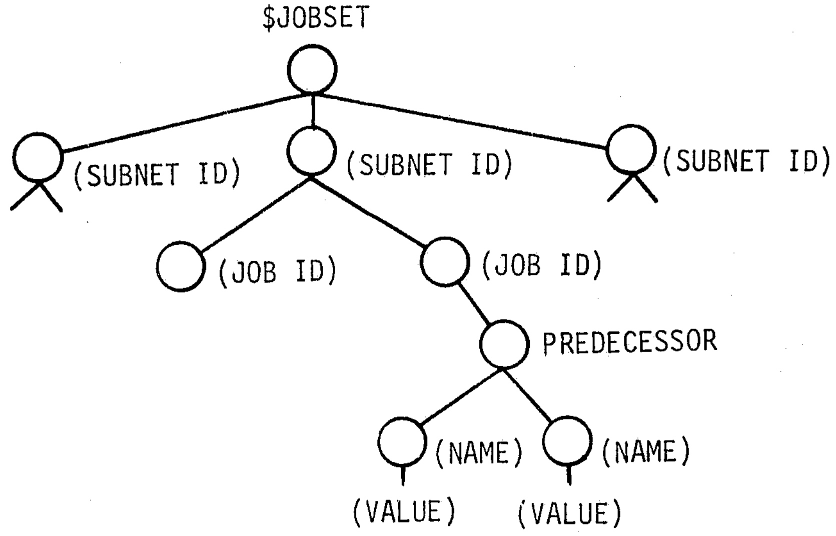
Tree defining the unique subproject decomposition $JOBSET

Subproject identifier (user supplied label)

Member activity or event identifer

Predecessor of activity or event identifer

.
.
.



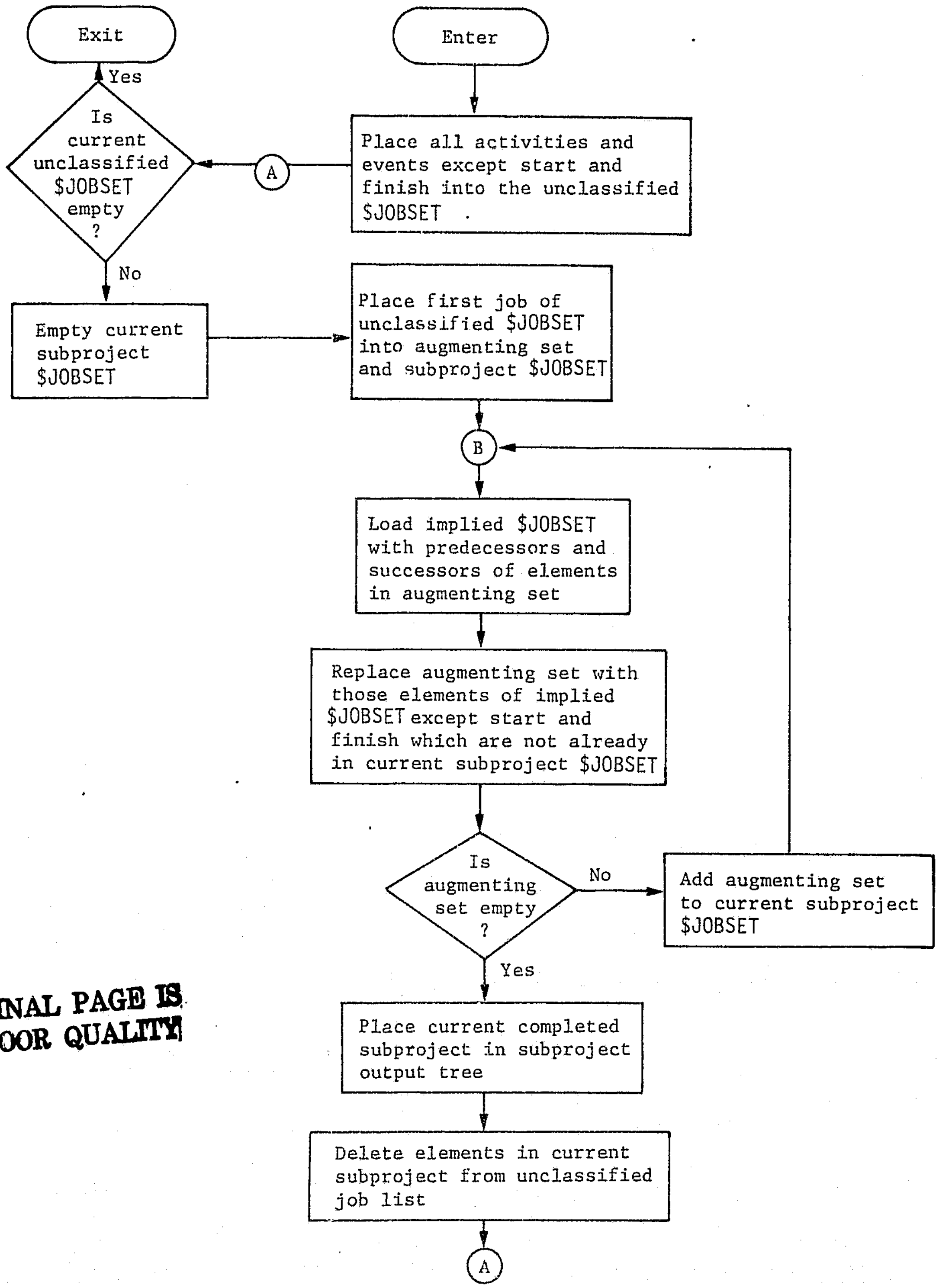
2.4.21.5 Functional Description

In order to construct an algorithm for identifying "subprojects" this term must be precisely defined. A subproject is a subnetwork containing all the predecessors and successors of its member activities. (These, of course, do not include the events START and FINISH.) Recall that a network for scheduling purposes is a set of activities and events denoted by nodes together with all their predecessor and successor relationships represented by branches. Clearly, then, each activity, i, in a network belongs to a unique subproject Q_i that can be generated inductively as follows:

1) Initialize sets $\mathcal{N}$ and $\mathcal{L}$ to the singleton {i}
2) $\mathcal{M} \leftarrow \bigcup_{j\varepsilon\mathcal{L}} \mathcal{P}_j \cup \mathcal{S}_j$ where $\mathcal{P}_j$ and $\mathcal{S}_j$ are the predecessor and successor sets of activity j, respectively
3) $\mathcal{L} \leftarrow \mathcal{M} - \mathcal{M} \cap [\mathcal{N} \cup \{s, f\}]$
4) If $\mathcal{L} = \phi$ go to 5; otherwise $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{L}$ and go to 2
5) The subproject Q_i is simply the set of activities, $\mathcal{N}$, along with the events START and FINISH together with their respective predecessor relations.

Every project can then be decomposed into a unique set of subprojects. To do so, pick an arbitrary activity or event in the project other than start or finish, and generate the unique subproject of which it is a member by the procedure outlined above. If this subproject does not exhaust the network, select any other activity or event other than start of finish not contained in the subprobject and generate its subproject. If this process is continued, the set of subnetworks will eventually exhaust the network thereby providing the desired decomposition.

2.4.21.6 Functional Block Diagram



2.4.21.7 Typical Applications

In order to effectively manage large projects, it is necessary to identify any independent subprojects they may contain. "Independent" here means that the activities in one subproject have no predecessors or successors in any other subproject. Such subprojects can then be analyzed separately as far as precedence calcuations are concerned. They are tied together only by a common start and finish date and any resource requirements they may share. Being smaller and more logically concise, their critical path analysis proceeds quickly. Hence, the critical path calculations for the complete projects can be performed much more quickly by analyzing each of its subprojects independently.

Although the scheduler can frequently decompose small projects into subprojects by simply categorizing the activities by function, this analysis for large projects is at best tedious and at worst unsuccessful. This module provides a simple and efficient automated procedure for decomposing large precedence networks.

2.4.21.8 References

Burman, P. J.: *Precedence Networks for Project Planning and Control*, McGraw Hill, London, 1972.

2.4.21.9 Detailed Design

Since this module will identify subprojects of a specified project, it is essential that the project to be decomposed is the first (or only) subnode of the input tree, $JOBSET. All activities and events (jobs) of the project must be subnodes of this first subnode of $JOBSET. The module starts by placing all activities and events of the project except START and FINISH into an **unclassified set. As each job is grouped into a** subproject, is is removed from the unclassified set, completion of the decomposition is signaled when the unclassified set is emptied. The first job is selected from the unclassified set and placed in the current subproject. Predecessors and successor of this job are identified and added to the current subproject. This process is repeated until no new predecessors, or successors are found and a subproject is completed. Then the next remaining unclassified job is selected and the process repeated, until the unclassified set is emptied. The functional block diagram serves as the flow chart for this module.

2.4.21.10 Interval Variable and Tree Name Definitions

$ADD - Identifier for each subnode of $AUGMENT when funding predecessors and successors

$AUGMENT - Tree containing jobs for which predecessors and successors are identified

$CURRENT - Tree containing jobs identified as belonging to the current subproject

$CURRENT_NAME - Single level tree containing only the names of jobs in the current subproject

$IMPLIED	-	Single level tree containing names of predecessors and successors of jobs in $AUGMENT
$JOB	-	Identifier for each subnode of $AUGMENT when augmenting current subproject
$JOB_ID	-	Identifier of each subnode of the project provided by input $JOBSET
$JOB_NAME	-	Identifier of each subnode of $IMPLIED
$OUTPUT	-	Tree to which identified subprojects are appended
$PRED	-	Identifier for each subnode of predecessors
$REMOVE	-	Identifier for each subnode of $CURRENT_NAME
$SUCC	-	Identifier for each subnode of successors
$UNCLASSIFIED	-	Tree consisting of jobs not associated with any subproject

2.4.21.11 Modifications to Functional Specification and/or Standard Data Structures Assumed

The project to be decomposed must be the first (or only) subnode of the input tree $JOBSET. If more than one project is included in $JOBSET, repeated calls to this module must be made to decompose each project. With each call, only the first project under $JOBSET will be decomposed. **Since this module expects both predecessors and successors to be named, the module PREDECESSOR_SET_INVERTER should be called prior to the first call to the module.**

2.4.21.12 Commented Code

```
PROJECT_DECOMPOSER: PROCEDURE($NETWORK,$DECOMPOSED_NETWORK)
   OPTIONS(EXTERNAL);
/*                                                                    */
/*      PROJECT_DECOMPOSER IDENTIFIES THE INDEPENDENT SUBPROJECTS     */
/*   THAT ARE CONTAINED WITHIN A LARGE PROJECT.  BY INDEPENDENT,      */
/*   WE MEAN THAT NO ACTIVITY IN ONE SUBPROJECT HAS PREDECESSORS      */
/*   OR SUCCESSORS IN ANY OTHER SUBPROJECT.                           */
/*                                                                    */
/*      ALL ACTIVITIES AND EVENTS OF THE PROJECT TO BE DECOMPOSED     */
/*   MUST BE SUBNODES OF THE FIRST SUBNODE OF THE INPUT $JOBSET.      */
/*   THE OUTPUT SUBPROJECTS ARE RETURNED AS THE FIRST 'N' SUBNODES    */
/*   OF $JOBSET, WHERE 'N' IS THE NUMBER OF SUBPROJECTS IDENTIFIED.   */
/*   ANY OTHER PROJECTS WILL FOLLOW THESE N SUBNODES.  IF ANY OF      */
/*   THE OTHER PROJECTS ARE TO BE DECOMPOSED, IT MUST BE MOVED TO     */
/*   THE POSITION OF THE FIRST SUBNODE AND THIS MODULE CALLED AGAIN.  */
/*                                                                    */
DECLARE $CURRENT,$IMPLIED,$AUGMENT,$UNCLASSIFIED,$CURRENT_NAME LOCAL;
DECLARE $JOB,$JOB_ID,$PRED,$SUCC,$JOB_NAME,$ADD,$REMOVE,$OUTPUT,
      $NETWORK,$DECOMPOSED_NETWORK,$TEMP_PRED,$TEMP_SUCC LOCAL;
/*      THE MODULE STARTS BY PLACING ALL ACTIVITIES AND EVENTS IN     */
/*   A SET TO BE EXAMINED.  ALL ACTIVITIES OR EVENTS WILL HAVE BEEN   */
/*   PLACED IN SUBNETWORKS AND CONTROL WILL BE RETURNED TO THE        */
/*   CALLING PROGRAM WHEN THIS SET HAS BEEN EMPTIED.                  */
DO FOR ALL SUBNODES OF $NETWORK USING $JOB_ID;
   IF (LABEL($JOB_ID) ='START'| LABEL($JOB_ID) ='FINISH')THEN;
      ELSE $UNCLASSIFIED(NEXT) = $JOB_ID;
END;
POINT_A:
   IF $UNCLASSIFIED IDENTICAL TO $NULL THEN DO;
    IF $NETWORK.START IDENTICAL TO $NULL   THEN;
    ELSE GRAFT INSERT $NETWORK.START BEFORE $OUTPUT(FIRST);
    IF $NETWORK.FINISH IDENTICAL TO $NULL    THEN;
    ELSE GRAFT INSERT $NETWORK.FINISH BEFORE $OUTPUT(NEXT);
    LABEL($OUTPUT)=LABEL($NETWORK);
    PRUNE $NETWORK;
         $DECOMPOSED_NETWORK = $OUTPUT;
         PRUNE $OUTPUT;
         RETURN;
         END;
      ELSE PRUNE $CURRENT;
/*      START A CURRENT SUBPROJECT SET AND AN AUGMENTING SET WITH     */
/*   THE FIRST OF THE SET TO BE EXAMINED.                             */
   $CURRENT(NEXT) = $UNCLASSIFIED(FIRST);
   $AUGMENT(NEXT) = $UNCLASSIFIED(FIRST);
   $CURRENT_NAME(NEXT) = LABEL($UNCLASSIFIED(FIRST));
POINT_B:
/*      FIND ALL UNIQUE PREDECESSORS AND SUCCESSORS OF ACTIVITIES     */
/*   AND EVENTS IN THE AUGMENTING SET.                                */
   DO FOR ALL SUBNODES OF $AUGMENT USING $JOB;
      DO FOR ALL SUBNODES OF $JOB.TEMPORAL_RELATION.PREDECESSOR USING
         $PRED;
```

```
        IF $PRED ELEMENT OF $IMPLIED THEN;
           ELSE $IMPLIED(NEXT) = $PRED;
        END;
        DO FOR ALL SUBNODES OF $JOB.TEMPORAL_RELATION.SUCCESSOR USING
           $SUCC;
        IF $SUCC ELEMENT OF $IMPLIED THEN;
           ELSE $IMPLIED(NEXT) = $SUCC;
        END;
     END;
/*        REPLACE THE AUGMENTING SET WITH THE PREDECESSORS AND             */
/*     SUCCESSORS JUST FOUND WHICH ARE NEITHER START NOR FINISH,           */
/*     NOR WITHIN THE CURRENT SUBPROJECT SET.                              */
     PRUNE $AUGMENT;
     DO FOR ALL SUBNODES OF $IMPLIED USING $JOB_NAME;
     IF ($JOB_NAME ='START'| $JOB_NAME ='FINISH')THEN;
        ELSE IF $JOB_NAME ELEMENT OF $CURRENT_NAME THEN;
           ELSE $AUGMENT(NEXT) = $UNCLASSIFIED.#($JOB_NAME);
     END;
     PRUNE $IMPLIED;
/*        IF THE AUGMENTING SET IS NOT EMPTY, ADD IT TO THE CURRENT        */
/*     SUBPROJECT SET AND RETURN TO FIND MORE PREDECESSORS AND             */
/*     SUCCESSORS.                                                         */
     IF $AUGMENT ¬ IDENTICAL TO $NULL
        THEN DO;
           DO FOR ALL SUBNODES OF $AUGMENT USING $ADD;
           $CURRENT(NEXT) = $ADD;
           $CURRENT_NAME(NEXT) = LABEL($ADD);
           END;
           GO TO POINT_B;
           END;
/*        IF THE AUGMENTING SET IS EMPTY, A SUBPROJECT HAS BEEN            */
/*     IDENTIFIED AND IT IS THEN PLACED IN THE OUTPUT TREE.  ALL           */
/*     ACTIVITIES AND EVENTS OF THIS SUBPROJECT ARE REMOVED FROM THE       */
/*     SET TO BE EXAMINED, AND THE PROCESS REPEATED UNTIL THE SET TO       */
/*     BE EXAMINED HAS BEEN EMPTIED.                                       */
        ELSE GRAFT $CURRENT AT $OUTPUT(NEXT);
     DO FOR ALL SUBNODES OF $CURRENT_NAME USING $REMOVE;
     PRUNE $UNCLASSIFIED.#($REMOVE);
     END;
     PRUNE $CURRENT_NAME;
  GO TO POINT_A;
  END_PROJECT_DECOMPOSER: END;
```

2.4.22 REDUNDANT_PREDECESSOR_CHECKER

2.4.22 REDUNDANT_PREDECESSOR_CHECKER

2.4.22.1 Purpose and Scope

Given a technologically ordered set of activities and respective predecessor sets, this module eliminates any redundant predecessors. A predecessor is said to be redundant if it is not an immediate predecessor; that is, there is at least one intervening activity between the predecessor and its successor. As an example, suppose activity A is a predecessor of activity B, and B is a predecessor of activity C. Then A is a redundant predecessor of C, while A and B are immediate predecessors of B and C, respectively.

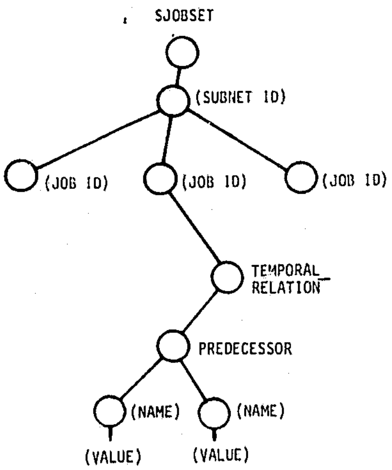
Expressing a project in terms of a collection of nonredundant predecessors serves two useful purpose: (1) it expedites considerably critical path calculations; (2) its facilities comprehension of the precedence relations by representing the project in terms of the most logically concise precedence network possible.

2.4.22.2 Modules Called

None

2.4.22.3 Module Input

Network definition $JOBSET - including redundant predecessors.



2.4.22.4 Module Output

Network definition $JOBSET - technologically ordered, excluding redundant predecessors.

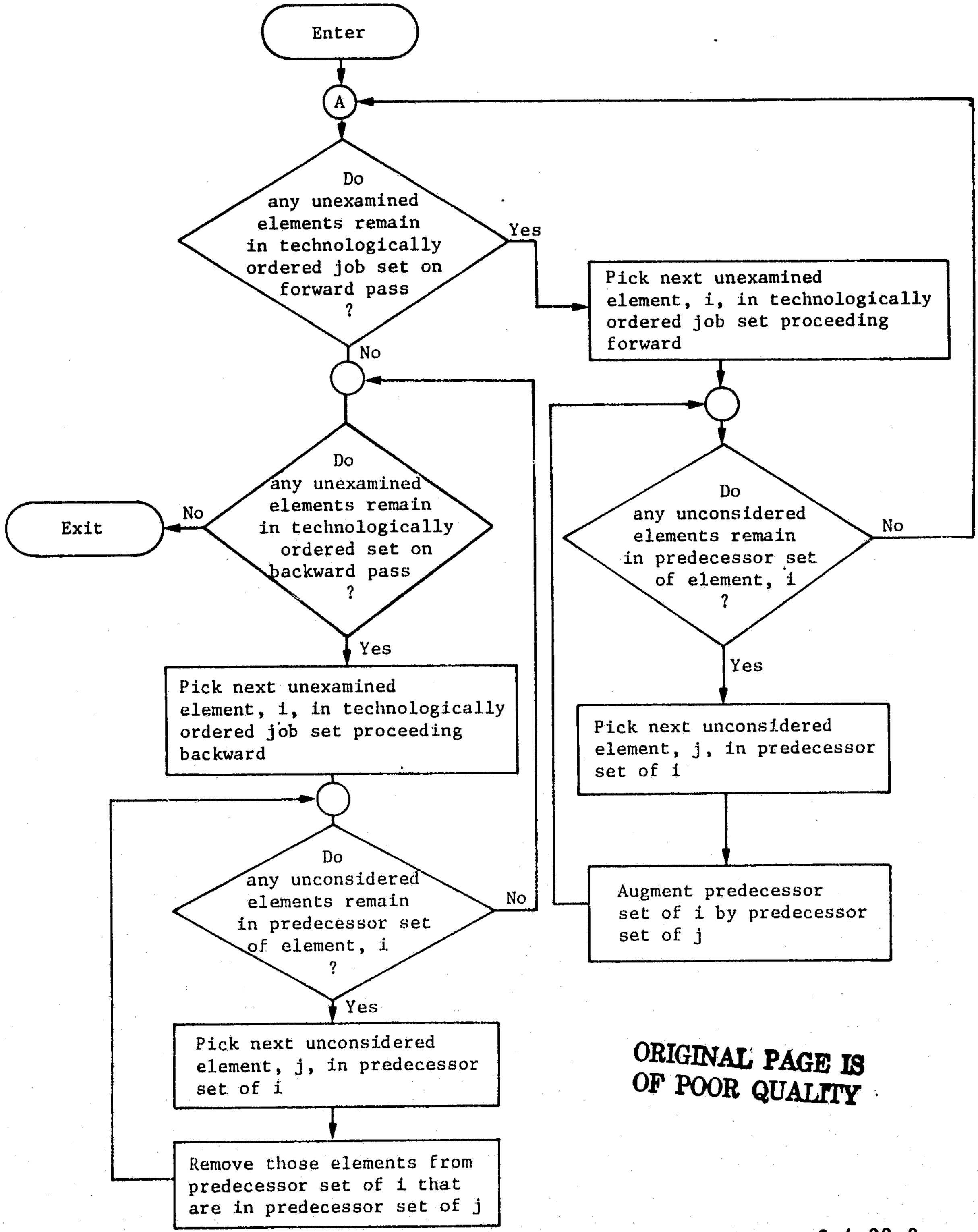
2.4.22.5 Functional Description

The most efficient redundant predessor elimination algorithm is a two-phase recursive procedure based on a technologically ordered job set.

The first, or forward phase, recursively augments the predecessor sets to introduce maximum redundancy beginning with the predecessor set of the first element in the technologically ordered job set. The second, or reverse phase, recursively decrements the maximally redundant predecessor sets to secure minimum redundancy beginning with the predecessor set of the last element in the technologically ordered job set. The major difficulty with this or any other algorithm designed to eliminate redundant predecessors is the excessive storage requirements. For a job set containing n activities up to $n^2/2$ memory cells can be required to store the intermediate maximally redundant predecessors.

2.4.22.6 Functional Block Diagram

REDUNDANT_PREDECESSOR_CHECKER



ORIGINAL PAGE IS OF POOR QUALITY

2.4.22.7 Typical Application

The module can be applied wherever the most logically concise precedence network representation of a project is desired. This includes critical path calculation, automated heuristic scheduling, and manual precedence relation analysis.

2.4.22.8 References

Muth, John F. and Gerald L. Thompson: *Industrial Scheduling*, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1963.

2.4.22.9 Detailed Design

The functional block diagram provides the flowchart for this module. The module takes each job element, i, of a given subnet in turn, and examines each predecessor, j, of the element. It augments the predecessor set of i with the predecessor set of j. When all job elements have been examined, the maximally redundant predecessor set has been formed. Next each job element, i, of the subnet is examined, but in reverse order. Each predecessor, j, of the element i is examined. If a predecessor of j is found as a predecessor of i, it is pruned from the predecessor set of i.

2.4.22.10 Internal Variable and Tree Name Definitions

$PRECESSOR - Pointer first subnode of PRECESSOR node

$SET - Temporary tree storage for predecessor names of job element under consideration

2.4.22.11 Modifications to Functional Sepcifications and/or Standard Data Structures Assumed

Since only a single subnode of $JOBSET is examined with each call to this module, the subnode is identified as $SUBNET through the parameter list.

```
REDUNDANT_PREDECESSOR_CHECKER: PROCEDURE($SUBNET)
                OPTIONS(EXTERNAL);

  DECLARE $SET, $SAVE, $PREDECESSOR   LOCAL;

/* FOREWORD PASS TO CREATE MAXIMALLY REDUNDANT PREDECESSOR SET          */

/*  PICK NEXT UNEXAMINED ELEMENT, I, IN TECHNOLOGICALLY ORDERED          */
/*  JOB SET PROCEDING FORWARD                                             */
DO I = 1 TO NUMBER($SUBNET) ;
/*  PICK NEXT UNCONSIDERED ELEMENT, J, IN PREDECESSOR SET OF I           */
   DO J= 1 TO NUMBER($SUBNET(I).TEMPORAL_RELATION.PREDECESSOR) ;
/*  AUGMENT PREDECESSOR SET OF I BY PREDECESSOR SET OF J                 */
      DO K = 1 TO NUMBER($SUBNET.#($SUBNET(I).TEMPORAL_RELATION
       .PREDECESSOR(J)).TEMPORAL_RELATION.PREDECESSOR) ;
          IF $SUBNET.#($SUBNET(I).TEMPORAL_RELATION
           .PREDECESSOR(J)).TEMPORAL_RELATION.PREDECESSOR(K)
          ¬ELEMENT OF $SUBNET(I).TEMPORAL_RELATION.PREDECESSOR
          THEN $SUBNET(I).TEMPORAL_RELATION.PREDECESSOR(NEXT) =
           $SUBNET.#($SUBNET(I).TEMPORAL_RELATION.PREDECESSOR(J))
           .TEMPORAL_RELATION.PREDECESSOR(K) ;
      END ;
   END ;
END ;


/* BACKWARD PASS TO CREATE MINIMALLY REDUNDANT PREDECESSOR SET          */

/*  PICK NEXT UNEXAMINED ELEMENT, I, IN TECHNOLOGICALLY ORDERED          */
/*  JOB SET PROCEDING BACKWARD                                            */
DO I = NUMBER($SUBNET) TO 1 BY -1 ;
   PRUNE $SET;
/*  PICK NEXT UNCONSIDERED ELEMENT, J, IN PREDECESSOR SET OF I           */
   DO J = 1 TO NUMBER($SUBNET(I).TEMPORAL_RELATION.PREDECESSOR) ;
/*  REMOVE THOSE ELEMENTS FROM PREDECESSOR SET OF I THAT ARE IN          */
/*  PREDECESSOR SET OF J                                                  */
      DO K = 1 TO NUMBER($SUBNET.#($SUBNET(I).TEMPORAL_RELATION
       .PREDECESSOR(J)).TEMPORAL_RELATION.PREDECESSOR) ;
         IF $SUBNET.#($SUBNET(I).TEMPORAL_RELATION.PREDECESSOR(J))
          .TEMPORAL_RELATION.PREDECESSOR(K) ELEMENT OF $SUBNET(I)
          .TEMPORAL_RELATION.PREDECESSOR  & $SUBNET.#($SUBNET(I)
          .TEMPORAL_RELATION.PREDECESSOR(J)).TEMPORAL_RELATION
          .PREDECESSOR(K) ¬ ELEMENT OF $SET
         THEN $SET(NEXT) = $SUBNET.#($SUBNET(I).TEMPORAL_RELATION
          .PREDECESSOR(J)).TEMPORAL_RELATION.PREDECESSOR(K) ;
      END ;
   END ;

   PRUNE $SAVE;
      IF $SUBNET(I).TEMPORAL_RELATION.PREDECESSOR ¬ IDENTICAL TO $NULL
```

```
        THEN DO;
           DEFINE $PREDECESSOR AS $SUBNET(I).TEMPORAL_RELATION.
                  PREDECESSOR(FIRST);
           DO WHILE ($PREDECESSOR ¬ IDENTICAL TO $NULL);
              IF $PREDECESSOR ELEMENT OF $SET
                THEN PRUNE $PREDECESSOR;
                ELSE ADVANCE $PREDECESSOR;
             END;
           END;
END ;
END REDUNDANT_PREDECESSOR_CHECKER;
```

2.4.23 CRITICAL_PATH_CALCULATOR

2.4.23 CRITICAL_PATH_CALCULATOR

2.4.23.1 Purpose and Scope

This module will calculate the critical path data for a project network. The variables computed are: (1) early-start, late-start, early-finish, and late-finish of each activity; (2) early occurrence and late occurrence of each event; and (3) total slack and free slack of each activity and event.

A project that is defined by a collection of activities and events, their precedence constraints, and their durations must meet several other requirements to be amendable to critical path analysis:

1) It must consist of a *finite* collection of well-defined activities and events (with no unspecified alternatives) which, when completed, mark the end of the project.

2) The activities may be started and stopped independently of each other within a given sequence. This requirement precludes the analysis of continuous flow processes.

3) The predecessor relationships among the activities and events must not contain cycles; that is there can be no predecessor chains implying that a job precedes itself. Thus a project is nonrepetitive. It is essentially a one-time effort such as a R&D task or a construction project.

2.4.23.2 Modules Called

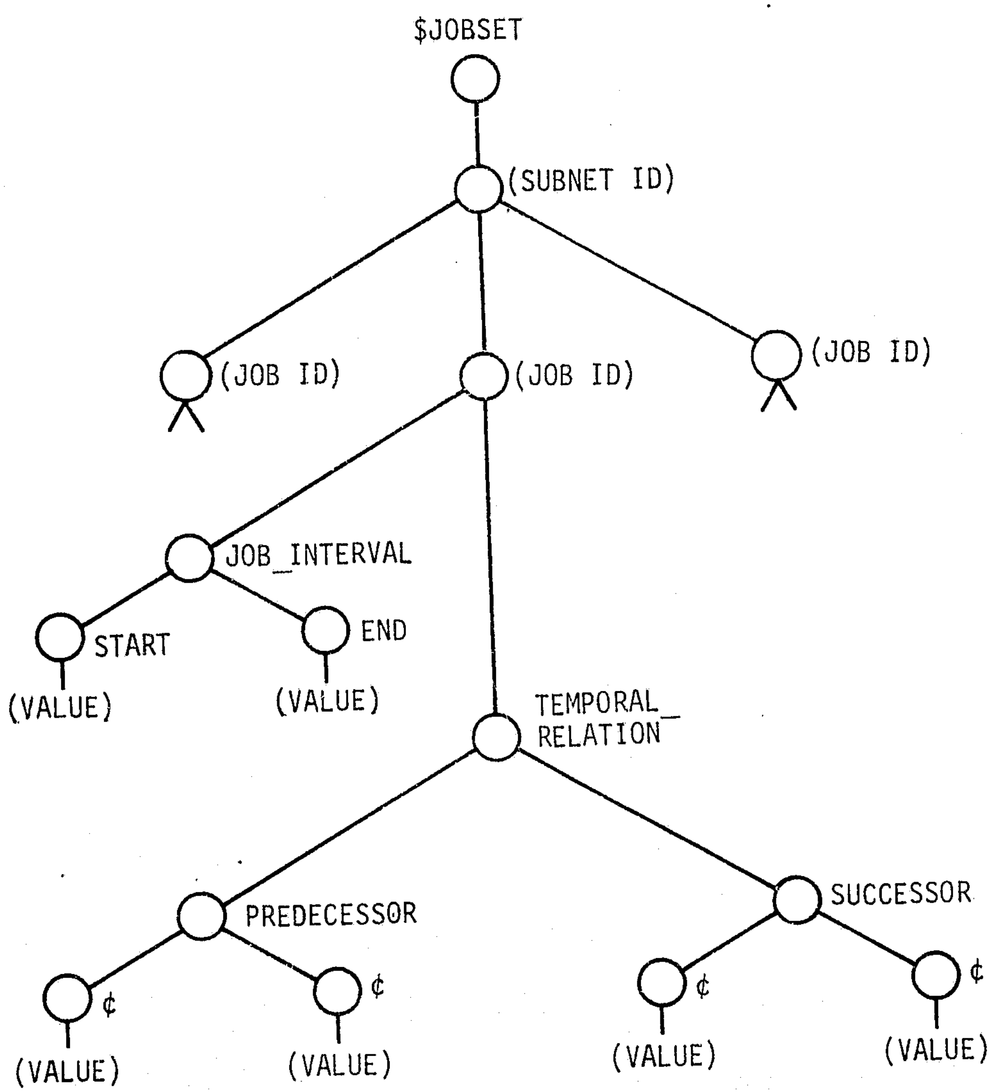
ORDER_BY_PREDECESSORS

FIND_MAX

FIND_MIN

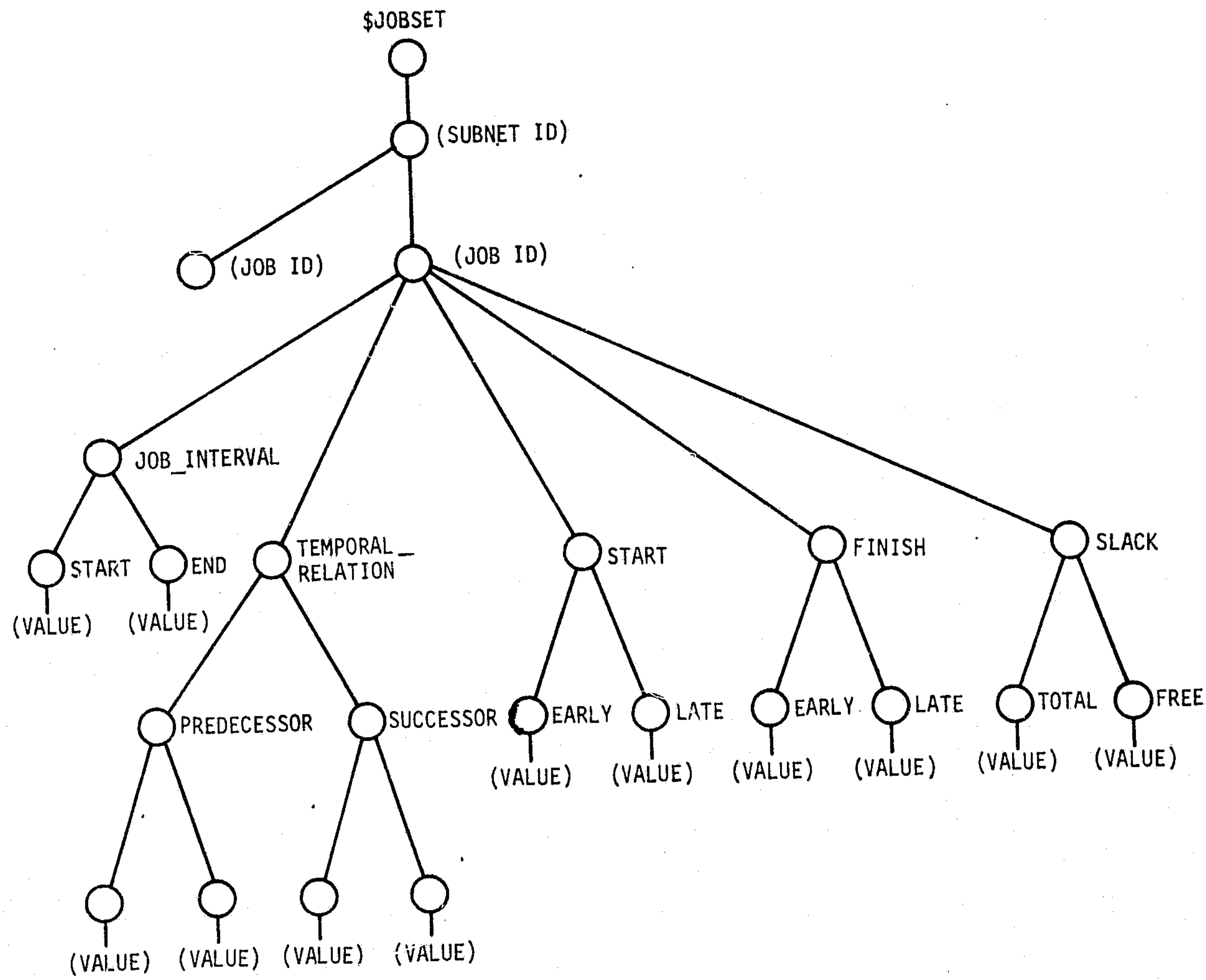
2.4.23.3 Module Input

Critical Path Input Data ($JOBSET)



2.4.23.4 Module Output

Critical Path Output Data ($JOBSET)



2.4.23.5 Functional Description

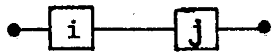
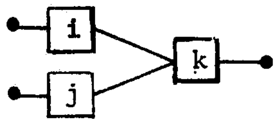
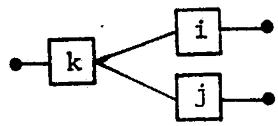
Critical path analysis is a powerful but simple technique for analyzing, planning, scheduling, and controlling complex projects. In essence, the method provides a means of determining (1) which activities are "critical" in their effect upon total project duration, and (2) how to schedule all activities to meet milestone dates.

Critical path analysis is based on the simple concept of predecessor/successor relationships between the activities and events defining the project network. A brief introduction to these fundamental scheduling concepts is presented below.

Let $\mathcal{A}$ = {i,j,k, ...} be a set of activities and events that must be completed to finish a project. Let the symbol "<<" denote the basic immediate predecessor relation. Thus the notation i<<j is interpretated to mean that activity i must be completed before activity j can start. If s_j denotes the start of activity j and f_i denotes the finish of activity i, then the relationship i<<j is equivalent to the standard inequality $s_j \geq f_i$. The set P_i = {j:j<<i} is said to be the *immediate predecessor set* of activity or event i. Similarly the set, $\mathcal{S}_i$ = {j:i<<j}, denotes the *immediate successor set* of the activity or event i.

A directed graph (network) is a useful topological representation of a project, and can provide valuable insight into many scheduling problems. A summary of predecessor/successor relationships in terms of their network representation is given in **Table 2.4.23-1. More general temporal relationships can be easily included within this simple framework by adding artificial activities.**

Table 2.4.23-1 Basic Precedence Relationship

Network Representation	Mathematical Representation
	$i \ll j,\ s_j \geq f_i,\ \mathbf{P}_j = \{i\},\ \mathcal{S}_i = \{j\}$
	$i \ll k,\ j \ll k,\ s_k \geq \max\{f_i, f_j\}$ $\mathbf{P}_k = \{i,j\},\ \mathcal{S}_i = \{k\} = \mathcal{S}_j$
	$k \ll i,\ k \ll j,\ s_i \geq f_k,\ s_j \geq f_k$ $\mathbf{P}_i = \mathbf{P}_j = \{k\},\ \mathcal{S}_k = \{i,j\}$

Suppose now that every activity in the project is started as soon as possible, that is, as soon as all of its predecessors are finished. It is then possible to calculate the early start of each activity as

[1] $$s_i^e = \max_{j \varepsilon P_i} \left\{ f_j^e \right\},$$

and the early finish of activity i is clearly

[2] $$f_i^e = s_i^e + d_i$$

where d_i is the duration of the ith activity ($d_i = 0$ for events). Similarly, the late finish for activity i is given by

[3] $$f_i^\ell = \min_{j \varepsilon \mathcal{S}_i} \left\{ s_j^\ell \right\},$$

and the late start is

[4] $$s_i^\ell = f_i^\ell - d_i.$$

For any activity, the quantity

[5] $$S_i = s_i^{\ell} - s_i^{e} = f_i^{\ell} - f_i^{e}$$

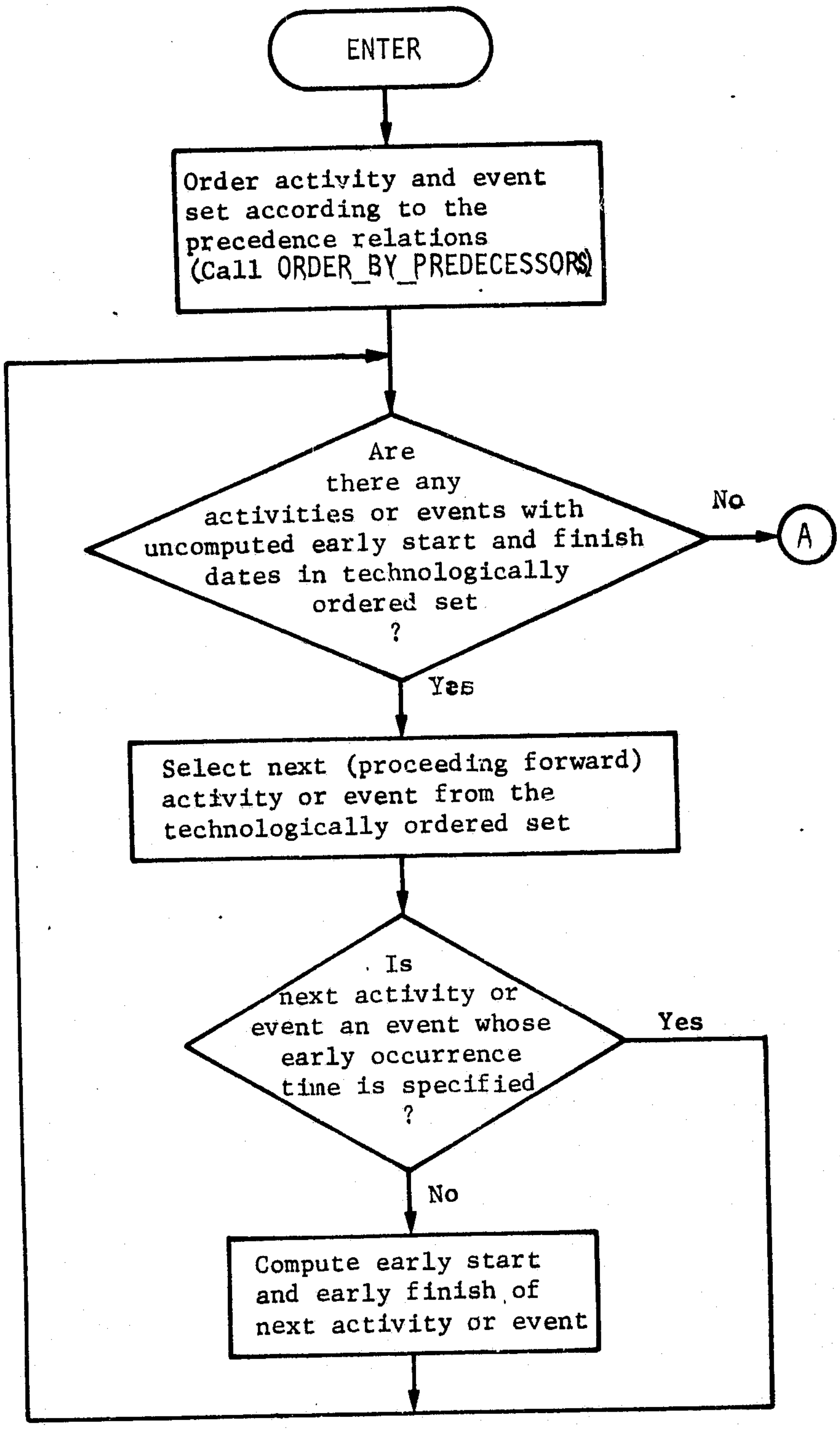
is defined to be the total slack. The set of critical activities is then the subset of activities having minimum total slack.

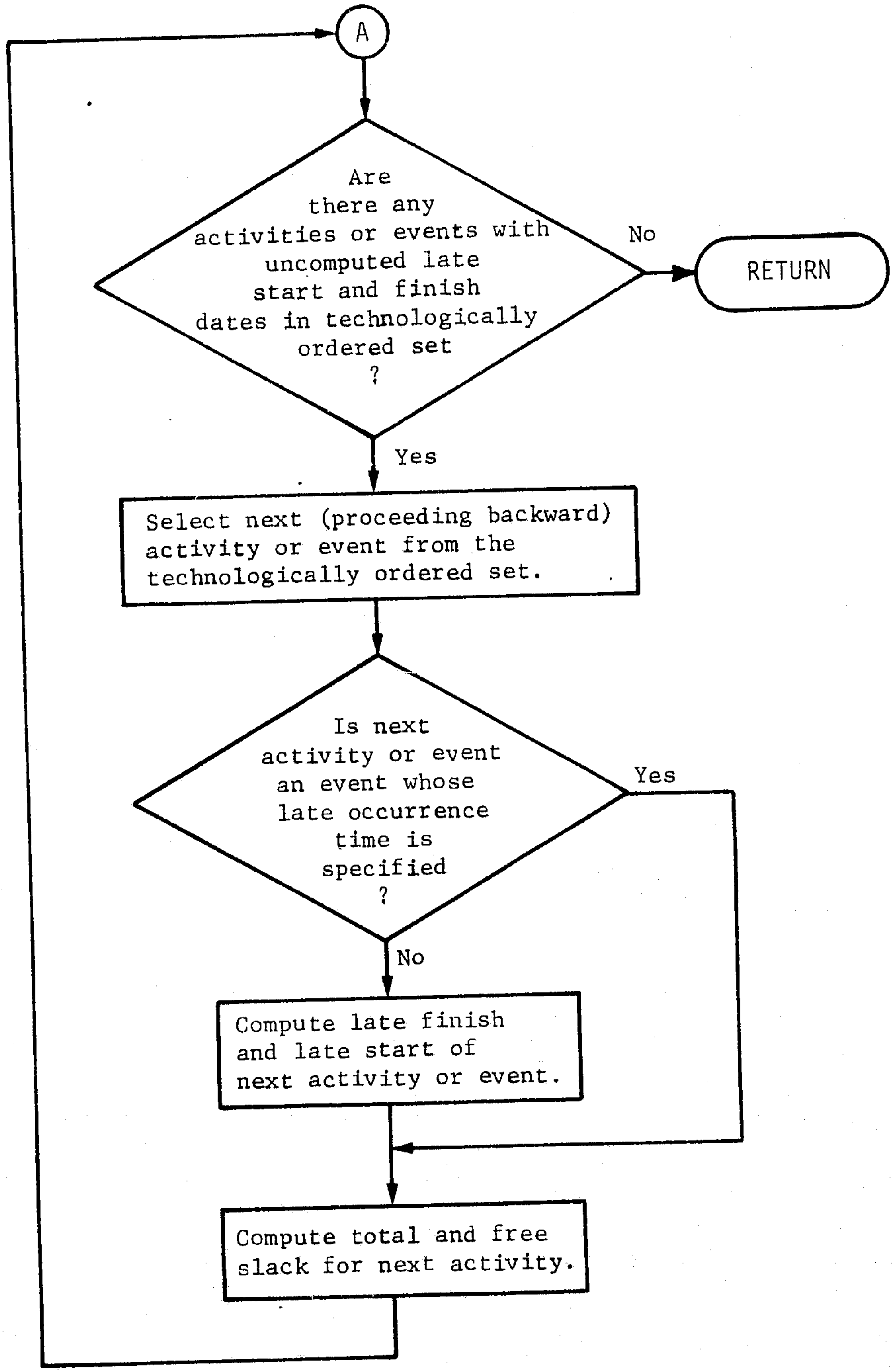
Another useful variable is free slack, S^f. Free slack is defined as the amount by which an activity may be delayed without affecting any other activity. It is computed as

[6] $$S_i^f = \min_{j \in \mathcal{S}_i} \left\{ s_j^e - f_i^e \right\}$$

The logic for the coordination of these calculations into an efficient computational procedure is given in the following block diagram.

2.4.23.6 Functional Block Diagram




A
Are there any activities or events with uncomputed late start and finish dates in technologically ordered set ?
No
RETURN
Yes
Select next (proceeding backward) activity or event from the technologically ordered set.
Is next activity or event an event whose late occurrence time is specified ?
Yes
No
Compute late finish and late start of next activity or event.
Compute total and free slack for next activity.

2.4.23.7 Typical Application

This module is used to compute the basic critical-path data under the direction of executive procedures such as the CRITICAL_PATH_PROCESSOR. The results are useful in manual and automatic heuristic scheduling as well as in project control.

2.4.23.8 Implementation Considerations

This module plays a fundamental role in any project management system. As a consequence, every effort should be made to ensure the computational efficiency of the algorithm. For example, it may well prove worthwhile to code the algorithm in assembly language to optimize execution efficiency. Further it may be advantageous to distinguish between activities and events by some other technique than merely noting the durations in the $JOBSET data structure. In general, the outline of the procedure presented here is intended only to specify the desired results of the module. Any modification to the suggested implementation that produces the same results more efficiently is to be preferred. Finally, it may be desirable to split the module into three submodules. The first would perform the forward pass calculations of early start and finish. The second would execute the backward pass computations of late start and finish. The third would carry out the slack calculations.

2.4.23.9 Reference

Kelley, J. E., Jr.: *Critical Path Scheduling: Mathematical Basis*, *Operations Research*, Volume 9, 1961.

Muth, John F. and Gerald L. Thompson: *Industrial Scheduling*, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1963.

2.4.23.10 Detailed Design

The functional block diagram provides the flowchart for this module. The individual activities and events (jobs) of the input subnetwork should be ordered according to precedence relations. This is accomplished by calling the module ORDER_BY_PREDECESSORS. Each activity or event in the ordered subnetwork is considered in turn and its early start and early finish times are calculated. The early start time of a job is equal to the latest early finish time of its predecessors. When early start and finish times have been determined, the activities and events in the ordered subnetwork are considered in their reverse order. Late start and finish times are then determined. The late finish time of a job is equal to the earliest start time of its successors. Also determined for each job are free and total slack. Free slack is the difference between the early finish of a job and the earliest start time of its successors. Total slack is the difference between the late start time and the early start time of the job.

2.4.23.11 Internal Variable and Tree Name Definitions

VAL_MAX	-	maximum of a set of values, returned from module FIND_MAX
VAL_MIN	-	minimum of a set of values, returned from module FIND_MIN
$FREE	-	tree formed with values representing free slack for each successor of a job. Actual free slack is the minimum of these values.

$SUBNET - The input subnet work after it has been technologically ordered by a call to ORDER_BY_PREDECESSORS

$INDICES - Output of modules FIND_MAX and FIND_MIN not used in this module

$PRED - Temporary single node tree, having only a label equal to the name of a predecessor of the job under consideration

$SUCC - Temporary single node tree, having only a label equal to the name of a successor of the job under consideration

$TEMP_PRED - Temporary tree containing values of early finish times of the predecessors of the job under consideration

$TEMP_SUCC - Temporary tree containing values of late start times of the successors of the job under consideration

2.4.23.12 Modifications to Functional Specifications and/or Standard Data Structures

Since only a single subnode of $JOBSET is considered with each call to this module, the subnode is identified as $UNORDERED_SUBNET through the parameter list. Additional first level subnodes of a job element (second level of $SUBNET, third level of $JOBSET) are added as shown in the module output structure. These subnodes are START (early and late), FINISH (early and late), and SLACK (total and free).

2.4.23.13 Commented Code

```
CRITICAL_PATH_CALCULATOR: PROCEDURE($UNORDERED_SUBNET,$ORDERED_SUBNET)
              OPTIONS(EXTERNAL);
DECLARE I,J,VAL_MAX,VAL_MIN,$PRED,$SUCC LOCAL;
DECLARE $TEMP_PRED,$TEMP_SUCC,$FREE,$DURATION,$INDICES,$SUBNET LOCAL ;
/*  ORDER ACTIVITY AND EVENT SET ACCORDING TO PRECEDENCE RELATIONS    */
CALL ORDER_BY_PREDECESSORS($UNORDERED_SUBNET,$SUBNET);
/*  SELECT NEXT (PROCEEDING FORWARD) ACTIVITY OR EVENT FROM THE       */
/*  TECHNOLOGICALLY ORDERED SET                                       */
DO I = 1 TO NUMBER($SUBNET);
/* COMPUTE EARLY START AND EARLY FINISH OF CURRENT ACTIVITY OR EVENT */
     PRUNE $TEMP_PRED;
     DO J = 1 TO NUMBER($SUBNET(I).TEMPORAL_RELATION.PREDECESSOR);
     $PRED = $SUBNET(I).TEMPORAL_RELATION.PREDECESSOR(J);
     $TEMP_PRED(NEXT) = $SUBNET.#($PRED).FINISH.EARLY;
     END;
     VAL_MAX = 0.;
     IF $TEMP_PRED IDENTICAL TO $NULL THEN;
          ELSE CALL FIND_MAX($TEMP_PRED,$INDICES,VAL_MAX);
/*  IS CURRENT ACTIVITY OR EVENT AN EVENT WHOSE EARLY OCCURRENCE      */
/*  TIME IS SPECIFIED?                                                */
IF $SUBNET(I).START.EARLY = ''
   THEN;
   ELSE IF VAL_MAX < $SUBNET(I).START.EARLY
      THEN VAL_MAX = $SUBNET(I).START.EARLY;
      ELSE;
      $SUBNET(I).START.EARLY = VAL_MAX;
      $SUBNET(I).FINISH.EARLY = VAL_MAX + $SUBNET(I).DURATION;
/*  ARE THERE ANY ACTIVITIES OR EVENTS WITH UNCOMPUTED EARLY START    */
/*  AND FINISH DATES IN TECHNOLOGICALLY ORDERED SET?                  */
END;
/*  SELECT NEXT (PROCEEDING BACKWARD)  ACTIVITY OR EVENT FROM THE     */
/*  TECHNOLOGICALLY ORDERED SET                                       */
DO I = NUMBER($SUBNET) TO 1 BY -1;
/*  COMPUTE LATE FINISH AND LATE START OF CURRENT ACTIVITY OR EVENT  */
      PRUNE $TEMP_SUCC;
      DO J = 1 TO NUMBER($SUBNET(I).TEMPORAL_RELATION.SUCCESSOR);
      $SUCC = $SUBNET(I).TEMPORAL_RELATION.SUCCESSOR(J);
      $TEMP_SUCC(NEXT) = $SUBNET.#($SUCC).START.LATE;
      $FREE(J) = $SUBNET.#($SUCC).START.EARLY - $SUBNET(I).
         FINISH.EARLY;
      END;
/*  IS CURRENT ACTIVITY OR EVENT AN EVENT WHOSE LATE OCCURRENCE TIME */
/*  IS SPECIFIED?                                                     */

      IF $TEMP_SUCC IDENTICAL TO $NULL
         THEN IF $SUBNET(I).FINISH.LATE = ''
            THEN $SUBNET(I).FINISH.LATE = $SUBNET(I).FINISH.EARLY;
            ELSE;
         ELSE DO;
              CALL FIND_MIN($TEMP_SUCC,$INDICES,VAL_MIN);
            IF ( $SUBNET(I).FINISH.LATE = '' | VAL_MIN < $SUBNET(I).
```

```
                FINISH.LATE )
             THEN $SUBNET(I).FINISH.LATE = VAL_MIN;
             ELSE;
          END;

      $SUBNET(I).START.LATE = $SUBNET(I).FINISH.LATE - $SUBNET(I).
            DURATION;
        DO J = 1 TO NUMBER($SUBNET(I).TEMPORAL_RELATION.SUCCESSOR);
      $SUCC = $SUBNET(I).TEMPORAL_RELATION.SUCCESSOR(J);
      $FREE(J) = $SUBNET.#($SUCC).START.EARLY - $SUBNET(I).
         FINISH.EARLY;
      END;
/*  COMPUTE TOTAL AND FREE SLACK FOR CURRENT ACTIVITY                      */
   IF $SUCC = '' THEN $FREE(FIRST) = 0;
    CALL FIND_MIN($FREE,$INDICES,VAL_MIN);
$SUBNET(I).SLACK.FREE = VAL_MIN;
$SUBNET(I).SLACK.TOTAL = $SUBNET(I).START.LATE - $SUBNET(I).START.
   EARLY;
/*  ARE THERE ANY ACTIVITIES OR EVENTS WITH UNCOMPUTED LATE START          */
/*  AND FINISH DATES IN TECHNOLOGICALLY ORDERED SET?                       */
END;
$ORDERED_SUBNET = $SUBNET;
END;     /*   CRITICAL_PATH_CALCULATOR   */
```

2.4.24 PREDECESSOR_SET_INVERTER

2.4.24 PREDECESSOR_SET_INVERTER

2.4.24.1 Purpose and Scope

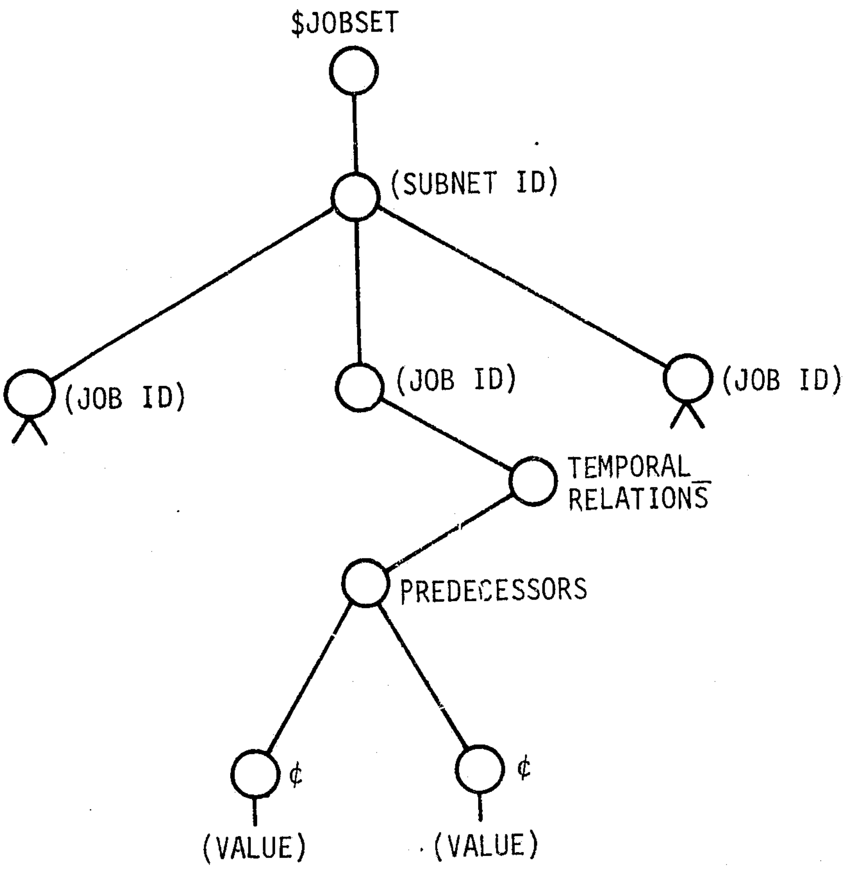
Given a set of activities and their respective predecessor sets, this module will form the respective successor sets. This inversion process is necessary for critical path computation. The project scheduling system assumes throughout that stating precedence relations in terms of predecessor sets is more natural than expressing them as successor sets. For this reason the user is asked to define all subnetwork topology in terms of predecessor sets in the input data structure $JOBSET.

2.4.24.2 Modules Called

None

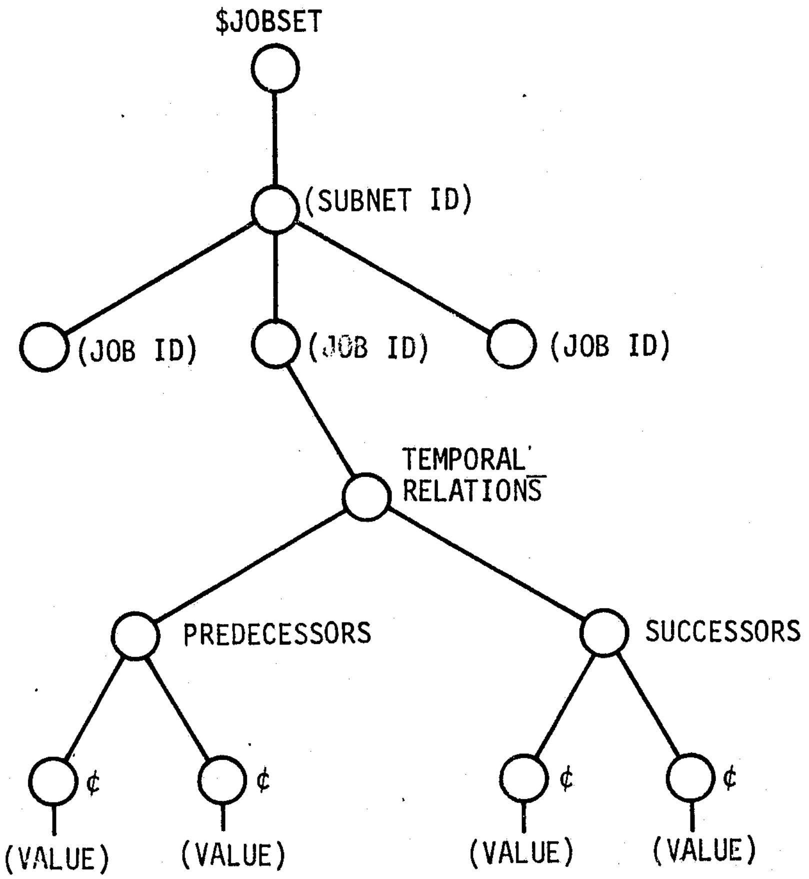
2.4.24.3 Module Input

Network definition ($JOBSET)- The substructures of the tree beginning at the nodes labeled SUCCESSORS are null upon input to the module.



2.4.24.4 Module Output

Redundant network definition ($JOBSET) - The substructures of the tree beginning at the nodes labeled SUCCESSORS are complete upon exit from the module.

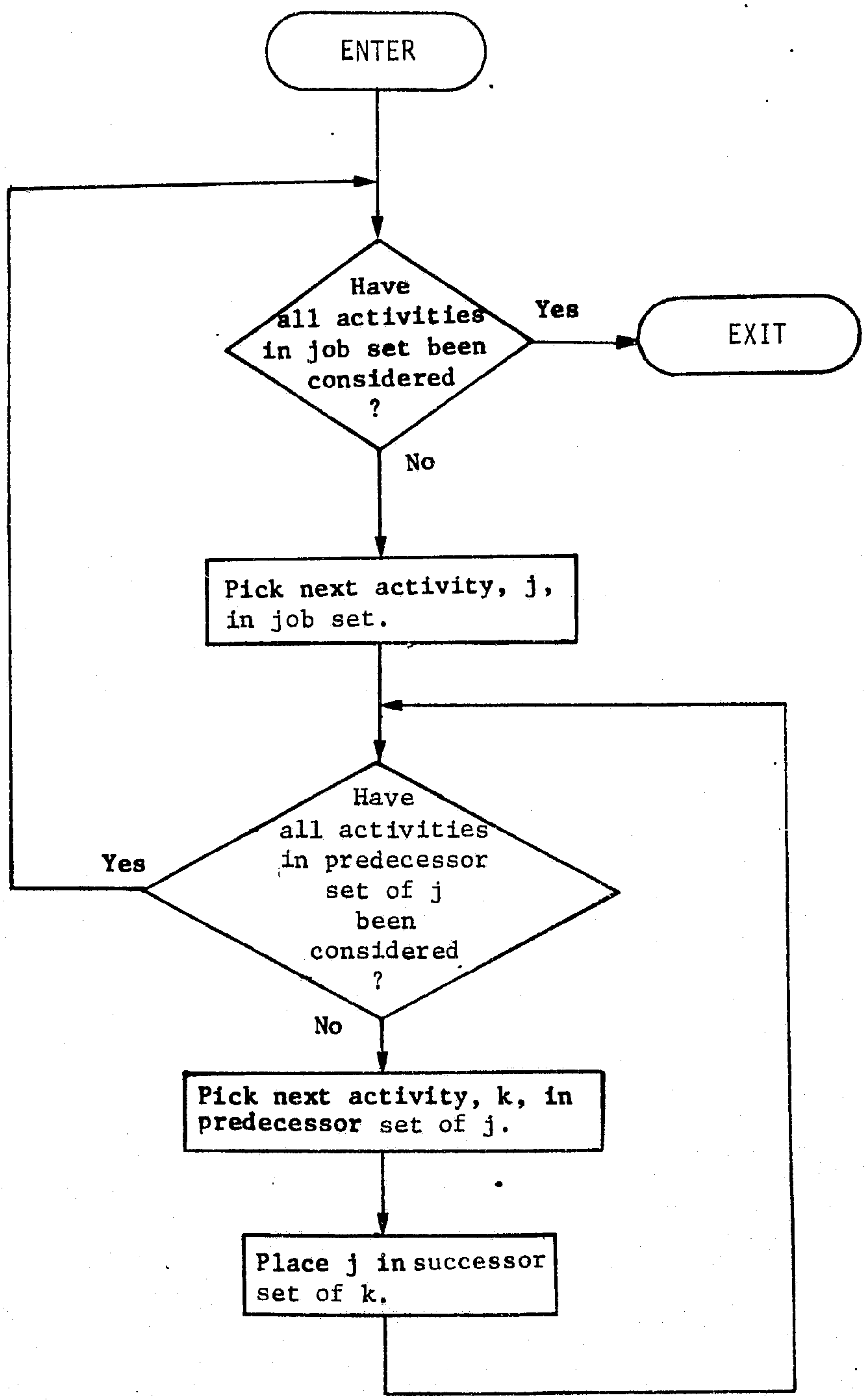


2.4.24.5 Functional Description

The logic of the inversion process from predecessor sets is simple and direct. Each activity in the job set is considered in turn. Whenever a given activity is found in the predecessor set of another, the latter is included in the successor set of the former. When all of the predecessor sets of all of the jobs have been examined, the collection of successor sets is complete. The following block diagram illustrates this straightforward yet efficient logic.

2.4.24.6 Functional Block Diagram

PREDECESSOR_SET_INVERTER



2.4.24.7 Typical Application

The module can be applied wherever successor sets rather than user input predecessor sets are required. This includes the modules CRITICAL_PATH_CALCULATOR.

2.4.24.8 Detailed Design

The functional block diagram provides the flow chart for this module. The module loops on subnet - i.e., job name, and predecessors. For each predecessor job, the name of the current job is inserted as a successor.

2.4.24.9 Internal Variable and Tree Names

$JOB_ID - Identifier for each subnode of $SUBNET

$SUBNET - Identifier for each subnode of $JOBSET

$TEMP_PRED - Identifier for each subnode of PREDECESSOR node

2.4.24.10 Modifications to Functional Specifications, and/or Standard Data Structures Assume

None

2.4.24.11 Commented Code

```
PREDECESSOR_SET_INVERTER: PROCEDURE($JOBSET)
               OPTIONS(EXTERNAL);
DECLARE $SUBNET,$JOB_ID,$TEMP_PRED LOCAL;
DO FOR ALL SUBNODES OF $JOBSET USING $SUBNET;
   DO FOR ALL SUBNODES OF $SUBNET USING $JOB_ID;
      DO FOR ALL SUBNODES OF $JOB_ID.TEMPORAL_RELATION.PREDECESSOR
            USING $TEMP_PRED;
      IF LABEL($JOB_ID) ¬ELEMENT OF $SUBNET.#($TEMP_PRED).
            TEMPORAL_RELATION.SUCCESSOR THEN
      $SUBNET.#($TEMP_PRED).TEMPORAL_RELATION.SUCCESSOR(NEXT) =
            LABEL($JOB_ID);
      END;
   END;
END;

END;  /*  PREDECESSOR_SET_INVERTER    */
```

2.4.25 NETWORK_CONDENSER

2.4.25 NETWORK_CONDENSER

2.4.25.1 Purpose and Scope

Condensing a network is the process of eliminating activities from the network leaving only events, as nodes, linked by delays, as branches. These delays are simply the maximum sum of activity durations along any path leading directly from one event to another. Condensation is useful in two contexts: (1) integrating subnetworks into master networks and (2) summarizing networks for management review. This module will perform such a condensation on a specified network.

The condensed event node network can be defined precisely in terms of the original network from which it is derived. The condensed network contains, as nodes, precisely those nodes that were events in the original network. A pair of nodes is linked by a branch wherever there is possible path in the original network between the respective events, which does not contain any other event. A critical delay, defined to be the longest path in the original network between the two events and passing through no third event, is assigned to each such branch in the condensed network.

The availability of this critical delay figure between any two **connected** events facilitates the merging of subnetworks into a master network. This integration facility is essential for any practical multilevel project analysis. The condensed version of any two networks having interfaces events can be merged together into a composite condensed network as follows. For each pair of interface nodes, which are linked in both condensed subnetworks, replace the critical delay on the resulting branch in the condensed composite network by the maximum of the respective values for the subnetworks. All other branches and critical dealys in either of the two condensed subnetworks are simply transcribed into the composite condensed network.

The critical path data can then be readily calculated from the composite condensed network. Each of the branches is treated as an activity with a duration equal to its critical delay. The early and late occurrence dates and the free and total float of each event can then be computed in the usual manner. Once critical path data for the events in the master condensed network have been calculated, they can be substituted back into the original subnetworks to determine the corresponding data for the original activities.

Thus, the condensation and merging processes make it possible to logically segment a project into tractable subnetworks of successively higher levels of detail so that the entire project, no matter what its size, can be viewed as one comprehensible summarized network. Without this capability network analysis would be of little value to project scheduling.

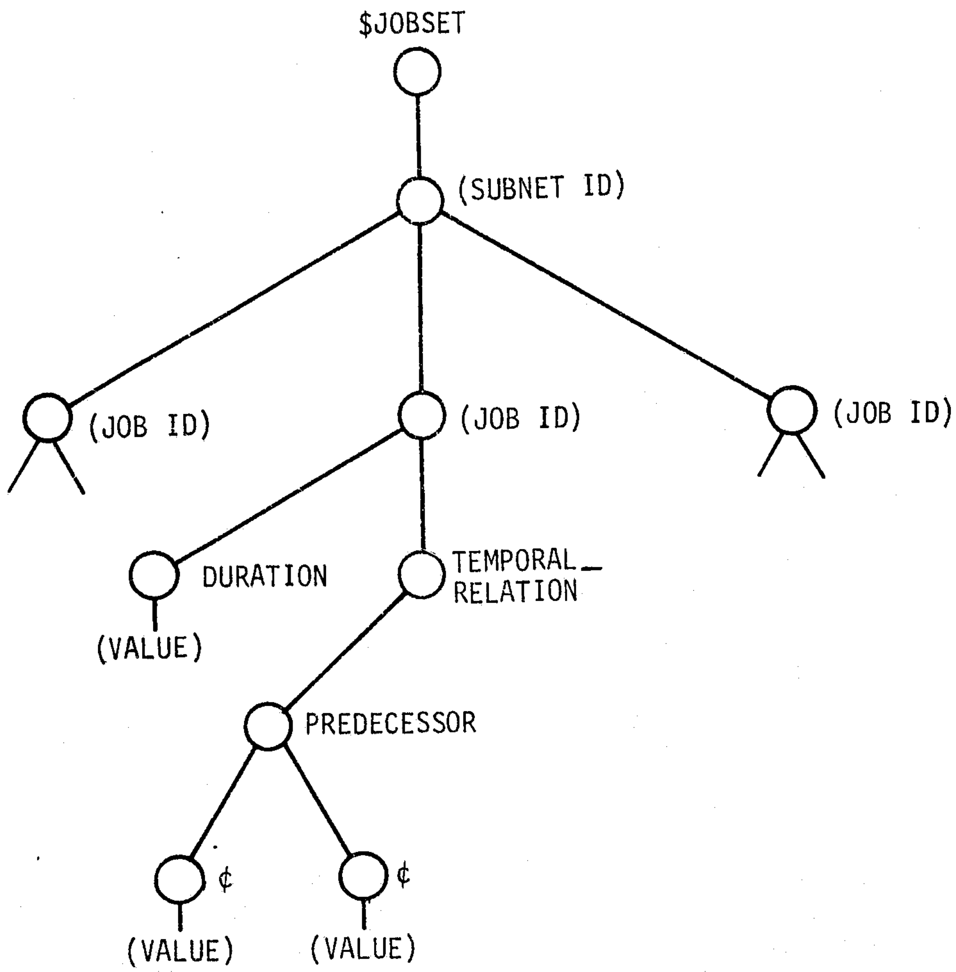
The purpose of this module is then to convert a network, specified in terms of a jobset with its corresponding family of predecessor sets and durations, into a condensed network defined by its event and pseudo-activity set with its corresponding collection of predecessor sets and durations.

2.4.25.2 Modules Called

None

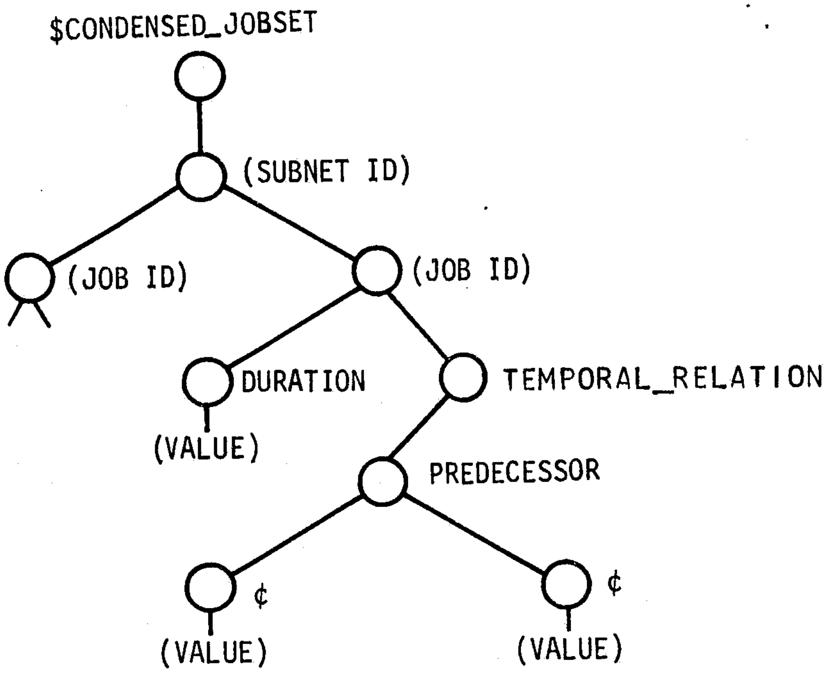
2.4.25.3 Module Input

Critical Path Input Data ($JOBSET)



2.4.25.4 Module Output

Tree Defining the Condensed network

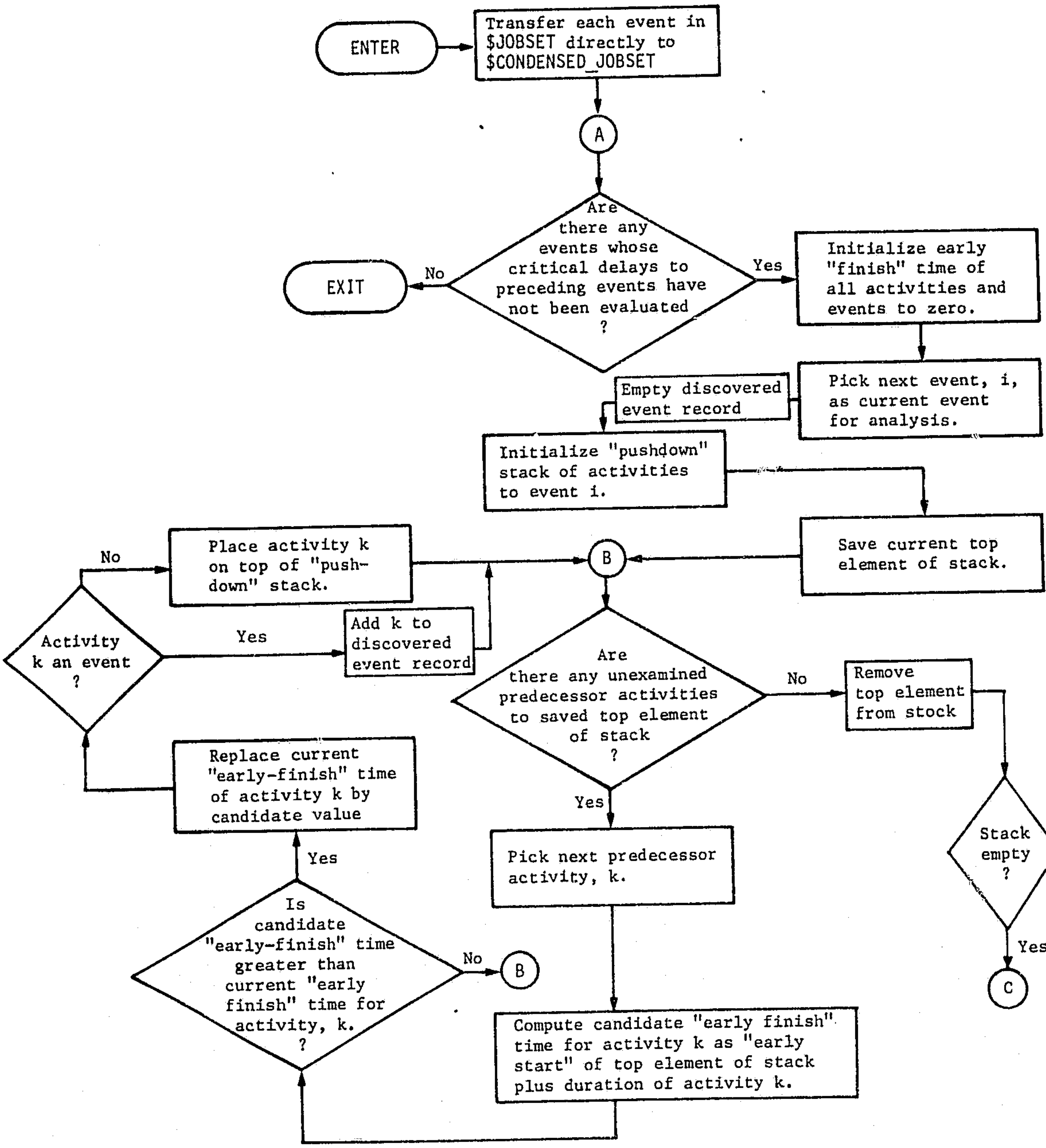


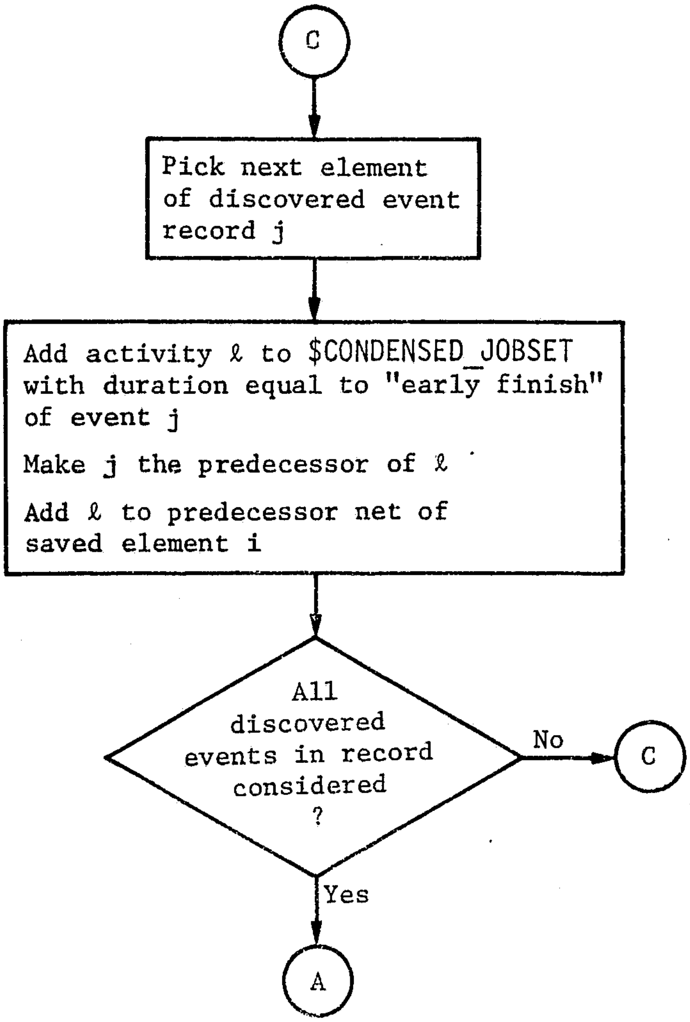
2.4.25.5 Functional Description

The problem of finding the critical delay between any pair of events is simply that of finding the longest directed path between two nodes in a network not passing through any third node. Because the critical delays between all directly connected events are desired, the following approach suggests itself. Consider each event in turn. Step by step, examine all possible paths that terminate at the current event under analysis. All branches of any path must be investigated and for this reason a "pushdown" stack is useful in recalling which alternatives remain unexamined. A path is eliminated from further consideration when it reaches an event or merges with some other path of greater length. Since the topology of the condensed networks are specified in terms of precedence sets rather than successor sets, it is convenient to proceed along the activity paths in reverse order to activity performance.

The macrologic of the module requires a few further words of explanation. First, when an event is transferred from the input tree $JOBSET to the output tree $CONDENSED_JOBSET, its predecessors are omitted and its duration is maintained at zero. Second, when candidate early start and finish times are computed, the calculations are performed as though the activities and events proceeded backward in time. This point of view is adopted to avoid the costly process of inverting the predecessor sets to obtain successor sets. Finally, the details of inserting a pseudo-activity into the output tree $CONDENSED_JOBSET are described. If pseudo-activity l represents a critical delay originating at event i and terminating at event j, then the pseudo-activity should be listed as a predecessor of event j and event i should be listed as a predecessor of pseudo-activity ℓ. The duration of the pseudo-activity is simply the critical delay between events i and j (that is, the early start of event i computed with respect to event j).

2.4.25.6 Functional Block Diagram




C
Pick next element of discovered event record j
Add activity ℓ to $CONDENSED_JOBSET with duration equal to "early finish" of event j
Make j the predecessor of ℓ
Add ℓ to predecessor net of saved element i
All discovered events in record considered ?
No
C
Yes
A

2.4.25.7 Typical Application

Network condensation has two basic applications. The first and most important is to facilitate integration of subnetworks into master networks. Condensed subnetworks with complex interfacing events can be merged into a simple composite condensed network equivalent to the complete composite network for critical path calculations concerning events. The condensed composite network is substantially smaller in terms of node and branch counts making critical path analysis feasible in high-speed memory. The complete composite network, on the other hand, is frequently too large to permit analysis without the use of slower mass storage. Furthermore, the relative execution times of small and large networks are such that critical path data can be generated more rapidly by treating a condensed master network and using the resulting event early occurrence times to solve the complete subnetworks than by directly solving the complete master network.

The second use of network condensation is in summarizing a complex network for top-level review. The webs of jobs connecting milestone events are replaced by simple sets of pseudo-jobs--one for each direct path between any two events. Thus, the criticality of the respective events as well as their interdependence is made more transparent.

2.4.25.8 References

Burman, P. J., *Precedence Networks for Project Planning and Control*, McGraw Hill, London, 1972.

2.4.25.9 Detailed Design

The functional block diagram provides the flow chart for this module. The module first examines each job element to determine whether the element is an activity or an event. An event has a duration of zero. Each event of the subnetwork is placed in the tree $CONDENSED_JOBSET, without transferring its predecessors. The early "finish" times of all activities and events are initialized to zero. The next event, i, is selected from $CONDENSED_JOBSET for analysis and its label is placed in the stack. Each predecessor of event i is used to call the recursive procedure, CHECK_DURA, in order to develop the critical delay to this event. The duration of the predecessor is added to the early "finish" of the event, and this value is compared with the early "finish" of the predecessor. If the calculated early "finish" time is greater than that of the predecessor, it replaces that of the predecessor. If the predecessor is an event (duration equal to zero) it is added to the stack of discovered events, otherwise, it is added to the stack to be examined. If the current predecessor has predecessors, each in turn is used to call the recursive procedure. When the stack to be examined is finally emptied, i.e. all jobs have been examined, the substructure of $CONDENSED_JOBSET is completed.

2.4.25.10 Internal Variable and Tree Name Definitions

DURA	-	Duration of the correct predecessor being examined in CHECK_DURA, to simplify calling
EARLY_FINISH	-	Value of early finish of event under examination plus duration of its current predecessor

$DISCOVERED_EVENT - Stack of events discovered in checking predecessors.

$JOB - Identifier for each subnode of $SUBNET

$JOB_NAME - Identifier for each subnode of $CONDENSED_SUBNET

L - Counter used as the label for the identified delays

$NEXT_EVENT - Identifier for each subnode of $DISCOVERED_EVENT

$PRED - Identifier for each subnode of the PREDECESSOR node

$PRED_NAME - Name of the events predecessor used in the recursive procedure CHECK_DURA

$SAVE_ELEMENT - Name of current event under examination

$STACK - List of jobs to be examined

2.4.25.11 Modification to Functional Specifications and/or Standard Data Structures Assumed

Since only one subnetwork is used to call this module, $SUBNET is the name used in the parameter list of the module. In order to maintain unique labels for the identified delays, the counter, L, is now an input parameter.

2.4.25.12 <u>Commented Code</u>

```
NETWORK_CONDENSER:  PROCEDURE(L,$SUBNET,$CONDENSED_SUBNET)
                    OPTIONS(EXTERNAL);
DECLARE L,$JOB,$JOB_NAME,$STACK,$SAVE_ELEMENT,$PRED LOCAL;
DECLARE $DISCOVERED_EVENT LOCAL;
/*  TRANSFER EACH EVENT IN $JOBSET TO $CONDENSED_JOBSET WITHOUT          */
/*   TRANSFERRING PREDECESSORS                                             */
DO FOR ALL SUBNODES OF $SUBNET USING $JOB;
   IF $JOB.DURATION IDENTICAL TO $NULL
      THEN IF $JOB.JOB_INTERVAL IDENTICAL TO $NULL
         THEN DO;
            WRITE 'NEITHER_DURATION_NOR_JOB_INTERVAL_INPUT';
            WRITE LABEL($SUBNET);
            WRITE LABEL($JOB);
            RETURN;
            END;
         ELSE $JOB.DURATION = $JOB.JOB_INTERVAL.END - $JOB.
               JOB_INTERVAL.START;
      ELSE;
   IF $JOB.DURATION = 0 THEN DO;
      LABEL($CONDENSED_SUBNET(NEXT)) = LABEL($JOB);
      $CONDENSED_SUBNET(LAST).DURATION = $JOB.DURATION;
      END;
   ELSE;
END;
DO FOR ALL SUBNODES OF $CONDENSED_SUBNET USING $JOB_NAME;
/*  INITIALIZE EARLY FINISH TIMES OF ALL JOBS TO ZERO                     */
   DO FOR ALL SUBNODES OF $SUBNET USING $JOB;
   $JOB.EARLY_FINISH = 0;
   END;
PRUNE $DISCOVERED_EVENT;
INSERT LABEL($JOB_NAME) BEFORE $STACK(FIRST);
$SAVE_ELEMENT = $STACK(FIRST);
POINT_B:
/*  PICK NEXT PREDECESSOR ACTIVITY, K                                      */
   DO FOR ALL SUBNODES OF $SUBNET.#($STACK(FIRST)).TEMPORAL_RELATION.
         PREDECESSOR USING $PRED;
/*  COMPUTE CANDIDATE 'EARLY FINISH' TIME FOR ACTIVITY, K                 */
   CALL CHECK_DURA($PRED);

   CHECK_DURA:  PROCEDURE($PRED_NAME)  RECURSIVE;
   DECLARE DURA, EARLY_FINISH,$NEXT_EVENT,$PRED_NAME,$TEMP_PRED LOCAL;
   DURA = $SUBNET.#($PRED_NAME).DURATION;
   EARLY_FINISH = $SUBNET.#($STACK(FIRST)).EARLY_FINISH + DURA;
/*  IS CANDIDATE 'EARLY FINISH' TIME GREATER THAN CURRENT 'EARLY         */
/*   FINISH' TIME FOR ACTIVITY K?                                          */
   IF EARLY_FINISH > $SUBNET.#($PRED_NAME).EARLY_FINISH
      THEN DO;
/*  REPLACE CURRENT 'EARLY FINISH' TIME OF ACTIVITY K BY CANDIDATE        */
         $SUBNET.#($PRED_NAME).EARLY_FINISH = EARLY_FINISH;
/*  IS K AN EVENT?                                                         */
         IF DURA = 0
```

```
/*  ADD K TO DISCOVERED EVENT RECORD                                              */
          THEN IF $PRED_NAME NOT ELEMENT OF $DISCOVERED_EVENT
             THEN DO;
                $DISCOVERED_EVENT(NEXT) = $PRED_NAME;
                GO TO POINT_B;
                END;
             ELSE;
          ELSE DO;
             INSERT $PRED_NAME BEFORE $STACK(FIRST);
             GO TO POINT_B;
             END;
       END;
    ELSE;
          END;      /*   CHECK_DURA     */
  END;
/* REMOVE TOP ELEMENT FROM STACK                                                  */
          PRUNE $STACK(FIRST);
          IF $STACK IDENTICAL TO $NULL
/*  PICK NEXT ELEMENT OF DISCOVERED EVENT RECORD J                                */
             THEN DO FOR ALL SUBNODES OF $DISCOVERED_EVENT USING
                   $NEXT_EVENT;
/*  ADD ACTIVITY L TO $CONDENSED_JOBSET WITH DURATION EQUAL TO                    */
/*   'EARLY FINISH' OF EVENT J                                                    */
                LABEL($CONDENSED_SUBNET(NEXT)) = L;
                $CONDENSED_SUBNET(LAST).DURATION = $SUBNET.
                      #($NEXT_EVENT).EARLY_FINISH;
/*  MAKE J THE PREDECESSOR OF L                                                   */
                $CONDENSED_SUBNET(LAST).TEMPORAL_RELATION.PREDECESSOR
                      (NEXT) = $NEXT_EVENT;
/*  ADD L TO PREDECESSOR SET OF I                                                 */
                $CONDENSED_SUBNET.#($SAVE_ELEMENT).TEMPORAL_RELATION.
                      PREDECESSOR(NEXT) = L;
                L = L + 1;
                END;
             ELSE GO TO POINT_B;

END;
DO FOR ALL SUBNODES OF $SUBNET USING $JOB;
PRUNE $JOB.EARLY_FINISH;
IF $JOB.JOB_INTERVAL ¬ IDENTICAL TO $NULL
   THEN PRUNE $JOB.DURATION;
END;
END;   /*   NETWORK_CONDENSER   */
```

2.4.26 CONDENSED_NETWORK_MERGER

2.4.26 CONDENSED_NETWORK_MERGER

2.4.26.1 Purpose and Scope

This module will merge two condensed subnetworks into a composite condensed network. This process is essential in merging subnetworks into a self-contained master network.

2.4.26.2 Modules Called

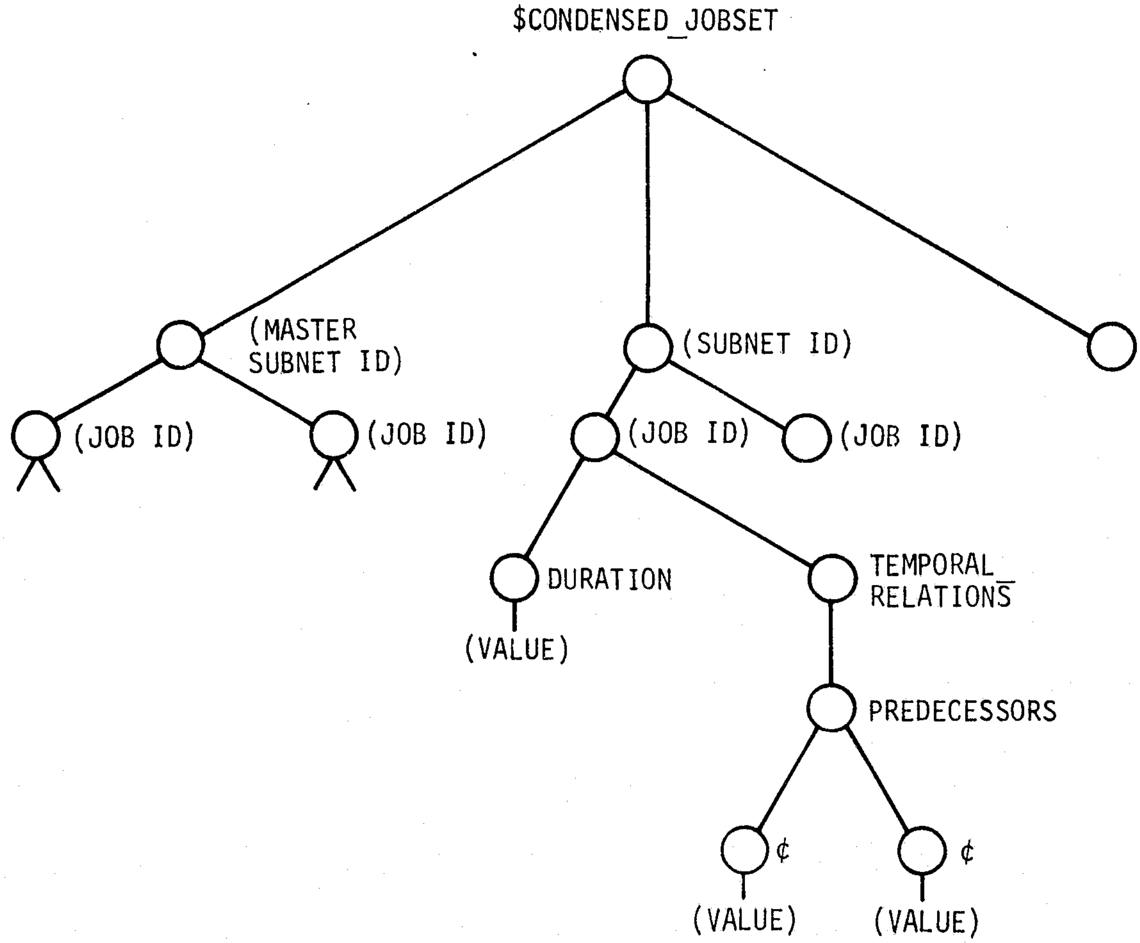
SET_INTERSECTION

SET_UNION

NETWORK_CONDENSER

2.4.26.3 Module Input

Critical path data for condensed subnetwork and condensed master subnetworks $CONDENSED_JOBSET



2.4.26.4 Module Output

Critical path input data for merged network contained under master subnetworks node of $CONDENSED_JOBSET.

2.4.26.5 Functional Description

The object of this module although specialized is critical to effective critical path analysis by subnetworks. By applying this module in conjunction with the NETWORK_CONDENSER module, the CRITICAL_PATH_PROCESSOR is able to assemble a self-contained condensed network. By applying the CRITICAL_PATH_CALCULATOR to the resulting network the critical-path data on the interfacing events is obtained. These data are then substituted back into the original subnetworks to obtain the critical path figures for their respective activities.

What is required of the merging routine in the context of the CRITICAL_PATH_PROCESSOR is the capability of combining two critical path input data structures for two condensed networks to yield a resultant structure representing the composite condensed network. The rules for performing the merger follow directly from the definition of a condensed network. Recall that a condensed network consists of all of the events of the original network connected by activities representing critical delays. Two events are connected by such an activity when there is at least one path between them in the original network that contains no third event. The durations along all paths directly connecting the two activities. The rules for merging two condensed networks are as follows:

1) Merge events. Let E_i denote the set of events in condensed network N_i. Then if N_j and N_k are the condensed networks to be merged the event set E of the merged network is simply

   $E = E_j \cup E_k$

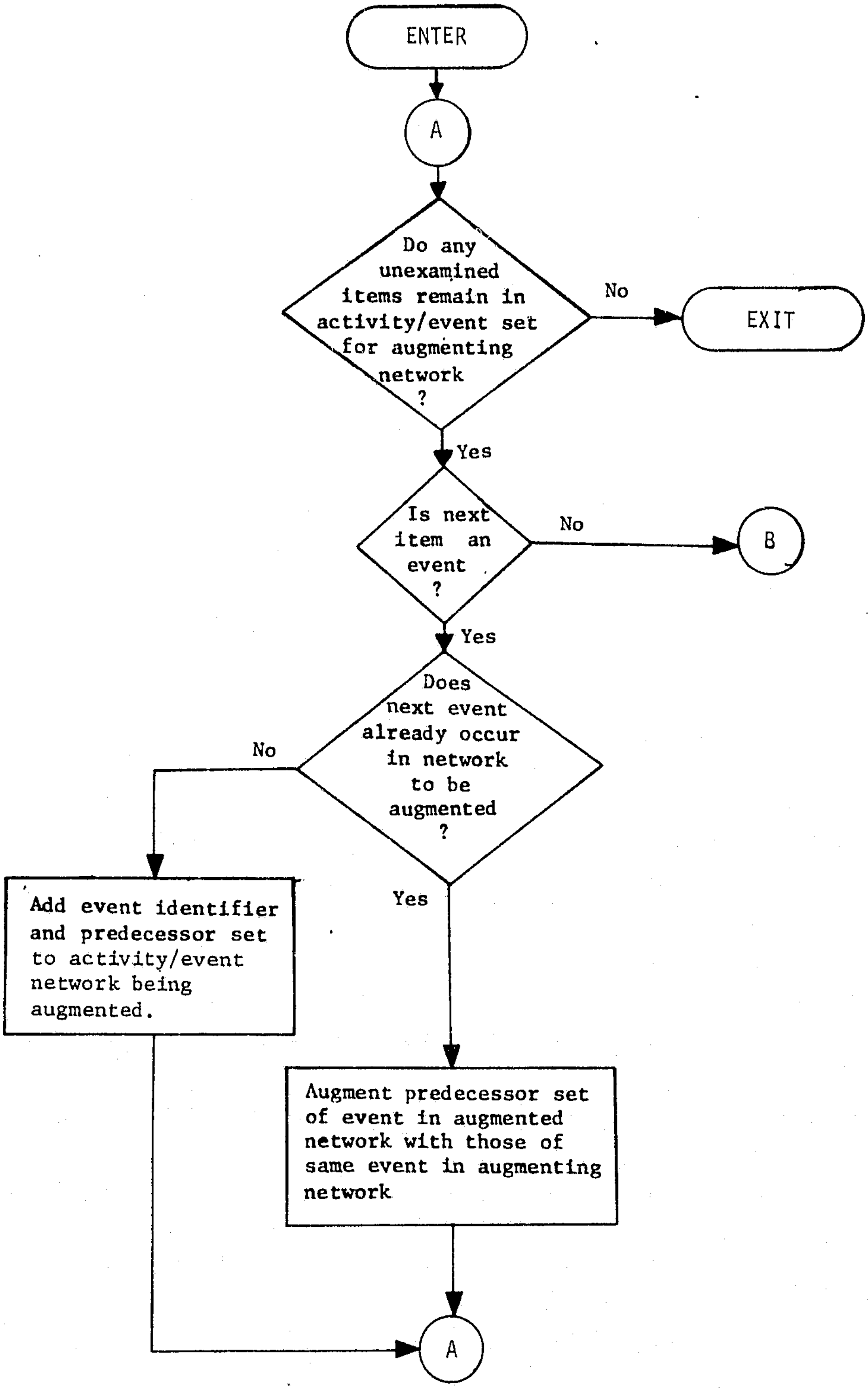
2) Merge Pseudo-Activities. Let e_m and e_n be two interface events common to both N_j and N_k, then the delay between these events in the merged network is calculated as

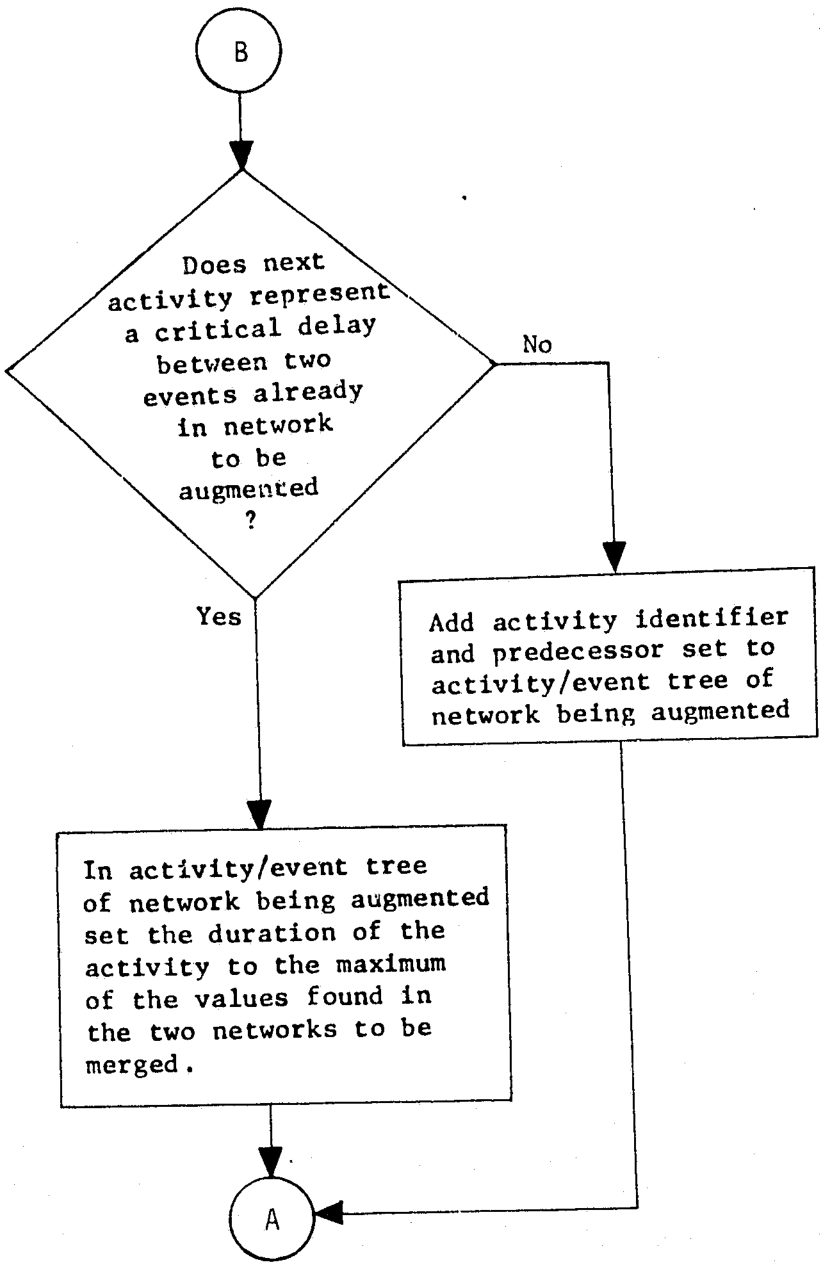
$$d\left(e_m, e_n\right) = \max \left\{d_j\left(e_m, e_n\right), d_k\left(e_m, e_n\right)\right\}$$

where $d_i\left(e_m, e_n\right)$ is the delay calculated in N_i.

The remaining pseudo-activities defining the delay between non-interfacing events can be added directly to the merged network.

2.4.26.6 Functional Block Diagram




B
Does next activity represent a critical delay between two events already in network to be augmented ?
No
Yes
Add activity identifier and predecessor set to activity/event tree of network being augmented
In activity/event tree of network being augmented set the duration of the activity to the maximum of the values found in the two networks to be merged.
A

2.4.26.7 Typical Applications

The sole application of this module is in the recursive construction of a master network from a specified master subnetwork and all of its interfacing subnetworks as directed by the executive procedure CRITICAL_PATH_PROCESSOR and supported by the procedure NETWORK_CONDENSER.

2.4.26.8 Reference

Burman, P. J., *Precedence Networks for Project Planning and Control*, McGraw Hill, London, 1972.

IBM, Project Management System IV, *Network Processor Program Description and Operations Manual*, Publication SH20-0899-1, 1972.

2.4.26.9 Detailed Design

The functional block diagram may be used as a flow chart for this module. Since one condensed network is to be merged with another, it is desirable that the resulting network have the same name as one of the initial two. In this case B will be merged into A, and returned as A. In order to preserve the initial A for operations, it will be duplicated as C and C will be used to build the merged network. Each job of network B is examined. If the job is an event and is not included in network A, the event is added to network C, otherwise the predecessor set of A is augmented by the predecessor set of B and placed in C. If the job is an activity and is a critical delay between events of A, the greater duration of the activity is included in C. If the activity is not a critical delay in A, the activity is added to C. When all jobs of B have been examined, network A is replaced by network C, the merged network.

2.4.26.10 Internal Variable and Tree Name Definitions

$NAME_A - tree whose single level substructure contains the labels of the jobs in network A.

$NAME_B - tree whose single level substructure contains the labels of the jobs in network B.

$PRED - temporary storage for a predecessor name, for ease in referencing.

$SUCC - temporary storage for a successor name, for ease in referencing.

2.4.26.11 Modifications to Functional Specifications and/or Standard Data Structures Assumed

The subnetworks A and B of $JOBSET are identified as $SUBNET_A and $SUBNET_B in the calling parameter list.

2.4.26.12 Commented Code

```
CONDENSED_NETWORK_MERGER: PROCEDURE($SUBNET_A,$SUBNET_B)
                          OPTIONS(EXTERNAL);

DECLARE NUMBER_DELAY LOCAL;
DECLARE $SUBNET_D LOCAL;
DECLARE $NAME_A,$NAME_B,$PRED,$SUCC,$SUBNET_C LOCAL;
DECLARE I,J,$INTERSECT,$UNION,$B LOCAL;

$SUBNET_C = $SUBNET_A;
DO I = 1 TO NUMBER($SUBNET_A);
   $NAME_A(I) = LABEL($SUBNET_A(I));
   END;
DO I = 1 TO NUMBER($SUBNET_B);
   $NAME_B(I) = LABEL($SUBNET_B(I));
   END;
DO I = 1 TO NUMBER($NAME_B);
   IF $SUBNET_B(I).DURATION IDENTICAL TO $NULL
      THEN IF $SUBNET_B(I).JOB_INTERVAL IDENTICAL TO $NULL
         THEN DO;
            WRITE 'NEITHER_DURATION_NOR_JOB_INTERVAL_INPUT',
               LABEL($SUBNET_B).LABEL($SUBNET_B(I));
               RETURN;
               END;
            ELSE $SUBNET_B(I).DURATION = $SUBNET_B(I).JOB_INTERVAL.END
                 - $SUBNET_B(I).JOB_INTERVAL.START;
      ELSE;
   IF $SUBNET_B(I).DURATION = 0
      THEN DO;
         PRUNE $B;
         $B(FIRST) = $NAME_B(I);
         CALL SETINTERSECTION($B,$NAME_A,$INTERSECT);
         IF $INTERSECT IDENTICAL TO $NULL
            THEN $SUBNET_C(NEXT) = $SUBNET_B(I);
            ELSE DO;
               CALL SETUNION($SUBNET_B(I).TEMPORAL_RELATION.
                  PREDECESSOR,$SUBNET_A.#($NAME_B(I)).
                  TEMPORAL_RELATION.PREDECESSOR,$UNION);
               GRAFT $UNION AT $SUBNET_C.#($NAME_B(I)).
                  TEMPORAL_RELATION.PREDECESSOR;
               END;
         END;
      ELSE DO FOR ALL SUBNODES OF $SUBNET_B(I).TEMPORAL_RELATION.
            PREDECESSOR USING $PRED;
         DO J = 1 TO NUMBER($SUBNET_B);
         IF $NAME_B(I) ELEMENT OF $SUBNET_B(J).TEMPORAL_RELATION.
                                                       PREDECESSOR
            THEN $SUCC = LABEL($SUBNET_B(J));
         END;
         IF ($PRED ELEMENT OF $SUBNET_A.#($NAME_B(I)).TEMPORAL_RELATION
            .PREDECESSOR & $NAME_B(I) ELEMENT OF $SUBNET_A.#($SUCC).
            TEMPORAL_RELATION.PREDECESSOR)
```

```
              THEN IF $SUBNET_B(I).DURATION > $SUBNET_A.#($NAME_B(I)).
                 DURATION THEN $SUBNET_C(NEXT) = $SUBNET_B(I);
                 ELSE $SUBNET_C(NEXT) = $SUBNET_A.#($NAME_B(I));
              ELSE $SUBNET_C(NEXT) = $SUBNET_B(I);
      END;
END;
NUMBER_DELAY = 1;
CALL NETWORK_CONDENSER(NUMBER_DELAY,$SUBNET_C,$SUBNET_D);
GRAFT $SUBNET_D AT $SUBNET_A;
END;    /*    CONDENSED_NETWORK_MERGER    */
```

2.4.27 NETWORK_ASSEMBLER

2.4.27 NETWORK_ASSEMBLER

2.4.27.1 Purpose and Scope

Given a master subnetwork and its prescribed interfacing events, this module will assemble this subnetwork and all of its interfacing subnetworks into a master network. This assembly capability facilitates the heuristic scheduling of any combination of subnetworks that may share common resources. The list of interfacing events need only be constructed to draw together all of the desired subnetworks.

2.4.27.2 Modules Called

REDUNDANT_PREDECESSOR_CHECKER

SET_INTERSECTION

SET_UNION

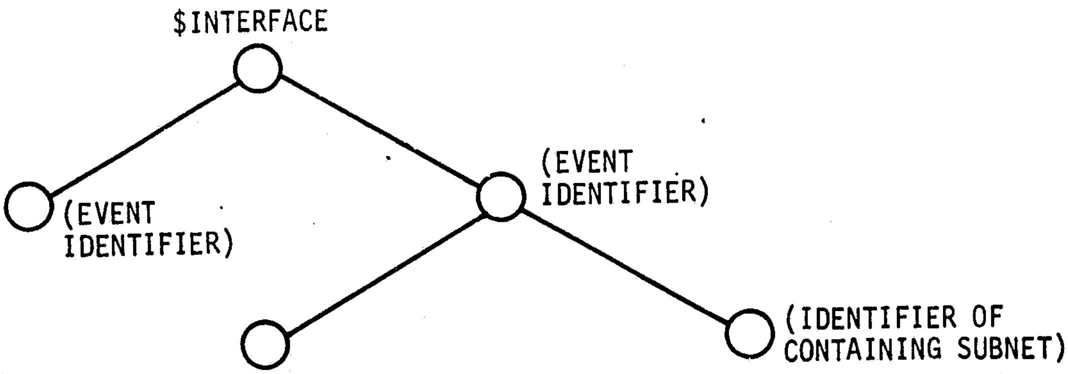
2.4.27.3 Module Input

1) Master subnetwork identifier ($MASTER_SUBNET_ID)

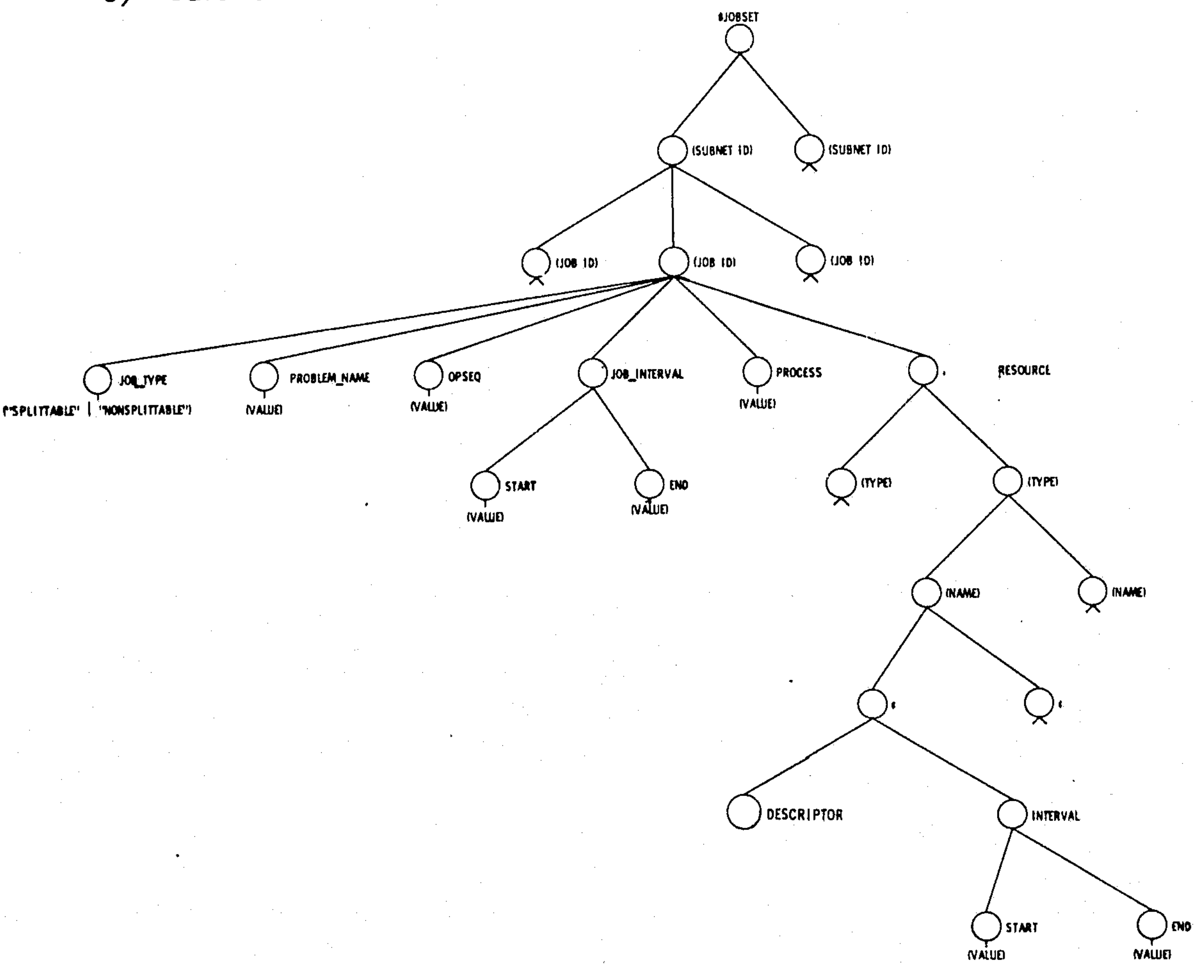
$MASTER_SUBNET_ID

(SUBNET ID)

2) Interface event definition ($INTERFACE)



3) Subnetwork definitions, including master subnetwork ($JOBSET)

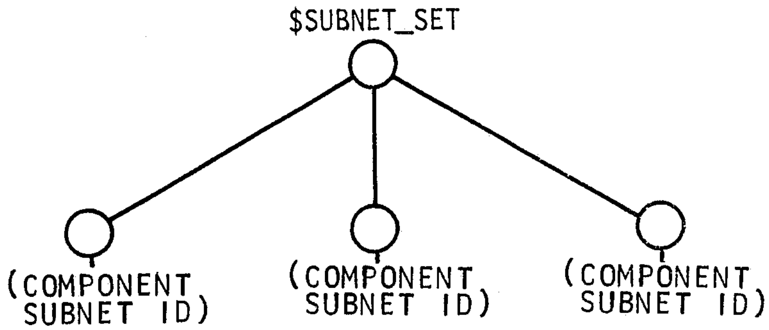


2.4.27.4 Module Output

1) Heuristic processor input data under master subnetwork node of $JOBSET

2) Component Subnetworks of Master Network ($SUBNET_SET)

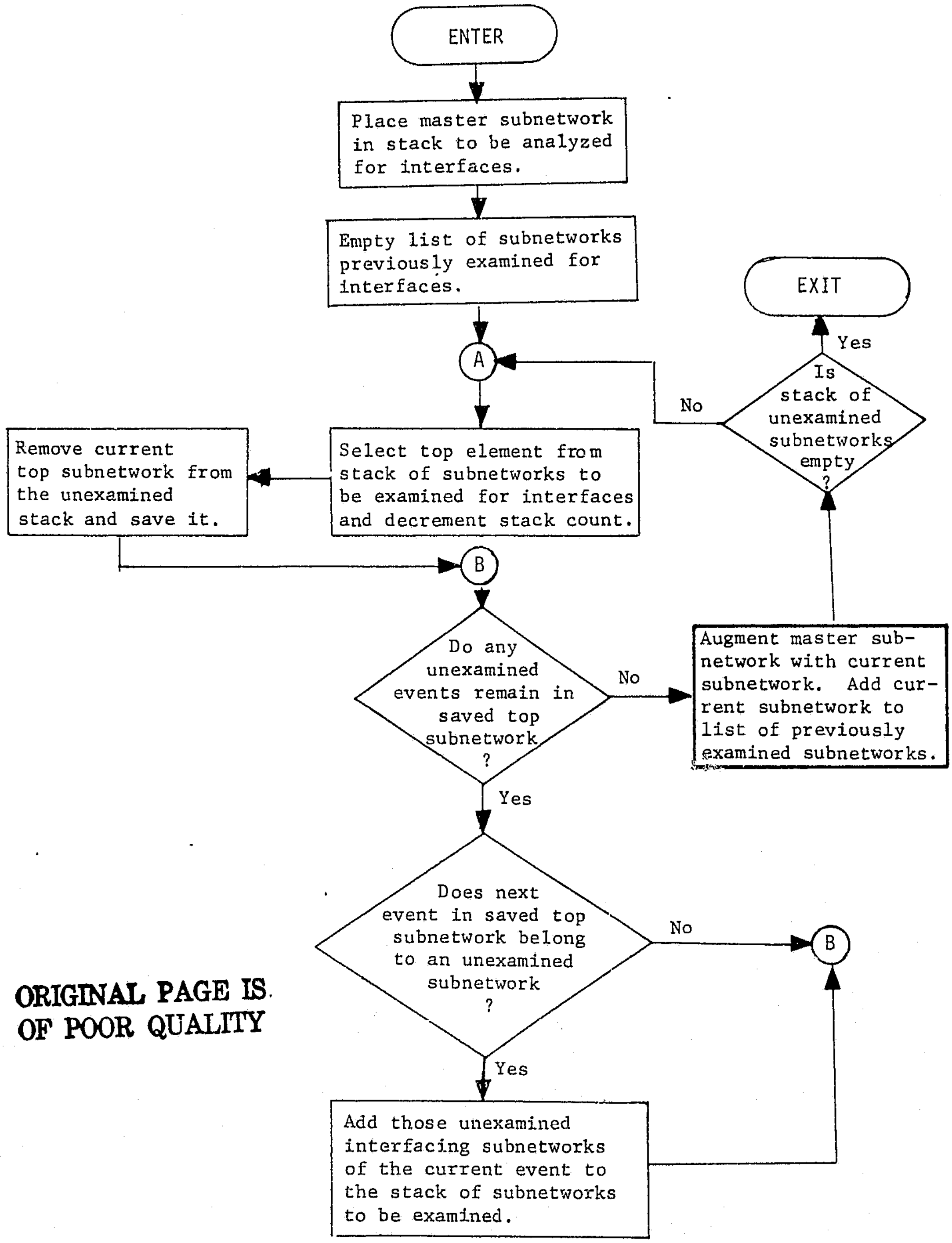
A. Component subnet identifier



2.4.27.5 Functional Description

The assembly of the master subnetwork and all of its interfacing subnetworks into a master network is straightforward. A "pushdown" stack of interfacing subnetworks to be examined is initialized to contain the master subnetwork. The top element of the stack is analyzed for interfacing subnetworks by successively examining each of its events for their presence in other unexamined subnetworks. Any such interfacing subnetworks found are added to the top of the stack. When all events in a subnetwork have been investigated it is added to the master network and removed from the unexamined stack. When the unexamined stack of interfacing networks is empty, the assembly process is complete.

2.4.27.6 Functional Block Diagram



ORIGINAL PAGE IS OF POOR QUALITY

2.4.27.7 Typical Applications

This module may be applied whenever it is necessary to assemble several subnetworks into a master network. This situation arises frequently in heuristic scheduling where several subnetworks must be scheduled simultaneously because their activities share common resources. Also small subnetworks may be assembled for display or subsequent critical path analysis. However, large subnetworks because of computer resource limitations will require condensation before merging using the two modules NETWORK_CONDENSER and CONDENSED_NETWORK_MERGER, respectively.

2.4.27.8 Detailed Design

The functional block diagram for this module may be used as a flow chart. The first subnetwork to be examined is the one identified as the master subnetwork, i.e. the subnetwork with controlling interfacing events. The name of the subnetwork to be analyzed is placed in the examined stack and is saved for future reference. Each job of the subnetwork being analyzed is examined, and if the job is an event, it is checked for interfaces. The stack remaining to be examined is augmented by the interfaces of this event. Each job of the current network is examined. Events not included in the assembled network are added to network C. If the event is included, then the predecessor set of the event in the assembled network is augmented by the predecessor set in the current network and the result placed in network C. If the job is an activity and is included in the assembled network, the great duration of the activity is included as the activity is placed in network C. If the activity is not in the assembled network, it is added to C. When all jobs of the current subnetwork have been examined, the assembled network is replaced by C and the process repeated for the next subnetwork.

2.4.27.9 Internal Variable and Tree Name Definitions

$EXAMINED_STACK	-	list of jobs which have been examined.
$NAME_A	-	list of job names in the assembled network.
$NAME_B	-	list of job names in the subnetwork being examined.
$PRED	-	temporary storage for a predecessor name, for ease in referencing.
$SAVE_NAME	-	name of the subnetwork being examined.
$SUBNET_C	-	temporary structure for the network being assembled.
$SUCC	-	temporary storage for a successor name, for case in referencing.

2.4.27.9 Internal Variable and Tree Name Definitions

$ASSEMBLED_NETWORK	-	The completed output network prior to grafting it at the master subnetwork node of $JOBSET
$DUMMY	-	Temporary storage for the name of the subnetwork being examined
$EVENT	-	Identifier for each subnode of $INTERFACE
$EXAMINED_STACK	-	List of jobs which have been examined
$INTERSECT	-	Output set returned from procedure SET_ INTERSECTION
$NAME_A	-	List of job names in the assembled network
$NAME_B	-	List of job names in the subnetwork being examined
$PRED	-	Temporary storage for the predecessor name, for ease in referencing
$SAVE_NAME	-	Name of the subnetwork being examined
$SUBNET_C	-	Temporary structure for the network being assembled
$SUCC	-	Temporary storage for a successor name, for ease in referencing
$TEMP	-	Temporary storage for $EVENT
$TEMP_NAME	-	Temporary storage for the current activity or event being examined
$TEMP_NETS	-	Identifier for each subnode of $TEMP
$TEMP2	-	Temporary storage for the name of the subnetwork being examined
$UNEXAMINED_STACK	-	List of jobs yet to be examined
$UNION	-	Output set returned from procedure SET_UNION

2.4.27.10 Modification to Functional Specifications and/or Standard Data Structure Assumed

None

2.4.27.11 Commented Code

```
NETWORK_ASSEMBLER: PROCEDURE($JOBSET,$INTERFACE,$MASTER_SUBNET_ID,
   $SUBNET_SET)
              OPTIONS(EXTERNAL);
DECLARE $UNEXAMINED_STACK,$EXAMINED_STACK,$TEMP_NAME,$SAVE_NAME LOCAL;
DECLARE $TEMP, $TEMP_NETS, $TEMP2  LOCAL;
DECLARE $NAME_A, $NAME_B, $DUMMY, $EVENT, $ASSEMBLED_NETWORK,
      $SUBNET_C, $INTERSECT, $SUCC, $PRED, I   LOCAL;
IF $INTERFACE IDENTICAL TO $NULL
   THEN IF $MASTER_SUBNET_ID IDENTICAL TO $NULL
      THEN DO;
         DO I = NUMBER($JOBSET) TO 2 BY -1;
            DO J = NUMBER($JOBSET(I)) TO 1 BY -1;
            GRAFT $JOBSET(I)(J) AT $JOBSET(FIRST)(NEXT);
            END;
         PRUNE $JOBSET(I);
         END;
         RETURN;
         END;
      ELSE DO;
         DO I = NUMBER($JOBSET) TO 1 BY -1;
         IF LABEL($JOBSET(I)) ¬= $MASTER_SUBNET_ID
            THEN DO;
               DO J = NUMBER($JOBSET(I)) TO 1 BY -1;
               GRAFT $JOBSET(I)(J) AT $JOBSET.#($MASTER_SUBNET_ID)
                     (NEXT);
               END;
               PRUNE $JOBSET(I);
               END;
            ELSE;
         END;
         RETURN;
         END;
   ELSE;
/*  PLACE MASTER SUBNETWORK IN STACK TO BE ANALYZED FOR INTERFACES    */
$UNEXAMINED_STACK(FIRST) = $MASTER_SUBNET_ID;
/*  EMPTY LIST OF SUBNETWORKS PREVIOUSLY EXAMINED FOR INTERFACES      */
PRUNE $EXAMINED_STACK;
PRUNE $ASSEMBLED_NETWORK(FIRST);
POINT_A:
/*  SELECT TOP ELEMENT FROM STACK OF SUBNETWORKS TO BE EXAMINED       */
   $EXAMINED_STACK(NEXT) = $UNEXAMINED_STACK(FIRST);
/*  SAVE CURRENT TOP SUBNETWORK FROM THE UNEXAMINED STACK             */
   $SAVE_NAME = $UNEXAMINED_STACK(FIRST);
   PRUNE $UNEXAMINED_STACK(FIRST);
POINT_B:
   DO I = 1 TO NUMBER($JOBSET.#($SAVE_NAME));
   IF $JOBSET.#($SAVE_NAME)(I).DURATION IDENTICAL TO $NULL
      THEN IF $JOBSET.#($SAVE_NAME)(I).JOB_INTERVAL IDENTICAL TO $NULL
         THEN DO;
            WRITE 'NEITHER_DURATION_NOR_JOB_INTERVAL_INPUT',
                     $SAVE_NAME ;
```

```
            RETURN;
            END;
         ELSE $JOBSET.#($SAVE_NAME)(I).DURATION = $JOBSET.#
               ($SAVE_NAME)(I).JOB_INTERVAL.END - $JOBSET.#
              ($SAVE_NAME)(I).JOB_INTERVAL.START;
      ELSE;
   IF $JOBSET.#($SAVE_NAME)(I).DURATION = 0
      THEN DO;
        DO FOR ALL SUBNODES OF $INTERFACE USING $EVENT;
          PRUNE $TEMP;
          IF $SAVE_NAME ELEMENT OF $EVENT
            THEN $TEMP = $EVENT;
            ELSE;
          LABEL($DUMMY) = $SAVE_NAME;
          PRUNE $TEMP(FIRST:($ELEMENT = LABEL($DUMMY)));
          IF $TEMP(FIRST) IDENTICAL TO $NULL
            THEN;
            ELSE DO;
              DO FOR ALL SUBNODES OF $TEMP USING $TEMP_NETS;
              IF $TEMP_NETS ¬ ELEMENT OF $EXAMINED_STACK
                THEN IF $TEMP_NETS ¬ ELEMENT OF $UNEXAMINED_STACK
/*  ADD THOSE UNEXAMINED INTERFACING SUBNETWORKS OF THE CURRENT         */
/*   EVENT TO THE STACK OF SUBNETWORKS TO BE EXAMINED                   */
                  THEN INSERT $TEMP_NETS BEFORE $UNEXAMINED_STACK
                        (NEXT);
                  ELSE;
                ELSE;
              END;
            END;
          END;
      END;
      ELSE;
   END;
/*  AUGMENT MASTER SUBNETWORK WITH CURRENT SUBNETWORK.  ADD TO LIST     */
/*   OF PREVIOUSLY EXAMINED SUBNETWORKS                                 */
START_AUGMENTATION:
  IF $ASSEMBLED_NETWORK ¬ IDENTICAL TO $NULL
   THEN DO;
      DO I = NUMBER($ASSEMBLED_NETWORK) TO 1 BY -1;
              INSERT $ASSEMBLED_NETWORK(I) BEFORE $SUBNET_C(FIRST);
      END;
   END;
   DO I = 1 TO NUMBER($ASSEMBLED_NETWORK);
      $NAME_A(I) = LABEL($ASSEMBLED_NETWORK(I));
      END;
   DO I = 1 TO NUMBER($JOBSET.#($SAVE_NAME));
      $NAME_B(I) = LABEL($JOBSET.#($SAVE_NAME)(I));
      END;
   DO I = NUMBER($NAME_B) TO 1 BY -1;
      IF $JOBSET.#($SAVE_NAME)(I).DURATION = 0
         THEN DO;
```

```
        $TEMP_NAME(FIRST)=$NAME_B(I);
           CALL SETINTERSECTION($TEMP_NAME,$NAME_A,$INTERSECT);
        IF $INTERSECT IDENTICAL TO $NULL
           THEN GRAFT $JOBSET.#($SAVE_NAME)(I) AT $SUBNET_C(NEXT);
           ELSE DO;
              CALL SETUNION($JOBSET.#($SAVE_NAME)(I).
                 TEMPORAL_RELATION.PREDECESSOR,$ASSEMBLED_NETWORK.
                 #($NAME_B(I)).TEMPORAL_RELATION.PREDECESSOR,
                 $UNION);
              GRAFT $UNION AT $SUBNET_C.#($NAME_B(I)).
                 TEMPORAL_RELATION.PREDECESSOR;
              END;
        END;
     ELSE DO;
        $PRED = $JOBSET.#($SAVE_NAME)(I).TEMPORAL_RELATION.
           PREDECESSOR;
        DO J = 1 TO NUMBER($JOBSET.#($SAVE_NAME));
        IF $NAME_B(I) ELEMENT OF
           $JOBSET.#($SAVE_NAME)(J).TEMPORAL_RELATION.PREDECESSOR
              THEN $SUCC = LABEL($JOBSET.#($SAVE_NAME)(J));
        END;
        IF ($PRED SUBSET OF $ASSEMBLED_NETWORK.#($NAME_B(I)).
           TEMPORAL_RELATION.PREDECESSOR & $NAME_B(I) ELEMENT OF
           $ASSEMBLED_NETWORK.#($SUCC).TEMPORAL_RELATION.
           PREDECESSOR)
           THEN IF $JOBSET.#($SAVE_NAME)(I).DURATION >
              $ASSEMBLED_NETWORK.#($NAME_B(I)).DURATION
              THEN GRAFT $JOBSET.#($SAVE_NAME)(I) AT $SUBNET_C
                 (NEXT);
              ELSE GRAFT $ASSEMBLED_NETWORK.#($NAME_B(I)) AT
                 $SUBNET_C(NEXT);
           ELSE GRAFT $JOBSET.#($SAVE_NAME)(I) AT $SUBNET_C(NEXT);
        END;
END;
PRUNE $ASSEMBLED_NETWORK;
GRAFT $SUBNET_C AT $ASSEMBLED_NETWORK;
IF $UNEXAMINED_STACK ¬ IDENTICAL TO $NULL
   THEN GO TO POINT_A;
   ELSE DO;
      GRAFT $ASSEMBLED_NETWORK AT $JOBSET.#($MASTER_SUBNET_ID);
      GRAFT $EXAMINED_STACK AT $SUBNET_SET;
       DO I = NUMBER($JOBSET) TO 2 BY -1;
       $TEMP2 = LABEL($JOBSET(I));
       IF $TEMP2 ELEMENT OF $SUBNET_SET
         THEN PRUNE $JOBSET(I);
       IF $JOBSET(I)(FIRST) IDENTICAL TO $NULL
         THEN PRUNE $JOBSET(I);
       END;
      DO I= NUMBER($JOBSET.#($MASTER_SUBNET_ID)) TO 1 BY -1;
        IF $JOBSET.#($MASTER_SUBNET_ID)(I).TEMPORAL_RELATION.
              PREDECESSOR(FIRST) IDENTICAL TO $NULL
```

```
            THEN PRUNE $JOBSET.#($MASTER_SUBNET_ID)(I).
                TEMPORAL_RELATION;
        END;
        CALL REDUNDANT_PREDECESSOR_CHECKER($JOBSET.#
                ($MASTER_SUBNET_ID));
        RETURN;
        END;

END NETWORK_ASSEMBLER;
```

2.4.28 CRITICAL_PATH_ PROCESSOR

2.4.28 CRITICAL_PATH_PROCESSOR

2.4.28 CRITICAL_PATH_PROCESSOR

2.4.28.1 Purpose and Scope

Given a master subnetwork and its prescribed interfacing events, this module will

1) Integrate the master subnetwork and all of its interfacing subnetworks into a condensed master network.
2) Compute the early- and late-occurrence dates of all the interface events.
3) Compute all critical-path data for the activities in the master subnetwork and all of its interfacing subnetworks.

The objective of the module is to facilitate critical path calculations on networks too large to permit direct computations because of computer resource limitations in high-speed memory and execution time.

2.4.28.2 Modules Called

NETWORK_CONDENSER

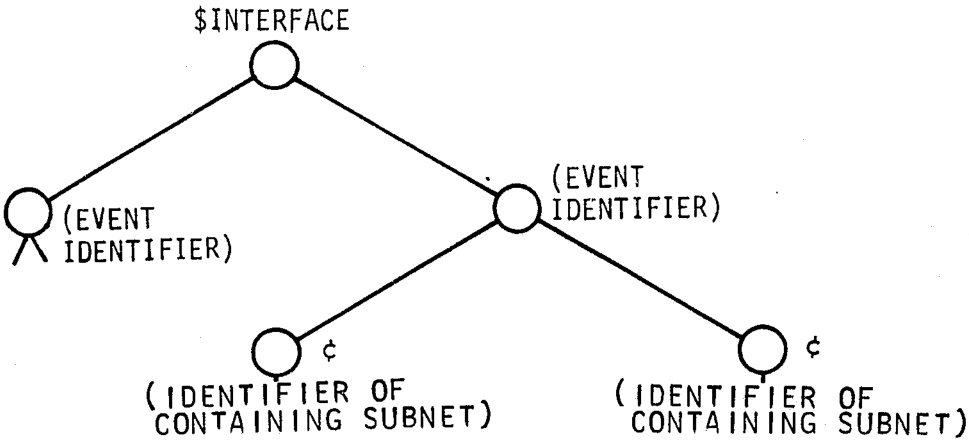
CONDENSED_NETWORK_MERGER

CRITICAL_PATH_CALCULATOR

PREDECESSOR_SET_INVERTER

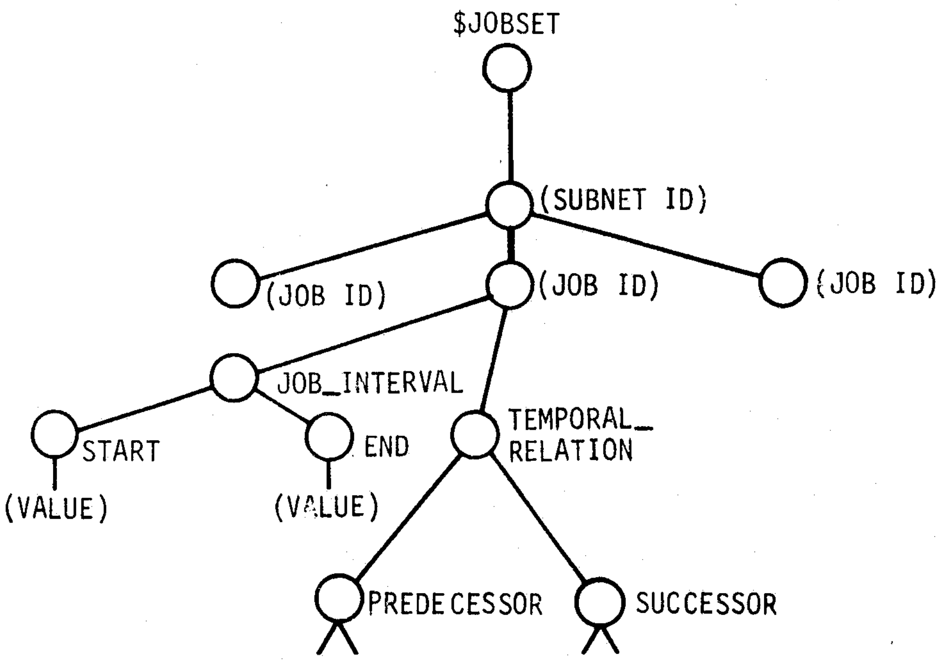
2.4.28.3 Module Input

1) Interface Event Definitions ($INTERFACE)



This data structure is illustrated in Fig. 2.4.28-1 for the subnetwork complex of Fig. 2.4.28-2

2) Subnetwork Definitions, Including Master Subnetwork ($JOBSET)



3) Master subnetwork identifier ($MASTER_SUBNET_ID)

$MASTER_SUBNET_ID

(SUBNET ID)

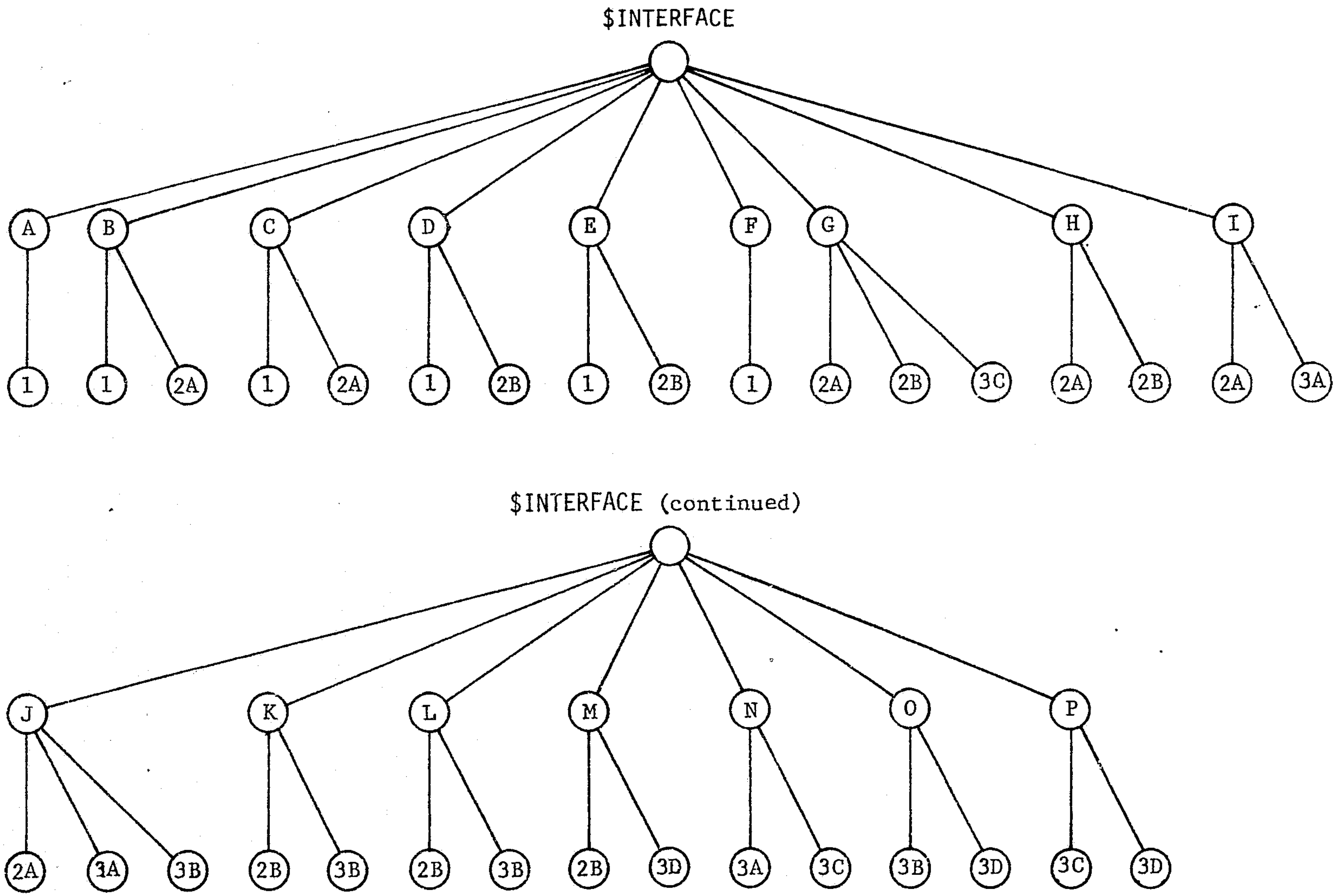


Fig. 2.4.28-1 Illustration of Interfacing-Event Data Structure for Sample Subnetwork Complex of Fig. 2.4.28-2

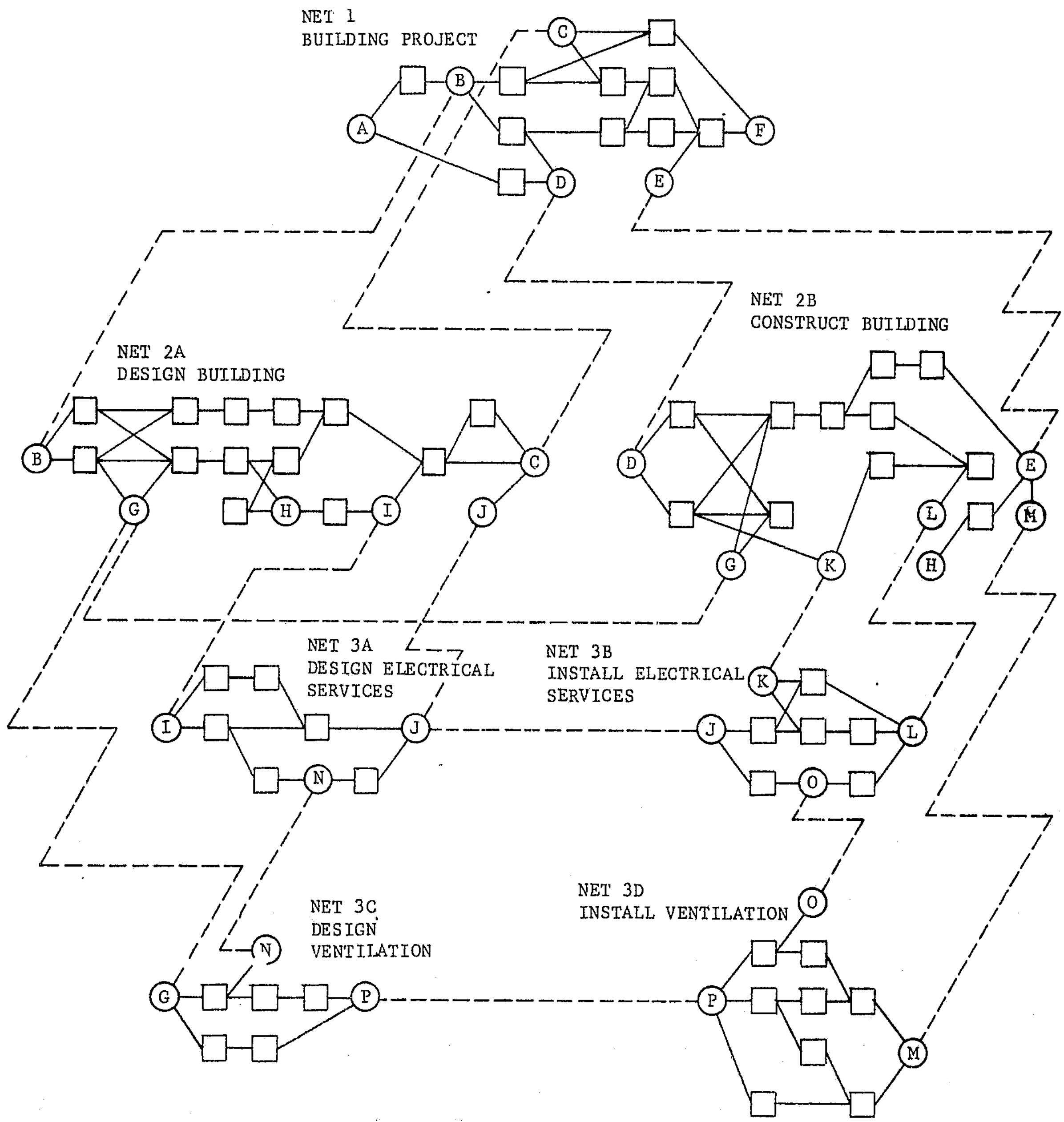
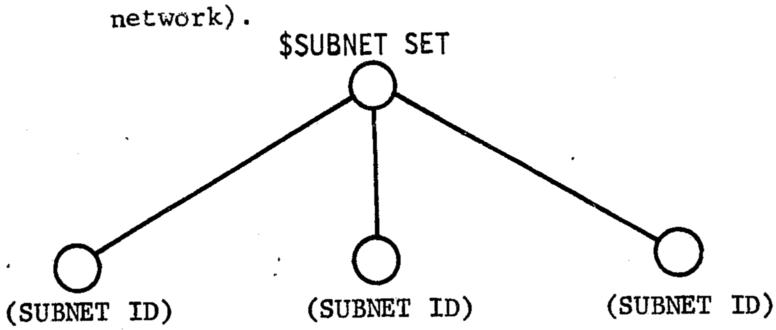


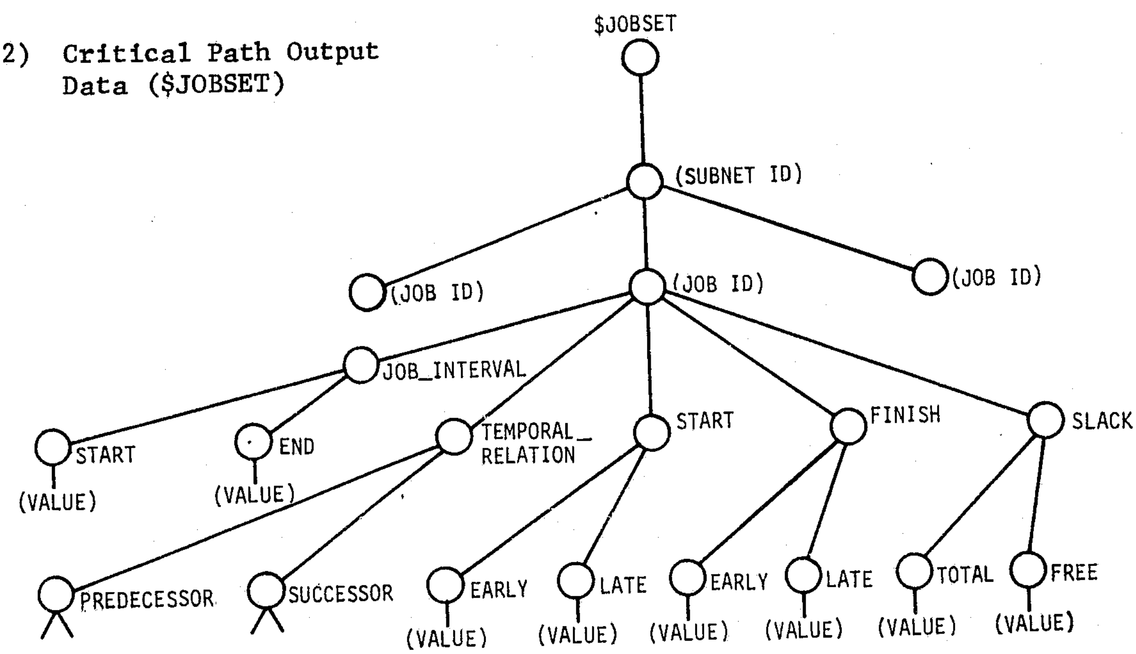
Fig. 2.4.28-2 Sample Subnetwork Complex

2.4.28.4 Module Output

1) Identifiers of subnetworks that are components of total network (all subnetworks in $JOBSET may not be connected to total network).



2) Critical Path Output Data ($JOBSET)



2.4.28.5 Functional Description

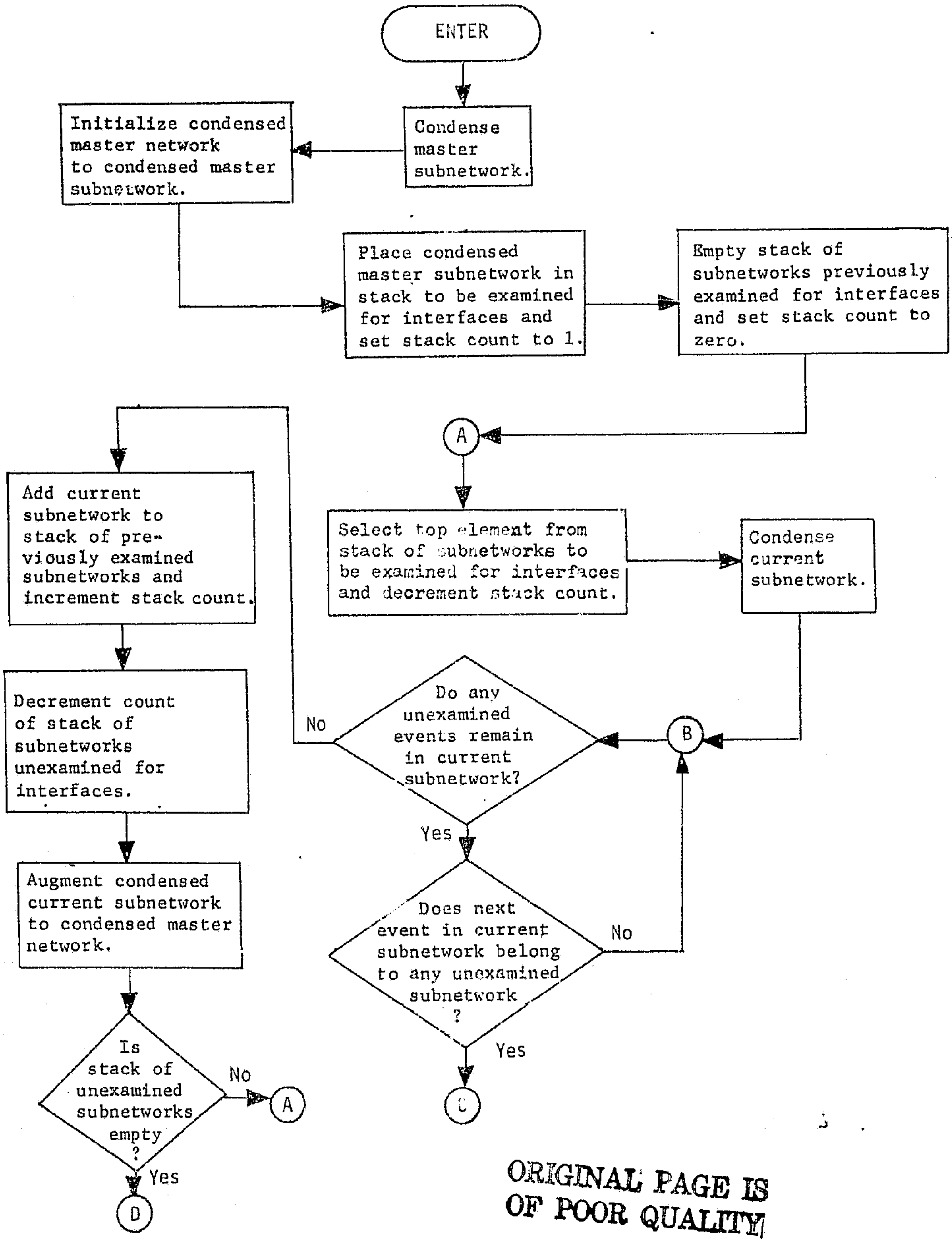
This module has three basic objectives. The first objective, assembling the subnetworks into a 'condensed' self-contained master network, is the most involved and facilitates ready accomplishment of the remaining two. Basically, it involves determining all of the subnetworks to which the specified master subnetwork is connected by interface events. These subnetworks are condensed and then merged into a condensed master network. These steps can best be accomplished in the recursive fashion. (See para 2.4.28.6.)

The master condensed network is initialized as the condensed master subnetwork. Next a 'pushdown' stack of interfacing subnetworks is created and initialized as the master subnetwork. Then, the top subnetwork of the stack is condensed and examined for interfacing subnetworks. All unanalyzed subnetworks found are added to the stack. When the interface examination of a given subnetwork is completed, it is merged into the current condensed master network. The merging process will be carried out by the module CONDENSED_NETWORK_MERGER. When the 'pushdown' stack of unexamined interfacing subnetworks is finally emptied, a self-contained master condensed network has been assembled and is ready for critical-path analysis.

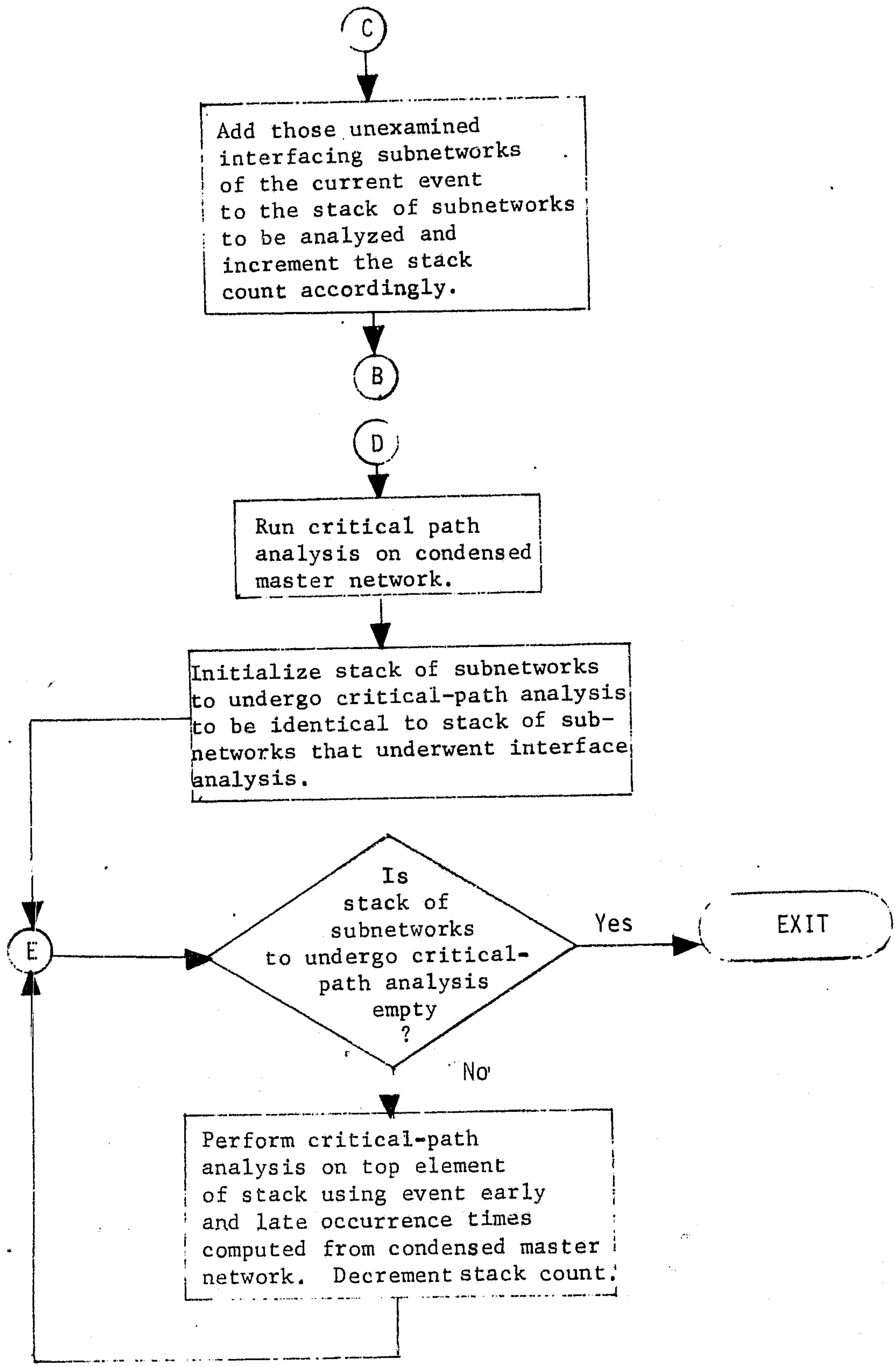
The second objective of the module, calculation of the early and late occurrence dates of all the interfacing events, is accomplished by applying the module CRITICAL_PATH_CALCULATOR to the condensed master network. To do so one need only construct the single tree SJOBSET, including the successor set substructure, required by the module CRITICAL_PATH_CALCULATOR. This input tree contains the network topology and duration information necessary for critical path analysis and can most conveniently be accumulated as the condensed master network is built. The successor sets can be built from the predecessor sets by the module PREDECESSOR_SET_INVERTER after the remainder of the tree is accumulated.

The third object of the module, computation of the critical path data for the activities in the various subnetworks, is also achieved by a simple call to the CRITICAL_PATH_CALCULATOR. This is possible provided $JOBSET contains the early- and late-occurrence date of each interface event. The input data structure $JOBSET must, of course, be filled with the network topology and the activity durations for the particular subnetwork in question. In particular the successor sets as well as the predecessor sets must be present in the tree. Hence a call to the PREDECESSOR_SET_INVERTER module is required.

2.4.28.6 Functional Block Diagram



ORIGINAL PAGE IS OF POOR QUALITY


C
Add those unexamined interfacing subnetworks of the current event to the stack of subnetworks to be analyzed and increment the stack count accordingly.
B
D
Run critical path analysis on condensed master network.
Initialize stack of subnetworks to undergo critical-path analysis to be identical to stack of subnetworks that underwent interface analysis.
E
Is stack of subnetworks to undergo critical-path analysis empty ?
Yes
EXIT
No
Perform critical-path analysis on top element of stack using event early and late occurrence times computed from condensed master network. Decrement stack count.

2.4.28.7 Typical Application

The primary application of the module is to permit critical path analyses of networks too large for the computer to handle as a single piece; it also permits network analysis using hierarchical levels of detail. Critical path data are obtained for high-level milestone events reflecting, without approximation, the effects of low-level activities. The savings in high-speed memory requirements and execution time can be considerable. Furthermore, data input tends to be more accurate when the networks involved are of comprehensible size. In general, critical path analyses based on subnetworks consistent with the normal hierarchical decomposition of the project is the most effective approach to the scheduling and control of large projects.

2.4.28.8 References:

Burman, P. J.: *Precedence Networks for Project Planning and Control*, McGraw Hill, London, 1972.

IBM: Project Management System IV, *Network Processor Program Description and Operations Manual*, Publication SH20-0899-1, 1972.

2.4.28.9 Detailed Design

The functional block diagram may be used as the flowchart for this module. The module first selects the master subnetwork and condenses it by a call to module NETWORK_CONDENSER. The name of the subnetwork is added to the stack of examined subnetworks. Each element of the current subnetwork is examined, and if events (i.e. duration equals zero) have interfaces with subnetworks not in the examined stack, the interfaces are added to the unexamined stack. When all elements have been examined, the condensed master network is augmented with the current subnetwork. This process is repeated until all subnetworks have been examined. A critical path analysis is run on the condensed master network, by a call to module CRITICAL_PATH_CALCULATOR. Each subnetwork which was examined for interfaces is subjected to critical path analysis using event early and late occurrence times computed from the condensed master network.

2.4.28.10 Internal Variable and Tree Name Definitions

$CONDENSED_MASTER	-	a tree structure representing a master network containing interface events and critical delays.
$CONDENSED_SUB	-	a tree structure containing one of the input subnetworks in condensed form.
$EXAMINED_STACK	-	a temporary tree whose first level subnode values are the labels of the subnetworks which have been examined for interfaces.

$DUMMY - Temporary storage at first subnode for subnetworks, used for call to procedure PREDECESSOR_SET_INVERTER

NUMBER_DELAY - Counter used in calls to procedure, NETWORK_CONDENSER, so that the identified delays between events are numbered consecutively

$ORDERED_MASTER - The condensed master subnetwork following, critical path analysis

$ORDERED_STACK - Any of the subnetworks following critical path analysis

$TEMP - A temporary tree whose single subnode is the label of the condensed master subnetwork

$TEMP_NAME - A temporary tree whose single subnode is the label of the subnetwork element being examined

$UNEXAMINED_STACK - A temporary tree whose first level subnodes are the labels of the subnetworks yet to be examined for interfaces

2.4.28.11 Modifications to Functional Specifications and/or Standard Data Structures Assumed

Since only one set of job elements forming a subnetwork are examined with each call to this module, the subnetwork is transmitted through the parameter list as $SUBNET_SET. The first level subnodes of $JOBSET will be only subnetwork identifiers, i.e., MASTER_SUBNET_ID will not be a sunbnode of $JOBSET. Instead, a single node tree, $MASTER_SUBNET_ID, will be used to identify the master subnetwork.

2.4.28.12 Commented Code

```
CRITICAL_PATH_PROCESSOR: PROCEDURE($JOBSET,$INTERFACE,
   $MASTER_SUBNET_ID,$SUBNET_SET) OPTIONS (EXTERNAL);
DECLARE NUMBER_DELAY LOCAL;
DECLARE I,J,$ORDERED_MASTER,$ORDERED_STACK,$TEMP LOCAL;
DECLARE $UNEXAMINED_STACK,$EXAMINED_STACK,$CONDENSED_MASTER,$TEMP_NAME,
        $DUMMY,$CONDENSED_SUB LOCAL;
/*  PLACE LABEL OF MASTER SUBNETWORK IN STACK TO BE EXAMINED                 */
$UNEXAMINED_STACK(FIRST) = $MASTER_SUBNET_ID;
PRUNE $EXAMINED_STACK;
PRUNE $CONDENSED_MASTER(FIRST);
NUMBER_DELAY = 1;
/*  MAKE TOP ELEMENT OF STACK TO BE EXAMINED 'CURRENT'                       */
/*  CONDENSE CURRENT SUBNETWORK                                              */
POINT_A:
   CALL NETWORK_CONDENSER(NUMBER_DELAY,$JOBSET.#($UNEXAMINED_STACK
        (FIRST)),$CONDENSED_SUB);
   GRAFT $CONDENSED_SUB AT $DUMMY(FIRST);
   CALL PREDECESSOR_SET_INVERTER($DUMMY);
   GRAFT $DUMMY(FIRST) AT $CONDENSED_SUB;
   LABEL($CONDENSED_SUB) = $UNEXAMINED_STACK(FIRST);
/*  ADD CURRENT SUBNETWORK TO STACK OF EXAMINED                              */
   $EXAMINED_STACK(NEXT) = $UNEXAMINED_STACK(FIRST);
   PRUNE $UNEXAMINED_STACK(FIRST);
POINT_B:
/*  DO ANY UNEXAMINED ELEMENTS REMAIN IN CURRENT SUBNETWORK?                 */
   DO I = 1 TO NUMBER($CONDENSED_SUB);
   IF $CONDENSED_SUB(I).JOB_INTERVAL IDENTICAL TO $NULL
      THEN IF $CONDENSED_SUB(I).DURATION IDENTICAL TO $NULL
         THEN DO;
            WRITE 'NEITHER_DURATION_NOR_JOB_INTERVAL_INPUT',
                  LABEL($CONDENSED_SUB(I));
            RETURN;
            END;
         ELSE;
      ELSE DO;
            $CONDENSED_SUB(I).START.EARLY = $CONDENSED_SUB(I).
                  JOB_INTERVAL.START;
            $CONDENSED_SUB(I).START.LATE  = $CONDENSED_SUB(I).
                  JOB_INTERVAL.START;
            $CONDENSED_SUB(I).FINISH.EARLY = $CONDENSED_SUB(I).
                  JOB_INTERVAL.END;
            $CONDENSED_SUB(I).FINISH.LATE  = $CONDENSED_SUB(I).
                  JOB_INTERVAL.END;
         IF $CONDENSED_SUB(I).DURATION IDENTICAL TO $NULL
            THEN $CONDENSED_SUB(I).DURATION = $CONDENSED_SUB(I).
                  JOB_INTERVAL.END - $CONDENSED_SUB(I).JOB_INTERVAL.
                  START;
            END;
/*  IS THIS ELEMENT AN EVENT?                                                */
   IF $CONDENSED_SUB(I).DURATION = 0
      THEN DO;
```

```
/*  DOES THIS EVENT HAVE INTERFACES WITH SUBNETWORKS NOT IN          */
/*   EXAMINED STACK?                                                */
          $TEMP_NAME = LABEL($CONDENSED_SUB(I));
          DO J = 1 TO NUMBER($INTERFACE.#($TEMP_NAME));
          IF $INTERFACE.#($TEMP_NAME)(J) ¬ELEMENT OF $EXAMINED_STACK
             THEN IF $INTERFACE.#($TEMP_NAME)(J) ¬ ELEMENT OF
                $UNEXAMINED_STACK
/*  ADD THE UNEXAMINED INTERFACES TO THE UNEXAMINED STACK           */
                THEN $UNEXAMINED_STACK(NEXT) = $INTERFACE.#($TEMP_NAME)
                     (J);
                ELSE;
             ELSE;
          END;
          END;
       ELSE;
    END;
/*  AUGMENT THE CONDENSED MASTER NETWORK WITH CONDENSED CURRENT     */
    CALL CONDENSED_NETWORK_MERGER($CONDENSED_MASTER,$CONDENSED_SUB);
LABEL($CONDENSED_MASTER) = $MASTER_SUBNET_ID;
  PRUNE $CONDENSED_SUB;
/*  IS STACK OF UNEXAMINED SUBNETWORKS EMPTY?                       */
    IF $UNEXAMINED_STACK ¬ IDENTICAL TO $NULL
       THEN GO TO POINT_A;
       ELSE DO;
/*  RUN CRITICAL PATH ANALYSIS ON CONDENSED MASTER NETWORK          */
          $TEMP = LABEL($CONDENSED_MASTER);
          GRAFT $CONDENSED_MASTER AT $DUMMY(FIRST);
          CALL PREDECESSOR_SET_INVERTER($DUMMY);
          GRAFT $DUMMY(FIRST) AT $CONDENSED_MASTER;
          LABEL($CONDENSED_MASTER) = $TEMP;
          CALL CRITICAL_PATH_CALCULATOR($CONDENSED_MASTER,
                $ORDERED_MASTER);
          GRAFT $ORDERED_MASTER AT $CONDENSED_MASTER;
          DO I = 1 TO NUMBER($EXAMINED_STACK);
             DO J = 1 TO NUMBER($JOBSET.#($EXAMINED_STACK(I)));
             IF $JOBSET.#($EXAMINED_STACK(I))(J).DURATION = 0
                THEN DO;
                   $JOBSET.#($EXAMINED_STACK(I))(J).START =
                      $CONDENSED_MASTER.#LABEL($JOBSET.#($EXAMINED_STACK
                            (I))(J)).START;
                   $JOBSET.#($EXAMINED_STACK(I))(J).FINISH =
                      $CONDENSED_MASTER.#LABEL($JOBSET.#($EXAMINED_STACK
                            (I))(J)).FINISH;
                   $JOBSET.#($EXAMINED_STACK(I))(J).SLACK =
                      $CONDENSED_MASTER.#LABEL($JOBSET.#($EXAMINED_STACK
                            (I))(J)).SLACK;
                   END;
             END;
/* PERFORM CRITICAL PATH ANALYSIS ON TOP ELEMENT OF STACK THAT HAS  */
/*   BEEN EXAMINED FOR INTERFACES.  USE EVENT EARLY AND LATE        */
/*   OCCURRENCE TIMES COMPUTED FROM CONDENSED MASTER NETWORK        */
```

```
        GRAFT $JOBSET.#($EXAMINED_STACK(I)) AT $DUMMY(FIRST);
        CALL PREDECESSOR_SET_INVERTER($DUMMY);
        GRAFT $DUMMY(FIRST) AT $JOBSET.#($EXAMINED_STACK(I));
        CALL CRITICAL_PATH_CALCULATOR($JOBSET.#($EXAMINED_STACK(I)),
             $ORDERED_STACK);
        GRAFT $ORDERED_STACK AT $JOBSET.#($EXAMINED_STACK(I));
        END;
        END;
GRAFT $EXAMINED_STACK AT $SUBNET_SET;
END;     /*    CRITICAL_PATH_PROCESSOR    */
```

2.4.29 NETWORK_EDITOR

2.4.29 NETWORK_EDITOR

2.4.29.1 Purpose and Scope

This module edits manually or automatically generated project scheduling precedence relations for logical inconsistencies.

Four types of errors may occur in precedence data:

1) The predecessor relationships may contain cycles; for example, job A is a predecessor of job B, B is a predecessor of C, and C is a predecessor of A.
2) The list of predecessors for a job may include more than immediate predecessors; for example job A is a predecessor of B, B is a predecessor of C, and A as well as B are listed as predecessors of C.
3) Some precedence relations may be overlooked.
4) Some predence relations may be listed that are spurious.

Errors of types (1) and (2) are inconsistencies in the data that can be detected by automated examination of the predecessor sets. Errors of types (3) and (4), however, appear to be legitimate data and, hence, cannot be discovered by computer procedures. Instead, manual checking (perhaps by a committee) is necessary to ensure that the predecessor relations are correctly reported.

Errors of type (1) are fatal to the critical path analysis. Errors of type (2), however, are not fatal and merely lengthen the execution of the critical path algorithm. For this reason the NETWORK_EDITOR has been divided into two separate editing procedures. The first, called ORDER_BY_PREDECESSORS, is mandatory. All efficient CPM processors require the job set to be arranged in a technological ordering (any job in the list precedes all of its successors). This ordering is a useful byproduct of the cycle-checking routine. The second procedure, called the REDUNDANT_PREDECESSOR_CHECKER, is optional. Its use is, however, recommended because, in addition to expediting the critical path processing, it generates the most logically concise precedence network possible.

2.4.29.2 Modules Called

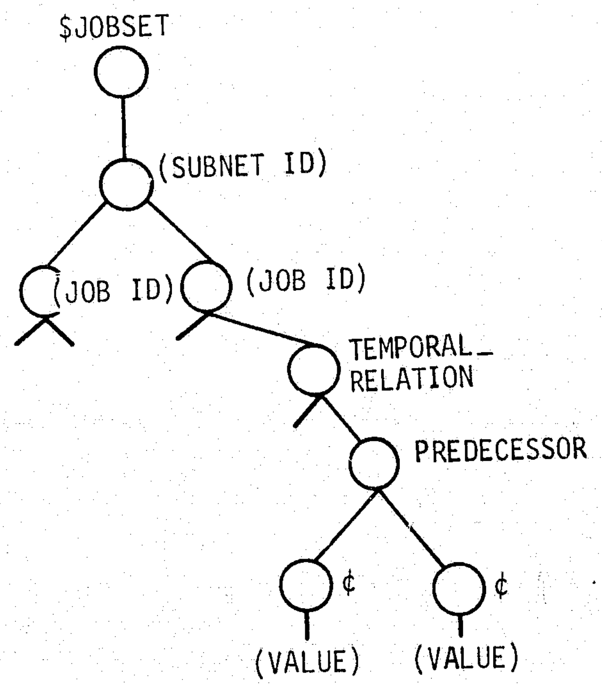
ORDER_BY_PREDECESSORS

REDUNDANT_PREDECESSOR_CHECKER

2.4.29.3 Module Input

1) Network definition $JOBSET - unedited version

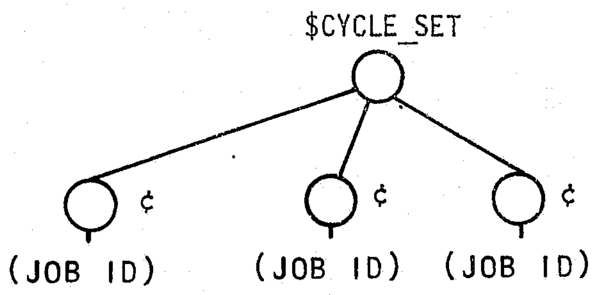
2) Redundant-predecessor-elimination option indicator (SIMPLIFY)



2.4.29.4 Module Output

1) Network definition $JOBSET - edited version

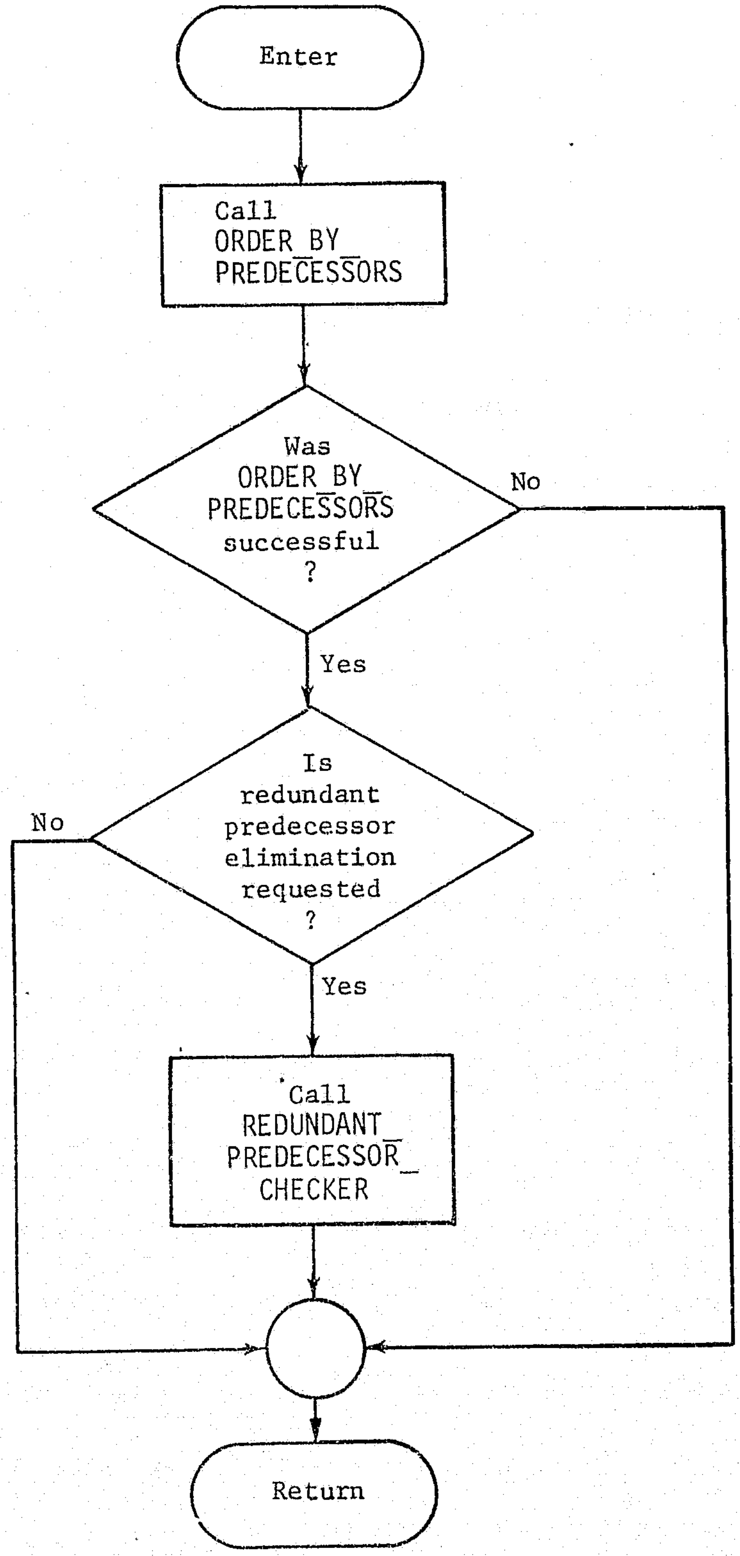
2) Cycle-containing subset of activities or events $CYCLE_SET



2.4.29.5 Functional Description

The module NETWORK_EDITOR serves primarily as a coordinator of the two editing modules, ORDER_BY_PREDECESSORS and REDUNDANT_PREDECESSOR_CHECKER. This module is intended to prevent the user from attempting to use REDUNDANT_PREDECESSOR_CHECKER without first having called ORDER_BY_PREDECESSORS to place the second level subnodes of $JOBSET in a technological ordering. The user may opt not to eliminate redundant predecessors by setting the flag SIMPLIFY.

2.4.29.6 Functional Block Diagram



2.4.29.7 Typical Application

Removal of logical inconsistencies from a precedence network is necessary in two contexts.

1) Facilitating an automated critical path analysis.

2) Preparing a consistent and concise precedence network for manual analysis to

   a) Eliminate errors of types (3) and (4) as discussed in 2.4.29.1.

   b) Improve the basic project organization.

2.4.29.8 Reference

Levy, F. K., Thompson, G. L., and Wiest, J. D.: "The ABC's of the Critical Path Method." *Harvard Business Review.* Vol 41, No. 5, September-October 1963, pp 98-107.

2.4.29.9 DETAILED DESIGN

This module simply consists of a call to ORDER_BY_PREDECESSORS and an optional call to REDUNDANT_PREDECESSOR_CHECKER. Since the functional definition of the parameters used to call ORDER_BY_PREDECESSORS have been changed, those changes have also been incorporated into this module. After calling this module, the user should always check $JOBLIST.FIRST to see if it is identical to $NULL. If not, the user can assume $JOBLIST contains a set of jobs containing a cycle and REDUNDANT_PREDECESSOR_CHECKER was not called (regardless of the option specified in the CALL statement). If $JOBLIST is empty, the user can assume $ORDERED_LIST contains the complete set of jobs and REDUNDANT_PREDECESSOR_CHECKER was called (assuming that option was specified).

2.4.29.10 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

I_ELIMINATE_REDUNDANCY_FLAG - Used to indicate whether or not REDUNDANT_PREDECESSOR_CHECKER is to be called

$JOBLIST - Is the set of jobs to be edited

$ORDER_LIST - Is the edited set of jobs output by the module

2.4.29.11 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES ASSUMED

In order to make this module compatible with the restart capability provided in ORDER_BY_PREDECESSORS, its parameters have been redefined. $JOBLIST has been substituted for $JOBSET. This is because all of the job nodes need to be one level below the root node, i.e., the SUBNET ID nodes of $JOBSET should not be present. This tree is still used to input the unedited list of jobs. However, as the jobs are ordered, they are transferred to $ORDERED_LIST. If a cycle is detected, the set of jobs containing the cycle is returned in $JOBLIST, thus eliminating the need for $CYCLE_SET. $ORDERED_LIST is the output tree of this module, but also can be used to input an already-ordered list of jobs. For better readability, the name of the option flag, SIMPLIFY, has been changed to I_ELIMINATE_REDUNDANCY_FLAG.

2.4.29.12 <u>COMMENTED CODE</u>

```
NETWORK_EDITOR:
/*************************************************************************/
/*                                                                       */
/*  THIS MODULE CHECKS FOR LOGICAL INCONSISTENCIES IN A SET OF           */
/*  PREDECESSOR RELATIONS.  ON OUTPUT, $ORDERED_LIST WILL CONTAIN        */
/*  THE JOBS (INPUT IN $JOBLIST) IN PRECEDENCE ORDER.  $ORDERED_         */
/*  LIST MAY ALSO BE USED TO INPUT A PREVIOUSLY ORDERED LIST.  IF A      */
/*  CYCLE IS DETECTED, THE MODULE IMMEDIATELY RETURNS, LEAVING THE       */
/*  SET OF JOBS CONTAINING THE CYCLE IN $JOBLIST.  THE INPUT VARI-       */
/*  ABLE 'I_ELIMINATE_REDUNDANCY_FLAG' INDICATES WHETHER OR NOT          */
/*  REDUNDANT PREDECESSOR RELATIONSHIPS ARE TO BE ELIMINATED.            */
/*                                                                       */
/*************************************************************************/
PROCEDURE ($JOBLIST, I_ELIMINATE_REDUNDANCY_FLAG, $ORDERED_LIST)
   OPTIONS(EXTERNAL);
   CALL ORDER_BY_PREDECESSORS($JOBLIST,$ORDERED_LIST) ;
  IF $JOBLIST(FIRST) NOT IDENTICAL TO $NULL  THEN RETURN ;
   IF I_ELIMINATE_REDUNDANCY_FLAG ¬= 0
      THEN CALL REDUNDANT_PREDECESSOR_CHECKER($ORDERED_LIST) ;
END NETWORK_EDITOR ;
```

2.4.30 CHECK_DESCRIPTOR_COMPATIBILITY

2.4.30 CHECK_DESCRIPTOR_COMPATIBILITY

2.4.30.1 Purpose and Scope

This module identifies incompatibilities between resource descriptors that arise when a single job is to be inserted on a timeline that contains jobs that have already been assigned. It applies to jobs that change the descriptors of one or more specific resources or that require resources with particular values of explicit resource descriptors. For example, a certain activity might require a camera with unexposed film. The same activity could leave the film in an exposed status; i.e., change the value of the film status from 'unexposed' to 'exposed.' If an attempt is made to schedule this activity ahead of another activity, which has already been assigned to the timeline and which requires unexposed film, a resource descriptor conflict would result. This module will identify such conflicts when given an existing schedule and a job to be inserted in that schedule at a specified time.

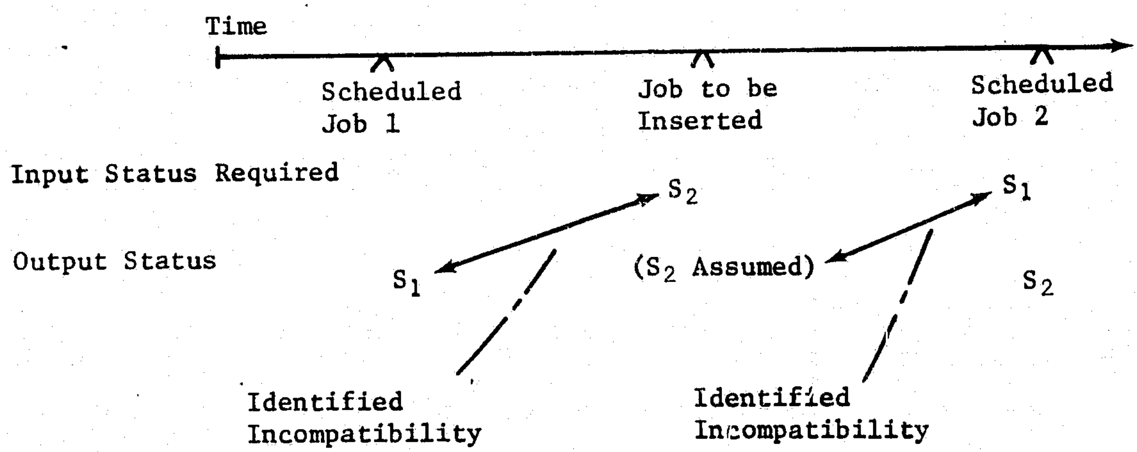
It is important to understand that this module is applicable to models with resources that have multiple explicit descriptors, but it is not suitable for use with resources that are described as pools. Pooled resources with multiple descriptors are extremely complex to model due to a proliferation of partitions of the pool. A discussion of modeling and solution strategies for pooled, explicit-descriptor resources is found in Volume II. The area of applicability of this module is illustrated as follows.

	Time Progressive Assignment Algorithms		Time Transcendent Assignment Algorithms	
	Implicit Descriptors Only	Explicit Descriptors	Implicit Descriptors Only	Explicit Descriptors
Item Specific Resources		This Module Applicable		This Module Applicable
Pooled Resources				Poor Assignment Strategy for This Type of Modeling

This module looks forward and backward in the input schedule from the assignment time for the job to be inserted. It searches backward to identify the jobs in the input schedule that place resources in improper statuses for use by the job to be inserted. The jobs in the schedule that create improper values of descriptors are identified and may or may not be those that immediately precede the job to be inserted.

Similarly, this module looks forward in time to identify the jobs in the schedule that no longer will have resources with correct descriptor values if the job to be inserted is placed in the schedules at the input assignment time. The module neither makes assignments nor cancels assignments, but merely builds a tree structure, which contains information that is useful for identifying the cause of conflicts.

This module assumes that the absence of a final descriptor means that the job does not change values of the resource that were input to the job. If the required resource descriptors for the job to be inserted are incompatible with the existing resource descriptors at the assignment time, the incompatibilities that are identified for times after the assignment time are those that result assuming compatibility between the scheduled resource descriptors and the required descriptors for the job to be inserted. This is illustrated below.



2.4.30.2 Modules Called

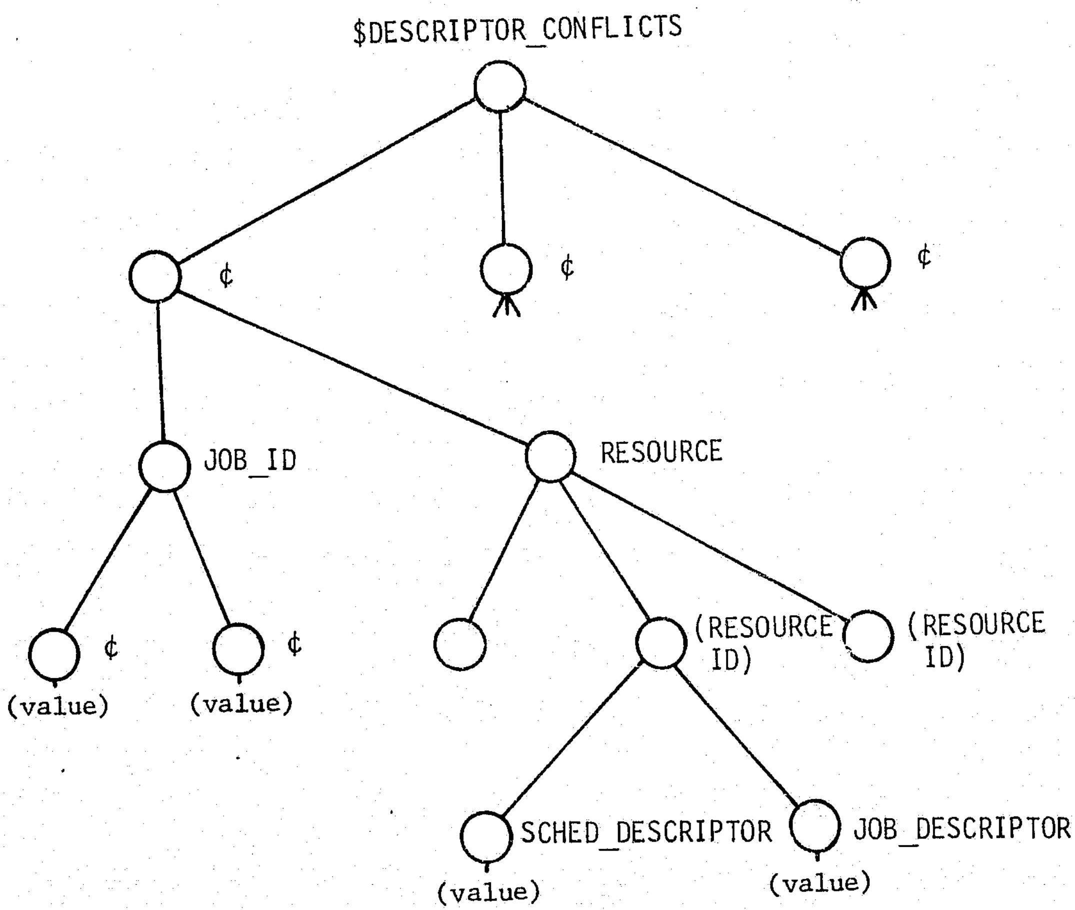
DESCRIPTOR_PROFILE

2.4.30.3 Module Input

This module is called with three input arguments. They are $CON_CHECK, $RESOURCE, and $SCHED_UNIT. $RESOURCE has the general structure given in Section 2.2 and must contain initial descriptors at a reference time and all assignment and descriptor changes that are to be considered after that time. This information is required by this module so that it can call DESCRIPTOR_PROFILE. $CON_CHECK is a string variable flag, used to indicate if contingency variables are to be examined.

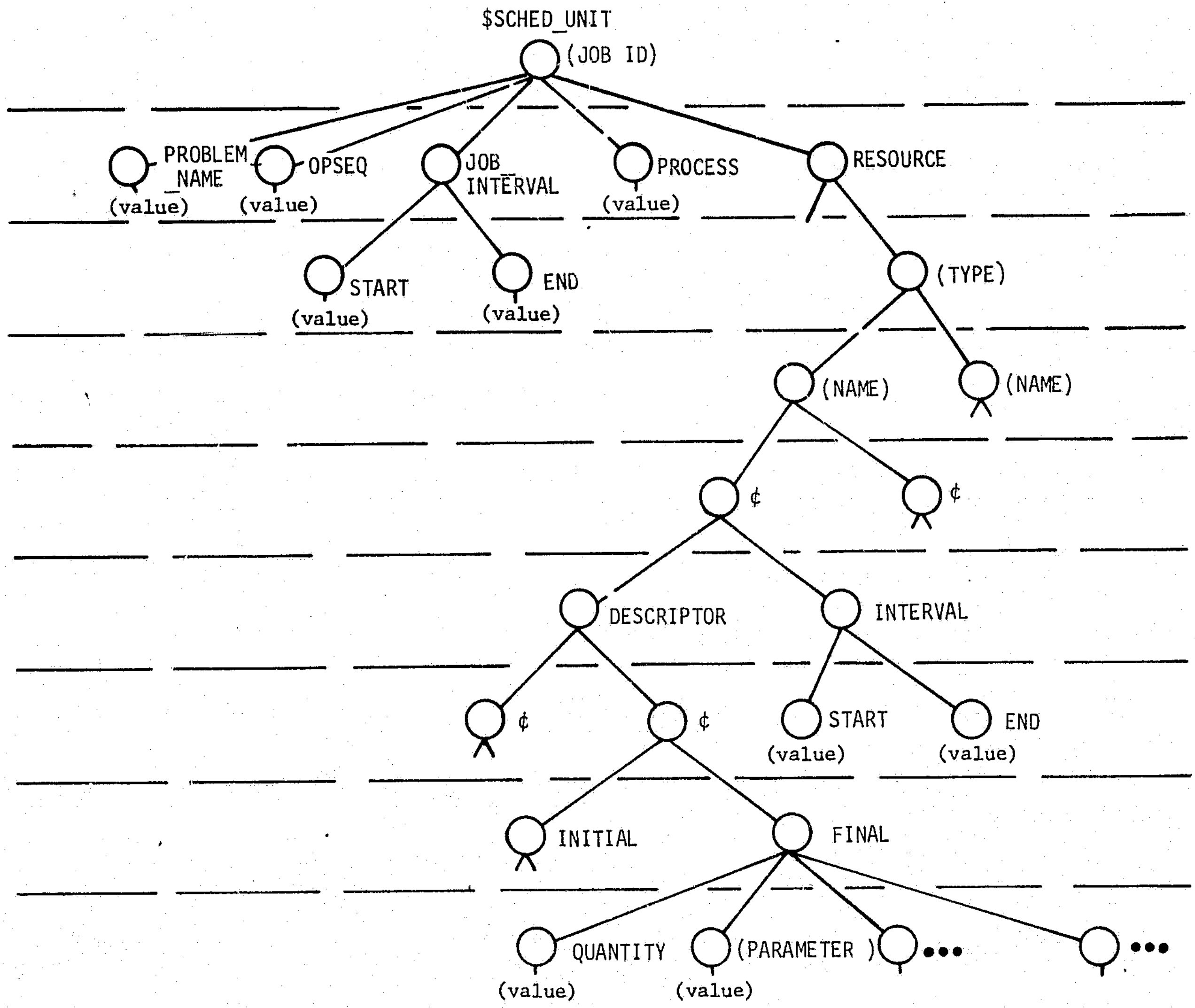
2.4.30.4 Module Output

This module returns a structure called $DESCRIPTOR_CONFLICTS, which contains information about the conflicts that would result if $SCHED_UNIT were assigned at its specified time. The general structure of $DESCRIPTOR_CONFLICTS is shown below:



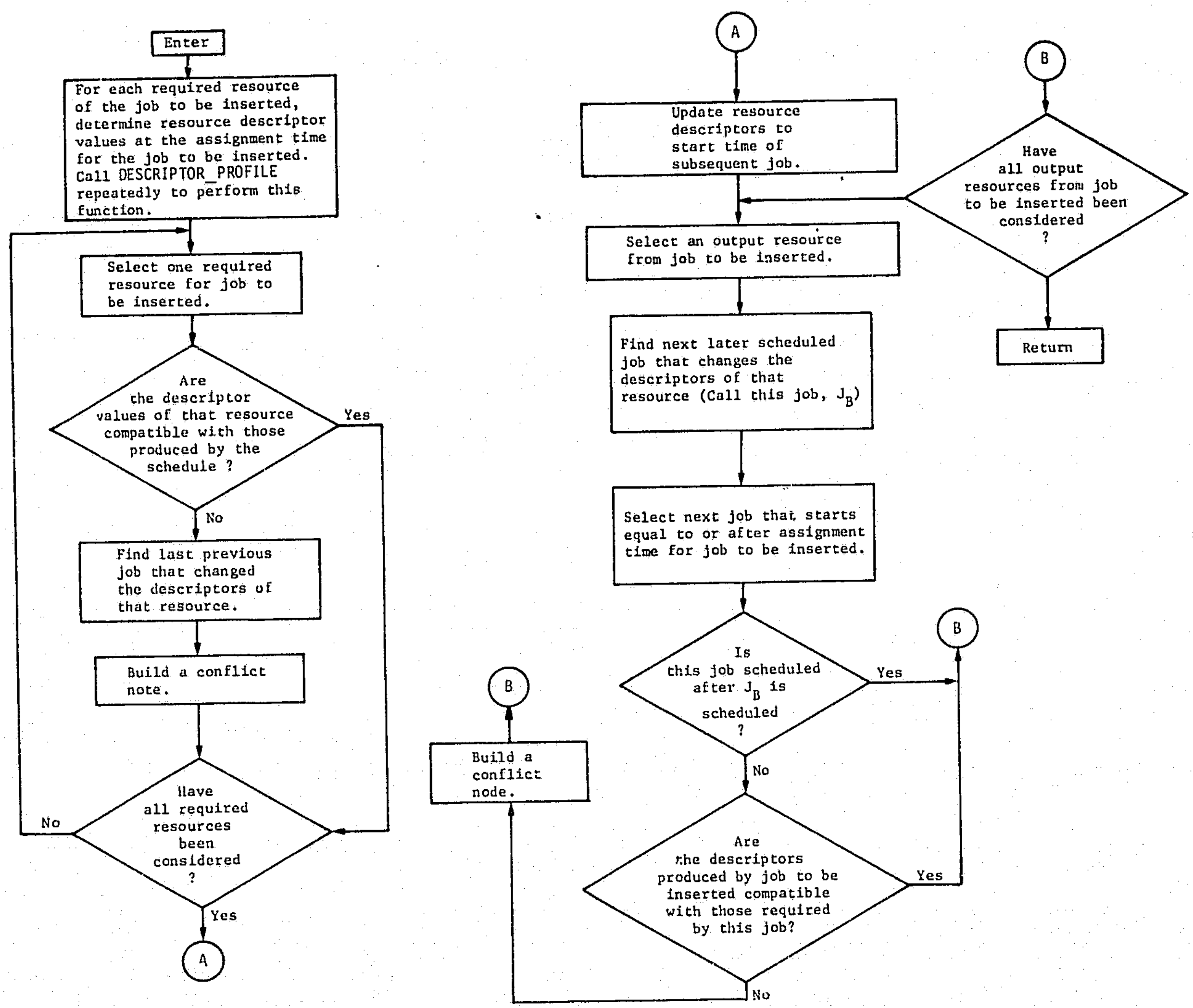
Each first-level subnode represents a resource status conflict that would result from the assignment of $SCHED_UNIT at the specified time.

$SCHED_UNIT has the general structure of a schedule unit shown below:



Note that in $SCHED_UNIT, the JOB_INTERVAL.START must contain the assignment time for the job to be inserted.

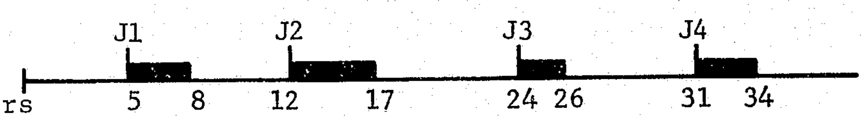
2.4.30.5 Functional Block Diagram



2.4.30.6 Typical Application

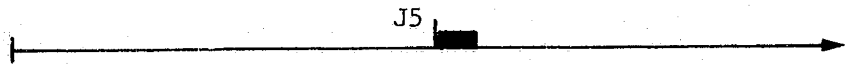
The most common use of this module is to service an assignment procedure that makes assignments in a time-transcendent manner; i.e., in a sequence that is not ordered on time. Such an assignment sequence might be: Assign job 1 at 9:00 AM, assign job 2 at 9:30 AM, assign job 3 at 9:18 AM. If each of these jobs has required resources with particular descriptors, then the insertion of job 3 between jobs 1 and 2 can cause an otherwise compatible schedule to become incompatible. Incompatibilities such as these are identified by this module.

An example of the function of this module is instructive. Consider a resource-feasible schedule consisting of 4 jobs, J1, J2, J3, and J4, and two resources, R1 and R2, each of which may have one of three descriptor values denote by S1, S2 or S3. Suppose the assignments for J1 through J4 are represented by the timeline below.



Input Descriptors Requirements					
R1	S_1	S_2	S_2	S_3	S_3
R2	S_1	S_1	S_2	S_1	S_2
Output Descriptors					
R1	S_2	S_3	S_2	S_3	S_3
R2	S_1	S_1	S_2	S_2	S_2

And suppose that the job to be inserted is represented as:



Input Descriptor
Requirements

R1 S_2

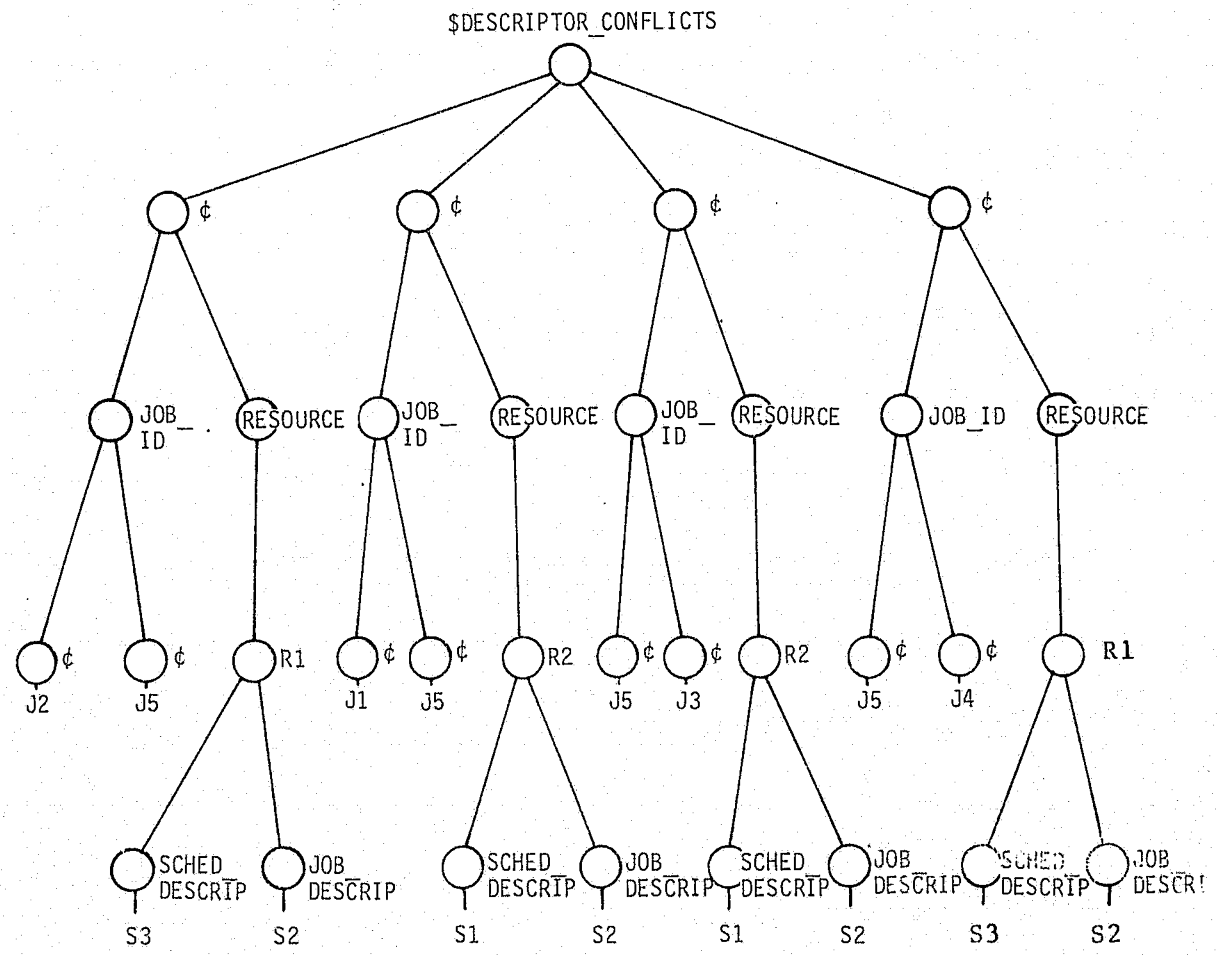
R2 S_2

Output Descriptor

R1

R2

This module will build the following structure.



2.4.30.7 Detailed Design

Each resource required by the input data structure, $SCHED_UNIT, must be checked against the preceeding resource assignments in the standard data structure, $RESOURCE, for correspondence of both intervals and descriptors. Failure of either interval or descriptor correspondence causes a node to be added to the output data structure, $DESCRIPTOR_CONFLICTS. If the resource is not specified by name then a resource name is associated with that resource. Contingent resource intervals are examined only when necessary and requested via the CON_CHECK input.

If resource descriptors are not specified in the input data structure, then any resource of the proper type and name is presumed acceptable, and the first one found is selected. For required resources which are not identified by name, the descriptors are examined first. Following successful correlation of descriptors, the intervals are examined. If this check is unsuccessful, contingency intervals are examined, if requested.

If the required resource name has been specified, the intervals are examined first, followed, if successful by descriptor examination. This order is preferred as the descriptor examination is expected to require more machine time.

When the job to be inserted has been compared with the preceding assignments it is compared with the suceeding assignments. The intervals are examined first, followed by examination of the descriptors.

Internal procedures are utilized to perform the functions of descriptor comparison, interval comparison and building the output data structure.

2.4.30.8 Interval Variable and Tree Name Definition

$ASSIGN	-	Identifier for each subnode of ASSIGNMENT for the resource being examined
$ASSIGNMENT	-	Pointer for the ASSIGNMENT node in internal procedure INTERVAL_CHECK
CHECK_FLAG	-	Flag in internal procedure, DESCRIP_CHECK, set to 'YES' if resource descriptors match, set to 'NO' if they do not match, corresponds to DESCRIP_OK
$CURRENT_PROFILE	-	Identifier for each subnode of $PROFILE in internal procedure, INTERVAL_CHECK
$DESC	-	Identifier for each subnode of last set of FINAL descriptors
DESCRIP_OK	-	Flag denoting descriptor matching, corresponds to CHECK_FLAG
$DESCRIPTION	-	Identifier for each subnode of $NAME when checking descriptors with following jobs
$DESCRIPTOR_1	-	Identifier in internal procedure, BUILD_CONFLICT, for descriptors of earlier job to be added to output data structure, $DESCRIPTOR_CONFLICTS
$DESCRIPTOR_2	-	As $DESCRIPTOR_1, for later job
$DESR	-	Identifier for each subnode of first set of INITIAL descriptors
$DUMMY	-	Temporary storage for substructure of last subnode of $RES-STATE
$DUMMY_1	-	Single node tree, whose value is the label of $SCHED-UNIT, used in call to internal procedure BUILD_CONFLICT

$DUMMY_2	-	Single node tree, whose value is the label of $TYPE, used as is $DUMMY_1
$DUMMY_3	-	Single node tree, whose value is the label of $NAME, used as is $DUMMY_1
$FOLLOWING_JOB	-	Identifier in internal procedure, BUILD_CONFLICT, for name of later job with descriptor conflicts to be added to output data structure
$FOUND_NAME	-	Name of resource returned from internal procedure DESCRIP-CHECK, whose descriptors match
1END	-	Numeric value of JOB-INTERVAL end time
$IFLAG	-	Flag in internal procedure, INTERVAL_CHECK, set to 'YES' if job intervals correspond, set to 'NO' if they do not. Same as $INTERVAL_OK
$INTERVAL_OK	-	Flag denoting interval agreement, corresponds to $IFLAG
1START	-	Numeric value of JOB-INTERVAL start time
$JOB_ID	-	Single node tree whose value is that of the last identified JOB-ID in $RES-STATE
$KSTART	-	Single node tree whose value is the start time of the schedule interval being examined
$NAME	-	Identifier for each subnode of $TYPE
$NAME_FOUND	-	Identifier in internal procedure, DESCRIP_CHECK, for each subnode of $RESOURCE_TYPE
$NAME_MISSING	-	Flag indicating the required resource name is not specified
$PRECEDING_JOB	-	Same as $FOLLOWING_JOB for earlier job

$PROFILE - Identifier in internal procedure, INTERVAL_CHECK for normal or contingency subnodes of INITIAL_PROFILE of the resource being examined

$RES_NAME - Identifier of the resource name substructure found in internal procedure, DESCRIP_CHECK. Corresponds to the single $FOUND_NAME transmitted back to main procedure.

$RES_STATE - Descriptor state at specified time returned from internal procedure, DESCRIP_CHECK. Corresponds to output of library module, DESCRIPTOR_PROFILE.

$RES_TYPE - Identifier in internal procedure, BUILD_CONFLICT, for resource type

$RESOURCE_NAME - Identifier in internal procedure DESCRIP_CHECK for resource name substructure being examined

$RESOURCE_TYPE - Identifier in internal procedure DESCRIP_CHECK for resource type substructure being examined

$STATE - Identifier in internal procedure DESCRIP_CHECK for output of library module, DESCRIPTOR_PROFILE

$START_TIME - Identifier in internal procedure DESCRIP_CHECK for initial time of job to be inserted

$SUBNAME - Identifier of each subnode of $NAME when checking descriptors of preceding jobs

$TEMP_NAME - Temporary storage for the resource name idenified for the required resource with name not specified

$TREE - Tree whose subnode values are the labels of the last set of FINAL descriptors

$TYPE - Identifier for each subnode of $RESOURCE

2.4.30.9 Modifications to Functional Specifications and/or Standard Data Structures Assumed

None

```
CHECK_DESCRIPTOR_COMPATIBILITY: PROCEDURE($CON_CHECK,$SCHED_UNIT,
     $RESOURCE,$DESCRIPTOR_CONFLICTS) OPTIONS(EXTERNAL);
/*                                                                        */
/*      THIS MODULE IDENTIFIES INCOMPATIBILITIES BETWEEN RESOURCE         */
/*  DESCRIPTORS ARISING WHEN A SINGLE JOB IS TO BE INSERTED ON A          */
/*  TIMELINE CONTAINING JOBS ALREADY ASSIGNED.                            */
/*                                                                        */
/*  INPUT:     $SCHED_UNIT - THE JOB TO BE INSERTED. THIS STRUC-          */
/*                  TURE HAS THE FORM OF A SINGLE NODE OF THE STAN-       */
/*                  DARD DATA STRUCTURE, $SCHEDULE.                       */
/*             $RESOURCE - THE STANDARD DATA STRUCTURE CONTAINING         */
/*                  THE RESOURCE ASSIGNMENTS.                             */
/*             CON_CHECK - INDICATES WHETHER CONTINGENT RESOURCES         */
/*                  ARE TO BE USED.  WITH ITEM SPECIFIC RESOURCES,        */
/*                  THIS MEANS THE USE OF CONTINGENCY INTERVALS.          */
/*                  VALUES - 'YES'  USE CONTINGENCY INTERVALS             */
/*                         - 'NO '  DO NOT USE CONTINGENCY INTERVALS      */
/*                                                                        */
/*  OUTPUT:    $DESCRIPTOR_CONFLICTS - A TREE IDENTIFYING JOBS IN         */
/*                  CONFLICT AND THE RESOURCES CAUSING THE CONFLICT.      */
/*                                                                        */
DECLARE $TYPE,$NAME,$DESCRIP_OK,$RES_NAME,$INTERVAL_OK,$DESCRIPTION,
     $NAME_MISSING,$SUBNAME, ISTART, IEND,$KSTART,
      $ASSIGN,TEMP_TYPE,$TEMP_NAME,$RES_STATE LOCAL;
DECLARE $DUMMY,$JOB_ID LOCAL;

/*      EACH RESOURCE MUST BE CHECKED AGAINST THE ASSIGNMENTS FOR         */
/*  CORRESPONDENCE OF BOTH INTERVALS AND DESCRIPTORS.  FAILURE OF         */
/*  EITHER CAUSES A SUBNODE TO BE CONSTRUCTED ON THE 'CONFLICT'           */
/*  TREE.  IF THE RESOURCE IS NOT IDENTIFIED BY NAME, THEN A NAME         */
/*  IS ASSOCIATED WITH THE RESOURCE.  CONTINGENT INTERVALS ARE            */
/*  EXAMINED ONLY WHEN NECESSARY AND REQUESTED.  ASSIGNMENTS ARE          */
/*  CHECKED PRIOR TO THE TIME THE JOB IS TO BE INSERTED, AND THEN         */
/*  FOR TIMES LATER THAN THE JOB IS TO BE INSERTED.                       */
PRUNE $DESCRIPTOR_CONFLICTS;
ISTART = $SCHED_UNIT.JOB_INTERVAL.START;
IEND = $SCHED_UNIT.JOB_INTERVAL.END;
DO FOR ALL SUBNODES OF $SCHED_UNIT.RESOURCE USING $TYPE;
   DO FOR ALL SUBNODES OF $TYPE USING $NAME;
   IF LABEL($NAME) = ''
      THEN DO FOR ALL SUBNODES OF $NAME USING $SUBNAME;
       $KSTART = $SUBNAME.INTERVAL.START + ISTART;
        $NAME_MISSING = 'YES';
         CALL DESCRIP_CHECK($NAME_MISSING,$SUBNAME,$RESOURCE.#LABEL(
                $TYPE),$KSTART,$DESCRIP_OK,
                 $RES_NAME,$RES_STATE);
         IF $DESCRIP_OK = 'YES'
            THEN DO;
               CALL INTERVAL_CHECK($SUBNAME,$RESOURCE.#LABEL($TYPE).
                     #LABEL($RES_NAME).INITIAL_PROFILE,NORMAL,
                     $RESOURCE.#LABEL($TYPE).#LABEL($RES_NAME).
```

```
        ASSIGNMENT,
       $INTERVAL_OK);
  IF $INTERVAL_OK = 'YES'
     THEN $TEMP_NAME = LABEL($RES_NAME);
     ELSE DO;
        IF $CON_CHECK = 'YES'
           THEN DO;
              CALL INTERVAL_CHECK($SUBNAME,$RESOURCE.
                     #LABEL($TYPE).#LABEL($RES_NAME).
                     INITIAL_PROFILE.CONTINGENCY,
                     $RESOURCE.#LABEL($TYPE).#LABEL
                     ($RES_NAME).ASSIGNMENT,
                    $INTERVAL_OK);
               IF $INTERVAL_OK = 'YES'
                  THEN $TEMP_NAME = LABEL($RES_NAME);
                 ELSE DO;
                   $DUMMY1=LABEL($SCHED_UNIT);
                   $DUMMY2 = LABEL($TYPE);
                   $DUMMY3 = LABEL($NAME);

                  IF $RES_STATE(LAST).JOB_ID IDENTICAL
                          TO $NULL THEN DO;
                     GRAFT $RES_STATE(LAST) AT $DUMMY;
                     $JOB_ID = $RES_STATE(LAST).JOB_ID;
                     GRAFT $DUMMY AT $RES_STATE(NEXT);
                     END;
                    ELSE $JOB_ID = $RES_STATE(LAST).JOB_ID;

                   CALL BUILD_CONFLICT($DUMMY1,
                          $JOB_ID,$DUMMY2,
                          $DUMMY3,$RES_STATE(
                             LAST).DESCRIPTOR,$SUBNAME.
                             DESCRIPTOR(FIRST).INITIAL,
                             $DESCRIPTOR_CONFLICTS);
 IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
    ASSIGNMENT(FIRST) IDENTICAL TO $NULL
  THEN PRUNE $DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#LABEL
     ($TYPE).#LABEL($NAME).SCHED_DESCRIPTOR ;
                 END;
               END;
           ELSE DO;
                  $DUMMY1=LABEL($SCHED_UNIT);
                  $DUMMY2 = LABEL($TYPE);
                  $DUMMY3 = LABEL($NAME);

                 IF $RES_STATE(LAST).JOB_ID IDENTICAL
                         TO $NULL THEN DO;
                    GRAFT $RES_STATE(LAST) AT $DUMMY;
                    $JOB_ID = $RES_STATE(LAST).JOB_ID;
                    GRAFT $DUMMY AT $RES_STATE(NEXT);
                    END;
```

```
                        ELSE $JOB_ID = $RES_STATE(LAST).JOB_ID;

                   CALL BUILD_CONFLICT($DUMMY1,
                              $JOB_ID,$DUMMY2,
                             $DUMMY3,$RES_STATE(LAST).DESCRIPTOR,
                      $SUBNAME.DESCRIPTOR(FIRST).INITIAL,
                      $DESCRIPTOR_CONFLICTS);
          IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
             ASSIGNMENT(FIRST) IDENTICAL TO $NULL
           THEN PRUNE $DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#LABEL
              ($TYPE).#LABEL($NAME).SCHED_DESCRIPTOR ;
                   END;
                END;
            END;
         ELSE DO;
                         $DUMMY1=LABEL($SCHED_UNIT);
                         $DUMMY2 = LABEL($TYPE);
                         $DUMMY3 = LABEL($NAME);

                        IF $RES_STATE(LAST).JOB_ID IDENTICAL
                              TO $NULL THEN DO;
                           GRAFT $RES_STATE(LAST) AT $DUMMY;
                           $JOB_ID = $RES_STATE(LAST).JOB_ID;
                           GRAFT $DUMMY AT $ ES_STATE(NEXT);

                           GRAFT $DUMMY AT $RES_STATE(NEXT);
                           END;
                          ELSE $JOB_ID = $RES_STATE(LAST).JOB_ID;

              CALL BUILD_CONFLICT($DUMMY1,
                $JOB_ID,$DUMMY2,$DUMMY3,          $RES_STATE(LAST).
               DESCRIPTOR,$SUBNAME.DESCRIPTOR(FIRST).INITIAL,
               $DESCRIPTOR_CONFLICTS);
           IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
              ASSIGNMENT(FIRST) IDENTICAL TO $NULL
            THEN PRUNE $DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#LABEL
              ($TYPE).#LABEL($NAME).SCHED_DESCRIPTOR ;
         END;
       END;
     ELSE DO FOR ALL SUBNODES OF $NAME USING $SUBNAME;
      $KSTART = $SUBNAME.INTERVAL.START + ISTART;
       $NAME_MISSING = 'NO';
        CALL INTERVAL_CHECK($SUBNAME,$RESOURCE.#LABEL($TYPE).#LABEL
             ($NAME).INITIAL_PROFILE.NORMAL,$RESOURCE.
             #LABEL($TYPE).#LABEL($NAME).ASSIGNMENT,
             $INTERVAL_OK);
        IF $INTERVAL_OK = 'YES'
          THEN DO;
             CALL DESCRIP_CHECK($NAME_MISSING,$NAME,    $RESOURCE.#
                   LABEL($TYPE),$KSTART,
                  $DESCRIP_OK,$RES_NAME,$RES_STATE);
```

```
   IF $DESCRIP_OK = 'YES'
      THEN;
      ELSE DO;
                    $DUMMY1=LABEL($SCHED_UNIT);
                    $DUMMY3 = LABEL($NAME);
                    $DUMMY2 = LABEL($TYPE);

                   IF $RES_STATE(LAST).JOB_ID IDENTICAL
                         TO $NULL THEN DO;
                      GRAFT $RES_STATE(LAST) AT $DUMMY;
                      $JOB_ID = $RES_STATE(LAST).JOB_ID;
                      GRAFT $DUMMY AT $RES_STATE(NEXT);
                      END;
                     ELSE $JOB_ID = $RES_STATE(LAST).JOB_ID;

           CALL BUILD_CONFLICT($DUMMY1,
                              $JOB_ID,$DUMMY2,$DUMMY3,
                  $RES_STATE(LAST).DESCRIPTOR,$SUBNAME,
           DESCRIPTOR(FIRST).INITIAL,$DESCRIPTOR_CONFLICTS)
                  ;
  IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
     ASSIGNMENT(FIRST) IDENTICAL TO $NULL
    THEN PRUNE $DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#LABEL
     ($TYPE).#LABEL($NAME).SCHED_DESCRIPTOR ;
       END;
    END;
 ELSE DO;
    IF $CON_CHECK = 'YES'
       THEN DO;
          CALL INTERVAL_CHECK($SUBNAME,$RESOURCE.#LABEL(
                $TYPE).#LABEL($NAME).INITIAL_PROFILE.
                CONTINGENCY,$RESOURCE.#LABEL($TYPE).
                #LABEL($NAME).ASSIGNMENT,$INTERVAL_OK) ;
          IF $INTERVAL_OK = 'YES'
             THEN DO;
                CALL DESCRIP_CHECK($NAME_MISSING,$SUBNAME,
                      $RESOURCE.#LABEL($TYPE),
                                  $KSTART,$DESCRIP_OK,
                      $RES_NAME,$RES_STATE);
               IF $DESCRIP_OK = 'YES'
                  THEN;
                 ELSE DO;
                   $DUMMY1=LABEL($SCHED_UNIT);
                   $DUMMY2 = LABEL($TYPE);
                   $DUMMY3 = LABEL($NAME);

                 IF $RES_STATE(LAST).JOB_ID IDENTICAL
                       TO $NULL THEN DO;
                    GRAFT $RES_STATE(LAST) AT $DUMMY;
                    $JOB_ID = $RES_STATE(LAST).JOB_ID;
                    GRAFT $DUMMY AT $RES_STATE(NEXT);
```

```
                    END;
                  ELSE $JOB_ID = $RES_STATE(LAST).JOB_ID;

                     CALL BUILD_CONFLICT($DUMMY1,
                                                $JOB_ID,
                  $DUMMY2,$DUMMY3,$RES_STATE(
                        LAST).DESCRIPTOR,$SUBNAME.
                        DESCRIPTOR(FIRST).INITIAL,
                        $DESCRIPTOR_CONFLICTS);
IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
   ASSIGNMENT(FIRST) IDENTICAL TO $NULL
 THEN PRUNE $DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#LABEL
   ($TYPE).#LABEL($NAME).SCHED_DESCRIPTOR ;
                  END;
                END;
             ELSE DO;
                    $DUMMY1=LABEL($SCHED_UNIT);
                    $DUMMY2 = LABEL($TYPE);
                    $DUMMY3 = LABEL($NAME);

                   IF $RES_STATE(LAST).JOB_ID IDENTICAL
                         TO $NULL THEN DO;
                      GRAFT $RES_STATE(LAST) AT $DUMMY;
                      $JOB_ID = $RES_STATE(LAST).JOB_ID;
                      GRAFT $DUMMY AT $RES_STATE(NEXT);
                      END;
                     ELSE $JOB_ID = $RES_STATE(LAST).JOB_ID;

                  CALL BUILD_CONFLICT($DUMMY1,
                                      $JOB_ID,$DUMMY2,
                        $DUMMY3,$RES_STATE(LAST).DESCRIPTOR,
                   $SUBNAME.DESCRIPTOR(FIRST).INITIAL,
                   $DESCRIPTOR_CONFLICTS);
 IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
    ASSIGNMENT(FIRST) IDENTICAL TO $NULL
  THEN PRUNE $DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#LABEL
    ($TYPE).#LABEL($NAME).SCHED_DESCRIPTOR ;
              END;
           END;
      ELSE DO;
           CALL DESCRIPTOR_PROFILE($RESOURCE.#LABEL($TYPE).
             #LABEL($NAME),$RESOURCE.#LABEL($TYPE).#
             LABEL($NAME).ASSIGNMENT,$RES_STATE,
            $KSTART);
                    $DUMMY1=LABEL($SCHED_UNIT);
                    $DUMMY2 = LABEL($TYPE);
                    $DUMMY3 = LABEL($NAME);

                   IF $RES_STATE(LAST).JOB_ID IDENTICAL
                         TO $NULL THEN DO;
                      GRAFT $RES_STATE(LAST) AT $DUMMY;
```

```
                              $JOB_ID = $RES_STATE(LAST).JOB_ID;
                              GRAFT $DUMMY AT $RES_STATE(NEXT);
                              END;
                             ELSE $JOB_ID = $RES_STATE(LAST).JOB_ID;

                      CALL BUILD_CONFLICT($DUMMY1,
                             $JOB_ID,$DUMMY2,$DUMMY3,
                        $RES_STATE(LAST).DESCRIPTOR,$SUBNAME.DESCRIPTOR
                        (FIRST).INITIAL,$DESCRIPTOR_CONFLICTS);
               IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
                  ASSIGNMENT(FIRST) IDENTICAL TO $NULL
                THEN PRUNE $DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#LABEL
                  ($TYPE).#LABEL($NAME).SCHED_DESCRIPTOR ;
                   END;
                 END;
          END;
/*       THIS SECTION OF CODE DOES THE CHECKING OF ASSIGNMENTS AT          */
/*   TIMES LATER THAN THE JOB IS TO BE INSERTED.                            */
   CHECK_FOLLOWING_JOBS:
      DO FOR ALL SUBNODES OF $NAME USING $DESCRIPTION;
       IF $NAME_MISSING = 'YES'
         THEN;
         ELSE $TEMP_NAME = LABEL($NAME);
         DO FOR ALL SUBNODES OF $RESOURCE.#LABEL($TYPE).#($TEMP_NAME)
              .ASSIGNMENT USING $ASSIGN;
         IF $ASSIGN.INTERVAL.START <($DESCRIPTION.INTERVAL.END +
                     ISTART)
            THEN;
            ELSE DO;
               IF ($DESCRIPTION.DESCRIPTOR(LAST).FINAL SUBSET OF
                     $ASSIGN.DESCRIPTOR(FIRST).INITIAL | $ASSIGN.
                     DESCRIPTOR(FIRST).INITIAL SUBSET OF $DESCRIPTION.
                     DESCRIPTOR(LAST).FINAL) THEN;
                  ELSE DO;
                    PRUNE $TREE;
                    DO FOR ALL SUBNODES OF $DESCRIPTION.DESCRIPTOR
                        (LAST).FINAL USING $DESC;
                       $TREE(NEXT) = LABEL($DESC);
                    END;
                    DO FOR ALL SUBNODES OF $ASSIGN.DESCRIPTOR(FIRST).
                        INITIAL USING $DESR;
                      IF LABEL($DESR) ¬ELEMENT OF $TREE THEN;
                        ELSE IF $ASSIGN.DESCRIPTOR(FIRST).INITIAL
                                  ¬SUBSET OF $DESCRIPTION.
                         DESCRIPTOR(LAST).FINAL
                           THEN DO;
                                $DUMMY1=LABEL($SCHED_UNIT);
                                $DUMMY2 = LABEL($TYPE);
                                $DUMMY3 = LABEL($NAME);
                     CALL BUILD_CONFLICT($ASSIGN.JOB_ID,
                          $DUMMY1,$DUMMY2,$DUMMY3,
```

```
                    $ASSIGN.DESCRIPTOR(FIRST).INITIAL,
                            $DESCRIPTION.DESCRIPTOR(LAST).FINAL,
                            $DESCRIPTOR_CONFLICTS)!
              IF $RESOURCE.#LABEL($TYPE).#LABEL($NAME).
                 ASSIGNMENT(FIRST) IDENTICAL TO $NULL
               THEN PRUNE $DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#LABEL
                 ($TYPE).#LABEL($NAME).SCHED_DESCRIPTOR !
                      GO TO NEXT_RESOURCE_NAME!
                      END!
                     END!
                     END!
                END!
          END!
IF $DESCRIPTOR_CONFLICTS(FIRST) IDENTICAL TO $NULL
   THEN LABEL($NAME) = $TEMP_NAME!
      END!
NEXT_RESOURCE_NAME:
   END!
END!


DESCRIP_CHECK: PROCEDURE($NAME_MISSING,$RESOURCE_NAME,$RESOURCE_TYPE,
      $START_TIME,$CHECK_FLAG,$FOUND_NAME,$STATE)!
/*     THIS INTERNAL PROCEDURE IS DESIGNED TO CHECK ONLY THE             */
/*  RESOURCE DESCRIPTORS.  THIS PROCEDURE CALLS THE MODULE               */
/*  'DESCRIPTOR_PROFILE' TO DETERMINE THE RESOURCE STATE AT THE          */
/*  TIME THE JOB IS TO BE INSERTED.  THE FINAL DESCRIPTORS OF THE        */
/*  RESOURCE STATE ARE COMPARED WITH THOSE REQUIRED BY THE JOB TO        */
/*  BE INSERTED.  THE INPUT VARIABLE 'NAME_MISSING' INFORMS THE          */
/*  PROCEDURE TO RETURN A RESOURCE NAME IF MATCHING DESCRIPTORS          */
/*  ARE FOUND.                                                           */
DECLARE $RESOURCE_NAME,$RESOURCE_TYPE,$START_TIME,CHECK_FLAG,
      $NAME_FOUND,$STATE,$NAME_MISSING LOCAL!
$CHECK_FLAG = 'NO'!
IF $NAME_MISSING = 'YES'
   THEN DO FOR ALL SUBNODES OF $RESOURCE_TYPE USING $NAME_FOUND:
      CALL DESCRIPTOR_PROFILE($NAME_FOUND,$NAME_FOUND.ASSIGNMENT,
                   $STATE,$START_TIME)!
      IF $RESOURCE_NAME       .DESCRIPTOR(FIRST).INITIAL SUBSET OF
         $STATE(LAST).DESCRIPTOR
         THEN DO!
           $CHECK_FLAG = 'YES'!
            $FOUND_NAME = $NAME_FOUND!
            RETURN!
            END!
      END!
   ELSE DO!
      CALL DESCRIPTOR_PROFILE($RESOURCE_TYPE.#LABEL($RESOURCE_NAME),
                  $RESOURCE_TYPE.#LABEL($RESOURCE_NAME).ASSIGNMENT,
            $STATE,$START_TIME)!
      IF $RESOURCE_NAME(FIRST).DESCRIPTOR(FIRST).INITIAL SUBSET OF
```

```
            $STATE(LAST).DESCRIPTOR
          THEN DO;
            $CHECK_FLAG = 'YES';
             $FOUND_NAME = $RESOURCE_NAME;
             RETURN;
             END;
       END;
END_DESCRIP_CHECK: END;


INTERVAL_CHECK: PROCEDURE($RESOURCE_NAME,$PROFILE,$ASSIGNMENT,$IFLAG);
/*       THIS INTERNAL PROCEDURE CHECKS THE INITIAL PROFILE OF THE      */
/*  PROPER RESOURCE TYPE AND NAME TO INSURE THAT THE RESOURCE IS       */
/*  AT LEAST POTENTIALLY AVAILABLE AT THE TIMES REQUESTED.             */
DECLARE $IFLAG,$PROFILE,$RESOURCE_NAME,$ASSIGN  LOCAL;
$IFLAG = 'YES';
   DO FOR ALL SUBNODES OF $PROFILE USING $CURRENT_PROFILE;
   IF  $RESOURCE_NAME.INTERVAL.START + ISTART  >= $CURRENT_PROFILE.
     START &  $RESOURCE_NAME.INTERVAL.END +ISTART <= $CURRENT_PROFILE.
    END
      THEN GO TO NEXT_SCHED_INTERVAL;
   END;
  $IFLAG = 'NO';
NEXT_SCHED_INTERVAL:
 IF $IFLAG='YES'
   THEN DO;
     DO FOR ALL SUBNODES OF $ASSIGNMENT USING $ASSIGN;
       IF $RESOURCE_NAME.INTERVAL.START+ISTART<=$ASSIGN.
                        INTERVAL.START
         THEN IF $RESOURCE_NAME.INTERVAL.END+ISTART<=$ASSIGN.INTERVAL.
                        START
           THEN;
           ELSE $IFLAG='NO';
         ELSE;
       END;
     END;
                        END;      /* END_INTERVAL_CHECK   */

BUILD_CONFLICT: PROCEDURE($PRECEDING_JOB,$FOLLOWING_JOB,$RES_TYPE,
     $RES_NAME,$DESCRIPTOR_1,$DESCRIPTOR_2,$DESCRIPTOR_CONFLICTS);
/*    THIS PROCEDURE CONSTRUCTS THE OUTPUT, '$DESCRIPTOR_CONFLICTS'   */
DECLARE $PRECEDING_JOB,$FOLLOWING_JOB,$PRECEDING_DESCRIPTOR,
   $FOLLOWING_DESCRIPTOR LOCAL;
$DESCRIPTOR_CONFLICTS(NEXT).JOB_ID(FIRST)= $FOLLOWING_JOB;
LABEL($DESCRIPTOR_CONFLICTS(LAST).JOB_ID(FIRST)) = '';
$DESCRIPTOR_CONFLICTS(LAST).JOB_ID(NEXT) = $PRECEDING_JOB;
LABEL($DESCRIPTOR_CONFLICTS(LAST).JOB_ID(LAST)) = '';
$DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#($RES_TYPE).#($RES_NAME).
      SCHED_DESCRIPTOR = $DESCRIPTOR_1;
$DESCRIPTOR_CONFLICTS(LAST).RESOURCE.#($RES_TYPE).#($RES_NAME).
      JOB_DESCRIPTOR = $DESCRIPTOR_2;


END;   /*   BUILD_CONFLICT   */
END;   /*   CHECK_DESCRIPTOR_COMPATIBILITY   */
```

2.4.31 ORDER_BY_PREDECESSORS

2.4.31 ORDER_BY_PREDECESSORS

2.4.31.1 Purpose and Scope

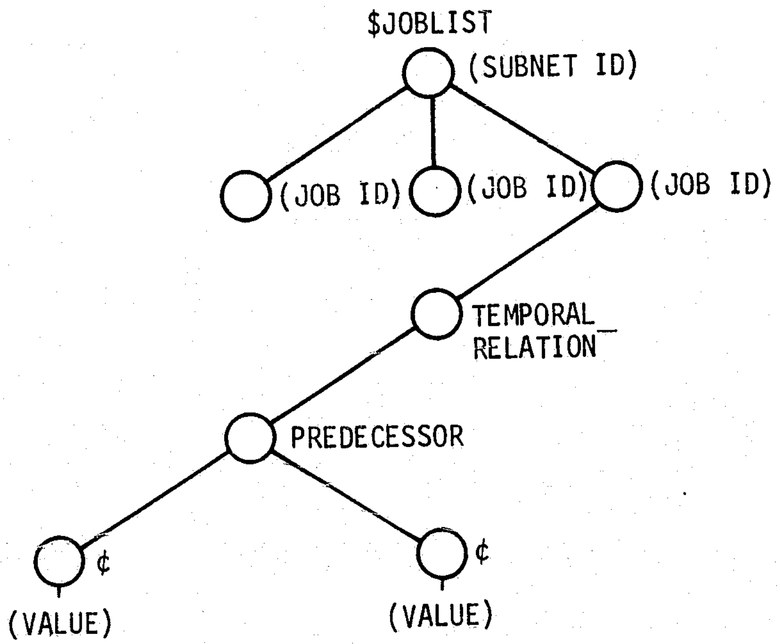
Given a set of activities and events and their respective predecessor sets, this module either places them in a technological order if one exists or identifies a subset of the activities containing a cycle. A technological ordering of the events and activities means an ordering such that any activity or event is preceded by all of its predecessors or equivalently followed by all of its successors. A cycle, on the other hand, is a chain of predecessor-successor related activities or events implying that some event or activity is a predecessor of itself. Such an activity or event could never be scheduled because one of its predecessors, namely itself, could never be completed beforehand. Hence, the presence of cycles in a precedence network precludes any scheduling or critical path analyses.

2.4.31.2 Modules Called

None

2.4.31.3 Module Input

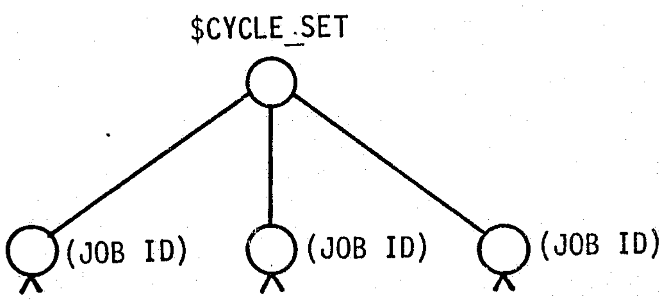
Network definition (**$JOBLIST**) - activities or events (first level subnodes) are not technologically ordered.



2.4.31.4 Module Output

1) Network definition ($JOBLIST) - activities or events (second-level subnodes) are technologically ordered.

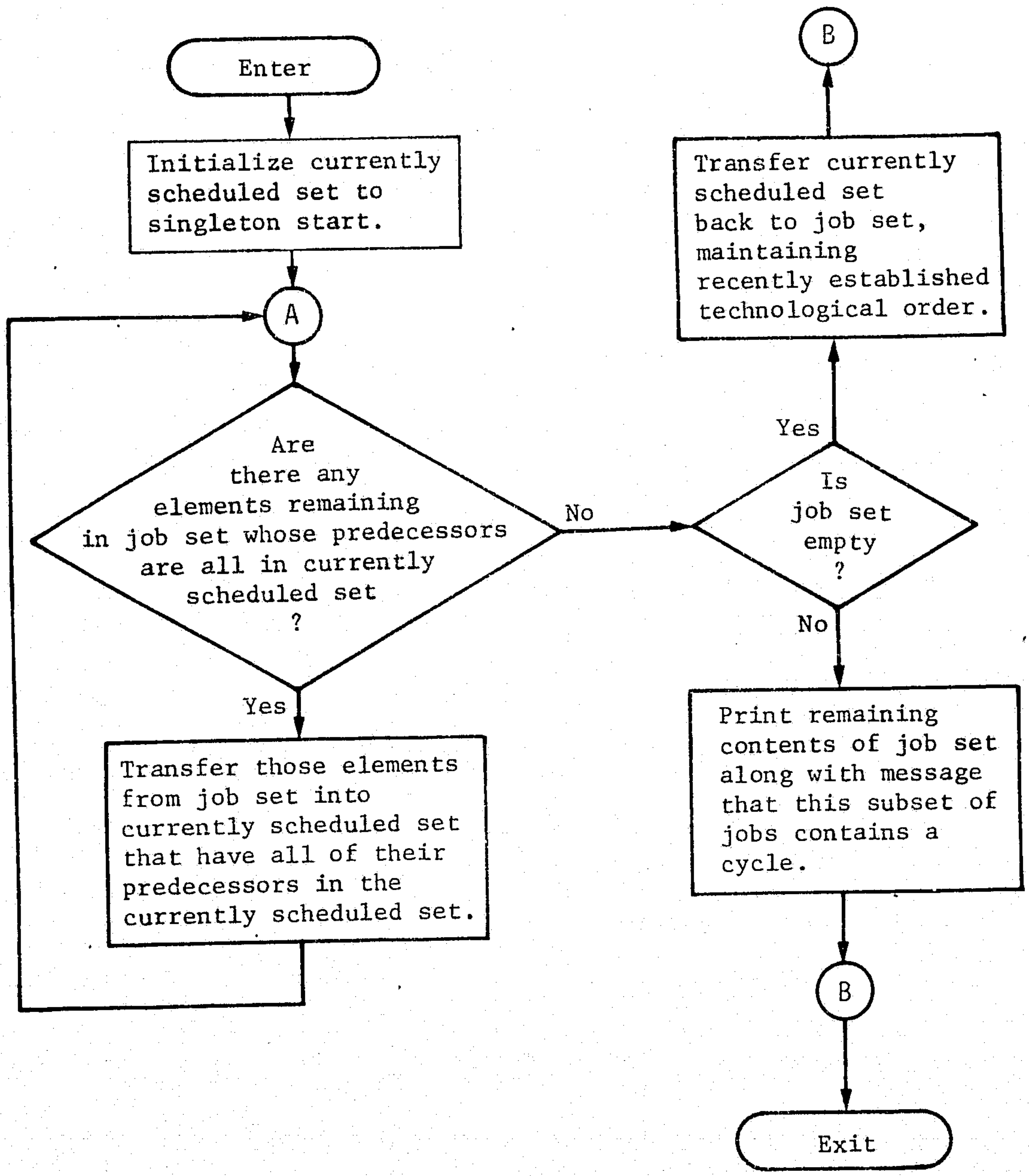
2) Subset of jobs containing cycles (if any exist) ($CYCLE_SET)



2.4.31.5 Functional Description

It can be shown that the activities and events of a project can be technologically ordered if, and only if, the precedence relations contain no cycles. It must be noted, however, that if cycles are absent, the technological ordering is by no means unique. The particular ordering produced by this module results from inductively "scheduling" in cycles all those activities or events whose predecessors are "scheduled." Eventually a cycle arises where there are no activities or events with all of their predecessors "scheduled." If some activities or events remain unscheduled, they contain a cycle. A more precise description of the logic of the module is provided in the functional block diagram.

2.4.31.6 Functional Block Diagram



2.4.31.7 Typical Application

The module is applied wherever a job set must be technologically ordered or wherever erroneous definition of the network may result in cycles that would invalidate further analysis. Examples of the former include the modules CRITICAL_PATH_CALCULATOR and REDUNDANT_PREDECESSOR_CHECKER. An example of the latter is the HEURISTIC_SCHEDULING_PRECESSOR.

2.4.31.8 Reference

Muth, John F. and Gerald L. Thompson, *Industrial Scheduling*, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1963.

2.4.31.9 DETAILED DESIGN

During the implementation of this module, it was determined that a restart capability could be added with very little extra logic. Since this provides an important service to the user, it has been incorporated in the code presented here.

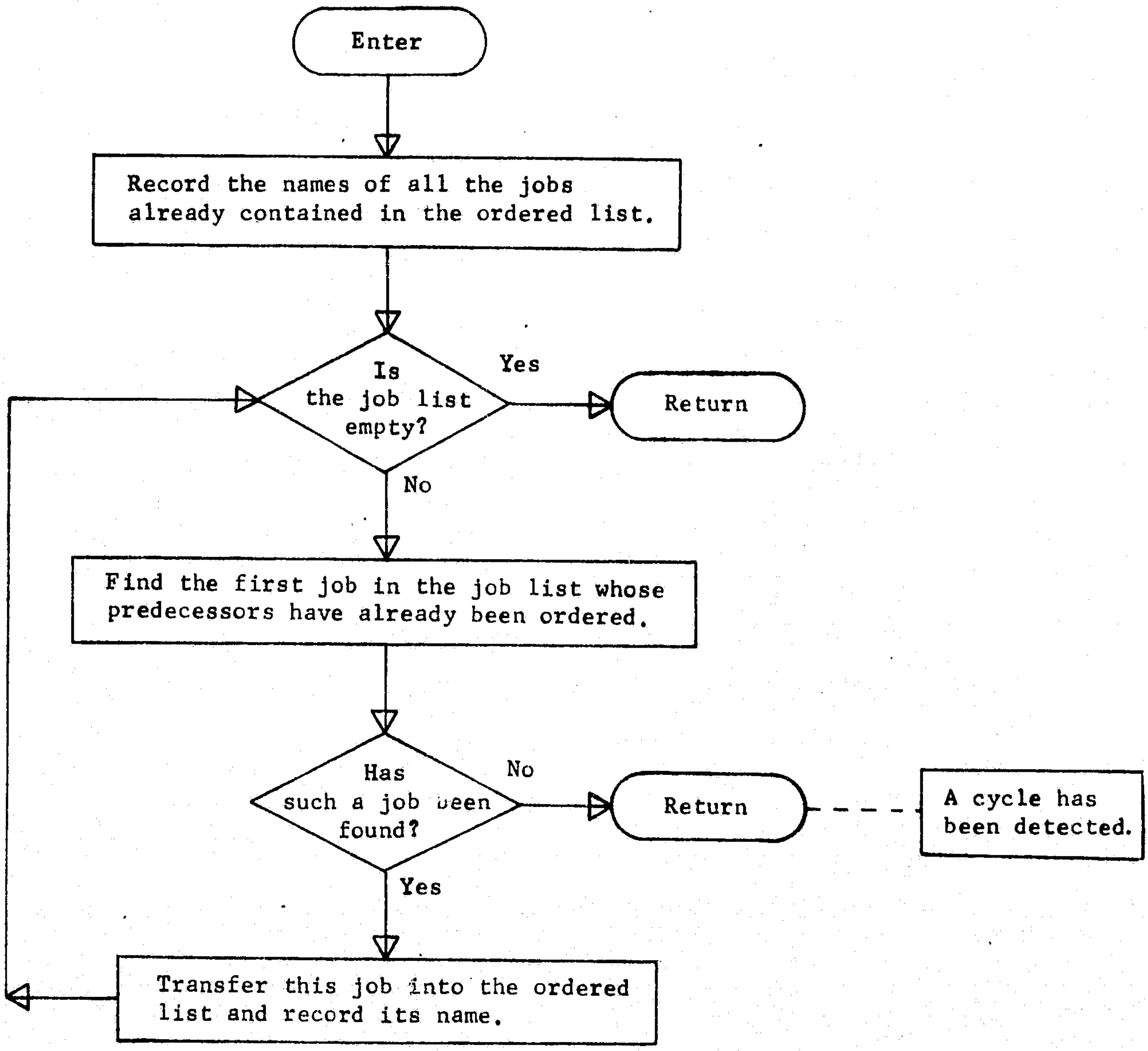
The module repeatedly searches through the list of jobs until it finds one whose predecessors are already in the technologically ordered list. This job is then transferred to the ordered list. This iterative process continues until all of the jobs have been ordered or until a cycle is detected in the remaining set of jobs.

The calling program should always check to make sure that $JOBLIST.FIRST is identical to $NULL before assuming that $ORDERED_LIST contains the complete set of jobs. If there are any jobs left in $JOBLIST after calling ORDER_BY_PREDECESSORS, the user can assume that they contain an input error.

2.4.31.10 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

$JOB	- Used to reference a subnode of $ORDERED_LIST
$JOBLIST	- Is the set of unordered jobs input by the calling program
$NAME_LIST	- Used to record the names of jobs that have already been ordered
$ORDERED_LIST	- Is the set of jobs that are technilogically ordered
$TEMP	- Is a temporary storage area

ORDER_BY_PREDECESSORS



2.4.31.11 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES

The functional definitions of the input and output parameters of this module have been changed. This was done to provide the user the ability to restart the ordering process by inputting an already-ordered list of jobs. The unordered jobs will then be added to this list.

If the user is working with large precedence networks, it is possible that his original specification of the network will contain several cycles. ORDER_BY_PREDECESSORS will identify these cycles (one per call) and also return a partially ordered list of jobs. Once the cycle has been eliminated, the user will need to call ORDER_BY_PREDECESSORS again. However, the jobs that were returned in the ordered list may be predecessors of those returned in $CYCLE_SET. If this module were implemented as originally specified, the above fact would force the user to recombine the ordered and unordered jobs, before calling the module again. This means that after detection and elimination of each cycle, ordering of the entire set of jobs would have to be reattempted. In order to avoid this cumbersome and wasteful process, a restart capability has been provided.

$JOBLIST is still used to input the list of unordered jobs, but is not used to return the list of ordered jobs. The general philosophy of the module is that as the jobs are ordered, they are transferred from $JOBLIST to the output tree, $ORDERED_LIST. This means that if a cycle is detected, the set of jobs containing the cycle will be returned in $JOBLIST, thus eliminating the need for $CYCLE_SET. This approach provides the restart capability by allowing the user to input an already-ordered set of jobs (in $ORDERED_LIST) onto which the jobs in $JOBLIST will be added.

The functions of $JOBLIST and $ORDERED_LIST are clearly separated. For purposes of both input and output, they will contain unordered and ordered lists of jobs, respectively.

2.4.31.12 COMMENTED CODE

```
ORDER_BY_PREDECESSORS:  PROCEDURE ($JOBLIST, $ORDERED_LIST)
                OPTIONS(EXTERNAL);
/*************************************************************************/
/*                                                                       */
/*   THIS MODULE TECHNOLOGICALLY ORDERS THE SET OF JOBS INPUT IN         */
/*   $JOBLIST AND RETURNS THEM IN $ORDERED_LIST.  $ORDERED_LIST CAN      */
/*   BE USED TO INPUT AN ALREADY-ORDERED SET OF JOBS ONTO WHICH THE      */
/*   JOBS CONTAINED IN $JOBLIST ARE TO BE ADDED.  IF A CYCLE IS DE-      */
/*   TECTED IN THE PRECEDENCE NETWORK, THE SUBSET OF JOBS CONTAINING     */
/*   THE CYCLE IS RETURNED IN $JOBLIST.                                  */
/*                                                                       */
/*************************************************************************/
   DECLARE $NAME_LIST,$JOB,$TEMP LOCAL ;
   DO FOR ALL SUBNODES OF $ORDERED_LIST USING $JOB ;
      INSERT LABEL($JOB) BEFORE $NAME_LIST(FIRST) ;
      END ;
   DO WHILE($JOBLIST(FIRST) NOT IDENTICAL TO $NULL) ;
      GRAFT $JOBLIST(FIRST:$ELEMENT.TEMPORAL_RELATION.PREDECESSOR
                                    SUBSET OF  $NAME_LIST) AT $TEMP ;
      IF $TEMP IDENTICAL TO $NULL  THEN RETURN ;
      $NAME_LIST(NEXT) = LABEL($TEMP) ;
      GRAFT $TEMP AT $ORDERED_LIST(NEXT) ;
      END ;
END;    /*   ORDER_BY_PREDECESSORS   */
```

2.4.32 RESOURCE_ALLOCATOR

2.4.32 RESOURCE_ALLOCATOR

2.4.32.1 Purpose and Scope

This module allocates resources to the various activities in a project to produce a schedule that satisfies all the resource constraints and heuristically minimizes the project's duration. What precisely is meant by a project is detailed in the purpose and scope section of the executive module HEURISTIC_SCHEDULING_PROCESSOR.

The scheduling heuristic takes a very pragmatic approach to the problem. It realistically assumes that the resource constraints are "soft." For example, additional labor or equipment can frequently be obtained by subcontracting or scheduling overtime. Further, within narrow limits resources can virtually be expanded by increasing the pace of the effort. To model the softness of the resource constraints, contingency threshold increments to the normal availability levels of critical resources are specified by the user. Then, whenever an activity cannot be scheduled by its critical path late start time, the current partial schedule is voided beyond the time that the resource-bound activity first had all of its predecessors completed. From that point onward, the schedule is rebuilt assuming that all of the critical resources that previously prevented scheduling are now available at their respective original normal levels plus their respective contingency threshold increments. When the previously resource-bound activity is finally scheduled the pool levels of its critical resources are returned to their normal levels (assuming the contingency increments are not simultaneously required for some other resource-bound activity).

The normal scheduling heuristic is a time progressive procedure employing the critical path late start time of each activity as a dynamic priority rule. Fortunately the late-start date of an activity does not depend on the actual start dates of its predecessors provided none of these is delayed beyond its late start date. If some activity is delayed beyond its late start date by an interval "a," the project completion date is slipped by "a" time units and, hence, the late starts of any activity with respect to any other is unaltered. Hence, the priority function does not require updating each time a new activity is added to the partial schedule. The procedure basically steps through process time scheduling those activities whose predecessors have all been completed in order of their late-start priority and within the existing resource availabilities. The contingency resource increments and the associated rescheduling can be viewed as a modifying heuristic on the earliest late-start date priority rule.

The RESOURCE_ALLOCATOR serves as the forward-pass segment of the combined forward- and backward-pass heuristic resource allocating procedure called the HEURISTIC_SCHEDULING_PROCESSOR. It produces a tentative front-loaded resource allocation together with its associated practical estimate of the project duration. Preserving this estimated minimum project duration, the RESOURCE_LEVELER delays activities in the tentative schedule within the limits of their residual slack to produce heuristically the most level resource-loaded schedule.

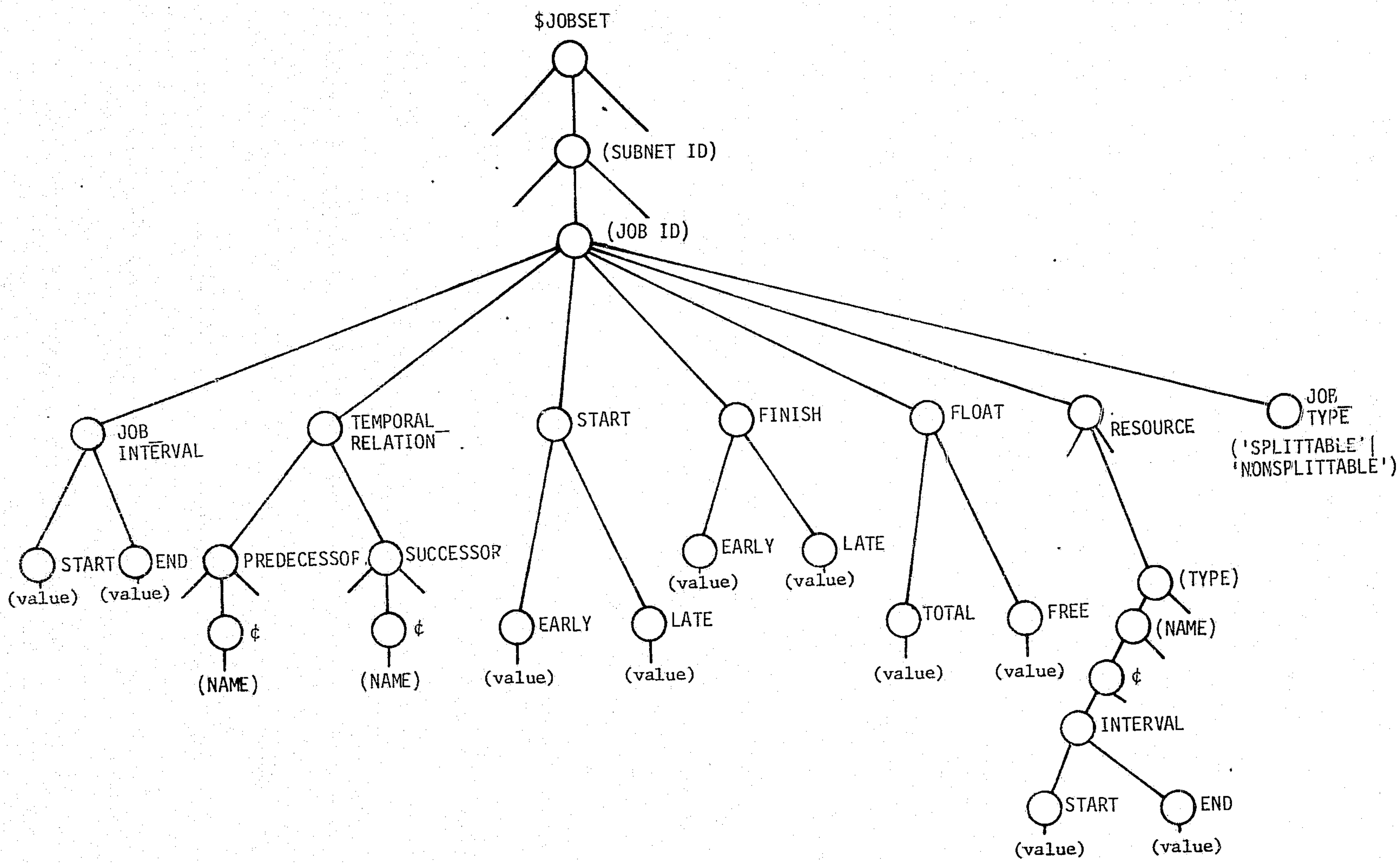
2.4.32.2 Modules Called

None

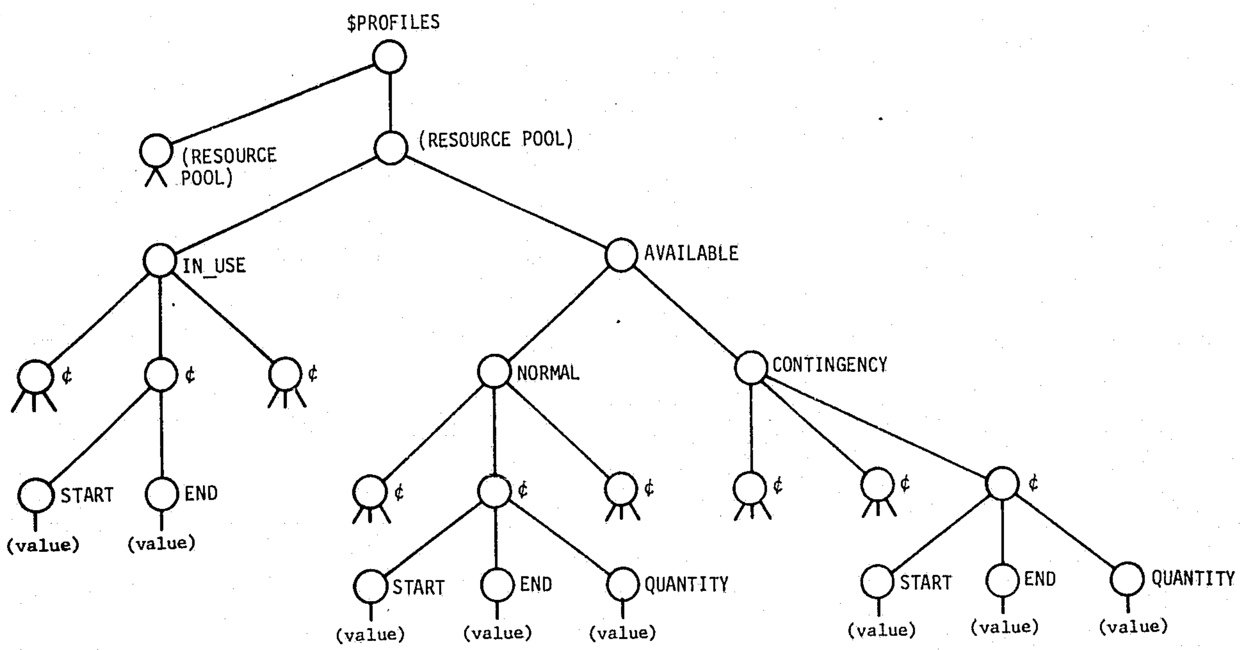
2.4.32.3 Module Input

1) Network, Critical Path Data and Activity or Event Definitions

$JOBSET

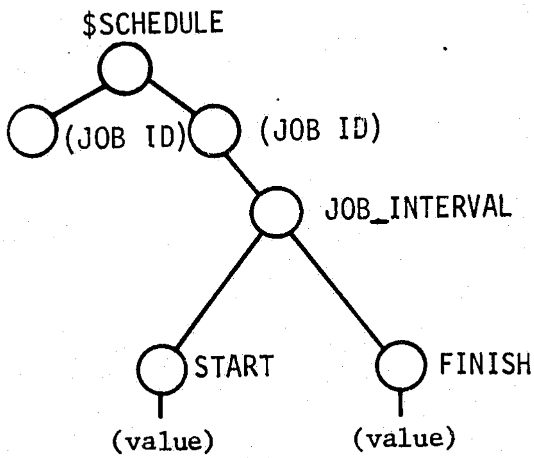

$JOBSET
(SUBNET ID)
(JOB ID)
JOB_ INTERVAL
START
(value)
END
(value)
TEMPORAL_ RELATION
PREDECESSOR
¢
(NAME)
SUCCESSOR
¢
(NAME)
START
EARLY
(value)
LATE
(value)
FINISH
EARLY
(value)
LATE
(value)
FLOAT
TOTAL
(value)
FREE
(value)
RESOURCE
(TYPE)
(NAME)
¢
INTERVAL
START
(value)
END
(value)
JOB_ TYPE
('SPLITTABLE' | 'NONSPLITTABLE')

2) Resource Definitions ($PROFILES)

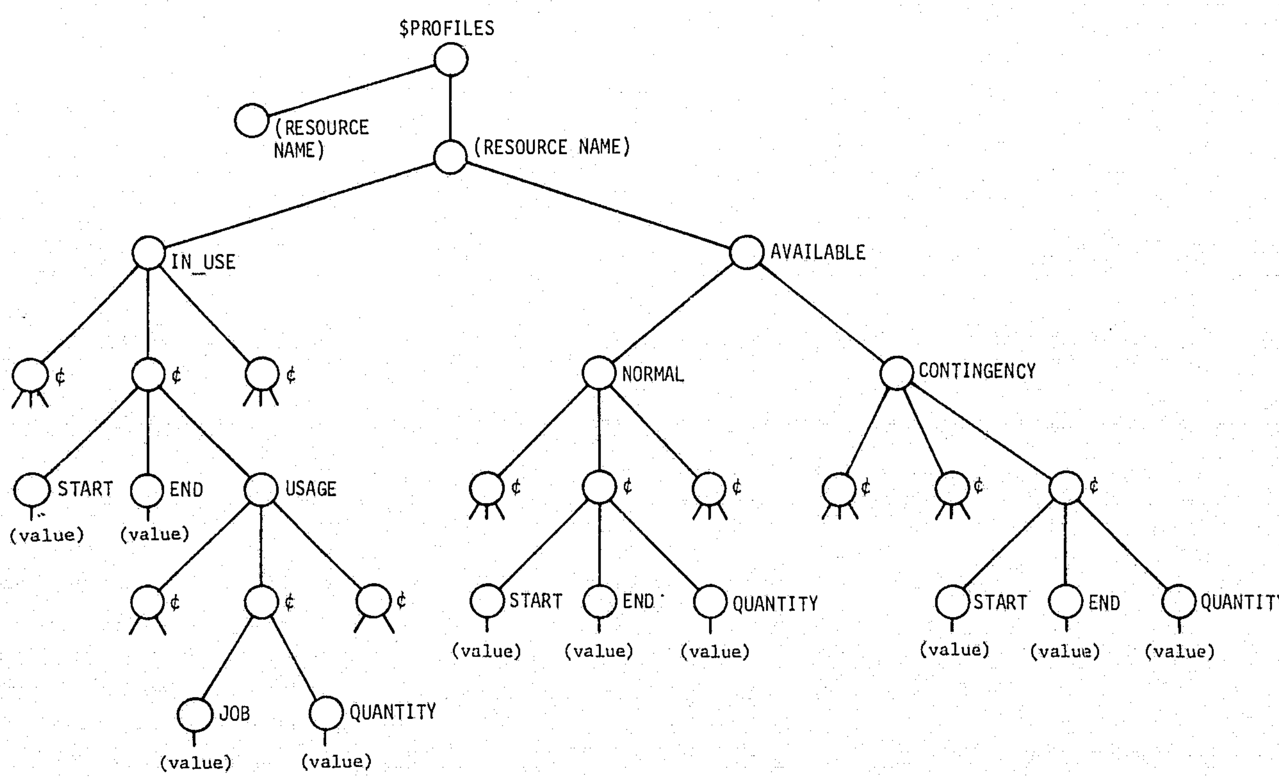


2.4.32.4 Module Output

1) Resulting Heuristic Schedule ($SCHEDULE)



2) Revised Resource Profile Including Usage ($PROFILES)



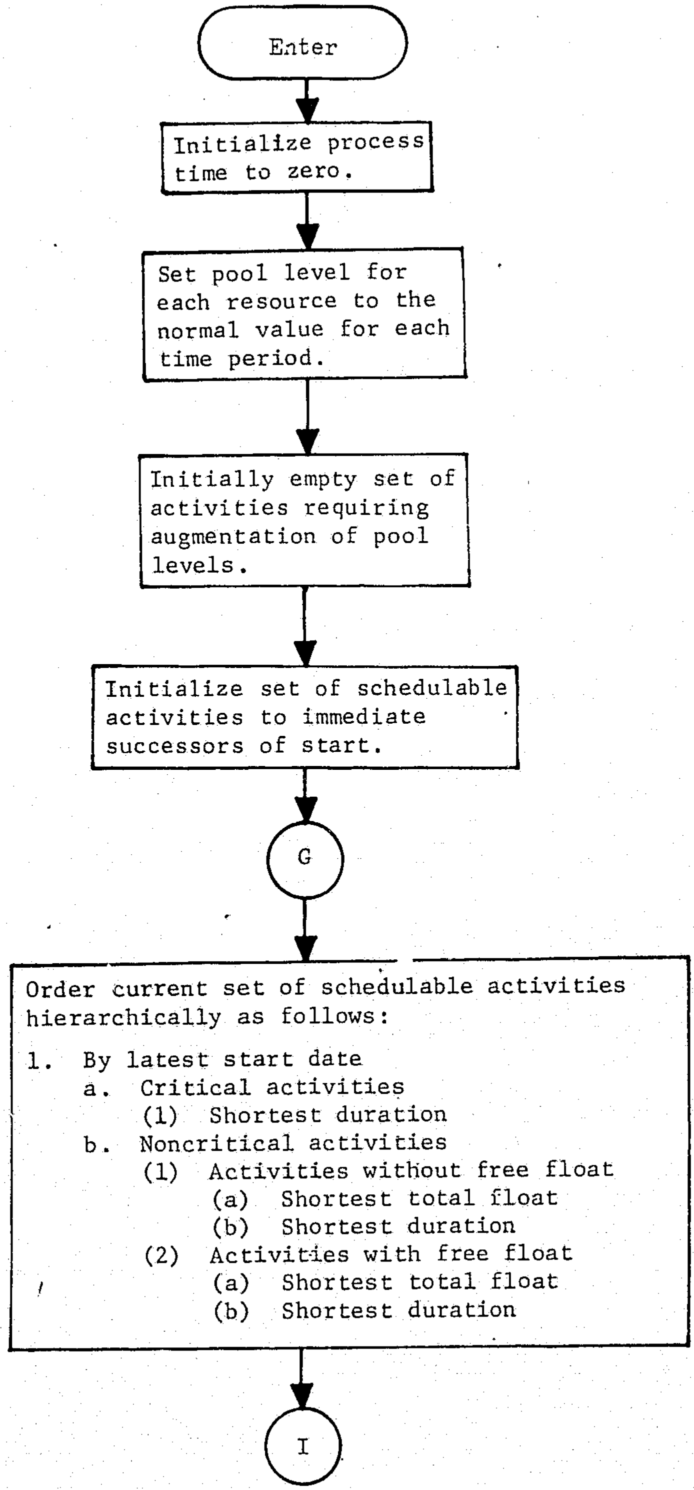
2.4.32.5 Functional Description

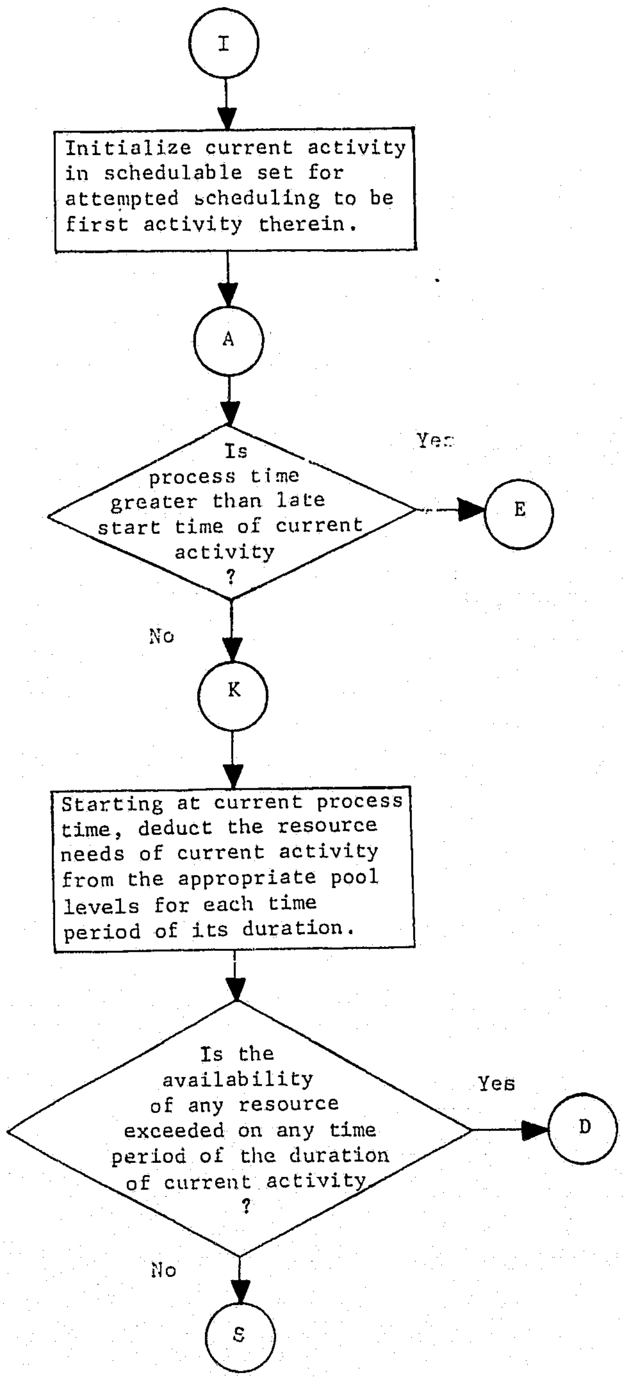
The RESOURCE_ALLOCATOR uses the policy of scheduling activities as soon after their predecessors are completed as resources become available and their priority warrants. Categorically, scheduling activities as soon as possible does indeed tend to produce minimum duration projects, but only at the expense of heavily front loaded resource utilization profiles. Unfortunately, however, it is usually desirable for reasons of economy to utilize resources at as constant a rate as possible. To achieve level resource utilization, the resulting schedule from the RESOURCE_ALLOCATOR is passed to the RESOURCE_LEVELER module. This routine delays jobs within their residual float to level out resource utilization while maintaining the project duration of the original schedule.

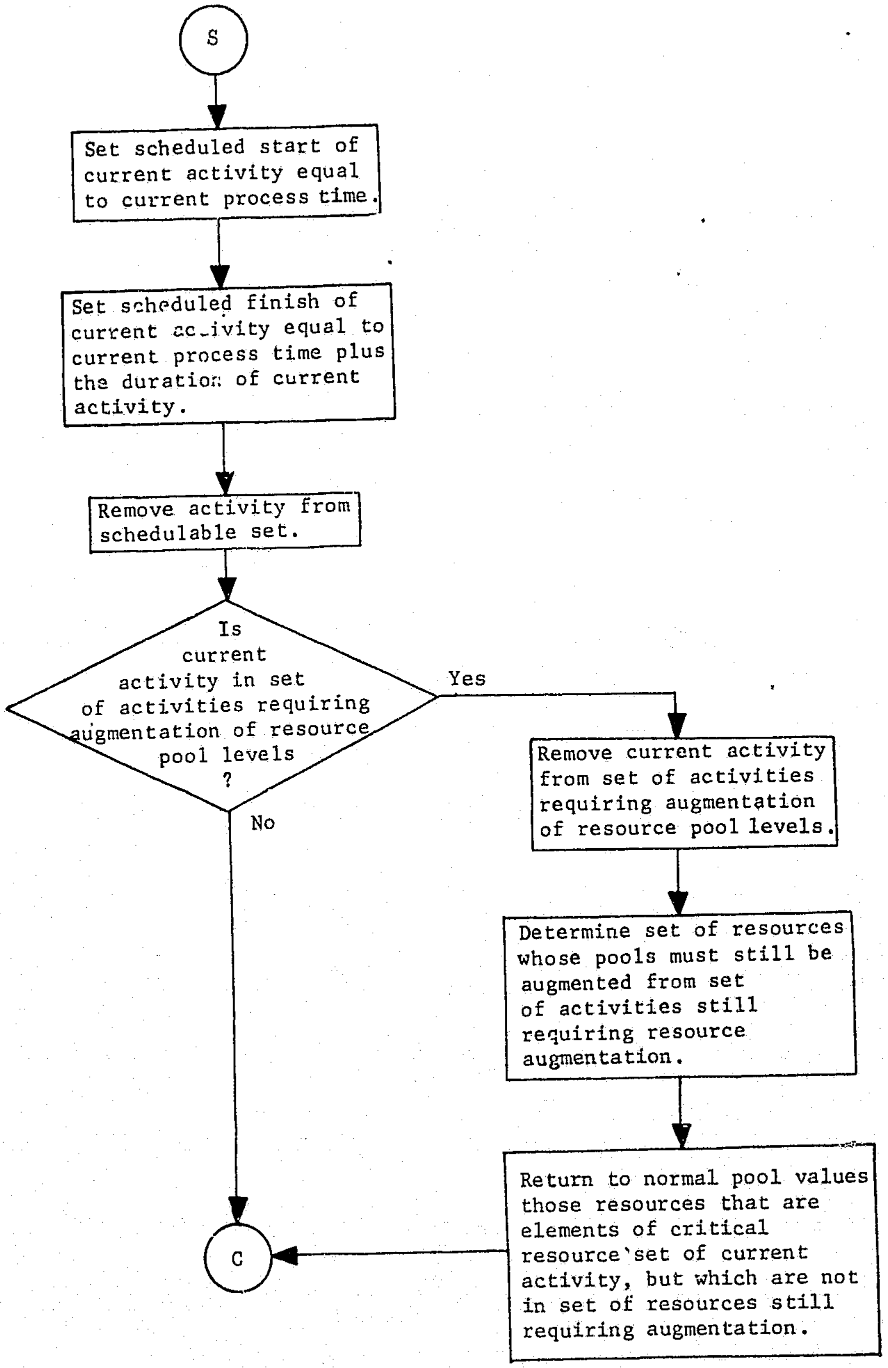
The scheduler tends to think chronologically beginning with the project start date. Hence, in imitating him, it is natural to use a time progressive heuristic. On any given day in his chronological scheduling effort, the next activity to be scheduled is that job whose predecessors are all complete and that is most likely to be slipped beyond its late-start date. Only the resource constraints can cause an activity to be so slipped because the precedence constraints are automatically satisfied by the set of critical-path early-start times for the resource-unconstrained situation. Nevertheless the interaction of the resource constraints with the precedence constraints can be extremely complicated. For example, a set of minor delays occasioned by various resource constraints can accumulate sufficiently over the precedence network to slip a subsequent activity beyond its late start time and, hence, delay the project. Modeling the resource constraints in the heuristic procedure is, therefore, extremely difficult. It has been attempted by several investigators but the results have never justified the complications. Thus the most reasonable approach appears to be resource bound. In this case, then, the most likely precedence-constraint-free candidate for slipping the project completion is that activity among those with completed predecessors that has the earliest late-start date.

Finally, if the same activity does slip its late-start date thereby delaying project completion, the scheduler considers enacting contingency measures such as obtaining more labor and equipment through subcontracting or scheduling overtime. The heuristic allocator takes the same approach, adding contingency threshold increments to the critical resource pools, binding the tardy activity beginning at the time its predecessors were first completed, the rescheduling the entire project from that point onward. When the resource-bound activity is finally successfully scheduled, the pool levels of the critical resources are returned to their normal values.

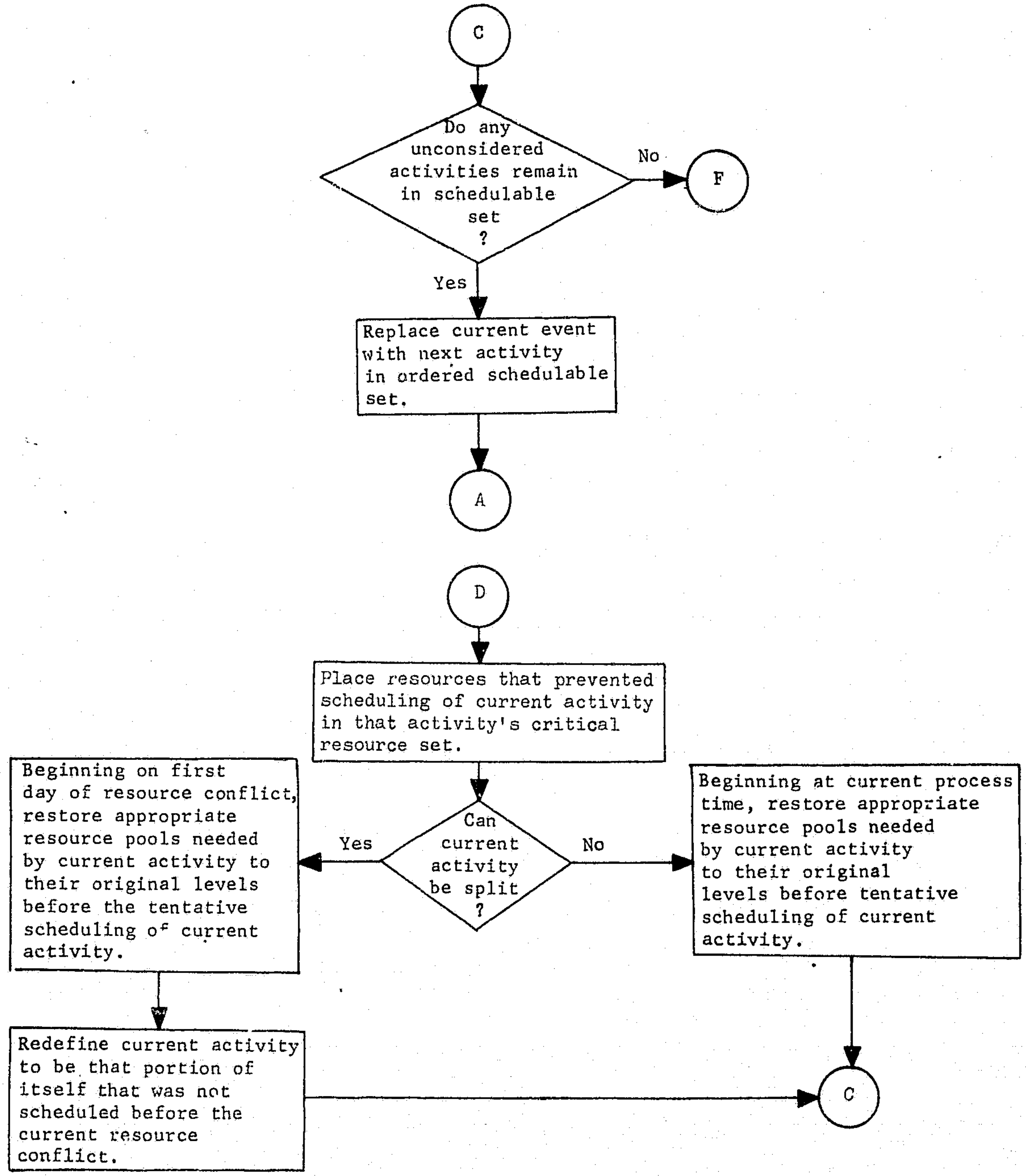
2.4.32.6 Functional Block Diagram

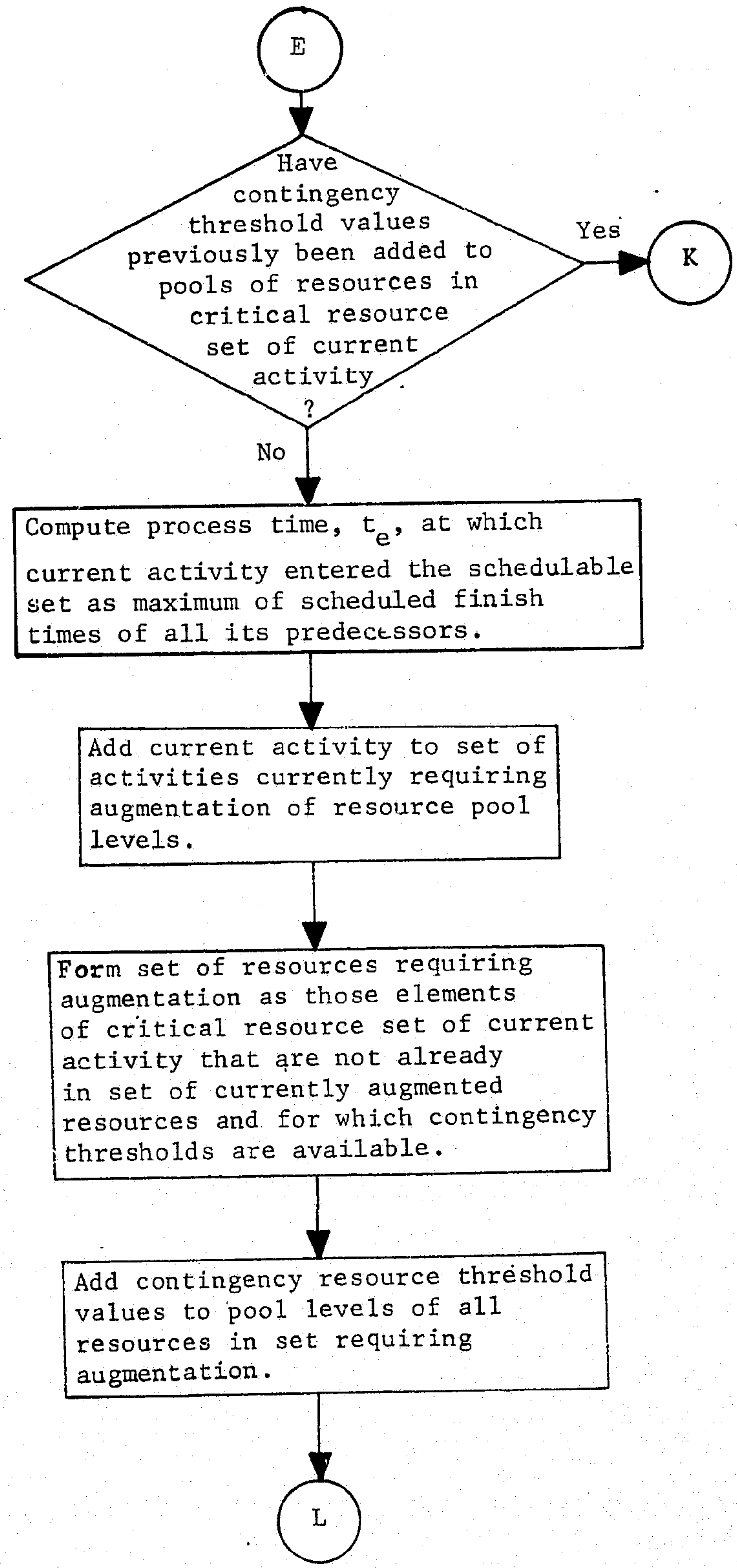


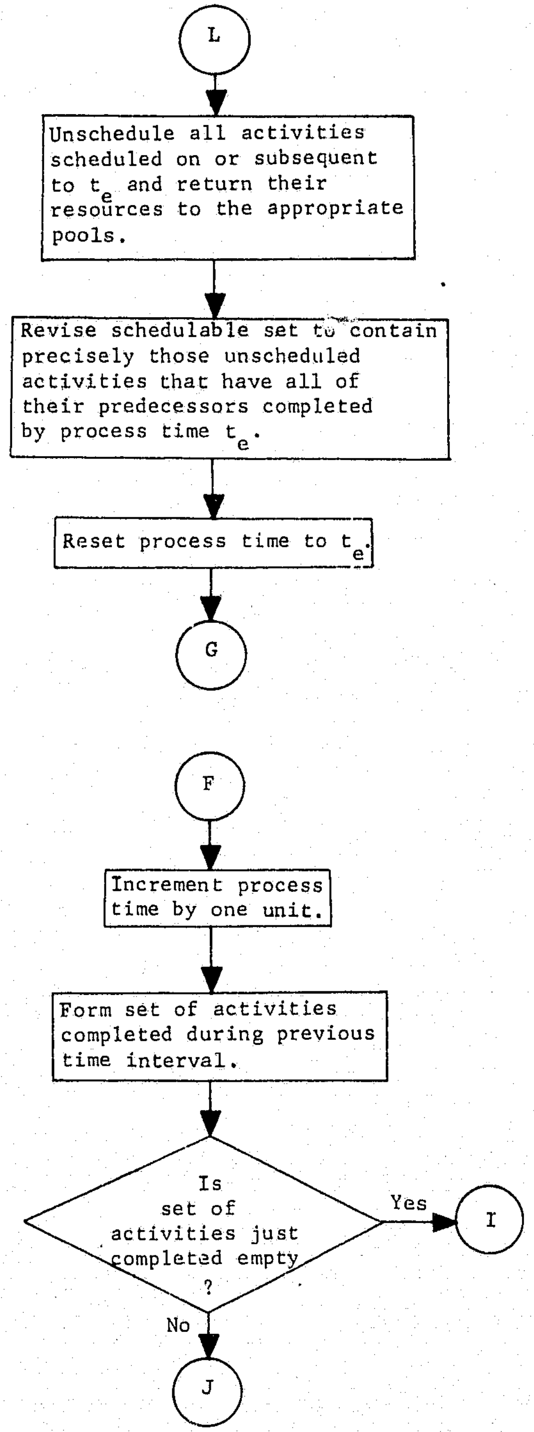

I
Initialize current activity in schedulable set for attempted scheduling to be first activity therein.
A
Is process time greater than late start time of current activity ?
Yes
E
No
K
Starting at current process time, deduct the resource needs of current activity from the appropriate pool levels for each time period of its duration.
Is the availability of any resource exceeded on any time period of the duration of current activity ?
Yes
D
No
S

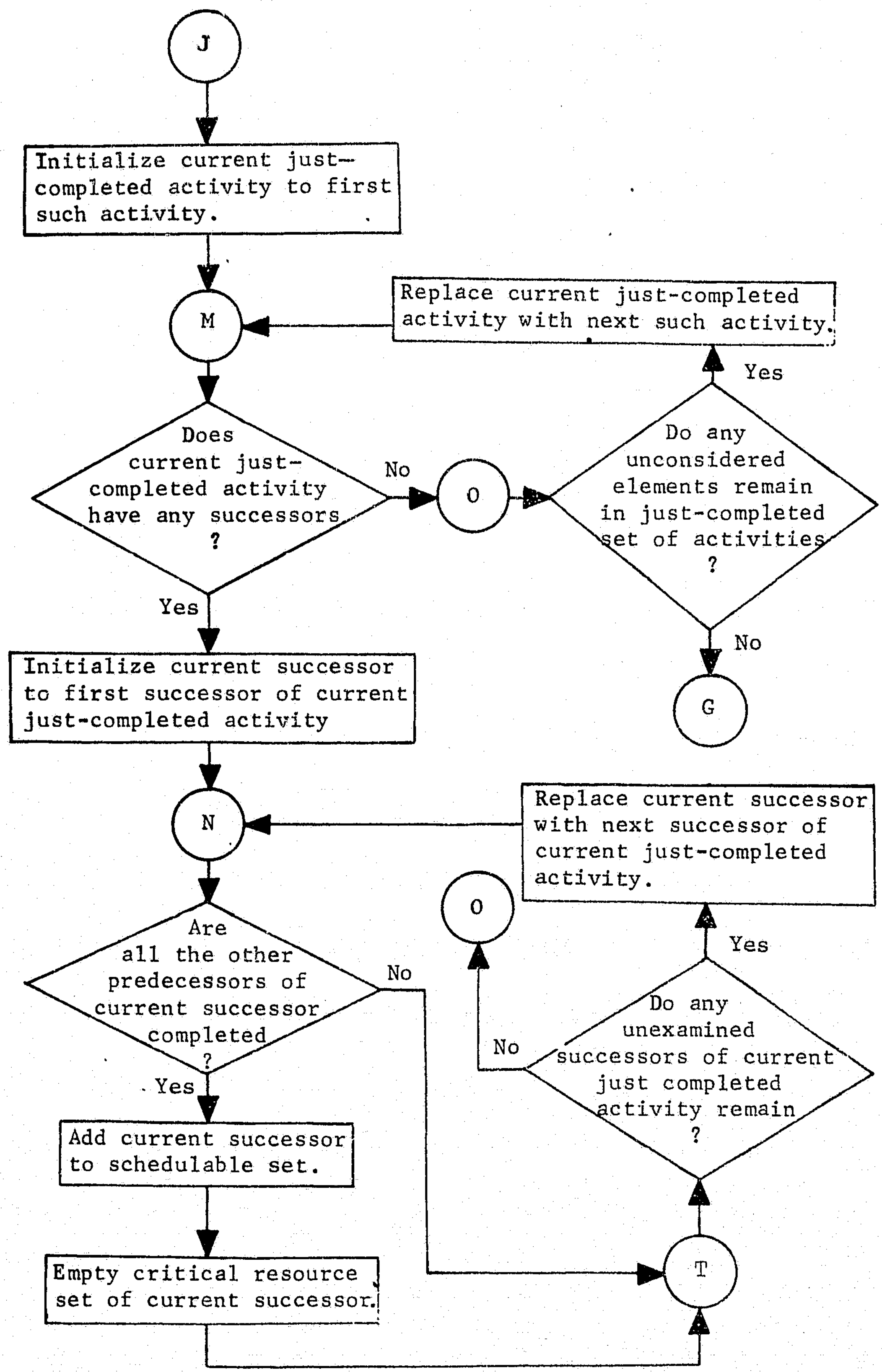


ORIGINAL PAGE IS
OF POOR QUALITY


C
Do any unconsidered activities remain in schedulable set ?
No
F
Yes
Replace current event with next activity in ordered schedulable set.
A
D
Place resources that prevented scheduling of current activity in that activity's critical resource set.
Can current activity be split ?
Yes
No
Beginning on first day of resource conflict, restore appropriate resource pools needed by current activity to their original levels before the tentative scheduling of current activity.
Beginning at current process time, restore appropriate resource pools needed by current activity to their original levels before tentative scheduling of current activity.
Redefine current activity to be that portion of itself that was not scheduled before the current resource conflict.
C


E
Have contingency threshold values previously been added to pools of resources in critical resource set of current activity ?
Yes
K
No
Compute process time, t_e, at which current activity entered the schedulable set as maximum of scheduled finish times of all its predecessors.
Add current activity to set of activities currently requiring augmentation of resource pool levels.
Form set of resources requiring augmentation as those elements of critical resource set of current activity that are not already in set of currently augmented resources and for which contingency thresholds are available.
Add contingency resource threshold values to pool levels of all resources in set requiring augmentation.
L

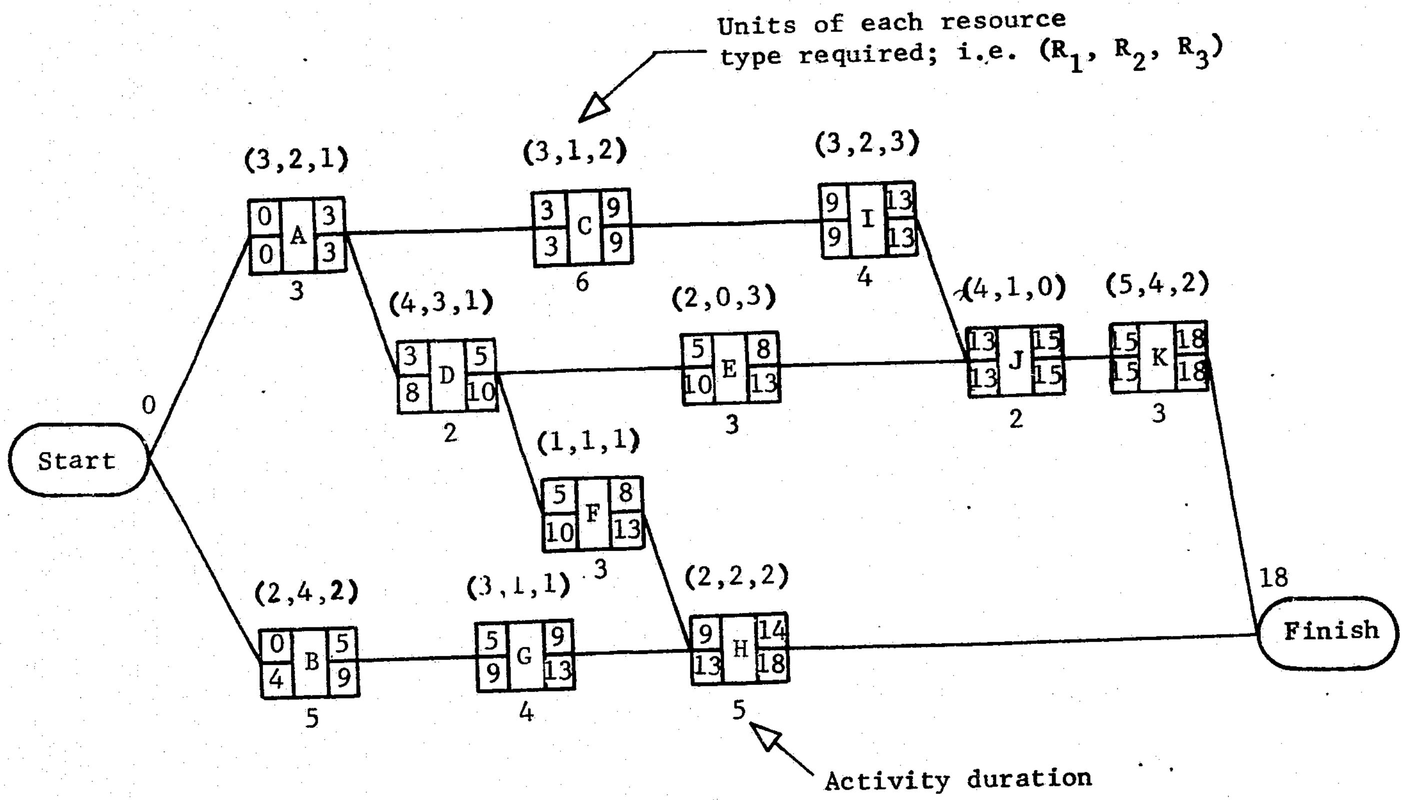

L
Unschedule all activities scheduled on or subsequent to t_e and return their resources to the appropriate pools.
Revise schedulable set to contain precisely those unscheduled activities that have all of their predecessors completed by process time t_e.
Reset process time to t_e.
G
F
Increment process time by one unit.
Form set of activities completed during previous time interval.
Is set of activities just completed empty ?
Yes
I
No
J


J
Initialize current just-completed activity to first such activity.
M
Replace current just-completed activity with next such activity.
Yes
Does current just-completed activity have any successors ?
No
O
Do any unconsidered elements remain in just-completed set of activities ?
Yes
No
G
Initialize current successor to first successor of current just-completed activity
N
Replace current successor with next successor of current just-completed activity.
O
Yes
Are all the other predecessors of current successor completed ?
No
No
Do any unexamined successors of current just completed activity remain ?
Yes
Add current successor to schedulable set.
Empty critical resource set of current successor.
T

2.4.32.7 Typical Application

The forward-pass RESOURCE_ALLOCATOR module can be used in conjunction with the backward-pass RESOURCE_LEVELER wherever a short duration and level-resource profile schedule for a project is desired. The heuristics involved are sophisticated enough to give a practical schedule, but simple enough to allow rapid execution. Thus, parametric runs on normal and contingency resource levels can be executed to arrive at a highly desirable schedule.

Because the logic involved in the RESOURCE_ALLOCATOR is somewhat involved, an illustrative example is presented. Consider the project shown in Fig. 2.4.32-1. This problem is taken from Davis' (Davis, 74) survey of resource allocation procedures. Although it is rather small to be representative of practical projects, it is of interest in that the optimal solution is available via the algorithm of Davis and Heidorn (Davis and Heidorn, 1971). Figure 2.4.32-2 is a detailed trace of the execution of the RESOURCE_ALLOCATOR logic for the illustrated resource-constrained project. Sufficient detail is presented to allow the reader to verify his understanding of the algorithm's logical flow as presented in the functional block diagram. Next, Fig. 2.4.32-3 presents the details of the resulting 18-day schedule. Contingency resource pool increments of 2 and 1 were available for the first and third resources, respectively.Figure 2.4.32-4 contains the details of an optimal 20-day schedule generated by the Davis and Heidorn algorithm when resource contingency thresholds are not allowed.



Resource Limits:

R_1 = 6 men

R_2 = 7 men

R_3 = 6 men

Fig. 2.4.32-1
Constrained-Resource Problem with Three Resource Types

Fig. 2.4.32-2

ACTIVITIES									
ACTIVITY	DURATION	LATEST START	RESOURCE REQUIREMENTS			CYCLE OF ENTRY	SCHEDULED START	SCHEDULED FINISH
A	3	0	3	2	1	1	0	3
B	5	4	2	4	2	1	0	5
C	6	3	3	1	2	2	3	9
D	2	8	4	3	1	2	5	7
E	3	10	2	0	3	4	7	10
F	3	10	1	1	1	4	10	13
G	4	9	3	1	1	3	7	11
H	5	13	2	2	2	6	13	18
I	4	9	3	2	3	5	9	13
J	2	13	4	1	0	6	13	15
K	3	15	5	4	2	7	15	18

CYCLES			
CYCLE	PROCESS TIME	SET	
1	0	A	B
	1		
	2		
2	3	C	D
	4	D	
3	5	D	G
	6	D	
	7	D	
	8	D	

RESOURCES						
DAY	RESOURCES AVAILABLE			NORMAL ALLOCATION	CONTINGENCY ALLOCATION	
1	~~6~~	~~3~~	1	6	0	
	7	~~5~~	1	7	0	
	~~6~~	~~5~~	3	6	0	
2	~~6~~	~~3~~	1	6	0	
	7	~~5~~	1	7	0	
	~~6~~	~~5~~	3	6	0	
3	~~6~~	~~3~~	1	6	0	
	7	~~5~~	1	7	0	
	~~6~~	~~5~~	3	6	0	
4	~~6~~	~~4~~	1	6	0	Reschedule to Expedite D (days 4–9)
	7	~~3~~	2	7	0	
	~~6~~	~~4~~	2	6	0	
5	~~6~~	~~4~~	1	6	0	
	7	~~3~~	2	7	0	
	~~6~~	~~4~~	2	6	0	
6	~~6~~	~~3~~	0	6	0	
	7	~~6~~	5	7	0	
	~~6~~	4	3	6	0	
7	~~6~~	~~3~~	0	6	0	
	7	~~6~~	5	7	0	
	~~6~~	~~4~~	3	6	0	
8	~~6~~	~~3~~	0	6	0	
	7	~~6~~	5	7	0	
	~~6~~	~~4~~	3	6	0	
9	~~6~~	~~3~~	0	6	0	
	7	~~6~~	5	7	0	
	~~6~~	~~4~~	3	6	0	

Fig. 2.4.32-2
Trace of the Execution of the RESOURCE_ALLOCATOR *Algorithm on the Constrained-Resource Problem Shown in Fig. 2.4.32-1, Using Contingency Resource Thresholds on the First and Third Resources, Respectively*

CYCLES			RESOURCES						
CYCLE	PROCESS TIME	SET	DAY	RESOURCES AVAILABLE			NORMAL ALLOCATION	CONTINGENCY ALLOCATION	Remarks
	9	I H		6			6	0	Continuation of rescheduling of activity D
				7			7	0	
				6			6	0	
2	3	D	4	~~8~~	~~6~~	3	6	2	
				7	~~3~~	2	7	0	
				~~6~~	~~4~~	2	6	0	
	4	D	5	~~8~~	~~6~~	3	6	2	
				7	~~3~~	2	7	0	
				~~6~~	~~4~~	2	6	0	
3	5	D G	6	~~8~~	~~5~~	1	6	2	
				7	~~6~~	3	7	0	
				~~6~~	~~4~~	3	6	0	
	6	G	7	~~8~~	~~5~~	1	6	2	
				7	~~6~~	3	7	0	
				~~6~~	~~4~~	3	6	0	
4	7	G E F	8	~~6~~	~~3~~	0	6	0	Reschedule to Expedite E
				7	~~6~~	5	7	0	
				~~6~~	~~4~~	3	6	0	
	8	E F	9	~~6~~	~~3~~	0	6	0	
				7	~~6~~	5	7	0	
				~~6~~	~~4~~	3	6	0	
5	9	I E F	10	~~6~~	~~3~~	0	6	0	
				7	~~6~~	4	7	0	
				~~6~~	~~5~~	2	6	0	
	10	E F	11	~~6~~	~~3~~	0	6	0	
				7	~~6~~	4	7	0	
				~~6~~	~~5~~	2	6	0	
	11		12	~~6~~	3		6	0	
				7	5		7	0	
				~~6~~	3		6	0	

Fig. 2.4.32-2 (cont)

CYCLES		
CYCLE	PROCESS TIME	SET
	12	
4	7	G E F
	8	F
5	9	I F
	10	F
	11	
	12	
	10	F
	11	
	12	

RESOURCES						
DAY	RESOURCES AVAILABLE				NORMAL ALLOCATION	CONTINGENCY ALLOCATION
13	~~6~~	3			6	0
	7	5			7	0
	~~6~~	3			6	0
8	~~8~~	~~5~~	~~2~~	0	6	2
	7	~~6~~	~~5~~	5	7	0
	7	~~5~~	~~4~~	1	6	1
9	~~8~~	~~5~~	~~2~~	0	6	2
	7	~~6~~	~~5~~	5	7	0
	7	~~5~~	~~4~~	1	6	1
10	~~8~~	~~5~~	~~3~~	0	6	2
	7	~~6~~	~~6~~	4	7	0
	7	~~6~~	~~3~~	0	6	1
11	~~6~~	~~3~~	0		6	0
	7	~~6~~	4		7	0
	~~6~~	~~5~~	2		6	0
12	~~6~~	3			6	0
	7	5			7	0
	~~6~~	3			6	0
13	~~6~~	3			6	0
	7	5			7	0
	~~6~~	3			6	0
11	~~8~~	~~5~~	~~2~~	1	6	2
	7	~~6~~	~~4~~	3	7	0
	7	~~6~~	~~3~~	2	6	1
12	~~8~~	~~5~~	4		6	2
	7	~~5~~	4		7	0
	7	~~4~~	3		6	1
13	~~8~~	~~5~~	4		6	2
	7	~~5~~	4		7	0
	7	~~4~~	3		6	1

Reschedule to Expedite F (days 11–13, rows bracketed)

Fig. 2.4.32-2 (cont)

CYCLES		
CYCLE	PROCESS TIME	SET
6	13	J H
	14	
7	15	K
	16	
	17	
7	15	K
	16	
	17	

RESOURCES					
DAY	RESOURCES AVAILABLE			NORMAL ALLOCATION	CONTINGENCY ALLOCATION
14	~~6~~	~~2~~	0	6	0
	7	~~6~~	4	7	0
	~~6~~	~~6~~	4	6	0
15	~~6~~	~~2~~	0	6	0
	7	~~6~~	4	7	0
	~~6~~	~~6~~	4	6	0
16	~~6~~	4		6	0
	7	5		7	0
	~~6~~	4		6	0
17	~~6~~	4		6	0
	7	5		7	0
	~~6~~	4		6	0
18	~~6~~	4		6	0
	7	5		7	0
	~~6~~	4		6	0
16	~~8~~	~~6~~	1	6	2
	7	~~5~~	1	7	0
	~~6~~	~~4~~	2	6	0
17	~~8~~	~~6~~	1	6	2
	7	~~5~~	1	7	0
	~~6~~	~~4~~	2	6	0
18	~~8~~	6	1	6	2
	7	5	1	7	0
	~~6~~	4	2	6	0

Reschedule to Expedite K (days 16–18, first set)

Fig. 2.4.32-2 (concl)

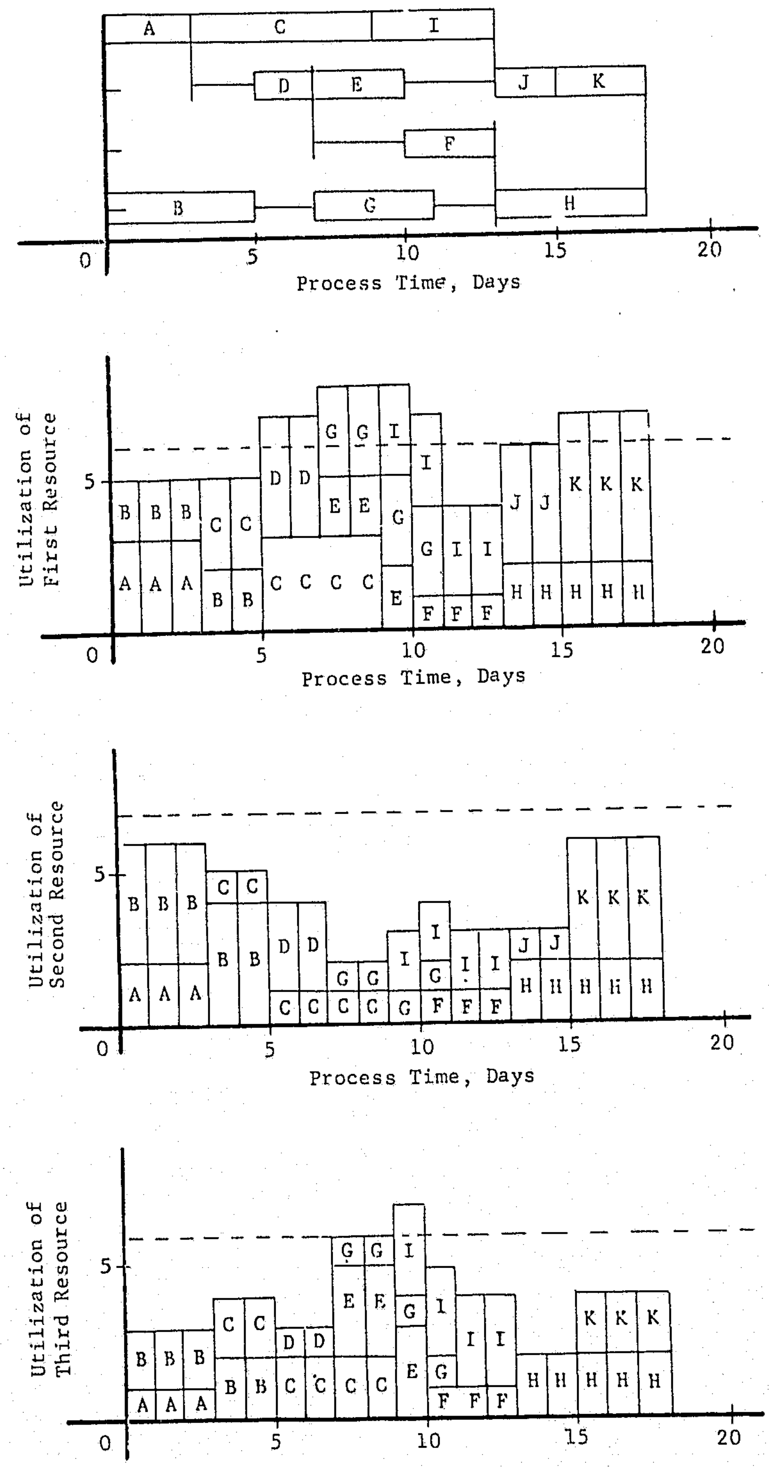


Fig. 2.4.32-3
RESOURCE_ALLOCATOR *Solution to Constrained-Resource Problem Using Resource Contingency Levels of 2, 0, and 1, Respectively*

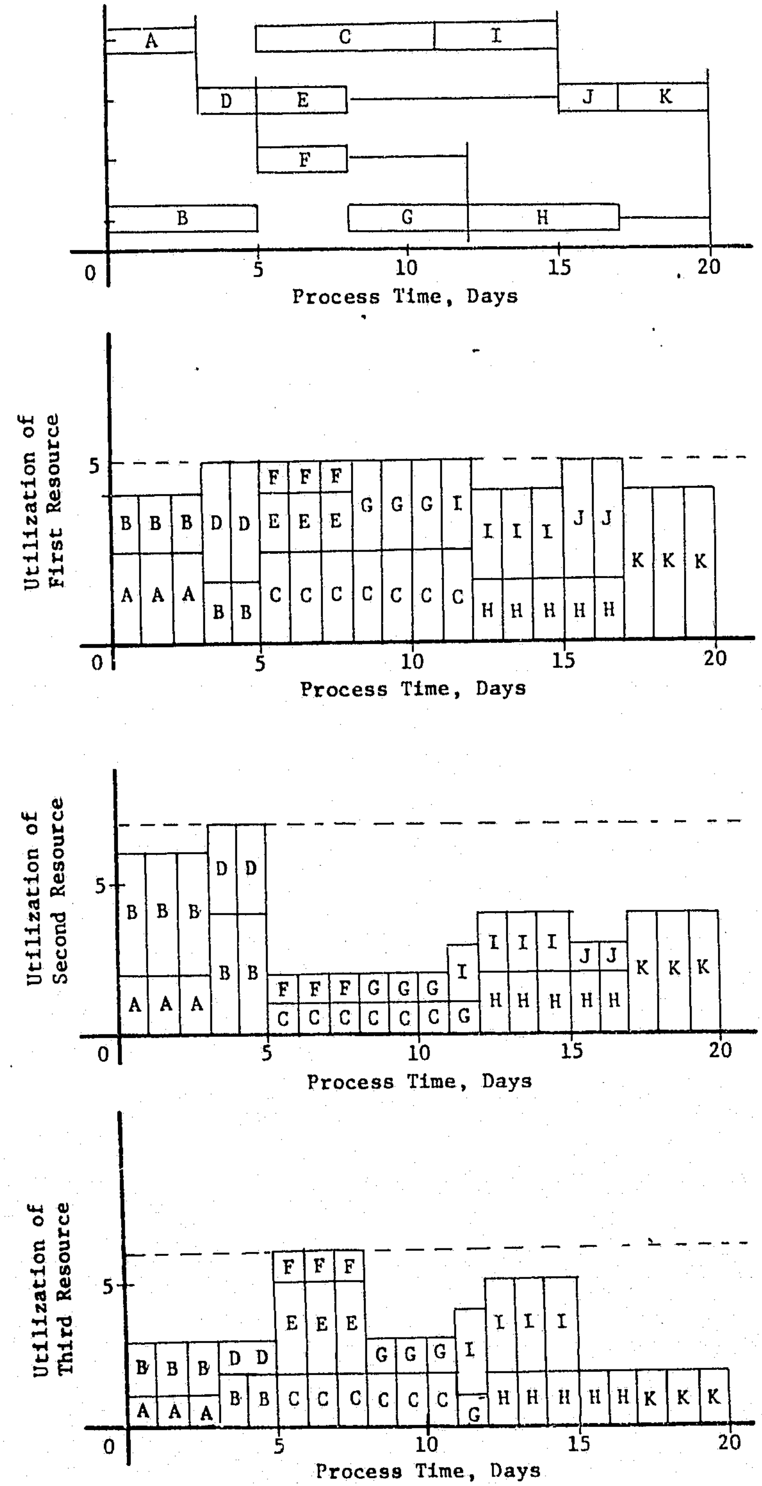


Fig. 2.4.32-4
Minimum Duration Solution to Constrained-Resource Problem Using No Resource Contingency Levels

The optimal schedule requires two more days than the contingency-resource schedule. Which schedule is suprior depends on the availability of supplemental resource units; that is, on the "hardness" of the resource constraints. It is obvious that the optimal schedule is superior to the 25-day RESOURCE_ALLOCATOR schedule generated assuming no resource contingency levels, as shown in Fig. 2.4.32-5. Thus, it is apparent that the simple priority rule scheduling of the RESOURCE_ALLOCATOR, which is in force when no resource thresholds are present, is greatly enhanced by the modifying heuristic that invokes contingency resources when an activity's late-start date is slipped. Finally, it should be noted that by executing a series of parametric **runs** with varying resource contingency thresholds, a thorough analysis of the tradeoff between project duration and resource availability can be made.

2.4.32.8 References

Davis, Edward W. and Heidorn, George E., "An Algorithm for Optimal Project Scheduling under Multiple Resource Constraints", *Management Science*, August 1971.

Davis, Edward W., "Networks: Resource Allocation", *Journal of Industrial Engineering*, April 1974.

Burman, P. J.: *Precedence Networks for Project Planning and Control*. McGraw Hill, London, 1972.

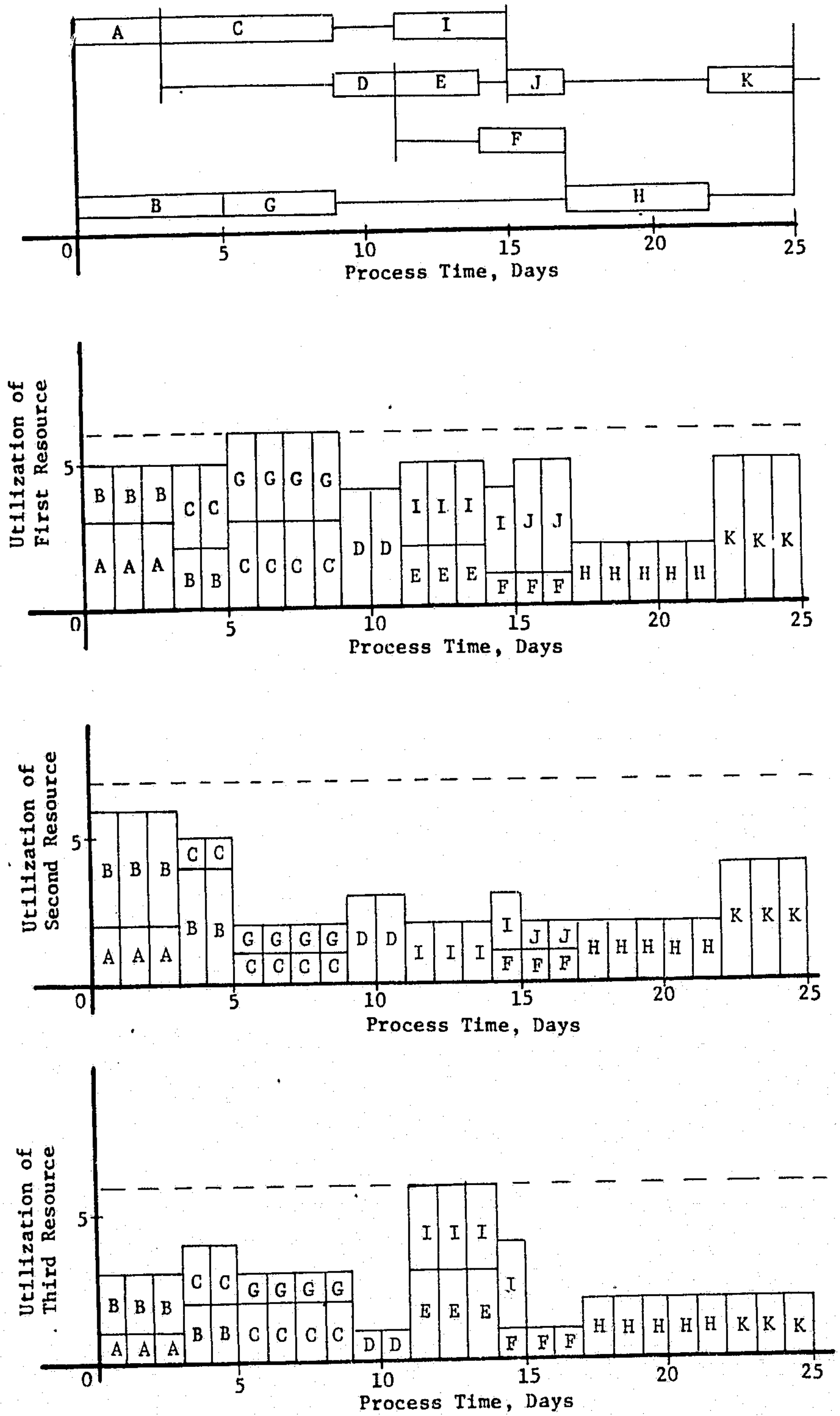


Fig. 2.4.32-5
RESOURCE_ALLOCATOR *Solution to Constrained-Resource Problem Using No Resource Contingency Levels*

2.4.32.9 DETAILED DESIGN

This module is an implementation of a time-progressive project scheduling heuristic that produces a near-minimum duration schedule that satisfies all resource constraints. On input, $JOBLIST will contain all of the information about the set of jobs to be scheduled including: resource requirements, temporal relation constraints and critical path data. $PROFILES will contain a description of each resource pool, including time and quantity constraints. The output $SCHEDULE will contain the set of jobs along with their start and end times. $PROFILES is updated to reflect the resource allocations performed during the scheduling process.

The timeliner steps through time, scheduling activities from a prioritized list at each time point. The activities are considered eligible as soon as time reaches the early start limit for the activity. The activities are prioritized based on late start time limit, slack and activity duration. Activities from the list are scheduled at the current time point as long as resources are available. If an activity is not scheduled at its early start time, its slippage may affect the early starts and thus the slacks of its successors. If the activity has negative or zero slack and must be slipped, the project length may be extended and the late starts and slacks of other activities may require adjustment. RESOURCE_ALLOCATOR dynamically adjusts early starts, late starts and slacks to account for slippage beyond the early start limits of activities. However, it does allow activities to be slipped beyond their original late start limits.

To determine whether sufficient resources are available to schedule an activity at a given time, it is necessary to know what is needed and what is available. The amount of each available resource is specified as a pool level and is allowed to vary as a function of time unit. The module also allows the user to specify for each resource required by an activity the time interval or intervals over which the resource is needed, and an initial and final quantity for each. The initial quantity is the amount which will be subtracted from the resource pool for the duration of the interval. The final quantity is the amount which is returned to the pool at the end of the interval. If no interval is specified, it is assumed that the resource is required for the duration of the activity.

2.4.32.10 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

$ALLOCATED_RESOURCES	-	lists the resources already allocated thus far
I_EVENT_JUST_SCHEDULED_FLAG	-	indicates whether-or-not the job being scheduled is an event
$NILL	-	equivalent to $NULL
LOCAL_INFINITY	-	equal to 16000
$FINISH_TIMES	-	contains the end times of the jobs from $JOBLIST
$JOB	-	specifies the job in $JOBLIST which is currently being processed
$JOBLIST	-	the list of jobs to be scheduled
$JOB_INFO	-	contains the necessary information from $JOB to perform a critical path analysis
$TEMP	-	temporary storage used for many purposes throughout the module
$JOBID_TREE	-	contains the label of the current job
I_JOB_START	-	the start time of the job being processed
I_THRESHOLD_FLAG	-	indicates whether-or-not contingency level resources can be allocated
I_ALLOCATE_OR_FREE_FLAG	-	indicates what type of update operation is to be performed
$FAILURE_INDICATOR	-	can be used to input a time after which resource constraints are allowed.
KLOCK	-	indicates the elapsed time of the schedule at any point in time
$SCHEDULABLE_JOBS	-	list of jobs which can be scheduled based on their predecessor constraints.

$TEMP_JOB	-	temporary storage for the job to be scheduled.
$FINISHED_JOBS	-	list of jobs already scheduled
$DELAYED_JOBS	-	list of jobs delayed due to resource unavailabilities
$IMPORTANT_JOBS	-	contains delayed jobs which have access to critical resources
I_DELAYED_JOB_COUNTER	-	counts number of delayed jobs
I_SCHEDULABLE_JOB_COUNTER	-	counts number of jobs able to be scheduled
$SCHEDULABLES	-	contains the name of all jobs able to be scheduled
$OUTPUT_SCHEDULE	-	contains the start time of all jobs on $SCHEDULE
$SCHEDULE_JOB	-	points at job already on $SCHEDULE
$SCHED_JOB	-	points at jobs able to be scheduled
I1, N1, I3, I , N	-	counters used for various purposes throughout the module
$CURRENT_JOB	-	points at the job able to be scheduled currently being processed
$CONFLICT_TIMES	-	contains the information in $FAILURE_INDICATOR
I_CRITICAL_WARNING_FLAG	-	indicates whether-or-not resources are at their critical point
I_SLACK_UPDATE_FLAG	-	indicates that job has been delayed and slack updates must be done
$POOL	-	points at subnodes of $ALLOCATED_RESOURCES
$CRITICAL_RESOURCES	-	lists resources which are delaying jobs being scheduled
$TEMP_POOL	-	contains the resource in $CRITICAL_RESOURCES currently being processed
$ALLOCATED_RESOURCES	-	contains resources which have been allocated

$RESOURCE_SHORTAGE	-	contains resources which are in short supply and therefore cause a delay in scheduling a job
$CURRENT_JOB_TYPE	-	contains the job type of the job being processed
$RESOURCES_DELAYING_JOB	-	contains the name of the resource delaying the scheduling of the current job
JOB_DURATION	-	duration of the job currently being processed
I_FINISH_TIME	-	the expected finish time of the current job
$MATCH_RESOURCE_TO_JOB	-	contains list of jobs delayed due to resource limitations with the resources causing the delay
$CRIT_RES	-	points at critical resources
$POOL_INDICATOR	-	points at subnodes of $RESOURCES_DELAYING_JOB
N_SCHEDULABLE_JOBS	-	number of jobs able to be scheduled
N_SCHEDULED_JOBS	-	number of jobs on $SCHEDULE
$TEMP_JOBLIST	-	temporary storage for $JOBLIST
I_SPLIT_TIME	-	minimum value of $CONFLICT_TIMES
$NEW_JOB	-	a new job definition, made as a result of splitting the current job
$RESOURCE_SAVE_AREA	-	condensed version of the resource information stored in $JOBLIST
SPLIT_TIME	-	indicates the time at which a splittable job is to be split
FIRST_JOB_DURATION	-	duration of the first part of the splittable job
I_FIRST_JOB_DURATION	-	interger valued FIRST_JOB_DURATION
$PREDECESSOR	-	points to the predecessors of $NEW_JOB's successors

$SUCCESSOR	-	points to the successors of $NEW_JOB
INDEX_OF_POOL	-	counter for number of resources required by the $NEW_JOB
INTERVAL_INDEX	-	points at different resource types required by $CURRENT_JOBS
NEW_INTERVAL_INDEX	-	points at different resource types required by $NEW_JOB
$NEW_POOL	-	points at resources required by $NEW_JOB
$JAB	-	points at subnodes of $SCHEDULE
$RESOURCE_LABEL	-	keeps the labels of resources in $JOBLIST so that the resource information can be returned to its original state in $JOBLIST
$TYPE	-	points at the subnodes of $RESOURCE_SAVE_AREA
$POOL_INFO	-	contains condensed resource information
$PARAMETER	-	temporary storage for condensed resource information before being put into $POOL_INFO
$INFO	-	points at subnodes of $POOL
$JOB_RESOURCES	-	points at subnodes of $RESOURCE_SAVE_AREA
$JOB_ID	-	contains label of $JOB_RESOURCES
$CONDENSED	-	intermediate tree in the return from condensed resource information to original state
ICRIT	-	index for 1 to NUMBER_OF_CRITERIA
$LIST	-	contains the list to be ordered by different criteria
$LIST_ELEMENT	-	points to the subnodes of $LIST

INTERVAL_COUNTER	-	internal counter for number of required resources
MINIMUM	-	equal to LOCAL_INFINITY, or I_START, whichever is less
NSECONDARY	-	number of jobs in $SECONDARY_LIST
$PRIME_LIST	-	a list of jobs which have information to calculate free slack for $CURRENT_JOB
$SECONDARY_LIST	-	list of secondary jobs needed to calculate free slack
I_START	-	indicates early start of SPRIME_LIST successors
$INCREMENT_CANDIDATES	-	successors of $CURRENT_JOB
INITIAL_PASS_FLAG	-	determines definition of $PUSHING_JOB
$PUSHING_JOB	-	defined as current job being processed or a successor of the current job depending upon value of INITIAL_PASS_FLAG
$SUCCESSOR_SET	-	equivalent to $JOBLIST
$JOB_TO_BE_PUSHED	-	a successor of $SUCCESSOR_SET
$CURRENT_SUCCESSOR	-	points at subnodes of $INCREMENT_CANDIDATES
$TARGET	-	equivalent to the description of the job being processed
NUMBER_OF_CRITERIA	-	specifies the number of criteria which the $SCHEDULABLE_JOBS should be ordered on
MAXMIN_FLAG	-	flag determining what criteria to order on
$TRANSFER_INDICES	-	tracks the indices of the variables in $LIST for ordering purposes
I_TRANSFER_INDEX	-	equals the value of $TRANSFER_INDICES
$NEW_LIST	-	contains the ordered $LIST

$IN_USE	-	the in use portion of the current resource
$POOL_PROFILE	-	contains the resource profile for a given resource
I_CHECK_TIME	-	contains the time in $FAILURE_INDICATOR
NUMBER_OF_UPDATES	-	equals 2 if contingency checks are to be made, equals 1 if not the case
$MODE	-	reflects the user choice to search contingency levels of resources or not
$JOB_INTERVAL	-	points at the subnodes of $POOL
I_START	-	the sum of I_JOB_START and the start of the resource availability in $POOL
I_END	-	the sum of I_JOB_START and the end of the resource availability in $POOL
QUANTITY_DELTA	-	the initial quantity of the resource in $POOL being processed
$INTERVAL	-	the inuse portion of $POOL_PROFILE
NLAST	-	the number of subnodes of $INTERVAL
$NEW_INTERVAL	-	equivalent to $NULL, used to insert null nodes onto $INTERVAL
$INTERVAL1	-	calling argument, containing interval information and resources to be updated by QUANTITY_DELTA
$INTERVAL2	-	if it is not empty, then its associated quantity is taken as the initial quantity of $INTERVAL1
$FAILURE_INDICATOR	-	start of $INTERVAL1
I_POINTER	-	pointer which is decremented if $INTERVAL1 and $INTERVAL2 can be combined

2.4.32.11 COMMENTED CODE

```
RESOURCE_ALLOCATOR:  PROCEDURE  ($JOBLIST, $PROFILES, $SCHEDULE )
                OPTIONS(EXTERNAL);
/*************************************************************************/
/*                                                                       */
/*  THIS MODULE IS AN IMPLEMENTATION OF A TIME-PROGRESSIVE PROJECT       */
/*  SCHEDULING HEURISTIC THAT PRODUCES A NEAR-MINIMUM DURATION           */
/*  SCHEDULE THAT SATISFIES ALL RESOURCE CONSTRAINTS.  A DETAILED        */
/*  DESCRIPTION OF THE HEURISTIC CAN BE FOUND IN THE FUNCTIONAL          */
/*  SPECIFICATIONS FOR THIS MODULE.  ON INPUT, $JOBLIST WILL CON-        */
/*  TAIN ALL OF THE INFORMATION ABOUT THE SET OF JOBS TO BE SCHED-       */
/*  ULED INCLUDING:  RESOURCE REQUIREMENTS, TEMPORAL RELATION CON-       */
/*  STRAINTS, AND CRITICAL PATH DATA.  $PROFILES WILL CONTAIN A          */
/*  DESCRIPTION OF EACH RESOURCE POOL, INCLUDING TIME AND QUANTITY       */
/*  CONSTRAINTS.  ON OUTPUT, $SCHEDULE WILL CONTAIN THE SET OF JOBS      */
/*  ALONG WITH THEIR START AND END TIMES.  $PROFILES IS UPDATED TO       */
/*  REFLECT THE RESOURCE ALLOCATIONS PERFORMED DURING THE SCHEDUL-       */
/*  ING PROCESS.                                                         */
/*                                                                       */
/*************************************************************************/
   DECLARE $ALLOCATED_RESOURCES,$CONFLICT_TIMES,$CRITICAL_JOBS,
       $CRITICAL_RESOURCES,$CURRENT_JOB,$DELAYED_JOB_RESOURCES,
       $DELAYED_JOBS,$FAILURE_INDICATOR,$FINISH_TIMES,$FINISHED_JOBS,I,
       I_SLACK_UPDATE_FLAG,I_CRITICAL_WARNING_FLAG,I_FINAL_QUANTITY,
       I_FINISH_TIME,I_SPLIT_TIME,I_START,I_THRESHOLD_FLAG,
       INITIAL_QUANTITY,J,$JOB,$JOB_INFO,KLOCK,M,MINIMUM,N,
       N_SCHEDULABLE_JOBS,N_SCHEDULED_JOBS,$NEW_JOB,$POOL,
       $RESOURCE_SAVE_AREA,$RESOURCE_SHORTAGE,$SCHEDULABLE_JOBS,
       LOCAL_INFINITY, $NILL,
       I_EVENT_JUST_SCHEDULED_FLAG, $IMPORTANT_JOBS,
       $MATCH_RESOURCE_TO_JOB,
                                               $SUCCESSOR,$TEMP LOCAL ;
/*  FIRST THE DATA STRUCTURE OF $JOBLIST IS CONDENSED TO INCREASE        */
/*  THE EFFICIENCY OF NODE ACCESSES IN THE REST OF THE PROGRAM.          */
   I_EVENT_JUST_SCHEDULED_FLAG = 0 ;
   $NILL = $NULL ;
   LOCAL_INFINITY = 10000 ;
   CALL CONDENSE_RESOURCE_INFORMATION($JOBLIST,$RESOURCE_SAVE_AREA) ;
   $FINISH_TIMES(FIRST)= LOCAL_INFINITY ;
   DO I = NUMBER($JOBLIST) TO 1 BY -1  ;
      DEFINE $JOB AS $JOBLIST(I) ;
      GRAFT $JOB.TEMPORAL_RELATION.PREDECESSOR AT $JOB_INFO.
                                                          PREDECESSOR ;
      GRAFT $JOB.TEMPORAL_RELATION.SUCCESSOR AT $JOB_INFO.SUCCESSOR ;
      GRAFT $JOB.DURATION AT $JOB_INFO.DURATION ;
      CALL RETRIEVE_CRITICAL_PATH_DATA($JOB_INFO) ;
      GRAFT $JOB.JOB_TYPE AT $JOB_INFO.JOB_TYPE ;
      CALL COMBINE_TREES($JOB_INFO,$JOB) ;
      IF $JOB.JOB_INTERVAL(FIRST) NOT IDENTICAL TO  $NULL
         THEN DO ;  $JOB.FROZEN = '' ;
              $TEMP.#LABEL($JOB) = $JOB.JOB_INTERVAL.END ;
              GRAFT INSERT $TEMP(FIRST) BEFORE $FINISH_TIMES(FIRST) ;
```


ORIGINAL PAGE IS OF POOR QUALITY

```
            $JOBID_TREE = LABEL($JOB) ;
            I_JOB_START = $JOB.JOB_INTERVAL.START ;
            I_THRESHOLD_FLAG = 0 ;
            I_ALLOCATE_OR_FREE_FLAG = 1 ;
            $FAILURE_INDICATOR = 0 ;
            DO FOR ALL SUBNODES OF $JOB.RESOURCE USING $POOL ;
            CALL UPDATE_POOL_LEVELS
                 ( $POOL, $JOBID_TREE, $PROFILES, I_JOB_START,
                   I_THRESHOLD_FLAG,I_ALLOCATE_OR_FREE_FLAG,
                   $FAILURE_INDICATOR ) ;
            END;
            GRAFT $JOB AT $SCHEDULE(NEXT) ;
            END ;
      END ;
   KLOCK = -1 ;
BEGIN_MAIN_SCHEDULING_LOOP:
/*  THIS IS THE MAIN LOOP OF 'RESOURCE_ALLOCATOR'.  IT PROGRESS-      */
/*  IVELY STEPS THROUGH TIME UNTIL ALL OF THE JOBS IN $JOBLIST HAVE   */
/*  BEEN SCHEDULED.                                                   */
   IF $JOBLIST(FIRST) IDENTICAL TO $NULL
              & $SCHEDULABLE_JOBS(FIRST) IDENTICAL TO $NULL
   THEN GO TO ALLOCATOR_FINAL_PROCEDURES ;
/*  IF AN EVENT HAS JUST BEEN SCHEDULED, DO NOT ADVANCE THE KLOCK.    */
/*  GIVE THE EVENT'S SUCCESSORS A CHANCE AT THE CURRENT TIME.         */
      IF I_EVENT_JUST_SCHEDULED_FLAG = 0
      THEN  KLOCK = KLOCK + 1 ;
      ELSE I_EVENT_JUST_SCHEDULED_FLAG = 0 ;
/*  IF ANY JOBS HAVE JUST FINISHED, THEIR ID NUMBERS ARE RECORDED     */
/*  IN $FINISHED_JOBS.                                                */
      DO I=1 TO NUMBER($FINISH_TIMES) ;
         IF $FINISH_TIMES(FIRST)   >   KLOCK
            THEN GO TO BUILD_SET_OF_SCHEDULABLE_JOBS ;
         INSERT LABEL($FINISH_TIMES(FIRST))
         BEFORE $FINISHED_JOBS(FIRST) ;
         PRUNE $FINISH_TIMES(FIRST);
         END ;
BUILD_SET_OF_SCHEDULABLE_JOBS:
/*  ALL JOBS WHOSE PREDECESSORS ARE FINISHED CAN NOW BE ADDED         */
/*  TO $SCHEDULABLE_JOBS.                                             */
      DO I = 1 TO NUMBER($JOBLIST) ;
         GRAFT $JOBLIST(FIRST) AT $TEMP_JOB ;
         IF $TEMP_JOB.PREDECESSOR SUBSET OF $FINISHED_JOBS
            & KLOCK >= $TEMP_JOB.EARLY_START
            THEN DO ;
                 $TEMP_JOB.ENTRY_TIME = KLOCK ;
                 GRAFT $TEMP_JOB AT $SCHEDULABLE_JOBS(NEXT) ;
                 END ;
         IF $TEMP_JOB NOT IDENTICAL TO $NULL
            THEN GRAFT $TEMP_JOB AT $JOBLIST(NEXT) ;
         END ;
ORDER_SCHEDULABLE_JOBS:
```

```
        DO FOR ALL SUBNODES OF $SCHEDULABLE_JOBS USING $JOB ;
           CALL UPDATE_SLACK($JOB, $JOBLIST, $SCHEDULE ) ;
        END;
/*************
      ORDER $SCHEDULABLE_JOBS BY -LATE_START,-FREE_SLACK,-TOTAL_SLACK,
                                                            -DURATION ;

****************/
   CALL SCHEDULABLE_JOB_ORDER($SCHEDULABLE_JOBS) ;
       IF $DELAYED_JOBS NOT IDENTICAL TO $NULL
       THEN DO ;
/* CREATE $IMPORTANT_JOBS. $IMPORTANT_JOBS ARE THE ONLY ONES WHO ARE */
/* ALLOWED ACCESS TO CRITICAL RESOURCES WHILE THERE ARE $DELAYED_JOBS*/
/* THEY CONSIST OF $DELAYED_JOBS AND OTHER JOBS WHICH ARE RANKED     */
/* HIGHER IN THE ORDERED $SCHEDULABLE_JOBS.                          */

            PRUNE $IMPORTANT_JOBS ;
            I_DELAYED_JOB_COUNTER = 0 ;
            I_SCHEDULABLE_JOB_COUNTER = 0 ;
            DO WHILE (I_DELAYED_JOB_COUNTER < NUMBER($DELAYED_JOBS) ) ;
               I_SCHEDULABLE_JOB_COUNTER = I_SCHEDULABLE_JOB_COUNTER+1;
               $IMPORTANT_JOBS(NEXT) =
                 LABEL($SCHEDULABLE_JOBS(I_SCHEDULABLE_JOB_COUNTER));
               IF LABEL($SCHEDULABLE_JOBS(I_SCHEDULABLE_JOB_COUNTER))
                  ELEMENT OF $DELAYED_JOBS
               THEN I_DELAYED_JOB_COUNTER = I_DELAYED_JOB_COUNTER +1 ;
               END;
               END;
      I_SLACK_UPDATE_FLAG = 0 ;
 /*  THIS LOOP EXAMINES ALL OF THE SCHEDULABLE JOBS AND DETERMINES     */
 /*  WHETHER OR NOT THEY CAN BE SCHEDULED WITHOUT VIOLATING ANY        */
 /*  RESOURCE CONSTRAINTS.                                             */
      I1=1;
      N1=NUMBER($SCHEDULABLE_JOBS);
 TEMP_LABEL1:
      IF I1>N1
        THEN GO TO TEMP_LABEL2;
         GRAFT $SCHEDULABLE_JOBS(FIRST) AT $CURRENT_JOB ;
         PRUNE $CONFLICT_TIMES ;
         I_CRITICAL_WARNING_FLAG = 0 ;
         IF I_SLACK_UPDATE_FLAG ¬= 0
            THEN CALL UPDATE_SLACK($CURRENT_JOB,$JOBLIST,$SCHEDULE) ;
         IF $DELAYED_JOBS NOT IDENTICAL TO $NULL
         THEN IF LABEL($CURRENT_JOB) NOT ELEMENT OF $IMPORTANT_JOBS
              THEN IF $CURRENT_JOB.LATE_START > KLOCK
                   THEN DO FOR ALL SUBNODES OF $CURRENT_JOB.RESOURCE
                                                            USING $POOL ;

                          IF LABEL($POOL) ELEMENT OF
                                      $CRITICAL_RESOURCES  &
                                      $POOL(FIRST).FINAL_QUANTITY
                                      <=$POOL(FIRST).INITIAL_QUANTITY
                             THEN GO TO PUT_JOB_IN_WAIT_STATE ;
```

```
                END ;
/*  THIS LOOP ATTEMPTS TO ALLOCATE RESOURCES FOR THE JOB CURRENTLY    */
/*  UNDER CONSIDERATION.                                              */
        DO I = 1 TO NUMBER($CURRENT_JOB.RESOURCE) ;
           PRUNE $FAILURE_INDICATOR ;
           GRAFT $CURRENT_JOB.RESOURCE(FIRST) AT $TEMP_POOL ;
           IF LABEL($TEMP_POOL) ELEMENT OF $CRITICAL_RESOURCES  &
           $PROFILES.#LABEL($TEMP_POOL).AVAILABLE.CONTINGENCY(FIRST)
                   NOT IDENTICAL TO $NULL
              THEN I_THRESHOLD_FLAG = 1 ;
              ELSE I_THRESHOLD_FLAG = 0 ;
           $JOBID_TREE = LABEL($CURRENT_JOB) ;
           CALL UPDATE_POOL_LEVELS($TEMP_POOL,$JOBID_TREE,
              $PROFILES,KLOCK,I_THRESHOLD_FLAG,1,$FAILURE_INDICATOR) ;
           IF $FAILURE_INDICATOR  IDENTICAL TO  $NULL
              THEN GRAFT INSERT $TEMP_POOL BEFORE
                                               $ALLOCATED_RESOURCES(1) ;
             ELSE DO;
                  GRAFT INSERT $TEMP_POOL BEFORE
                                              $RESOURCE_SHORTAGE(FIRST);
                  LABEL($CURRENT_JOB_TYPE)= $CURRENT_JOB.JOB_TYPE ;
                  IF LABEL($CURRENT_JOB_TYPE) = 'SPLITTABLE'
                  THEN GRAFT INSERT $FAILURE_INDICATOR BEFORE
                                   $CONFLICT_TIMES(FIRST) ;
                  IF $CURRENT_JOB.LATE_START <= KLOCK
                  THEN INSERT LABEL($RESOURCE_SHORTAGE(FIRST)) BEFORE
                             $RESOURCES_DELAYING_JOB(FIRST) ;
                  ELSE IF LABEL($CURRENT_JOB_TYPE) ¬= 'SPLITTABLE'
                       THEN GO TO PUT_JOB_IN_WAIT_STATE ;
                  END;
           END ;
/*  IF THE PRECEDING ATTEMPTED RESOURCE ALLOCATION WAS SUCCESSFUL,    */
/*  THE JOB CAN NOW BE SCHEDULED.                                     */
        IF $RESOURCE_SHORTAGE  IDENTICAL TO  $CONFLICT_TIMES
           THEN DO ;
                JOB_DURATION = $CURRENT_JOB.DURATION ;
                IF JOB_DURATION = 0
                THEN I_EVENT_JUST_SCHEDULED_FLAG = 1 ;
                GRAFT $ALLOCATED_RESOURCES AT $CURRENT_JOB.RESOURCE ;
                $CURRENT_JOB.JOB_INTERVAL.START = KLOCK ;
                I_FINISH_TIME = KLOCK + $CURRENT_JOB.DURATION ;
                $CURRENT_JOB.JOB_INTERVAL.END = I_FINISH_TIME ;
                $TEMP.#LABEL($CURRENT_JOB) = I_FINISH_TIME ;
                GRAFT INSERT $TEMP(FIRST) BEFORE
                $FINISH_TIMES(FIRST: $ELEMENT > I_FINISH_TIME) ;
               IF LABEL($CURRENT_JOB) ELEMENT OF $DELAYED_JOBS
               THEN DO ;
                    PRUNE $DELAYED_JOBS(FIRST: $ELEMENT =
                                            LABEL($CURRENT_JOB)) ;
                    DO FOR ALL SUBNODES OF
                      $MATCH_RESOURCE_TO_JOB.#LABEL($CURRENT_JOB).
```

```
                    CRITICAL_RESOURCE USING $CRIT_RES ;
                    PRUNE $CRITICAL_RESOURCES(FIRST: $ELEMENT
                         IDENTICAL TO $CRIT_RES) ;
                  END;
                  PRUNE $MATCH_RESOURCE_TO_JOB.#LABEL($CURRENT_JOB);
                  END;
              GRAFT $CURRENT_JOB AT $SCHEDULE(NEXT) ;
              END ;
         ELSE DO ;
              DO FOR ALL SUBNODES OF $ALLOCATED_RESOURCES USING
                                                              $POOL ;
                 $JOBID_TREE = LABEL($CURRENT_JOB);
                 CALL UPDATE_POOL_LEVELS($POOL,$JOBID_TREE,
                           $PROFILES,KLOCK,0,0,$FAILURE_INDICATOR) ;
                 END ;
              CALL COMBINE_TREES
                   ($ALLOCATED_RESOURCES,$CURRENT_JOB.RESOURCE) ;
              CALL COMBINE_TREES
                   ( $RESOURCE_SHORTAGE, $CURRENT_JOB.RESOURCE ) ;
              IF $CURRENT_JOB.LATE_START <= KLOCK
              THEN DO;
/*  DETERMINE WHETHER CONTINGENCY RESOURCES WOULD HELP                 */
                   DO FOR ALL SUBNODES OF $RESOURCES_DELAYING_JOB
                                  USING $POOL_INDICATOR ;
                      IF $PROFILES.#($POOL_INDICATOR).AVAILABLE.
                         CONTINGENCY(FIRST) NOT IDENTICAL TO $NULL
                      THEN GO TO CONSIDER_RESCHEDULING ;
                   END;
/* THERE ARE NO CONTINGENCY RESOURCES TO HELP THIS JOB, SO DROP OUT   */
/*  AND SPLIT THE JOB OR PUT IT IN THE WAIT STATE                      */
                        PRUNE $TEMP ;
                        LABEL($TEMP) = LABEL($CURRENT_JOB) ;
                        $TEMP.CRITICAL_RESOURCE
                          = $RESOURCES_DELAYING_JOB ;
                        GRAFT INSERT $TEMP BEFORE
                            $MATCH_RESOURCE_TO_JOB(FIRST) ;
                        CALL COMBINE_TREES
                             ( $RESOURCES_DELAYING_JOB,
                               $CRITICAL_RESOURCES) ;
                   IF $CONFLICT_TIMES IDENTICAL TO $NULL
                   THEN GO TO PUT_JOB_IN_WAIT_STATE ;
                   ELSE GO TO ATTEMPT_JOB_SPLIT ;
CONSIDER_RESCHEDULING:
/* IF THE RESOURCES DELAYING THIS JOB HAVE ALREADY BEEN MADE CRITICAL*/
/* DON'T RESCHEDULE BUT ADD THE JOB TO THE DELAYED JOBS AND ITS       */
/* RESOURCES TO $CRITICAL RESOURCES                                    */
                   IF $RESOURCES_DELAYING_JOB SUBSET OF
                      $CRITICAL_RESOURCES
                   THEN DO ;
                        INSERT LABEL($CURRENT_JOB) BEFORE
                               $DELAYED_JOBS(FIRST) ;
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
                        PRUNE $TEMP ;
                        LABEL($TEMP) = LABEL($CURRENT_JOB) ;
                        $TEMP.CRITICAL_RESOURCE
                          = $RESOURCES_DELAYING_JOB ;
                        GRAFT INSERT $TEMP BEFORE
                            $MATCH_RESOURCE_TO_JOB(FIRST) ;
                        CALL COMBINE_TREES
                             ( $RESOURCES_DELAYING_JOB,
                               $CRITICAL_RESOURCES) ;
                        END;
                    ELSE DO;
/* ALL THE CONDITIONS NECESSARY FOR RESCHEDULING HAVE NOW BEEN MET    */
/* SO GO TO IT                                                        */
                        I_EVENT_JUST_SCHEDULED_FLAG = 0 ;
                      INSERT LABEL($CURRENT_JOB) BEFORE
                             $DELAYED_JOBS(FIRST) ;
                        PRUNE $TEMP ;
                        LABEL($TEMP) = LABEL($CURRENT_JOB) ;
                        $TEMP.CRITICAL_RESOURCE
                          = $RESOURCES_DELAYING_JOB ;
                        GRAFT INSERT $TEMP BEFORE
                            $MATCH_RESOURCE_TO_JOB(FIRST) ;
                      CALL COMBINE_TREES
                           ( $RESOURCES_DELAYING_JOB,
                             $CRITICAL_RESOURCES ) ;
                      KLOCK = $CURRENT_JOB.ENTRY_TIME ;
                      DO I3= NUMBER($SCHEDULABLE_JOBS) TO 1 BY -1 ;
                         IF $SCHEDULABLE_JOBS(I3).ENTRY_TIME > KLOCK
                         THEN GRAFT INSERT $SCHEDULABLE_JOBS(I3)
                              BEFORE $JOBLIST(FIRST) ;
                      END;
                      DO J=1 TO NUMBER($SCHEDULE) ;
                         GRAFT $SCHEDULE(FIRST) AT $TEMP_JOB ;
                         IF $TEMP_JOB.FROZEN IDENTICAL TO $NULL
                            THEN IF $TEMP_JOB.JOB_INTERVAL.START >=
                                                              KLOCK
                                 THEN DO ;
                                      PRUNE $FINISH_TIMES.#LABEL
                                      ($TEMP_JOB),$FINISHED_JOBS.
                                             #LABEL($TEMP_JOB) ;
                                      $JOBID_TREE =
                                            LABEL($TEMP_JOB) ;
                                      I_JOB_START =
                                   $TEMP_JOB.JOB_INTERVAL.START ;
                                      I_THRESHOLD_FLAG = 0;
                                      I_ALLOCATE_OR_FREE_FLAG=0;
                                      $FAILURE_INDICATOR = $NULL;
                                      DO FOR ALL SUBNODES OF
                                 $TEMP_JOB.RESOURCE USING $POOL ;
                                      CALL UPDATE_POOL_LEVELS
                                      ( $POOL, $JOBID_TREE,
```

```
                                         $PROFILES, I_JOB_START,
                                         I_THRESHOLD_FLAG,
                                         I_ALLOCATE_OR_FREE_FLAG,
                                         $FAILURE_INDICATOR ) ;
                                           END;
                                   IF $TEMP_JOB.ENTRY_TIME > KLOCK
                                           THEN GRAFT INSERT
                                                    $TEMP_JOB
                                                 BEFORE
                                                 $JOBLIST(FIRST);
                                           ELSE GRAFT INSERT
                                                    $TEMP_JOB
                                               BEFORE
                                               $SCHEDULABLE_JOBS
                                                        (FIRST);
                                        END ;
                          IF $TEMP_JOB NOT IDENTICAL TO $NULL
                             THEN GRAFT $TEMP_JOB AT $SCHEDULE(NEXT);
                          END ;
                        GRAFT INSERT $CURRENT_JOB BEFORE
                                                $SCHEDULABLE_JOBS(FIRST);
/*  IN ORDER TO RECALCULATE THE CRITICAL PATH DATA FOR THE JOB NET-  */
/*  WORK, ALL OF THE JOBS FOUND IN $SCHEDULE AND $SCHEDULABLE_JOBS   */
/*  ARE COMBINED WITH THOSE IN $JOBLIST.                             */
                        DO FOR ALL SUBNODES OF $SCHEDULE USING $JOB ;
                          $JOB.START.EARLY = $JOB.JOB_INTERVAL.START;
                          END ;
                        N_SCHEDULABLE_JOBS=NUMBER($SCHEDULABLE_JOBS) ;
                        N_SCHEDULED_JOBS = NUMBER($SCHEDULE) ;
                        CALL COMBINE_TREES
                                     ($SCHEDULABLE_JOBS,$JOBLIST) ;
                                     DO FOR ALL SUBNODES OF
                                     $JOBLIST USING $JOB ;
                                     PRUNE $JOB.EARLY_START,
                                          $JOB.EARLY_FINISH,
                                          $JOB.LATE_START,
                                          $JOB.LATE_FINISH ;
                                     END;
/****************
                        ORDER $SCHEDULE BY -JOB_INTERVAL.START ;
******************/
                        CALL ORDER_BY_JOB_START($SCHEDULE) ;
                        CALL COMBINE_TREES($SCHEDULE,$JOBLIST) ;
                        DO FOR ALL SUBNODES OF $JOBLIST USING $JOB ;
                        GRAFT $JOB.PREDECESSOR AT
                              $JOB.TEMPORAL_RELATION.PREDECESSOR ;
                        GRAFT $JOB.SUCCESSOR AT
                              $JOB.TEMPORAL_RELATION.SUCCESSOR ;
                          END ;
                        GRAFT $JOBLIST AT $TEMP_JOBLIST ;
                        CALL CRITICAL_PATH_CALCULATOR
```

```
                        ( $TEMP_JOBLIST, $JOBLIST ) ;
                    PRUNE $TEMP_JOBLIST ;
                    DO FOR ALL SUBNODES OF $JOBLIST  USING $JOB ;
                    GRAFT $JOB.TEMPORAL_RELATION.PREDECESSOR
                          AT $JOB.PREDECESSOR ;
                    GRAFT $JOB.TEMPORAL_RELATION.SUCCESSOR
                          AT $JOB.SUCCESSOR ;
                       CALL RETRIEVE_CRITICAL_PATH_DATA($JOB) ;
                       END ;
                    DO K=1 TO N_SCHEDULED_JOBS ;
                       GRAFT $JOBLIST(FIRST) AT $SCHEDULE(NEXT) ;
                       END ;
                    DO K=1 TO N_SCHEDULABLE_JOBS ;
                       GRAFT $JOBLIST(FIRST) AT $SCHEDULABLE_JOBS
                                                        (NEXT) ;
                       END ;
                    GO TO ORDER_SCHEDULABLE_JOBS ;
                    END ;
                    END ;
                ELSE DO ;
ATTEMPT_JOB_SPLIT:      CALL COMBINE_TREES
                  ($ALLOCATED_RESOURCES,$CURRENT_JOB.RESOURCE) ;
                    CALL COMBINE_TREES
                      ($RESOURCE_SHORTAGE,$CURRENT_JOB.RESOURCE) ;
                    IF $CONFLICT_TIMES  NOT IDENTICAL TO  $NULL
/*  SINCE ALL OTHER SCHEDULING ATTEMPTS HAVE FAILED, SPLITTING OF     */
/*  THE JOB IS DONE AS A LAST RESORT.                                 */
                       THEN DO ;
                           CALL FIND_MIN
                      ($CONFLICT_TIMES,$DUMMY,VALUE_MINIMUM) ;
                           I_SPLIT_TIME = VALUE_MINIMUM ;
                           CALL JOB_SPLITTER
                           ($CURRENT_JOB,I_SPLIT_TIME,$NEW_JOB) ;
                           IF $NEW_JOB  NOT IDENTICAL TO  $NULL
                              THEN DO ;
                                  INSERT $RESOURCE_SAVE_AREA.
                                     #LABEL($JOB) BEFORE
                                     $RESOURCE_SAVE_AREA(FIRST);
                                  LABEL($RESOURCE_SAVE_AREA
                                     (FIRST))= LABEL($NEW_JOB) ;
                                  GRAFT INSERT $CURRENT_JOB
                                   BEFORE $SCHEDULABLE_JOBS(1) ;
                                  GRAFT $NEW_JOB AT
                                            $JOBLIST(NEXT) ;
                                  GO TO ORDER_SCHEDULABLE_JOBS;
                                  END ;
                           END ;
                    END ;
               END ;
PUT_JOB_IN_WAIT_STATE:
      IF $CURRENT_JOB  NOT IDENTICAL TO  $NULL
```

ORIGINAL PAGE IS OF POOR QUALITY

```
            THEN DO ;
/*  SINCE THIS JOB HAS BEEN DELAYED, ITS CRITICAL PATH DATA (AND        */
/*  POSSIBLE THAT OF ITS SUCCESSORS) MUST BE READJUSTED.                */
                   $CURRENT_JOB.EARLY_START = KLOCK+1 ;
                   $CURRENT_JOB.EARLY_FINISH=$CURRENT_JOB.EARLY_FINISH+1;
                   IF $CURRENT_JOB.FREE_SLACK = 0
                      THEN DO ;  I_SLACK_UPDATE_FLAG = 1 ;
                           CALL INCREMENT_SUCCESSOR_TIMES
                                      ($CURRENT_JOB, $JOBLIST ) ;
                           END ;
                   IF $CURRENT_JOB.TOTAL_SLACK <= 0
                      THEN DO ;
                           $CURRENT_JOB.LATE_START =
                                    $CURRENT_JOB.LATE_START + 1 ;
                           $CURRENT_JOB.LATE_FINISH =
                                        $CURRENT_JOB.LATE_FINISH + 1 ;
                           DO FOR ALL SUBNODES OF $SCHEDULABLE_JOBS USING
                                                                  $JOB ;
                              $JOB.LATE_START = $JOB.LATE_START + 1 ;
                              $JOB.LATE_FINISH = $JOB.LATE_FINISH + 1 ;
                              END ;
                           DO FOR ALL SUBNODES OF $JOBLIST USING $JOB ;
                              $JOB.LATE_START = $JOB.LATE_START + 1 ;
                              $JOB.LATE_FINISH = $JOB.LATE_FINISH + 1 ;
                              END ;
                           DO FOR ALL SUBNODES OF $SCHEDULE USING $JOB ;
                              $JOB.LATE_START = $JOB.LATE_START + 1 ;
                              $JOB.LATE_FINISH = $JOB.LATE_FINISH + 1 ;
                              END ;
                           END ;
/*  NOW THE RESOURCES PREVIOUSLY ALLOCATED CAN BE FREED.  THE            */
/*  RESOURCE POOLS ARE THEN REORDERED SO THAT THE CRITICAL POOLS         */
/*  WILL BE CONSIDERED FIRST ON THE NEXT SCHEDULING ATTEMPT FOR          */
/*  THIS JOB.                                                            */
                   DO FOR ALL SUBNODES OF $ALLOCATED_RESOURCES USING
                                                                 $POOL ;
                      $JOBID_TREE = LABEL($CURRENT_JOB) ;
                      CALL UPDATE_POOL_LEVELS($POOL,$JOBID_TREE,
                              $PROFILES,KLOCK,0,0,$FAILURE_INDICATOR) ;
                      END ;
                   CALL COMBINE_TREES
                          ($ALLOCATED_RESOURCES,$CURRENT_JOB.RESOURCE) ;
                   CALL COMBINE_TREES
                            ($RESOURCE_SHORTAGE,$CURRENT_JOB.RESOURCE) ;
                   GRAFT $CURRENT_JOB AT $SCHEDULABLE_JOBS(NEXT) ;
                   END ;
      I1=I1+1;
      GO TO TEMP_LABEL1;
TEMP_LABEL2:
   GO TO BEGIN_MAIN_SCHEDULING_LOOP ;
ALLOCATOR_FINAL_PROCEDURES:
```

```
/*  FINALLY, THE DATA STRUCTURE OF THE JOB NODES CAN BE EXPANDED              */
/*  BACK TO THEIR ORIGINAL FORM BEFORE RETURNING $SCHEDULE TO THE             */
/*  CALLING PROGRAM.                                                          */
   CALL RESTORE_RESOURCE_INFORMATION($SCHEDULE,$RESOURCE_SAVE_AREA) ;
   DO FOR ALL SUBNODES OF $SCHEDULE USING $JOB ;
      CALL UPDATE_SLACK($JOB, $SCHEDULE, $JOBLIST ) ;
      GRAFT $JOB.PREDECESSOR AT $JOB.TEMPORAL_RELATION.PREDECESSOR ;
      GRAFT $JOB.SUCCESSOR AT $JOB.TEMPORAL_RELATION.SUCCESSOR ;
      GRAFT $JOB.EARLY_START AT $JOB.START.EARLY ;
      GRAFT $JOB.LATE_START AT $JOB.START.LATE ;
      GRAFT $JOB.EARLY_FINISH AT $JOB.FINISH.EARLY ;
      GRAFT $JOB.LATE_FINISH AT $JOB.FINISH.LATE ;
      GRAFT $JOB.TOTAL_SLACK AT $JOB.SLACK.TOTAL ;
      GRAFT $JOB.FREE_SLACK AT $JOB.SLACK.FREE ;
      GRAFT INSERT $JOB.JOB_INTERVAL BEFORE $JOB(FIRST) ;
      PRUNE $JOB.ENTRY_TIME ;
      END ;
/*************
   ORDER $SCHEDULE BY -JOB_INTERVAL.START,-DURATION ;
*************/
      CALL ALLOCATOR_FINAL_ORDER($SCHEDULE) ;




JOB_SPLITTER:  PROCEDURE ($JOB, SPLIT_TIME, $NEW_JOB) ;
/*                                                                            */
/*    THIS PROCEDURE BREAKS $JOB INTO TWO SMALLER COMPONENT JOBS.             */
/*    THE SPLIT IS MADE AT THE TIME INDICATED BY 'SPLIT_TIME'.  THE           */
/*    TWO RESULTANT JOBS ARE RETURNED IN $JOB AND $NEW_JOB.                   */
/*                                                                            */
      DECLARE $PREDECESSOR, $SUCCESSOR    LOCAL ;
      PRUNE $NEW_JOB ;
      FIRST_JOB_DURATION = SPLIT_TIME - KLOCK ;
      I_FIRST_JOB_DURATION = FIRST_JOB_DURATION ;
      IF I_FIRST_JOB_DURATION = 0   THEN RETURN ;
      $NEW_JOB = $JOB ;
      TEMP = LABEL($JOB) ;
      TEMP = TEMP + 0.1 ;
      LABEL($NEW_JOB) = TEMP ;
/*                                                                            */
/*    FIX THE TEMPORAL QUANTITIES OF $JOB AND $NEW_JOB                        */
/*                                                                            */
      $JOB.DURATION = FIRST_JOB_DURATION ;
      $JOB.EARLY_FINISH = $JOB.EARLY_START + $JOB.DURATION ;
      $JOB.LATE_FINISH = $JOB.LATE_START + $JOB.DURATION ;
      PRUNE  $JOB.SUCCESSOR  ;
      $JOB.SUCCESSOR(FIRST) = LABEL($NEW_JOB) ;
      $JOB.FREE_SLACK = 0;
      $NEW_JOB.DURATION = $NEW_JOB.DURATION - FIRST_JOB_DURATION ;
```

```
    PRUNE $NEW_JOB.PREDECESSOR ;
    $NEW_JOB.PREDECESSOR(FIRST) = LABEL($JOB) ;
    DO FOR ALL SUBNODES OF $NEW_JOB.SUCCESSOR USING $SUCCESSOR ;
       DEFINE $PREDECESSOR AS $JOBLIST.#($SUCCESSOR).PREDECESSOR ;
       PRUNE $PREDECESSOR(FIRST: $ELEMENT = LABEL($JOB) ) ;
       INSERT LABEL($NEW_JOB) BEFORE $PREDECESSOR(FIRST) ;
    END;
    $NEW_JOB.EARLY_START = $NEW_JOB.EARLY_START + FIRST_JOB_DURATION;
    $NEW_JOB.LATE_START  = $NEW_JOB.LATE_START  + FIRST_JOB_DURATION;
    $NEW_JOB.EARLY_FINISH = $NEW_JOB.EARLY_START + $NEW_JOB.DURATION;
    $NEW_JOB.LATE_FINISH = $NEW_JOB.LATE_START + $NEW_JOB.DURATION ;
    $NEW_JOB.ENTRY_TIME = KLOCK ;
/*                                                                    */
/*  FIX UP THE RESOURCE REQUIREMENTS IN THE SPLIT JOBS                */
/*                                                                    */
    DO INDEX_OF_POOL = 1 TO NUMBER($JOB.RESOURCE) ;
    INTERVAL_INDEX = 1;
    NEW_INTERVAL_INDEX = 1;
    DEFINE $POOL AS $JOB.RESOURCE(INDEX_OF_POOL) ;
    DEFINE $NEW_POOL AS $NEW_JOB.RESOURCE(INDEX_OF_POOL) ;
    DO INTERVAL_COUNTER = 1 TO NUMBER($POOL) ;
      IF KLOCK + $POOL(INTERVAL_INDEX).END          <=  SPLIT_TIME
      THEN DO;
              PRUNE $NEW_POOL(NEW_INTERVAL_INDEX) ;
              INTERVAL_INDEX = INTERVAL_INDEX + 1 ;
           END;
      ELSE IF KLOCK + $POOL(INTERVAL_INDEX).START
                                                   >= SPLIT_TIME
           THEN DO;
                  $NEW_POOL(NEW_INTERVAL_INDEX).START          =
                  $NEW_POOL(NEW_INTERVAL_INDEX).START
                                             - FIRST_JOB_DURATION ;
                  $NEW_POOL(NEW_INTERVAL_INDEX).END    =
                  $NEW_POOL(NEW_INTERVAL_INDEX).END
                                             - FIRST_JOB_DURATION ;
                  PRUNE $POOL(INTERVAL_INDEX) ;
                  NEW_INTERVAL_INDEX = NEW_INTERVAL_INDEX + 1;
                END;
           ELSE DO;
                  $POOL(INTERVAL_INDEX).END  =
                                              FIRST_JOB_DURATION  ;
                  $POOL(INTERVAL_INDEX).FINAL_QUANTITY =
                      $POOL(INTERVAL_INDEX).INITIAL_QUANTITY ;
                  $NEW_POOL(NEW_INTERVAL_INDEX).START = 0 ;
                  $NEW_POOL(NEW_INTERVAL_INDEX).END =
                  $NEW_POOL(NEW_INTERVAL_INDEX).END
                                             - FIRST_JOB_DURATION ;
                  INTERVAL_INDEX = INTERVAL_INDEX + 1 ;
                  NEW_INTERVAL_INDEX = NEW_INTERVAL_INDEX + 1;
                END;
    END;
```

```
      END;
END;  /* JOB_SPLITTER */
UPDATE_SLACK:  PROCEDURE ($JOB, $PRIME_LIST, $SECONDARY_LIST ) ;
/*  THIS PROCEDURE RECALCULATES THE TOTAL AND FREE SLACK FOR THE      */
/*  JOB PASSED TO IT IN $JOB.                                          */
   DECLARE I, MINIMUM, NSCHEDULE      LOCAL ;
   $JOB.TOTAL_SLACK = $JOB.LATE_START - $JOB.EARLY_START ;
   MINIMUM = LOCAL_INFINITY ;
   NSECONDARY = NUMBER($SECONDARY_LIST) ;
   CALL COMBINE_TREES( $SECONDARY_LIST, $PRIME_LIST ) ;
   DO FOR ALL SUBNODES OF $JOB.SUCCESSOR USING $SUCCESSOR ;
     I_START = $PRIME_LIST.#($SUCCESSOR).EARLY_START ;
      IF I_START < MINIMUM  THEN MINIMUM = I_START ;
      END ;
   IF MINIMUM ¬= LOCAL_INFINITY
      THEN $JOB.FREE_SLACK = MINIMUM - $JOB.EARLY_FINISH ;
   DO I = NSECONDARY TO 1 BY -1 ;
      GRAFT INSERT $PRIME_LIST(I) BEFORE $SECONDARY_LIST(FIRST) ;
   END;
 END;  /*   UPDATE_SLACK  */
INCREMENT_SUCCESSOR_TIMES:  PROCEDURE ($JOB, $SUCCESSOR_SET) ;
  DECLARE  $INCREMENT_CANDIDATES, $PUSHING_JOB, $JOB_TO_BE_PUSHED,
          INITIAL_PASS_FLAG      LOCAL  ;
   $INCREMENT_CANDIDATES(FIRST) = $JOB.SUCCESSOR ;
   LABEL($INCREMENT_CANDIDATES(FIRST)) = LABEL($JOB) ;
   INITIAL_PASS_FLAG = 1 ;
   DO WHILE ($INCREMENT_CANDIDATES(FIRST) NOT IDENTICAL TO $NULL) ;
      IF INITIAL_PASS_FLAG = 1
      THEN DO;  DEFINE $PUSHING_JOB AS $JOB ;
                INITIAL_PASS_FLAG = 0 ;
           END;
      ELSE  DEFINE $PUSHING_JOB AS
                $SUCCESSOR_SET.#LABEL($INCREMENT_CANDIDATES(FIRST)) ;
      DO FOR ALL SUBNODES OF $INCREMENT_CANDIDATES(FIRST)
                                        USING $CURRENT_SUCCESSOR ;

         DEFINE $JOB_TO_BE_PUSHED AS
                $SUCCESSOR_SET.#($CURRENT_SUCCESSOR) ;
         IF $PUSHING_JOB.EARLY_FINISH > $JOB_TO_BE_PUSHED.EARLY_START
            THEN DO;
                 $JOB_TO_BE_PUSHED.EARLY_START =
                                     $PUSHING_JOB.EARLY_FINISH ;
                 $JOB_TO_BE_PUSHED.EARLY_FINISH =
                                  $JOB_TO_BE_PUSHED.EARLY_START
                                + $JOB_TO_BE_PUSHED.DURATION ;
                 $INCREMENT_CANDIDATES(NEXT) =
                                     $JOB_TO_BE_PUSHED.SUCCESSOR ;
                 LABEL($INCREMENT_CANDIDATES(LAST)) =
                                     LABEL($JOB_TO_BE_PUSHED) ;

                 END;
      END;
      PRUNE $INCREMENT_CANDIDATES(FIRST) ;
```

```
      END;
   END;    /*  INCREMENT_SUCCESSOR_TIMES   */

   RETRIEVE_CRITICAL_PATH_DATA:  PROCEDURE ($TARGET) ;
   /*  THIS CODE TRANSFERS THE CRITICAL PATH DATA OF $JOB INTO $TARGET. */
      GRAFT $JOB.START.EARLY AT $TARGET.EARLY_START ;
      GRAFT $JOB.FINISH.EARLY AT $TARGET.EARLY_FINISH ;
      GRAFT $JOB.START.LATE AT $TARGET.LATE_START ;
      GRAFT $JOB.FINISH.LATE AT $TARGET.LATE_FINISH ;
      GRAFT $JOB.SLACK.TOTAL AT $TARGET.TOTAL_SLACK ;
      GRAFT $JOB.SLACK.FREE AT $TARGET.FREE_SLACK ;
   END;    /*  RETRIEVE_CRITICAL_PATH_DATA   */
     SCHEDULABLE_JOB_ORDER: PROCEDURE ($SCHEDULABLE_JOBS) ;
   /*                                                                      */
   /*  ORDER $SCHEDULABLE_JOBS BY LOWEST VALUE OF LATE_START,              */
   /*     FREE_SLACK, TOTAL_SLACK AND DURATION                             */
   /*                                                                      */
        IF $SCHEDULABLE_JOBS(FIRST) IDENTICAL TO $NULL  THEN RETURN ;
        DO FOR ALL SUBNODES OF $SCHEDULABLE_JOBS USING $SJOB ;
          GRAFT INSERT $SJOB.LATE_START  BEFORE $SJOB(FIRST) ;
          GRAFT INSERT $SJOB.FREE_SLACK  BEFORE $SJOB(FIRST) ;
          GRAFT INSERT $SJOB.TOTAL_SLACK BEFORE $SJOB(FIRST) ;
          GRAFT INSERT $SJOB.DURATION    BEFORE $SJOB(FIRST) ;
        END;
        NUMBER_OF_CRITERIA = 4 ;
        MAXMIN_FLAG = -1 ;
        CALL MULTI_CRITERIA_ORDER
             ( $SCHEDULABLE_JOBS,NUMBER_OF_CRITERIA,MAXMIN_FLAG ) ;
     END;  /* SCHEDULABLE_JOB_ORDER   */
   ORDER_BY_JOB_START: PROCEDURE ($SCHEDULE) ;
   /*  ORDERS $SCHEDULE BY  -JOB_INTERVAL.START                            */
      DECLARE $JAB LOCAL ;
      DO FOR ALL SUBNODES OF $SCHEDULE USING $JAB ;
         INSERT $JAB.JOB_INTERVAL.START BEFORE $JAB(FIRST) ;
      END;
      CALL MULTI_CRITERIA_ORDER($SCHEDULE, 1, -1) ;
      DO FOR ALL SUBNODES OF $SCHEDULE USING $JAB ;
         PRUNE $JAB(FIRST) ;
      END;
   END;  /*  ORDER_BY_JOB_START   */
   ALLOCATOR_FINAL_ORDER: PROCEDURE ($SCHEDULE) ;
   /*  ORDERS $SCHEDULE BY -JOB_INTERVAL.START, -DURATION                  */
      DECLARE $JAB LOCAL ;
      DO FOR ALL SUBNODES OF $SCHEDULE USING $JAB ;
         INSERT $JAB.JOB_INTERVAL.START BEFORE $JAB(FIRST) ;
         GRAFT INSERT $JAB.DURATION BEFORE $JAB(FIRST) ;
      END;
      CALL MULTI_CRITERIA_ORDER($SCHEDULE, 2, -1 ) ;
      DO FOR ALL SUBNODES OF $SCHEDULE USING $JAB ;
         PRUNE $JAB(2) ;
      END ;
    END;  /*  ALLOCATOR_FINAL_ORDER   */
      END;  /*  RESOURCE_ALLOCATOR   */
```

ORIGINAL PAGE IS
OF POOR QUALITY

2.4.33 RESOURCE_LEVELER

2.4.33 RESOURCE_LEVELER

2.4.33.1 Purpose and Scope

In many project scheduling situations, the pattern of resource utilization is often more important than the quantity of resources used. For example, a resource feasible schedule that results in rapidly changing resource requirements is clearly undesirable from the project control standpoint. In these situations it is useful to perform resource leveling in order to reduce resource profile fluctuations.

Conceptually, a resource utilization profile is level when the actual quantity of resource used in each time period is constant. Unfortunately, it is not generally possible to maintain perfectly level profiles and simultaneously satisfy all of the scheduling constraints. As a consequence, some fluctuations will inevitably remain in the resource profiles. The purpose of this module is then to minimize these remaining resource variations. This is accomplished by heuristically minimizing the sum of the squares of the resources over time, subject to the network, resource availability, and activity completion constraints.

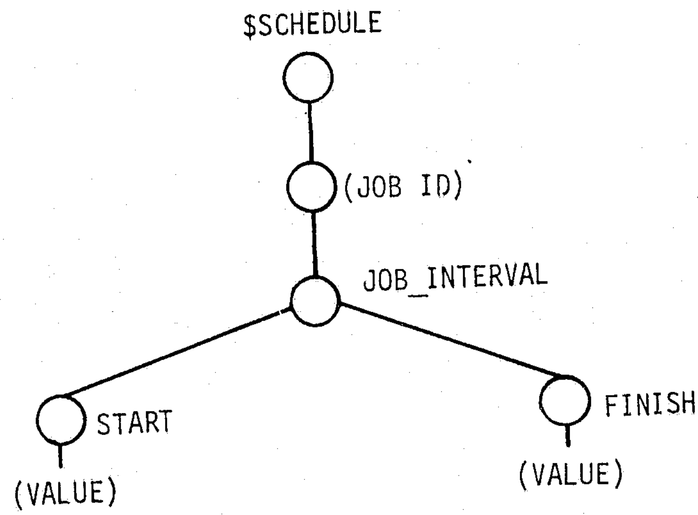
This module is applicable to the general class of project scheduling problems that includes multiple resources with time varying pool levels.

2.4.33.2 Modules Called

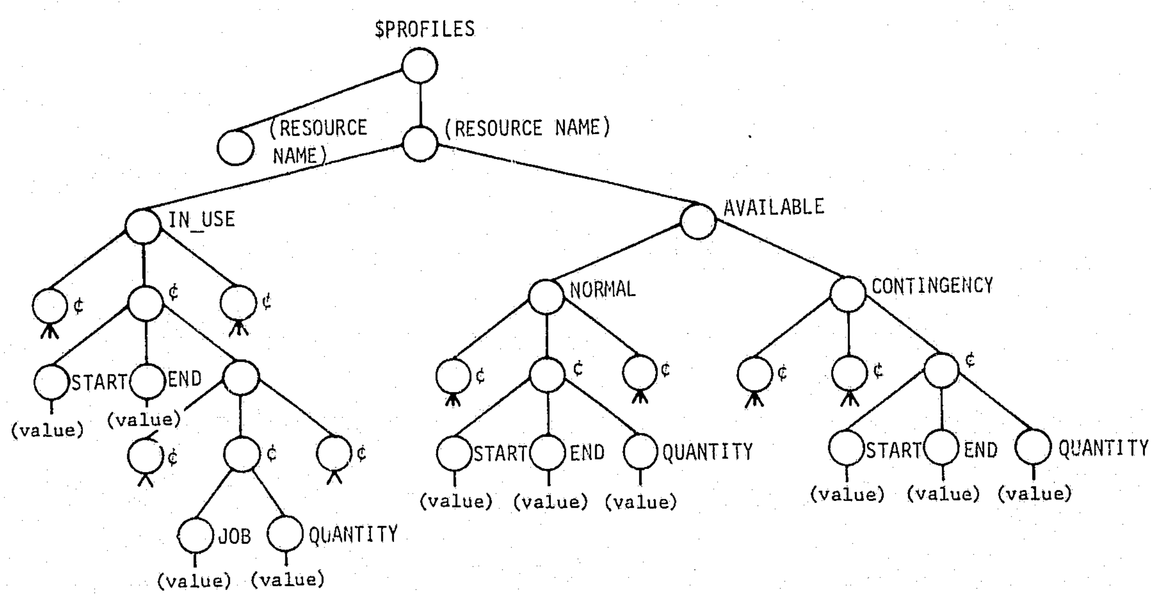
None.

2.4.33.3 Module Input

1) Nominal Schedule ($SCHEDULE)



2) Nominal Resource Profile ($PROFILES)



2.4.33.4 Module Output

1) Revised "Level" Schedule ($SCHEDULE)

   (Same structure as input.)

2) Revised "Level" Resource ($PROFILE)

   (Same structure as input.)

2.4.33.5 Functional Description

The resource leveling procedure is based on the minimization of the sum of the squares of the resources over time. The formulation of resource leveling as a least squares minimization problem is motivated by the fact that a level profile minimizes the sum of the squares of the resources subject to the constaint that the area under the profile is constant. A simple example best illustrates this principle. Consider a single resource defined over T equally spaced time periods of unit duration, as illustrated in Fig. 2.4.33-1. Furthermore, let r_i denote the quantity of resource used in period i. It is then possible to show that the level profile

$r_i = R/T, \; i = 1,2,\ldots, I;$

minimizes : $\sum_i r_i^2,$

subject to the

constraint: $\sum_i r_i = R$.

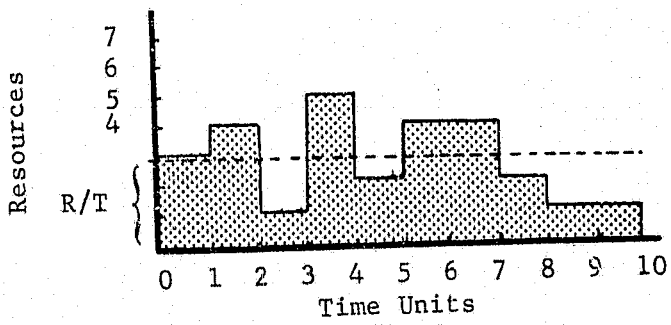


Fig. 2.4.33-1
Profile for Single Resource

Unfortunately, this simple leveling concept neglects the network, the resource, and the integer start time constraints. When these constraints are included the resource leveling problem is stated as follows.

Determine the activity start times s_i, i = 1, 2, ... , I, that minimize

[1] $$F\left(s_1, s_2, \ldots, s_I\right) = \sum_{k=1}^{K} \sum_{t=1}^{T} \left(\sum_{i \in \mathcal{A}_t} r_{ki} \left(t - s_i\right) \right)^2,$$

subject to network constraints

[2] $$s_i \geq f_j \text{ for all } j \in \mathcal{P}_i;\ i = 1, 2, \ldots, I$$

Resource constraints

[3] $$\sum_{i \in \mathcal{A}_i} r_{ki} \left(t - s_i\right) \leq R_k(t), \text{ for all } t \in (s,f)$$

Activity completion constraints

[4] $$\sum_{k} \sum_{t} \sum_{i} r_{ki} \left(t - s_i\right) = C$$

Integer start times

[5] $$s_i \in \mathcal{A}^{\ell}, \text{ for all } i.$$

In this formulation, K is the number of resource types, T is the number of scheduling time intervals, $\mathcal{A}_t$ is the set of indices for the resources being used during time interval t, $r_{ki}(t - s_i)$ is the quantity of resource type k used by activity i in period t, and $R_k(t)$ is the time varying resource level for resource k.

This formulation, which represents a nonlinear integer program, cannot be cost-effectively solved with an existing algorithm. As a result, a heuristic algorithm is used to obtain a solution. This heuristic assumes that a nominal schedule is input that satisfied all of these constraints. A new start time is then determined for activity i by minimizing the partial sum

[6] $$F\left(s_i\right) = \sum_{k=1}^{K} \sum_{t=s_i^*}^{f_i^\ell} \left(\sum_{j \in \mathcal{A}_t - \{i\}} r_{kj} \left(t - s_i^*\right) + r_{ki} \left(t - s_i\right) \right)^2$$

In the above experession, s_i^* is obtained from input and is the original scheduled start time of activity i, and f_i^ℓ is the late finish of activity i. The restriction of the minimization of $F\left(s_i\right)$ to the interval defined by s_i^* and f_i^ℓ insures that the network constraints remain satisfied. The resource limits are satisfied by setting the function $F(\cdot)$ to an arbitrarily large value for all start times that result in a resource violation in some time period. This ensures that start times that are resource infeasible are not selected in the minimization process. If more than one start time produces the same minimum value of F, given by

[7] $$F = \underset{s_i^* \leq s_i \leq s_i^\ell}{\text{minimum}} \left\{F\left(s_i\right)\right\},$$

than the latest start time is selected. This gives the algorithm more freedom to delay earlier scheduled activities.

The sequence in which the new start times are calculated is determined by ordering the set of activities according to: (1) latest scheduled finish and (2) largest residual total slack. Thus, the first activity to be rescheduled is the last activity completed in the nominal schedule. Rescheduling proceeds accordingly until a new start time for all activities are calculated. The entire leveling process can then be repeated until the resulting profile remains unchanged.

Clearly, this sequential one-dimensional minimization heuristic does not necessarily produce the actual optimum start times. However, our limited experience indicates that this simple procedure can significantly smooth resource profiles. This is particularly true when this algorithm is applied to schedules generated by resource allocation heuristics that schedule as soon as the required resources are available. Rescheduling in a reverse least square fashion naturally delays activities providing more free float to earlier activities that must be rescheduled to level resources. This natural delaying action tends to shift resource "peaks" that occur early in the scheduling horizon in order to fill later resource "valleys." This also offsets the "tailoff" problem associated with project completion.

As indicated in the formulation, the resource requirements can vary over the duration of the activity and, similarly, the resource pool levels can vary over the duration of the project. This is illustrated in Fig. 2.4.33-2 for an arbitrary resource type.

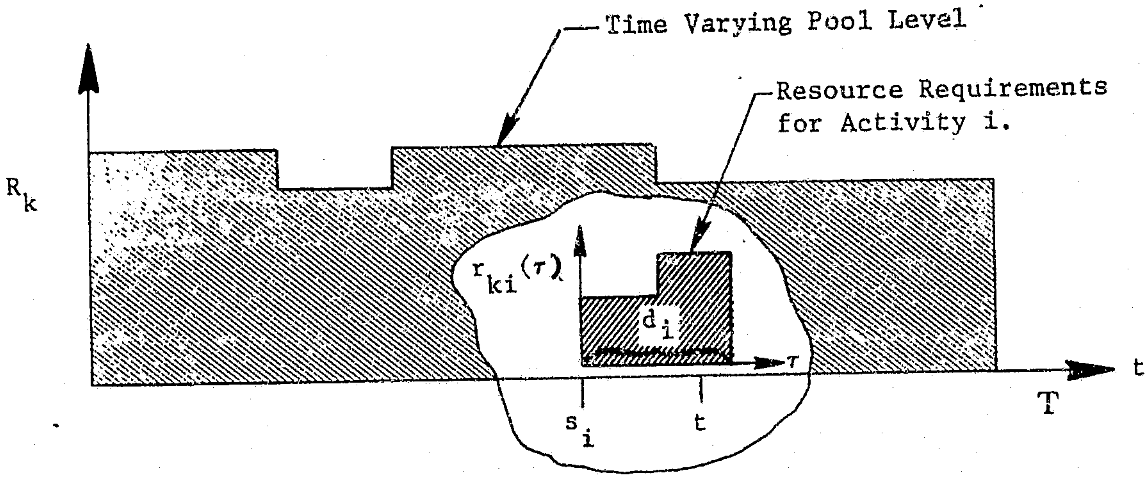
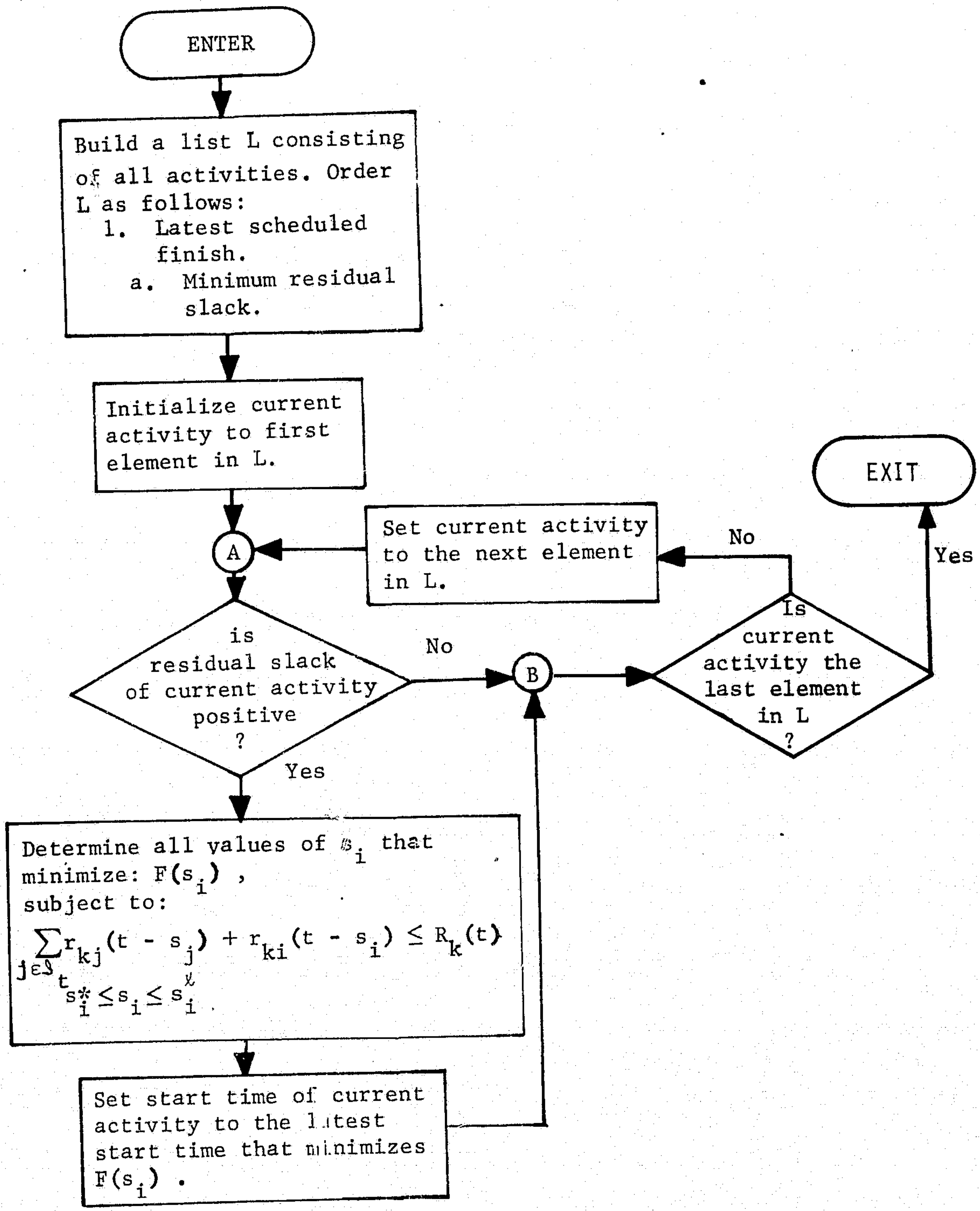


Fig. 2.4.33-2 Time-Varying Resource Variables

This module can also be easily modified to solve the resource profile shaping problem. This can be accomplished by minimizing the square of the differences between actual and desired resource profiles.

2.4.33.6 Functional Block Diagram



C. 6

2.4.33.7 Typical Application

Resource leveling techniques are used to distribute resources over time to reduce profile fluctuations. They can also be used to determine the maximum resource levels required to meet project completion dates.

A simple example is the best way to illustrate the application of this module. Consider, for example, the simple network shown in Fig. 2.4.33-3. The schedule given in Fig. 2.4.33-4 was generated by setting the activity start times equal to their critical path early starts. This schedule requires the resource profile shown in Fig. 2.4.33-4. As usual, the critical path early start heuristic gives a schedule that is very front loaded. Application of the resource leveling module delays several activities in order to reduce the peak resource requirement. The new schedule and the corresponding resource profile is given in Fig. 2.4.33-5. In this example, the least squares heuristic reduced the sum of the squares from 2137 to 1261 units^2 and reduced the peak level from 24 to 19 units. However, this heuristic solution is not optimal. For example, at least one better solution is illustrated in Fig. 2.4.33-6. For this schedule, the sum of the squares is 1215 units^2 and the peak resource requirement is only 15 units. This example clearly illustrates the fact that heuristic algorighms give "good" but not "optimal" solutions.

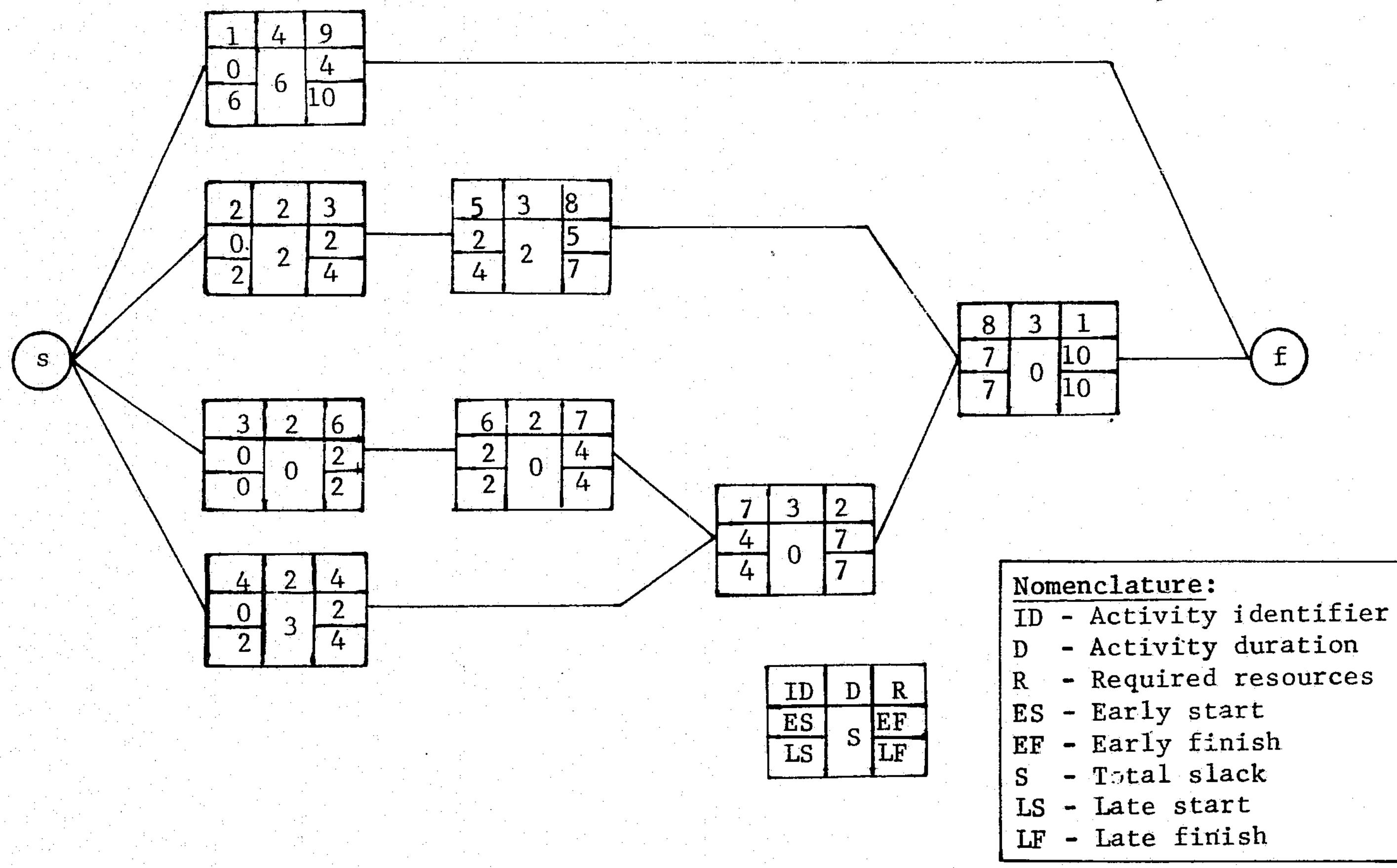


Fig. 2.4.33-3 Example Project Network

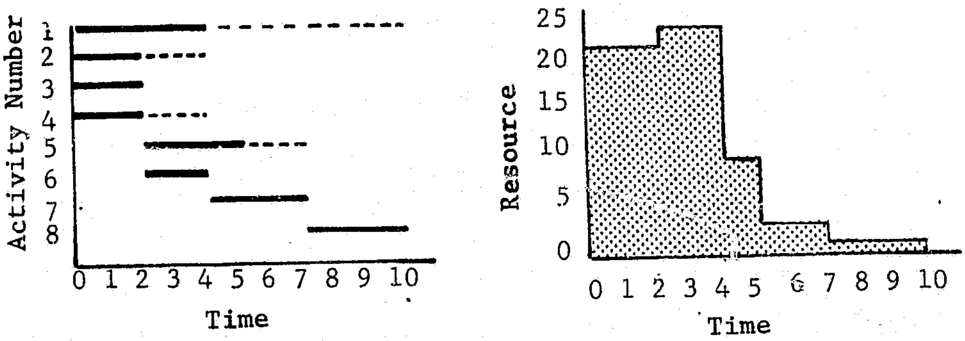


Fig. 2.4.33-4 Nominal Schedule Using CPM Early Starts

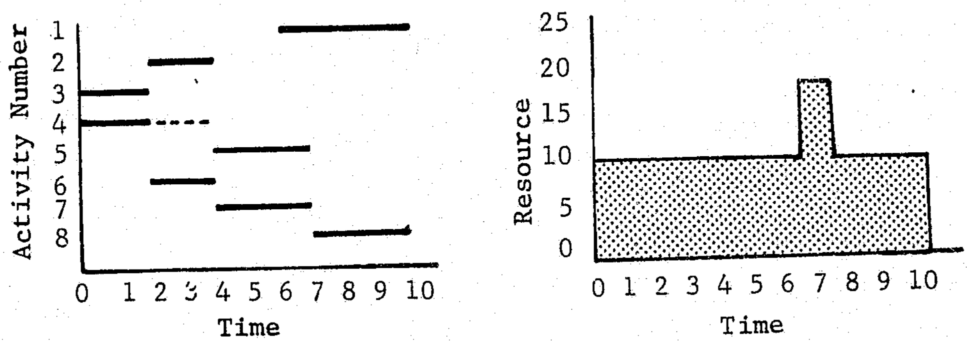


Fig. 2.4.33-5 Rescheduled Using RESOURCE_LEVELER

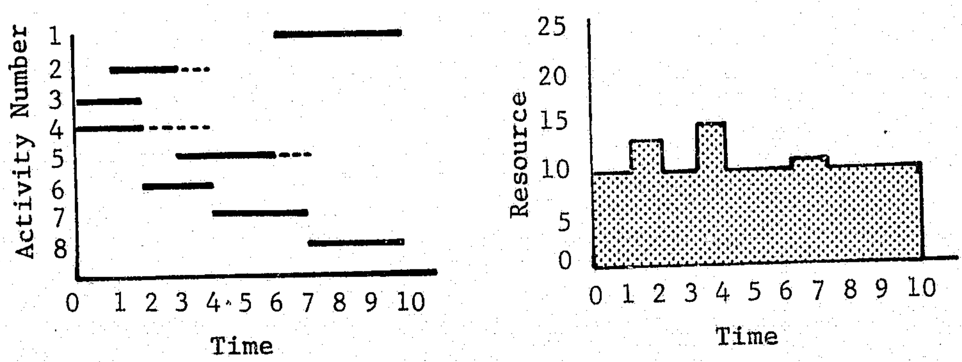


Fig. 2.4.33-6 "Hand" Scheduled Solution

2.4.33.8 Implementation Considerations

Implementation of this simple heuristic is straightforward. The only significant computation is the calculation of the minimum of $F(s_i)$. One approach for obtaining this minimum is outlined in Fig. 2.4.33-7.

When the individual resource quantities are different by more than two orders of magnitude over an appreciable segment of the schedule, then weighted least squares are required. This amounts to adding a weighting matrix to the least squares formula. In most normal situations scaling the resources on a percentage basis is adequate to ensure that each resource type contributes reasonably to the sum of the squares. Percentage scaling gives the diagonal weighting matrix

[8] $$W_i = \begin{bmatrix} w_{1i} & & & & \\ & w_{2i} & & & \\ & & \cdot & & \\ & & & \cdot & \\ & & & & w_{Ki} \end{bmatrix}$$

where

[9] $$w_{Ki} = \left[\sum_t r_{ki}(t)/T\right]^{-1}$$

2.4.33.9 References

Burman, P. J., *Precedence Networks for Project Planning and Control*, McGraw Hill, London, 1972.

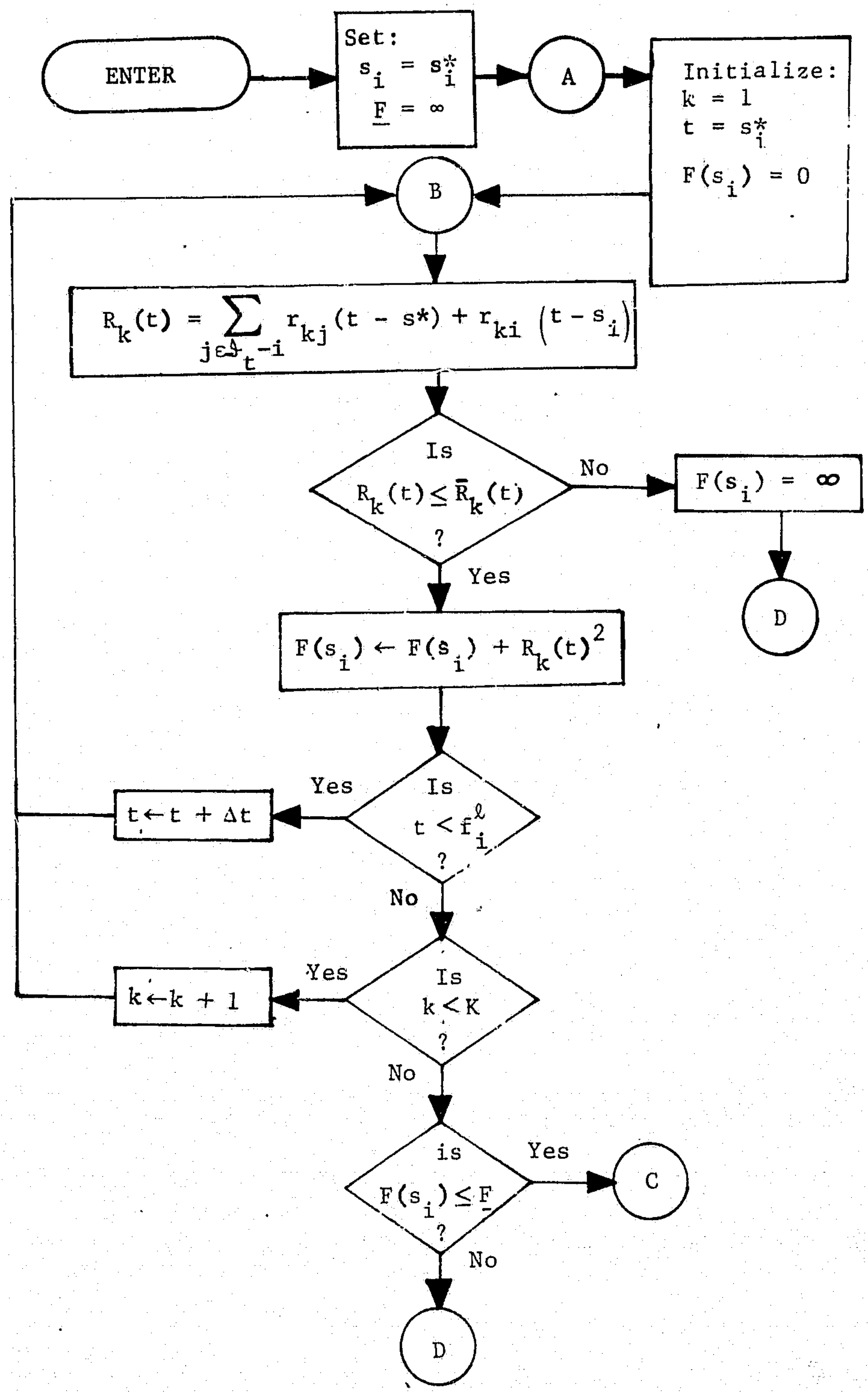
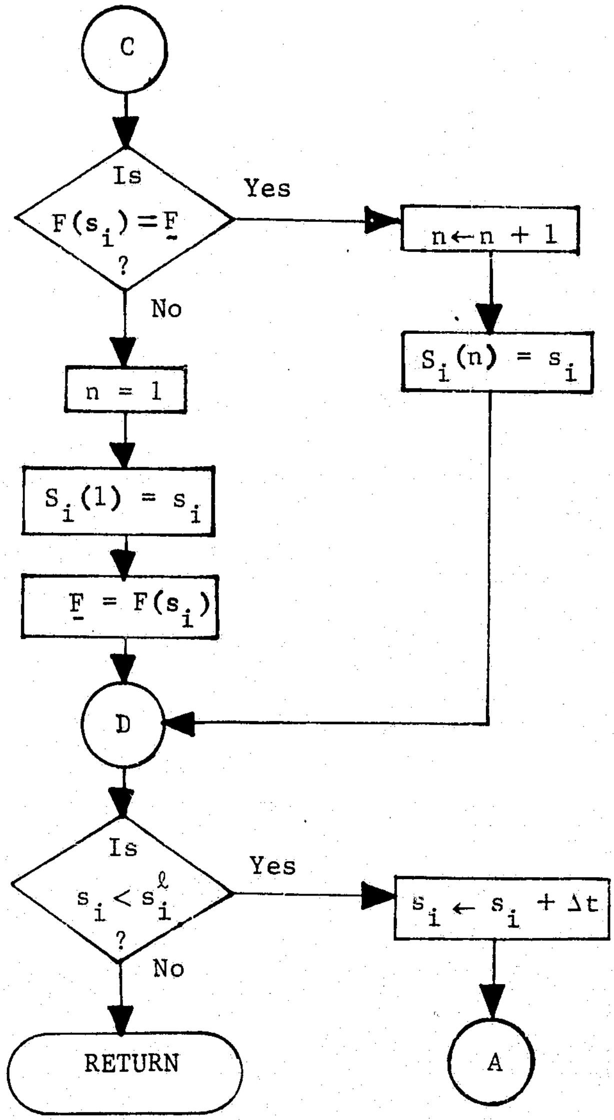


Fig. 2.4.33-7 Detailed Diagram of $\min(F(s_i))$



Nomenclature:

t	- Basic Time Increment
$\underline{F}$	- Lower Bound on the Least Squares Function
$\overline{S}_i(n)$	- Array Containing the Minimum Least Squares Start Times

Fig. 2.4.33-7 (concl)

2.4.33.8 DETAILED DESIGN

This module levels the resource utilization profiles (in $PROFILES) for the set of jobs input in $SCHEDULE. This is accomplished by systematically moving the jobs around on the time line within their given resource and temporal relation constraints. The selection of rescheduling time points is based on minimization of the sum of squares of the resource usage levels over time. On output, $SCHEDULE and $PROFILES will contain revisions to reflect the scheduling charges.

2.4.33.9 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

$SCHEDULE	-	the schedule built by RESOURCE_ALLOCATOR
$PROFILES	-	tree containing all pooled resource information
$RESOURCE_SAVE_AREA	-	temporary storage area for excess resource information
$JOB	-	points at subnodes of $SCHEDULE
$JOB_INFO	-	summarizes critical path information of $JOB
MODIFIED_SCHEDULE_FLAG	-	if equal to 1, indicates job can be rescheduled.
I_EARLIEST_SUCCESSOR_START	-	indicates the earliest end time of all successors of $JOB
I_START	-	temporary variable used to determine I_EARLIEST_SUCCESSOR_START
I_JOB_START	-	start of $JOB
LATEST_START	-	difference between the start of the earliest successor and the end of $JOB plus $JOB's start
FUNCTION_MINIMUM	-	cumulative sum of squares of $JOB_PROFILES
$POOL	-	points at pooled resources required by $JOB
I, J	-	internal counters
$JOB_PROFILES	-	contains pooled resource information about $JOB
$FAILURE_INDICATOR	-	indicates the time after which resource constraint violations are allowed
$JOBID_TREE	-	label of $JOB
I_RESCHEDULE_TIME	-	the time a job can be rescheduled
$TEMP_PROFILES	-	equivalent to $JOB_PROFILES
I_THRESHOLD_FLAG	-	indicates whether-or-not contingency level resources can be allocated
FUNCTION	-	cummulative sum of squares of $PROFILE
$PROFILE	-	points at subnodes of $TEMP_PROFILES

$POOL_PROFILE	-	contains pooled resource information for a given resource
SUM_OF_SQUARES	-	cumulative totals of SUM for each interval in $POOL_PROFILE
TOTAL	-	cumulative totals of resource quantities
SUM	-	equals the product of $TOTAL^2$ and the interval length of the $POOL_PROFILE
$JAB	-	points at subnodes of $SCHEDULE
$RESOURCE_LABEL	-	keeps the labels of resources in $JOBLIST so that the resource information can be returned to its original state in $JOBLIST
$TYPE	-	points at the subnodes of $RESOURCE_SAVE_AREA
$POOL_INFO	-	contains condensed resource information.
$PARAMETER	-	temporary storage for condensed resource information before being put into $POOL_INFO
$INFO	-	points at subnodes of $POOL
$JOB_RESOURCES	-	points at subnodes of $RESOURCE_SAVE_AREA
$JOB_ID	-	contains label of $JOB_RESOURCES
$CONDENSED	-	intermediate tree in the return from condensed resource information to original state
ICRIT	-	index for 1 to NUMBER_OF_CRITERIA
$LIST	-	contains the list to be ordered by different criteria
$LIST_ELEMENT	-	points to the subnodes of $LIST
$TRANSFER_INDICES	-	tracks the indices of the variables in $LIST for ordering purposes

I_TRANSFER_INDEX	-	equals the value of $TRANSFER_INDICES
$NEW_LIST	-	contains the ordered $LIST
$IN_USE	-	the in use portion of the current resource
$POOL_PROFILE	-	contains the resource profile for a given resource
I_CHECK_TIME	-	contains the time in $FAILURE_INDICATOR
NUMBER_OF_UPDATES	-	equals 2 if contingency checks are to be made, equals 1 if not the case
$MODE	-	reflects the user choice to search contingency levels of resources or not
$JOB_INTERVAL	-	points at the subnodes of $POOL
I_START	-	the sum of I_JOB_START and the start of the resource availability in $POOL
I_END	-	the sum of I_JOB_START and the end of the resource availability in $POOL
QUANTITY_DELTA	-	the initial quantity of the resource in $POOL being processed
$INTERVAL	-	the inuse portion of $POOL_PROFILE
NLAST	-	the number of subnodes of $INTERVAL
$NEW_INTERVAL	-	equivalent to $NULL, used to insert null nodes onto $INTERVAL
$INTERVAL1	-	calling argument, containing interval information and resources to be updated by QUANTITY_DELTA
$INTERVAL2	-	if it is not empty, then its associated quantity is taken as the initial quantity of $INTERVAL1
I_POINTER	-	pointer which is decremented if $INTERVAL1 and $INTERVAL2 can be combined.

2.4.33.10 COMMENTED CODE

```
RESOURCE_LEVELER:  PROCEDURE ($SCHEDULE, $PROFILES)
               OPTIONS(EXTERNAL);
/*******************************************************************/
/*                                                                 */
/*  THIS MODULE LEVELS THE RESOURCE UTILIZATION PROFILES (IN       */
/*  $PROFILES) FOR THE SET OF JOBS INPUT IN $SCHEDULE.  THIS IS    */
/*  ACCOMPLISHED BY SYSTEMATICALLY MOVING THE JOBS AROUND ON THE   */
/*  TIME LINE WITHIN THEIR GIVEN RESOURCE AND TEMPORAL RELATION    */
/*  CONSTRAINTS.  THE SELECTION OF RESCHEDULING TIME POINTS IS     */
/*  BASED ON MINIMIZATION OF THE SUM OF SQUARES OF THE RESOURCE    */
/*  USAGE LEVELS OVER TIME.  ON OUTPUT, $SCHEDULE AND $PROFILES    */
/*  WILL CONTAIN REVISIONS TO REFLECT THE SCHEDULING CHANGES.      */
/*                                                                 */
/*******************************************************************/
   DECLARE $FAILURE_INDICATOR,FUNCTION,FUNCTION_MINIMUM,
           I,I_EARLIEST_SUCCESSOR_START,I_JOB_START,
           I_RESCHEDULE_TIME,I_START,$INTERVAL,J,$JOB,$JOB_INFO,
           $JOB_PROFILES,LATEST_START,MODIFIED_SCHEDULE_FLAG,
           $POOL,$PROFILE,$RESOURCE_SAVE_AREA,$SUCCESSOR,SUM,
           LOCAL_INFINITY, $NILL, $JOBID_TREE,
           SUM_OF_SQUARES,$TEMP_PROFILES,TOTAL,TOTAL_QUANTITY LOCAL ;
           $NILL = $NULL ;
           LOCAL_INFINITY = 16000 ;
/*  THIS CODE CONDENSES THE DATA STRUCTURE OF $SCHEDULE TO INCREASE  */
/*  THE EFFICIENCY OF NODE ACCESSES IN THE REST OF THE PROGRAM.      */
   CALL CONDENSE_RESOURCE_INFORMATION($SCHEDULE,$RESOURCE_SAVE_AREA) ;
   DO FOR ALL SUBNODES OF $SCHEDULE USING $JOB ;
      GRAFT $JOB.TEMPORAL_RELATION.SUCCESSOR AT $JOB_INFO.SUCCESSOR ;
      GRAFT $JOB.SLACK.TOTAL AT $JOB_INFO.TOTAL_SLACK ;
      GRAFT $JOB.JOB_INTERVAL.START AT $JOB_INFO.START ;
      GRAFT $JOB.JOB_INTERVAL.END AT $JOB_INFO.END ;
      CALL COMBINE_TREES($JOB_INFO,$JOB) ;
      END ;
/*  NOW THE SUM OF SQUARES DATA IS COMPUTED FOR EACH USAGE INTERVAL  */
/*  AND TOTALED FOR EACH POOL BEING CONSIDERED.                      */
   DO FOR ALL SUBNODES OF $PROFILES USING $PROFILE ;
      CALL COMPUTE_SUM_OF_SQUARES($PROFILE);
      END ;
SMOOTH_RESOURCE_PROFILE:
   MODIFIED_SCHEDULE_FLAG = 0 ;
/*******************
   ORDER $SCHEDULE BY END,TOTAL_SLACK ;
*****************/
 CALL SCHEDULE_ORDER($SCHEDULE) ;
/*  EXAMINE EACH JOB IN THE SCHEDULE TO DETERMINE IF IT CAN BE       */
/*  RESCHEDULED AT A LATER TIME TO ACHIEVE A MORE LEVEL RESOURCE     */
/*  USAGE PROFILE.                                                   */
   DO FOR ALL SUBNODES OF $SCHEDULE USING $JOB ;
      IF $JOB.TOTAL_SLACK > 0
         THEN DO ;
/*  DETERMINE THE SET OF TIMES AT WHICH RESCHEDULING CAN BE DONE     */
```

```
/*  WITHOUT VIOLATING ANY TEMPORAL RELATION CONSTRAINTS.                   */
            IF $JOB.SUCCESSOR(FIRST) IDENTICAL TO  $NULL
               THEN I_EARLIEST_SUCCESSOR_START =$SCHEDULE(FIRST).END;
               ELSE DO ;  I_EARLIEST_SUCCESSOR_START=LOCAL_INFINITY;
                    DO FOR ALL SUBNODES OF $JOB.SUCCESSOR USING
                                                            $SUCCESSOR ;
                       I_START = $SCHEDULE.#($SUCCESSOR).START ;
                       IF I_START < I_EARLIEST_SUCCESSOR_START
                          THEN I_EARLIEST_SUCCESSOR_START = I_START ;
                       END ;
                    END ;
            I_JOB_START = $JOB.START ;
            LATEST_START = I_EARLIEST_SUCCESSOR_START - $JOB.END
                                                       + I_JOB_START ;
            IF LATEST_START > I_JOB_START
/*  DETERMINE THE SET OF TIMES AT WHICH RESCHEDULING CAN BE DONE         */
/*  WITHOUT VIOLATING ANY RESOURCE CONSTRAINTS.                           */
               THEN DO ;  FUNCTION_MINIMUM = 0.0 ;   I = 0 ;
/*  FIRST, RESOURCES FOR THIS JOB MUST BE DEALLOCATED TO MAKE            */
/*  $PROFILES LOOK AS THOUGH THIS JOB HAS BEEN UNSCHEDULED.              */
                    DO FOR ALL SUBNODES OF $JOB.RESOURCE USING $POOL ;
                       I = I+1 ;
                       GRAFT INSERT $PROFILES.#LABEL($POOL) BEFORE
                                                  $JOB_PROFILES(FIRST);
                       FUNCTION_MINIMUM = FUNCTION_MINIMUM +
                                $JOB_PROFILES(FIRST).SUM_OF_SQUARES;
                       $FAILURE_INDICATOR = LATEST_START +
                                                    $POOL(LAST).END ;
                       $JOBID_TREE = LABEL($JOB) ;
                       CALL UPDATE_POOL_LEVELS
                               ($POOL,$JOBID_TREE,$JOB_PROFILES,
                               I_JOB_START,0,0,$FAILURE_INDICATOR) ;
                       IF $FAILURE_INDICATOR  NOT IDENTICAL TO  $NULL
                        . THEN DO ;  J = 1 ;
                               DO FOR ALL SUBNODES OF $JOB.RESOURCE
                                                       USING $POOL ;
                                  IF I=J  THEN GO TO CHECK_NEXT_JOB ;
                       $JOBID_TREE = LABEL($JOB) ;
                                  CALL UPDATE_POOL_LEVELS($POOL,
                                       $JOBID_TREE,$JOB_PROFILES,
                                            I_JOB_START,0,1,$NULL) ;
                                  J = J+1 ;
                                  END ;
                               END ;
                       END ;
                    I_RESCHEDULE_TIME = I_JOB_START ;
/*  RESCHEDULING OF THIS JOB IS ATTEMPTED AT EACH FEASIBLE TIME          */
/*  POINT LATER THAN THE JOBS CURRENT SCHEDULED START TIME.              */
                    DO I=I_JOB_START+1 TO LATEST_START ;
                       $TEMP_PROFILES = $JOB_PROFILES ;
                       J = 0 ;
```

```
                         DO FOR ALL SUBNODES OF $JOB.RESOURCE USING
                                                            $POOL ;
                            J = J+1 ;
 IF $TEMP_PROFILES.#LABEL($POOL).AVAILABLE.CONTINGENCY(FIRST)=$NULL
 THEN I_THRESHOLD_FLAG=0;
 ELSE I_THRESHOLD_FLAG=1;
                            $JOBID_TREE = LABEL($JOB) ;
                            CALL UPDATE_POOL_LEVELS($POOL,$JOBID_TREE,
                        $TEMP_PROFILES,I,I_THRESHOLD_FLAG,1,
                        $FAILURE_INDICATOR);
                            IF $FAILURE_INDICATOR  NOT IDENTICAL TO
                                                                 $NULL
                               THEN DO ;
                                    GRAFT INSERT $POOL BEFORE $JOB.
                                                   RESOURCE(FIRST);
                                    GO TO CHECK_NEXT_TIME_POINT ;
                                    END ;
                            END ;
/*   SINCE THE ABOVE RESCHEDULING ATTEMPT WAS SUCCESSFUL, THE SUM OF   */
/*   SQUARES DATA IS CALCULATED TO SEE IF THIS WOULD PROVIDE AN        */
/*   IMPROVED SCHEDULE.                                                */
                         FUNCTION = 0.0 ;
                         DO FOR ALL SUBNODES OF $TEMP_PROFILES USING
                                                          $PROFILE ;
                            CALL   COMPUTE_SUM_OF_SQUARES($PROFILE) ;
                            FUNCTION = FUNCTION +
                                       $PROFILE.SUM_OF_SQUARES ;
                            END ;
                         IF FUNCTION <= FUNCTION_MINIMUM
                            THEN DO ;  FUNCTION_MINIMUM = FUNCTION ;
                                       I_RESCHEDULE_TIME = I ;
                                 END ;
CHECK_NEXT_TIME_POINT:   END ;
/*   THE JOB BEING CONSIDERED IS NOW RESCHEDULED, EITHER AT ITS        */
/*   ORIGINAL START TIME OR AT A LATER TIME RESULTING IN A MORE        */
/*   LEVEL RESOURCE USAGE PROFILE.                                     */
                      IF I_RESCHEDULE_TIME = LATEST_START
                         THEN GRAFT $TEMP_PROFILES AT $JOB_PROFILES ;
                         ELSE DO FOR ALL SUBNODES OF $JOB.RESOURCE
                                                      USING $POOL ;
 IF $JOB_PROFILES.#LABEL($POOL).AVAILABLE.CONTINGENCY(FIRST)=$NULL
  THEN I_THRESHOLD_FLAG=0;
  ELSE I_THRESHOLD_FLAG=1;
                              $JOBID_TREE = LABEL($JOB) ;
                              CALL UPDATE_POOL_LEVELS($POOL,
                                    $JOBID_TREE,$JOB_PROFILES,
                                    I_RESCHEDULE_TIME,
                                  I_THRESHOLD_FLAG,1,$NULL) ;
                              CALL COMPUTE_SUM_OF_SQUARES
                                   ($JOB_PROFILES.#LABEL($POOL)) ;
                              END ;
```

```
                        IF I_RESCHEDULE_TIME ¬= I_JOB_START
                           THEN DO ;
                                MODIFIED_SCHEDULE_FLAG = 1 ;
                                $JOB.START = I_RESCHEDULE_TIME ;
                                $JOB.END = $JOB.END + I_RESCHEDULE_TIME -
                                                      I_JOB_START ;
                                END ;
                        CALL COMBINE_TREES($JOB_PROFILES,$PROFILES) ;
                        END ;
                END ;
CHECK_NEXT_JOB:
      END ;
/*  IF ANY OF THE JOBS WERE MOVED AS A RESULT OF THE LAST LEVELING     */
/* PASS, MAKE ANOTHER PASS TO SEE IF A MORE LEVEL SCHEDULE CAN BE      */
/*  ATTAINED.                                                          */
   IF MODIFIED_SCHEDULE_FLAG ¬= 0  THEN GO TO SMOOTH_RESOURCE_PROFILE ;
/*  NOW THE SUM OF SQUARES DATA IS ELIMINATED FROM $PROFILES AND       */
/*  $SCHEDULE IS RESTORED TO ITS ORIGINAL FORM.                        */
   DO FOR ALL SUBNODES OF $PROFILES USING $PROFILE ;
      PRUNE $PROFILE.SUM_OF_SQUARES ;
      DO FOR ALL SUBNODES OF $PROFILE.IN_USE USING $INTERVAL ;
         PRUNE $INTERVAL.SUM ;
         END ;
      END ;
/******************
   ORDER $SCHEDULE BY -START,-DURATION ;
******************/
   CALL FINAL_ORDER($SCHEDULE) ;
   CALL RESTORE_RESOURCE_INFORMATION($SCHEDULE,$RESOURCE_SAVE_AREA) ;
   DO FOR ALL SUBNODES OF $SCHEDULE USING $JOB ;
      GRAFT $JOB.SUCCESSOR AT $JOB.TEMPORAL_RELATION.SUCCESSOR ;
      GRAFT $JOB.TOTAL_SLACK AT $JOB.SLACK.TOTAL ;
      GRAFT $JOB.START AT $JOB.JOB_INTERVAL.START ;
      GRAFT $JOB.END AT $JOB.JOB_INTERVAL.END ;
      END ;



  COMPUTE_SUM_OF_SQUARES:  PROCEDURE ($POOL_PROFILE) ;
/*  THIS PROCEDURE RECOMPUTES THE SUM OF SQUARES DATA FOR A            */
/*  RESOURCE USAGE PROFILE ($POOL_PROFILE.IN_USE).                     */
   DECLARE  SUM_OF_SQUARES, $INTERVAL, $JAB, TOTAL,SUM   LOCAL ;
      SUM_OF_SQUARES = 0.0 ;
      DO FOR ALL SUBNODES OF $POOL_PROFILE.IN_USE USING $INTERVAL ;
         TOTAL = 0.0 ;
         DO FOR ALL SUBNODES OF $INTERVAL.USAGE USING $JAB ;
            TOTAL = TOTAL + $JAB.QUANTITY ;
            END ;
         $INTERVAL.TOTAL = TOTAL ;
```

```
        SUM = TOTAL*TOTAL*($INTERVAL.END - $INTERVAL.START) ;
        $INTERVAL.SUM = SUM ;
        SUM_OF_SQUARES = SUM_OF_SQUARES + SUM ;
        END ;
     $POOL_PROFILE.SUM_OF_SQUARES = SUM_OF_SQUARES ;
END;  /*      COMPUTE_SUM_OF_SQUARES    */
SCHEDULE_ORDER: PROCEDURE ($SCHEDULE) ;
/*   ORDER $SCHEDULE BY END, TOTAL_SLACK                                   */
   DO FOR ALL SUBNODES OF $SCHEDULE USING $JOB ;
      GRAFT INSERT $JOB.END BEFORE $JOB(FIRST) ;
      GRAFT INSERT $JOB.TOTAL_SLACK BEFORE $JOB(FIRST) ;
   END;
   NUMBER_OF_CRITERIA = 2 ;
   CALL MULTI_CRITERIA_ORDER($SCHEDULE,NUMBER_OF_CRITERIA,1) ;
END;  /* SCHEDULE_ORDER   */
   FINAL_ORDER: PROCEDURE ($SCHED) ;
/*                                                                         */
/*   ORDERS $SCHEDULE BY LOWEST START AND DURATION FOR OUTPUT              */
    DECLARE  $SCHED_JOB     LOCAL ;
    DO FOR ALL SUBNODES OF $SCHED USING $SCHED_JOB  ;
      GRAFT INSERT $SCHED_JOB.START BEFORE $SCHED_JOB(FIRST) ;
      $SCHED_JOB.DUR = $SCHED_JOB.END - $SCHED_JOB.START ;
      GRAFT INSERT $SCHED_JOB.DUR BEFORE $SCHED_JOB(FIRST) ;
    END;
    CALL MULTI_CRITERIA_ORDER($SCHED, 2, -1 ) ;
    DO FOR ALL SUBNODES OF $SCHED USING $SCHED_JOB ;
      PRUNE $SCHED_JOB.DUR ;
    END;
   END;  /*  FINAL_ORDER  */
END;   /*   RESOURCE_LEVELER   */
```

2.4.34 HEURISTIC_SCHEDULING_PROCESSOR

2.4.34 HEURISTIC_SCHEDULING_PROCESSOR

2.4.34 HEURISTIC_SCHEDULING_PROCESSOR

2.4.34.1 Purpose and Scope

This processor heuristically schedules the class of projects definable by precedence networks. Activity start times are selected in an effort to heuristically minimize project duration while satisfying resource constraints. A resource leveling facility is also provided to improve a previously obtained heuristically "short" project duration. Thus, heuristic scheduling of the shortest project duration, consistent with both normal and contingency resource availabilities, is determined by the RESOURCE_ALLOCATOR. Then using this schedule as a basis, the RESOURCE_LEVELER reduces to a heuristic minimum the day-to-day variation in utilization levels of the various resources while preserving the project duration. The combination of a forward-pass resource allocation to determine the minimum feasible project duration followed by a backward-pass resource leveling to smooth out the resource loading is a technique frequently used by practical schedulers. The two-phase procedure provides an effective look-ahead capability without the need for the usual complicated logic.

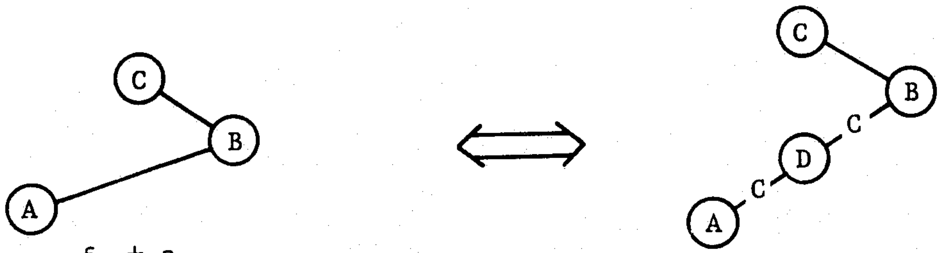
The processor addresses the broad class of high-level scheduling problems expressable as projects. By a project is meant a collection of activities each with a specified duration, set of predecessor activities, and resource requirements. A predecessor of a given activity is defined as a second activity that must precede the former by an arbitrary interval. The precedence relations of such projects are described graphically by precedence networks. Each activity can require multiple resources at utilization rates that vary with time. The resource pool structure is made versatile by providing both normal and contingency availability levels.

Unfortunately, some scheduling problems have temporal relations among their activities that cannot be described in terms of simple predecessor sets even with the addition of dummy jobs. The most prominent class of such problems are those that must have a fixed interval between two activities. To convert such a problem to the project format, the two jobs are usually lumped into a single composite job. However, the predecessors and successors of the original component jobs must then be made predecessors and successors, respectively, of the new composite job. This ploy may not yield and accurate representation of the original situation.

Most other forms of general temporal relations can be modeled in terms of precedence sets by the simple addition of "dummy" activities, which require no resources. For example, the relationship that the start of activity B follow the end of activity A by at least an interval of length "a," can be modeled by placing A in the predecessor set of a dummy activity D, that has a duration "a" and requires no resources, while placing D, in turn, in the predecessor set of A. Another simulation alternative would be to alter the definition of activity A to include the idle interval of length "a." In fact most high-level scheduling problems can be reasonably well represented by a precedence network. Reasonably accurate modeling can, however, require a great deal of ingenuity. Furthermore, questions that arise in modeling the project as a precedence network, frequently shed light on the entire scheduling problem.

Burman (Burman, 72) has suggested a sophistication of the ordinary precedence network that would permit the simple representation of all temporal relations among activities and events. Indeed, a somewhat more involved critical path algorithm can be developed to generate critical path data for his sophisticated networks. Unfortunately, however, the new networks hopelessly complicate any heuristic scheduling process. As is so often the case in problem solving, it is far easier to generalize a problem than to solve it.

Basically, what Burman has done is to identify a new type of successor--the closely-continuous successor. Such a successor must begin at the instant of completion of its predecessor. To see how this new concept facilitates the simulation of general temporal relations, consider the following examples. Consider the most difficult case of two activities whose respective start and finish are constrained to differ by a fixed time interval with the successor activity having an ordinary second predecessor as shown in Fig. 2.4.34-1.



$s_B = f_A + a$

$s_B \geq f_c$

$d_D = a$

Fig. 2.4.34-1
Sample Representation of a General Temporal Relation Using Closely-Continuous Successors

To represent this temporal relation in terms of closely-continuous successors one has only to introduce a single dummy activity D requiring no resources of duration equal to the fixed interval length "a." Activity D is then made a closely-continuous successor of activity A and B, in turn, is made a closely continuous successor of D. Activity B is made an ordinary successor of activity C. Consider next the case illustrated in Fig. 2.4.34-2, wherein one activity cannot start until a second activity has started.

$s_B \geq s_A$

Fig. 2.4.34-2
Sample Representation of a General Temporal Relation Using Closely Continuous Successors

To represent this temporal relation, one need only introduce a single dummy event E. Then activity A is made a closely-continuous successor of event E while activity B is made an ordinary successor.

Although the closely-continuous successor concept provides a generalized network presentation of all of the general temporal relations, no simple heuristic procedure can be devised to schedule such a network. Long multibranch trees of closely-continuous successors of a given activity have to be scheduled before that activity itself can be scheduled. This considerably complicates the resource allocation logic perhaps to the point of diminishing returns. Any complications in a heuristic procedure must be justified by their results. Without establishing the utility of the relatively simple resource allocator for ordinary precedence networks, it seems pointless to build a vastly more complicated allocator for generalized precedence networks. Nonetheless, in Subsection 2.4.34.7, a proof is given that any general temporal relation can be modeled using only ordinary and closely continuous successors.

This module has the capability of scheduling interfacing subnetworks. It assembles a user supplied master subnetwork and all of its interfacing subnetworks into a master network. All the activities of this master network are to be scheduled subject to common resource availability levels.

A time-progressive heuristic program is used to obtain short, but not necessarily minimal, project durations. The heuristic employs a critical-path-based priority rule tempered by a modifying heuristic using contingency resource thresholds. By utilizing late-start time as the priority value of each activity or event, a dynamic priority function is obtained that does not require updating each time a new acticity is scheduled. This results from the fact that the late-start-date of an activity is independent of the actual scheduled start dates of any of its predecessors as long as none of them are delayed beyond its late-start date. Nonetheless, the late-start date does represent a good priority rule in terms of scheduling the least flexible activities first. That unscheduled activity with the earliest late-start date, other factors being equal, is the activity most likely to lengthen project duration beyond the critical-path value. The modifying heuristic is activated whenever an activity cannot be scheduled before its late-start date. The resource that prevents the scheduling of the activity is augmented by a user-input contingency threshold from the time the activity's predecessors were all completed until the activity is successfully scheduled.

Finally, an option is provided for leveling the resource utilization profiles via a least squares heuristic after a tentative initial schedule has been obtained from the late-start-date heuristic. The leveling procedure involves sequentially considering the activities in order of latest scheduled finish. A weighted sum of squares of the resource profiles over time is then computed for each activity for each start date in its residual float. That start date in the float interval is selected that will minimize the weighted resource sum of squares. Two underlying principles motivate this heuristic procedure. First, by sequentially delaying activities considered, in order of their latest scheduled finish, the float of activities with earlier scheduled finishes can only be increased, thereby improving their subsequent scheduling flexibility. Second, the weighted sum of squares of the resource profiles over time is decreased by reducing any jump in the utilization level of any resource from one time interval to the next. In fact, the unconstrained minimum sum of the squares is achieved when all the resource profiles are such that the utilization levels of any given resource in each time period is a constant.

2.4.34.2 Modules Called

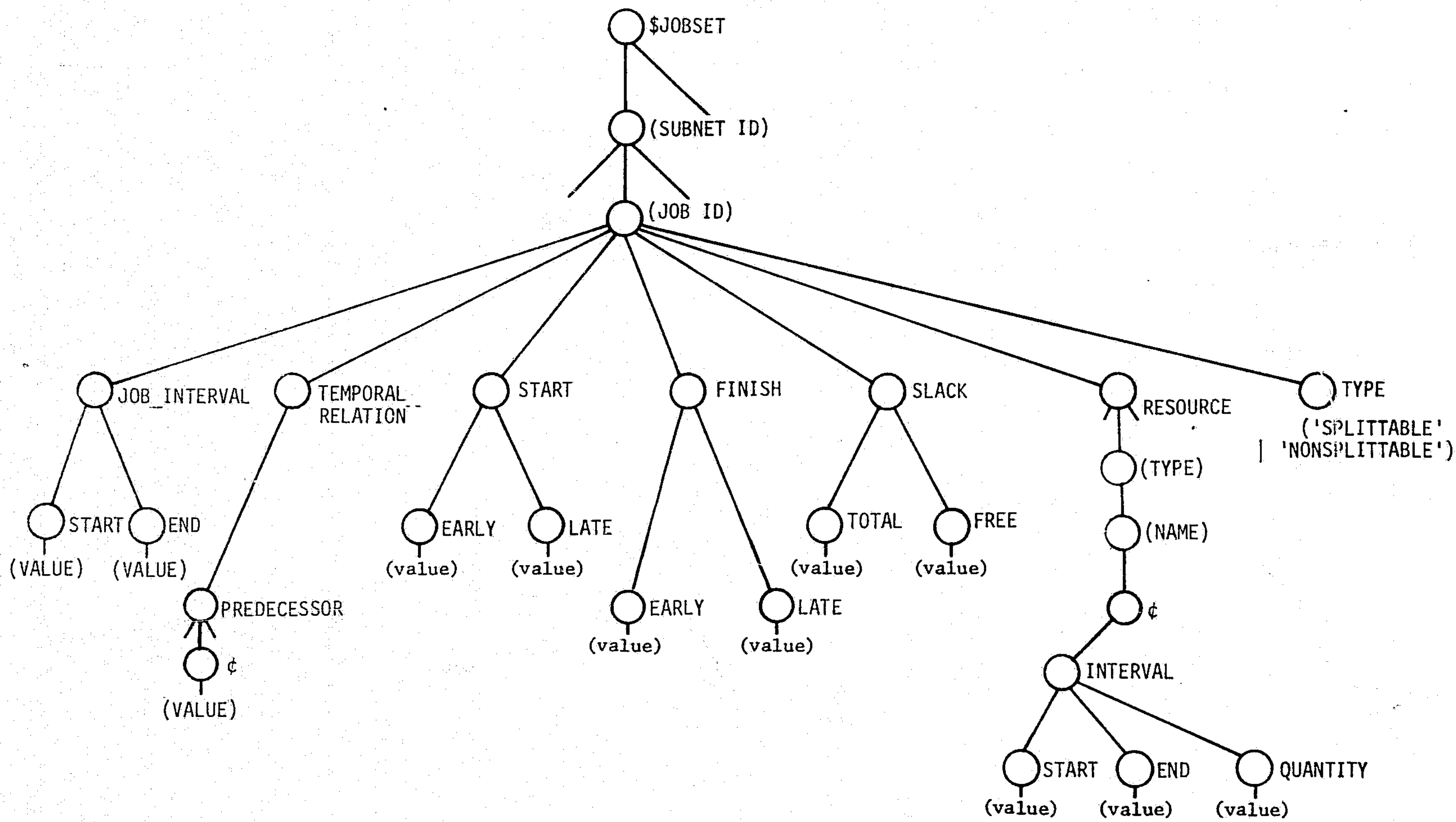
NETWORK_ASSEMBLER

RESOURCE_ALLOCATOR

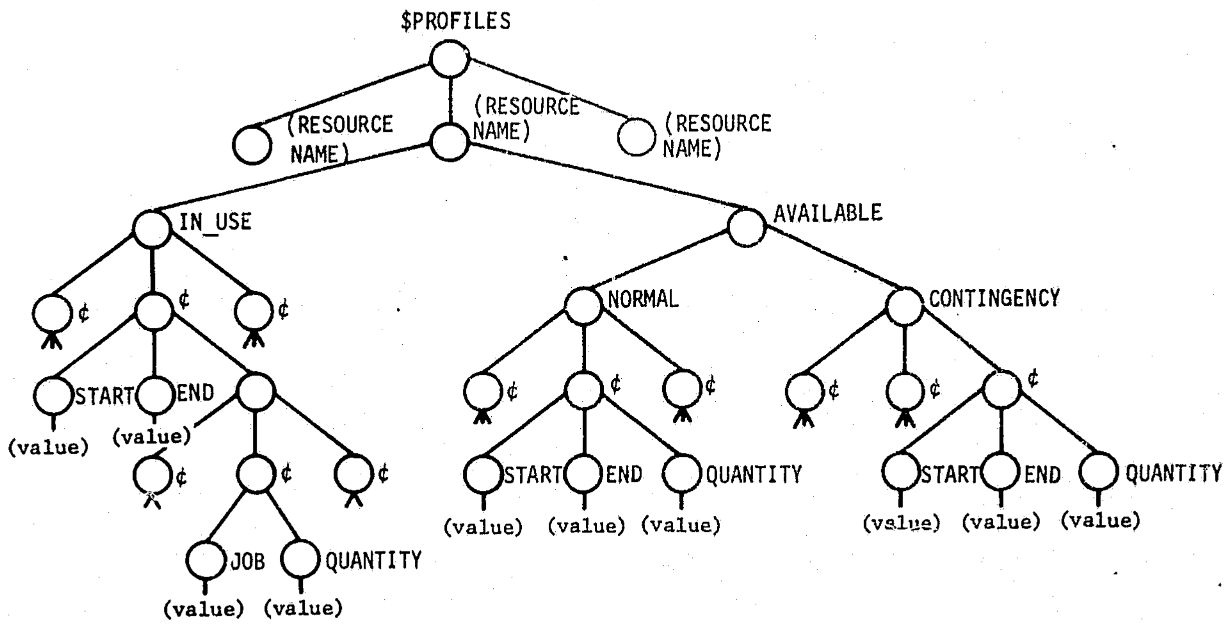
RESOURCE_LEVELER

2.4.34.3 Module Input

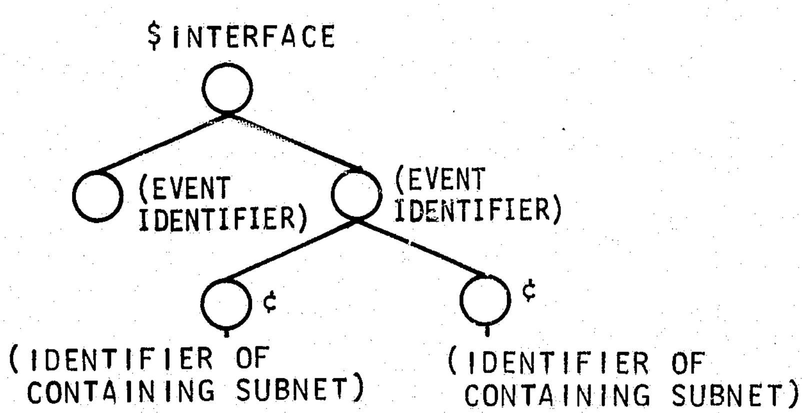
1) Network, Critical Path Data and Activity or Event Definitions ($JOBSET)


$JOBSET
(SUBNET ID)
(JOB ID)
JOB_INTERVAL
TEMPORAL_
RELATION
START
FINISH
SLACK
RESOURCE
TYPE
('SPLITTABLE'
| 'NONSPLITTABLE')
START
END
(VALUE)
(VALUE)
PREDECESSOR
¢
(VALUE)
EARLY
LATE
(value)
(value)
EARLY
LATE
(value)
(value)
TOTAL
FREE
(value)
(value)
(TYPE)
(NAME)
¢
INTERVAL
START
END
QUANTITY
(value)
(value)
(value)

2) Resource Definitions ($PROFILES)



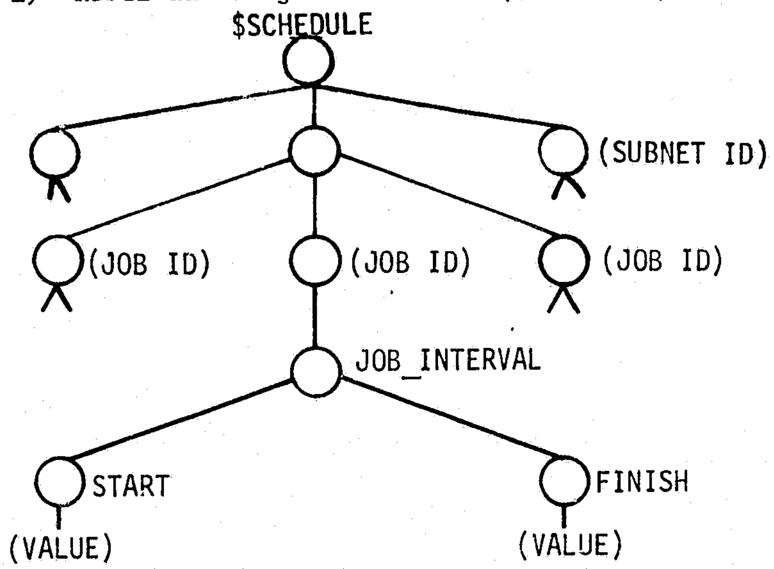
3) Interfacing Event Definitions ($INTERFACE)



4) Resource Leveling Option Indicator (LEVEL)

2.4.34.4 Module Output

1) Resultant Project Schedule ($SCHEDULE)



2) Revised Resource Profiles ($PROFILES)

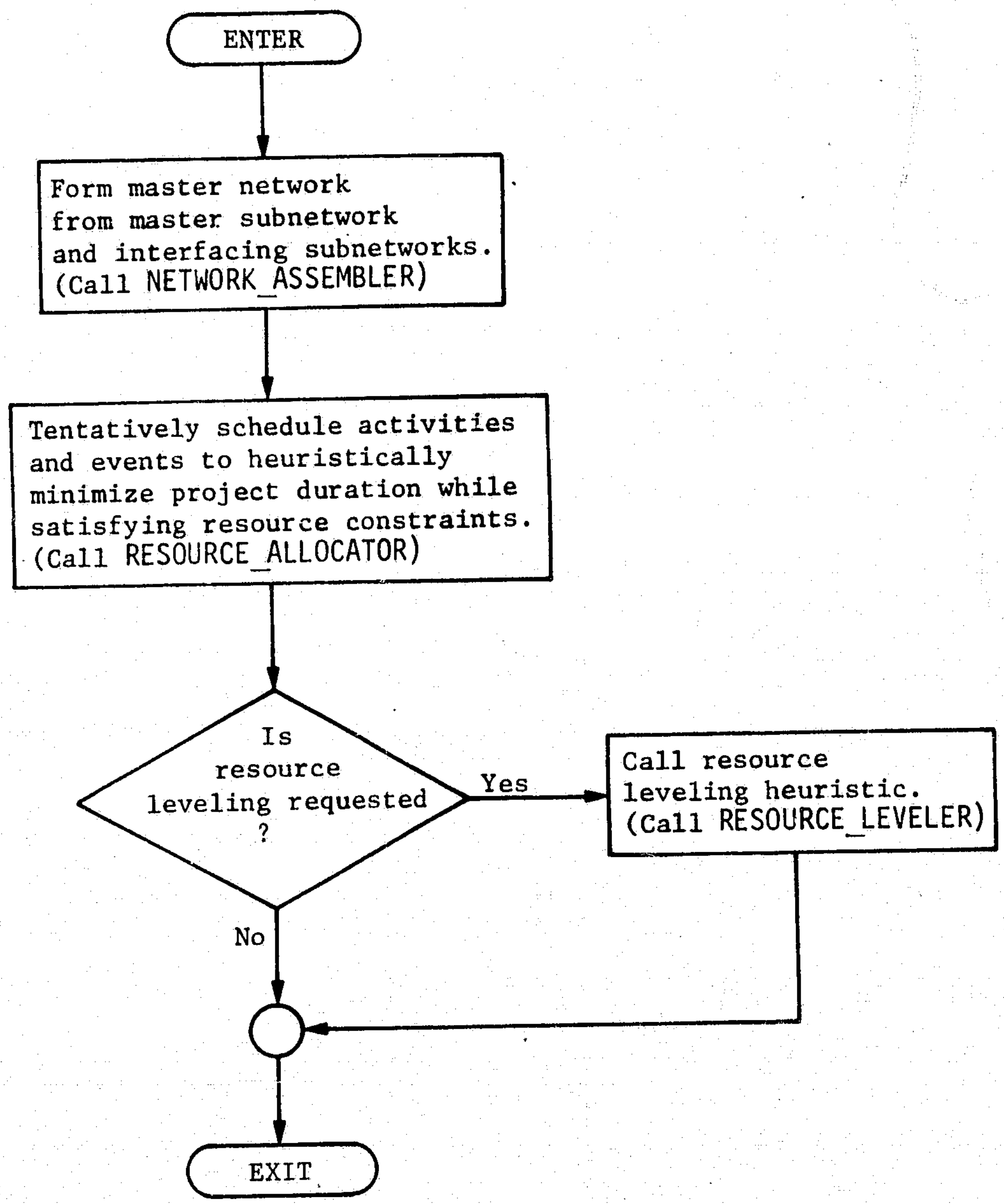
Same as for Module Input.

2.4.34.5 Functional Description

The HEURISTIC_SCHEDULING_PROCESSOR serves as an executive procedure for controlling and coordinating the entire heuristic scheduling process. First the network must be built whose activities are to be scheduled sharing the same common resources. By means of a call to the module NETWORK_ASSEMBLER, the user-specified master subnetwork and all of its interfacing subnetworks, as detailed in the interfacing event definitions, are assembled into the desired network. Next, the RESOURCE_ALLOCATOR is called to schedule the activities of the network according to the minimum project duration heuristic procedure described above. Earliest late-start is used as the priority function for each activity.

If an activity is delayed beyond its late-start date because of a resource shortage, a modifying heuristic is invoked to increase the availability of the deficient resource by a user input contingency threshold. If the user does not request any resource leveling effort by leaving the leveling option indicator, LEVEL, unset, the heuristic scheduling process ends here. Otherwise the module RESOURCE_LEVELER is called to heuristically reduce to a minimum the jumps in the resource utilization rate. The heuristic operates by considering the activities in order of latest scheduled finish. The weighted sum of the resource profiles squares over time is then computed for each possible start time of the activity under consideration within its remaining total float. That start time is selected that minimizes the sum. When all the activities have been considered for delay, the leveling effort is complete and the heuristic scheduling terminates. The simple macrologic for the processor is illustrated in the functional block diagram. M re detailed information on the resource allocation and leveling heuristics can be found in the respective specifications for the modules, RESOURCE_ALLOCATOR and RESOURCE_LEVELER.

2.4.34.6 Functional Block Diagram



2.4.34.7 Implementation Consideration

The salient feature of precedence network methods development has been the appearance of a tremendous number of elaborate computerized heuristic routines for constrained-resource scheduling. Most of these computer codes have been developed by organizations for internal and external use on a proprietary basis. Hence, their operating details are not available in the open literature. However, some are available which disclose their operating principles. Table 2.4.34-1 presents a sampling of the more prominent programs known to be available in the USA and United Kingdom. Each program produces a wide variety of resource- and activity-oriented reports in both tabular and graphic form.

For many applications, the complete capability of any of the commercial systems is not required. A smaller more specialized system could be built around the basic modules outlined above. Such a flexible system could then evolve to meet the user's ever-changing needs.

In closing this section, the claim that any generalized temporal relation can be expressed in terms of ordinary and closely-continuous successors with only the addition of dummy activities will be verified. Recall the generic form of a generalized temporal relation

[1] $$\begin{Bmatrix} s_i \\ f_i \end{Bmatrix} \begin{Bmatrix} \leq \\ \geq \\ = \end{Bmatrix} \begin{Bmatrix} s_j \\ f_j \end{Bmatrix} \begin{Bmatrix} + \\ - \end{Bmatrix} k$$

Table 2.4.34-1
Sample Characteristics of Some Commercially-Available Computer Programs with Contrained-Resource Network Scheduling Capabilities

Program Name and Company Responsible	Features
CPM-RPSM (Resource Planning and Scheduling Method) CEIR, Inc	2000 to 8000 jobs per project, 4 resource types per project, 26 total variables or constraint-resource limits, job splitting allowed, job start or finish constraints allowed. Uses fixed scheduling heuristic.
ICT 1900 Series PEWTER (PERT without Tears) International Computers, Ltd	Multiproject capability, 60,000 activities, 60 resources per activity, 125 resources per project, flexible updating and reporting options. Uses advanced resource allocation heuristic, enabling user to invoke input resource contingency thresholds. Levels resources with a separate heuristic. Employs a network condensation capability to accurately process large networks in pieces.
MSCS ("Management Scheduling and Control System") McDonnell Automation Co.	Multiproject capability (25 projects), 18,000 activities, 12 resource types per activity. Many flexible assumptions of job conditions, easy updating. Allows project costing and includes report generation. Scheduling heuristics are based on complex priority function approach controllable by user.
PMS/360 (Project Management System) IBM Corporation	A large complex management information system consisting of four main modules (of which resource allocation is one). Handles activity-on-arrow or precedence diagrams; up to 225 multiple projects allowed with 32,000 activities and 250 resource types. Numerous costing, updating, and report options. A choice of sequencing heuristics is provided.
PPS IV (Project Planning System) Control Data Corporation	2000 jobs per project, 20 resource types per job and project, multiple or single projects, allows overlapping jobs, resource costing, and progress reporting. Will also do resource leveling with fixed duration. Resource priorities may be specified, and multishift work is allowed. Uses one fixed heuristic procedure.
PROJECT/2 Project Software, Inc	Allows 50 multiple networks, 32,000 jobs, several hundred resource types. Includes automatic network generation for repetitive sequences, easy updating, and many cost analysis features. Choice of sequencing heuristics specified by user. Handles activity-on-arrow or activity-on-node input.

where i and j are any activities or events in the project and "s" denotes a start time while "f" signifies a finish time. But, because $s_i = f_i - d_i$ and $f_j = s_j + d_j$, then relation [1] can be simplified to

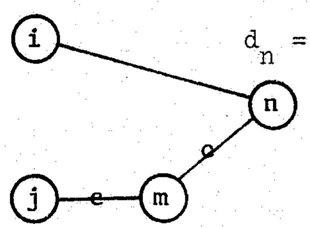
[2] $$f_i \begin{Bmatrix} \leqslant \\ \geqslant \\ = \end{Bmatrix} s_j + k$$

where k is a constant of arbitrary polarity. From the three relational operators three basic cases can be identified. These will be considered in turn.

$$f_i \leqslant s_j + k \text{ and } k \geqslant d_j$$

[3] Case IA:

$d_n = 0$



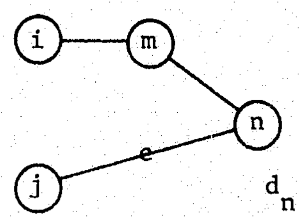
$d_m = k - d_j$

$$f_i \leqslant s_n = s_j + d_j + d_m$$
$$f_i \leqslant s_j + d_j + (k-d_j)$$
$$f_i \leqslant s_j$$

$$f_i \leqslant s_j + k \text{ and } k < d_j$$

[4] Case IB:

$d_m = d_j - k$



$d_n = 0$

$$f_i + d_m \leqslant s_n = f_j$$
$$f_i + d_j - k \leqslant s_j + d_j$$
$$f_i \leqslant s_j + k$$

[5] Case II $f_i \geqslant s_j + k$

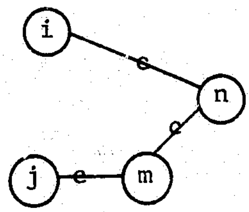
[6] $s_i + d_i \geqslant f_j - d_j + k$

[7] $f_j \leqslant s_i + (d_i + d_j - k)$

Relation [7], however, has already been modeled as a general precedence relation in Case I if the labels i and j are interchanged.

[8] Case IIIA $f_i = s_j + k$ and $k \geqslant d_j$

$$f_i = s_n = s_j + d_j + d_m$$
$$f_i = s_j + d_j + k - d_j$$
$$f_i = s_j + k$$

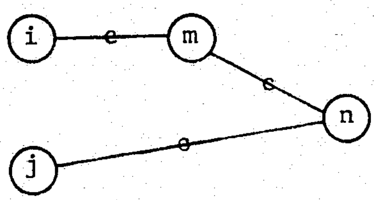


$$d_m = k - d_j$$

[9] Case IIIB $f_i = s_j + k$ and $k \quad d_j$

$$d_m = d_j - k$$

$$f_i + d_m = s_n = s_j + d_j$$
$$f_i + d_j - k = s_j + d_j$$
$$f_i = s_j + k$$



Thus, any general temporal relation between two activities, i and j, can be represented in terms of closely continuous and ordinary successors without redefining or combining any activities. Only new dummy activities need be introduced. Hence, the identity of all the original activities is maintained so that an ordinary predecessor or successor relation can be represented as usual.

2.4.34.8 References

IBM, *Project Management System IV Network Processor Program Description and Operations Manual*, Publication SH20-0899-1, 1972.

ICT 1900 Series PEWTER (PERT without Tears). ICT Technical Publications Group, London 1967.

Burman, P. J.: *Precedence Networks for Project Planning and Control.* McGraw Hill, London, 1972.

2.4.34.9 DETAILED DESIGN

This module heuristically schedules the broad class of scheduling problems definable by precedence networks. It serves primarily as an executive routine calling NETWORK_ASSEMBLER, PREDECESSOR_SET_INVERTER, CRITICAL_PATH_CALCULATOR and then RESOURCE_ALLOCATOR. The user has the option to call RESOURCE_LEVELER as well.

2.4.34.10 INTERNAL VARIABLE AND TREE NAME DEFINITIONS

$INTERFACE	-	Contains the network interfacing event definitions
$JOBSET	-	Contains the master subnetwork which contains the set of jobs that are to be scheduled
LEVEL_RESOURCE_FLAG	-	A flag used to indicate whether or not RESOURCE_LEVELER is to be called
$PROFILES	-	Contains all resource pool profile data
$SCHEDULE	-	The final schedule output by the module containing absolute time and resource assignments
$SUBNET_SET	-	Contains the names of the subnetworks eliminated by NETWORK_ASSEMBLER, when it built the master subnetwork

2.4.34.11 MODIFICATIONS TO FUNCTIONAL SPECS AND/OR STANDARD DATA STRUCTURES

It was decided that $SUBNET_SET should also be an output of this module. Since NETWORK_ASSEMBLER builds it anyway, it takes no extra effort to return this information to the calling program. For better readability, the name of the resource leveling option indicator was changed from LEVEL TO LEVEL_RESOURCE_FLAG.

2.4.34.12 COMMENTED CODE

```
HEURISTIC_SCHEDULING_PROCESSOR: PROCEDURE
/*************************************************************************/
/*                                                                       */
/*  THIS MODULE HEURISTICALLY SCHEDULES THE BROAD CLASS OF SCHEDUL-      */
/*  ING PROGLEMS DEFINABLE BY PRECEDENCE NETWORKS.  ITS INPUT AND        */
/*  OUTPUT PARAMETERS ARE DESCRIBED BELOW:                               */
/*       $JOBSET ------ THE SET OF JOBS THAT ARE TO BE SCHEDULED         */
/*       $INTERFACE --- THE NETWORK INTERFACING EVENT DEFINITIONS        */
/*       LEVEL_RESOURCE_FLAG --- A FLAG INDICATING WHETHER OR NOT        */
/*                               RESOURCE LEVELING IS TO BE DONE         */
/*       $PROFILES ---- CONTAINS ALL RESOURCE POOL PROFILE DATA          */
/*       $SUBNET_SET -- CONTAINS THE NAMES OF THE SUBNETWORKS USED       */
/*                      TO BUILD THE MASTER SUBNETWORK                   */
/*       $SCHEDULE ---- A HEURISTICALLY MINIMUM DURATION SCHEDULE        */
/*                      WITH ABSOLUTE TIME AND RESOURCE ASSIGNMENTS      */
/*                                                                       */
/*************************************************************************/
    ($JOBSET, $INTERFACE, $MASTER_SUBNET_ID,
              LEVEL_RESOURCE_FLAG, $PROFILES, $SUBNET_SET, $SCHEDULE)
               OPTIONS(EXTERNAL);
DECLARE $TEMP  LOCAL;
   CALL NETWORK_ASSEMBLER($JOBSET,$INTERFACE,$MASTER_SUBNET_ID,
                                  $SUBNET_SET) ;
 CALL PREDECESSOR_SET_INVERTER($JOBSET);
  CALL CRITICAL_PATH_CALCULATOR($JOBSET.
              #($MASTER_SUBNET_ID),$TEMP);
  GRAFT $TEMP AT $JOBSET.#($MASTER_SUBNET_ID);
WRITE 'OUTPUT $SUBNET_SET AND $JOBSET AFTER ASSEMBLING THE NETWORK ' ;
WRITE $SUBNET_SET ;
 WRITE $JOBSET ;
   CALL RESOURCE_ALLOCATOR
                   ($JOBSET.#($MASTER_SUBNET_ID),$PROFILES,$SCHEDULE) ;
WRITE 'OUTPUT $SCHEDULE AND $PROFILES AFTER ALLOCATION ' ;
WRITE $SCHEDULE, $PROFILES;
   IF LEVEL_RESOURCE_FLAG ¬= 0
      THEN CALL RESOURCE_LEVELER($SCHEDULE,$PROFILES) ;
END HEURISTIC_SCHEDULING_PROCESSOR ;
```

2.4.35 GUB_LP

2.4.35 GUB_LP

2.4.35 GUB_LP

2.4.35.1 Purpose and Scope

Generalized upper bounding (GUB) is a simplex type algorithm designed specifically for linear programs that contain large numbers of convexity constraints. The principal advantage of GUB for this class of problems is that it requires a significantly smaller working basis. Use of this smaller working basis reduces fast core memory requirements and increases the computational speed by nearly an order of magnitude for GUB problems.

GUB applies to LPs of the form: minimize u^o, subject to:

[1] m rows $$u^i + \sum_{j=1}^{N_o} a_j^i x^j + \sum_{p=1}^{P} \sum_{k=1}^{N_p} a_{kp}^i x_p^k = b^i, \quad i = 1,2,\ldots,m$$

[2] P rows $$\sum_{k=1}^{N_p} x_p^k = 1, \quad p = 1,2,\ldots,P$$

$$u^1, u^2, \ldots, x_p^{N_p} \geq 0.$$

In the above formulation, u^i denotes the logical variables (slack, surplus, and/or artifical) augmented to transform the i-th constraint into an equation, x^i for j=1,...N_o denotes the structural variables that are not contained in any convexity constraints, x_p^k for k=1,...,N_p denotes the structural variables constituting the p-th GUB set, a_j^i denotes the constraint coefficients for the i-th equation, and b^i is the right hand side (RHS) for constraint i. Equation [1] represents the interconnecting constraints and Equation [2] represents the GUB convexity constraints.

This special structure arises naturally in many problems, for example, transportation, distribution, and multi-item scheduling. GUB structure also results when the Dantzig-Wolf decmoposition principle is used to solve linear programs whose constraint matrices have block angular structures.

GUB can also be used to obtain approximate solutions to binary multiple choice programs. In these situations, the convexity constraints represent the multiple choice restrictions for the binary decision variables. This means that the convexity constraint, $\Sigma x^j = 1$, combined with the binary restriction, $x^j = 0$ or 1, ensures that one and only one x^j will be nonzero. The resulting binary problem is then solved as a continuous LP using GUB. Clearly, not all of the resulting optional decision variables will be binary. Fortunately, if the number of GUB rows (P) is much larger than the number of interconnection constraints (m), then most of the variable (P-m) will be binary in the optimal solution. This important result is often used to yield approximate solutions to large multiple-choice decision problems.

2.4.35.2 Modules Called

None

2.4.35.3 Module Input

1) The total number of structural variables - $N = \sum_{i=1}^{P} n_i$

2) The number of non-GUB constraints - m

3) The number of GUB constraints - P

4) The number of elements in each GUB set - n_p

5) The RHS vector - b^i (including GUB rows)

6) The type of each constraint (equality, inequality, etc)

7) The constraint matrix for nonGUB rows - $\left(a_j^i\right)_{mxn}$

2.4.35.4 Module Output

1) Output option indicator

2) Iteration summary

   a) Key columns

   b) Indices of current basis elements

   c) Values of variables in current basis

   d) Entering column

   e) Simplex multipliers

   f) Current cost function value

3) Solution summary

   a) Indices of variables in optimal basis

   b) Optimal values for the structural variables

   c) Value of each run GUB row in constraint matrix

   d) Cost function

   e) Simplex multipliers

2.4.35.5 Functional Description

The generalized upper bounding procedure, developed by Dantzig and Van Slyke(1), is a specialization of the simplex method. The key feature of GUB is that it solves the LP defined in Equations [1] and [2], while maintaining a "working" basis of dimension mxm. Thus, all quantities required to make a simplex like iteration are computed in terms of this reduced "working" basis.

GUB can be motivated by considering the following facts:

1) Any feasible bases for Equation [1] - [2] must contain at least one element from each GUB set.

2) An elementary matrix transformation that transforms the basis into upper block triangular form can be defined; hence, enabling a basis-feasible solution to be computed in terms of a mxm submatrix.

Clearly, in order to satisfy each GUB convexity constraint, $\Sigma\ x^j = 1$, at least one of the variables must take on a nonzero value. This implies that any feasible basis must contain at least one column from each GUB set. Suppose we select one such column from each GUB set and enter these columns as the first p columns in the basis. The remaining m columns are then selected from the nonkey columns. The basis can then be partitioned as follows:

[3] $$B = \left[\begin{array}{c|c} A_{mxp} & B_{mxm} \\ \hline I_{pxp} & C_{pxm} \end{array}\right]_{m+p\ x\ m+p}$$

key columns   nonkey columns

The important property of B is that the submatrix C_{pxm} is composed binary elements, which enables on elementary matrix to be constructed as follows:

[4] $$BE = \left[\begin{array}{c|c} A_{mxp} & B_{mxm} \\ \hline I_{pxp} & 0_{pxm} \end{array}\right]$$

is upper block triangular. The elementary transformation E is easily constructed by subtracting the appropriate columns of $\left[\frac{A}{I}\right]$ from $\left[\frac{B}{C}\right]$.

The elementary matrix that performs these simple column operations is given by

[5] $$E = \left[\begin{array}{c|c} I_{pxp} & -C_{pxm} \\ \hline 0_{mxp} & I_{mxm} \end{array}\right]$$

Now suppose that we have a basic feasible solution

[6] $$x_B = B^{-1}b,$$

and define

[7] $$y_B = E^{-1}x_B = E^{-1}\ B^{-1}\ b = (BE)^{-1}b.$$

Clearly, y_B is then a basic feasible solution of the transformed system

[8] $$(BE)\ y_B = b.$$

The solution of the transformed system

[9] $$\left[\begin{array}{c|c} A_{mxp} & B_{mxm} \\ \hline I_{pxp} & 0_{pxm} \end{array}\right] \begin{bmatrix} y_p \\ y_m \end{bmatrix} = \begin{bmatrix} b_m \\ 1_p \end{bmatrix}$$

is easily calculated from the reduced problem

[10] $$B_{mxm}y_m = b_m - A_{mxp}y_p$$

$$y_p = 1_p.$$

It is evident from Equation [10] that the determination of a basic feasible solution is, in essence, equivalent to solving for y_m. The calculation of y_m requires only the inverse of the "working" mxm basis, B^{-1}. This fact motivates the principal advantage of the GUB algorithm, and enables each operation required in applying the simplex method to Equation [1] - [2] to be performed in terms of the quantities associated with the reduced basis, B_{mxm}.

The outline of key modifications to the simplex operations required by GUB follow. These equations are derived in Reference 1, and are based on the working basis inverse.

Calculating the Simplex Multipliers - Let (π, μ) be the simplex multipliers for the basis B, where π is a m-component row vector associated with the first m equations and μ is a p-component row vector associated with the last p equations.

These multipliers are calculated as

[11] $$(\pi_1, \pi_2, \ldots, \pi_m) = (0,0,\ldots, 0,1)B^{-1} = ((B^{-1})_1^m, \ldots, (B^{-1})_m^m)$$

[12] $$\mu_i = -\pi' A_{k_i}, \; i = 1,2,\ldots,P,$$

where k_i denotes the index of the i-th key column.

Calculating the Relative Cost Coefficients - This is done in the usual way

[13] $$\bar{c}_j = -\Sigma \, \pi_i A_j^i - \mu_k, \text{ if } A_j \in \mathcal{S}_k$$

where $\mathcal{S}_k$ denotes the k-th GUB set.

Representing the Entering Column in Terms of the Current Basis - The column that enters the basis is selected in the standard manner by computing $c_s = \min \bar{c}_j$ over all nonbasic columns. If $c_s \geq 0$, the current solution is optimal. If $c_s < 0$, then A_s enters the basis. For the purpose of discussion, let $A_s \in \mathcal{S}_\sigma$. Now to see which column leaves the basis, the transformed column

[14] $\bar{A}_s = B^{-1} A_s$

must be computed. GUB structure enables the transformed columns to be calculated from the reduced "working" basis. The equations that perform this transformation are:

$$[15] \quad \bar{A}_s^{i} = \begin{cases} - \sum_{\kappa \varepsilon \Gamma_i} \bar{D}_s^{k}, & \text{for } 1 \leq i \leq p,\ i \neq \sigma \\ 1 - \sum_{\kappa \varepsilon \Gamma_i} \bar{D}_s^{k}, & \text{for } i = \sigma \end{cases}$$

$$[16] \quad \bar{A}_s^{p+k} = \bar{D}_s^{k}, \text{ for } k = 1,2,\ldots,m,$$

where

$$[17] \quad \bar{D}_s = B^{-1}\left(A_s - A_{k_\sigma}\right),$$

$$[18] \quad \Gamma_i = \left\{k : k \ \{1,2,\ldots,m\} \text{ and } r_k = i\right\}$$

and

$$[19] \quad r_k = i, \text{ for } k = 1,2,\ldots,m,$$

if the p+k-th column of B is an element of the i-th GUB set, that is,

$$[20] \quad B_{p+k} \ \varepsilon \left\{a_{1i}, a_{2i}, \ldots, a_{n_i i}\right\}.$$

Choosing the Column to Leave the Basis - This is done in the usual way, by computing

$$[21] \quad \theta = x^{B_r} / \bar{A}_s^{r} = \min_i \left\{ x^{B_i} / \bar{A}_s^{i} : \bar{A}_s^{i} \geq 0 \right\}$$

if all $\bar{A}_s^{i} \geq 0$, the solution is unbounded.

Updating the Basic Variables - The basic variables are updated according to the formula

[22] $x^{B_i} \leftarrow x^{B_i} - \theta A_s^{-i}$, for $i = 1,2,\ldots,m+p$, $i \neq r$

$x^{B_i} \leftarrow \theta$, for $i = r$

Updating the Inverse of the Working Basis - Let $A_s \in \mathcal{J}_\sigma$ be the column entering and $A_{j_r} \in \mathcal{J}_\rho$ be the column leaving. There are two cases to be considered.

Case 1. A_{j_r} is not a key column. In this case, A_{j_r} is replaced in B by A_s in one of the last m columns. The only resulting change in the working basis is that the column of B corresponding to the leaving column is replaced by $\left\{A_s - A_{k_\sigma}\right\}$. B^{-1} is updated by adjoining the column

[23] $\bar{D}_s = B^{-1}\left(A_s - A_{k_\sigma}\right)$

and performing a pivot.

Case 2. A_{j_r} is a key column.

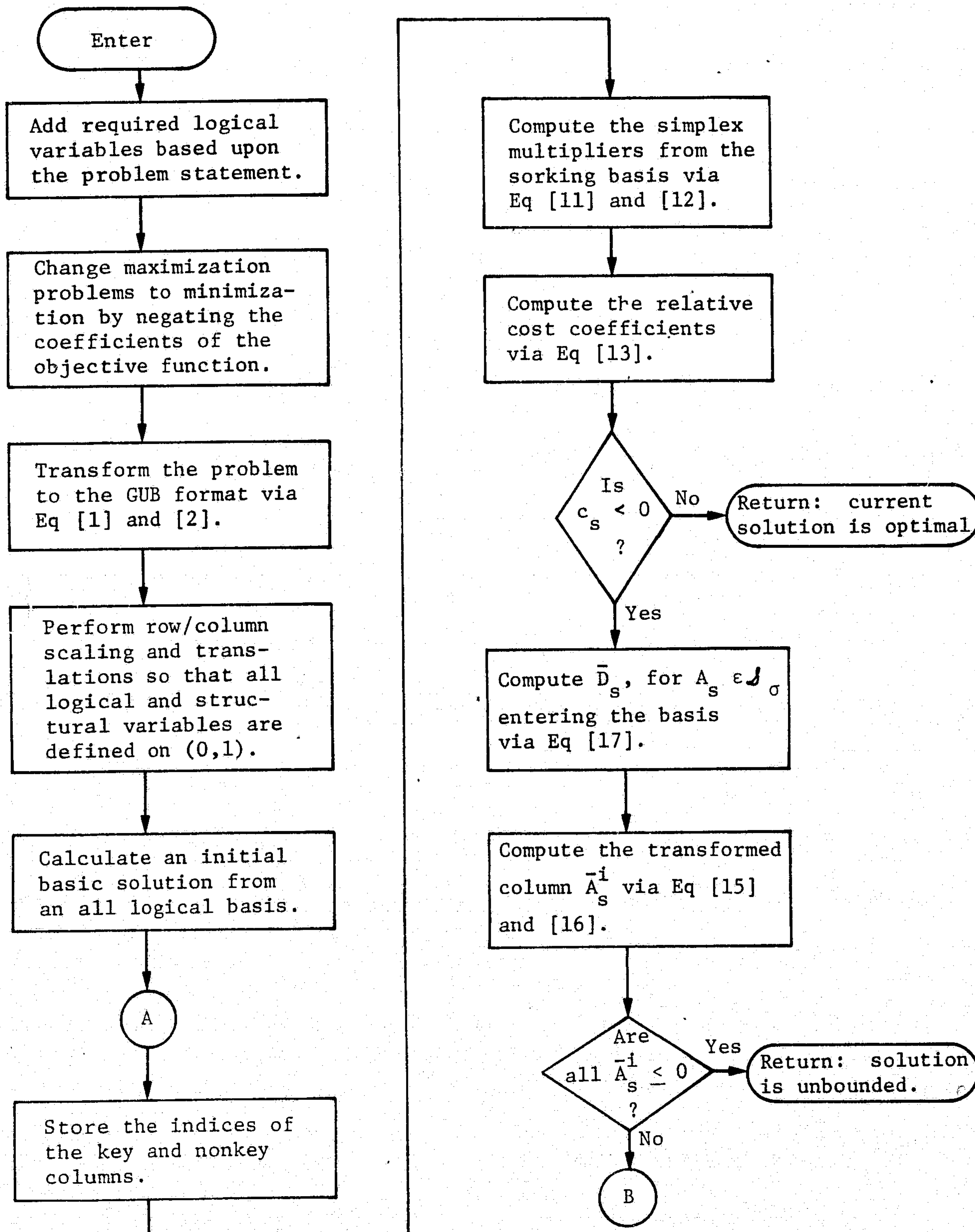
1) A_{j_r} and A_s are from different GUB sets (i.e., $\sigma \neq \rho$). Select one nonkey column in $\mathcal{J}_\rho$ and interchange it with A_{j_r}. The situation then becomes that of Case 1. Clearly, such a nonkey column must exist in order for $A_{j_r} \in \mathcal{J}_\rho$ to be selected for removal.

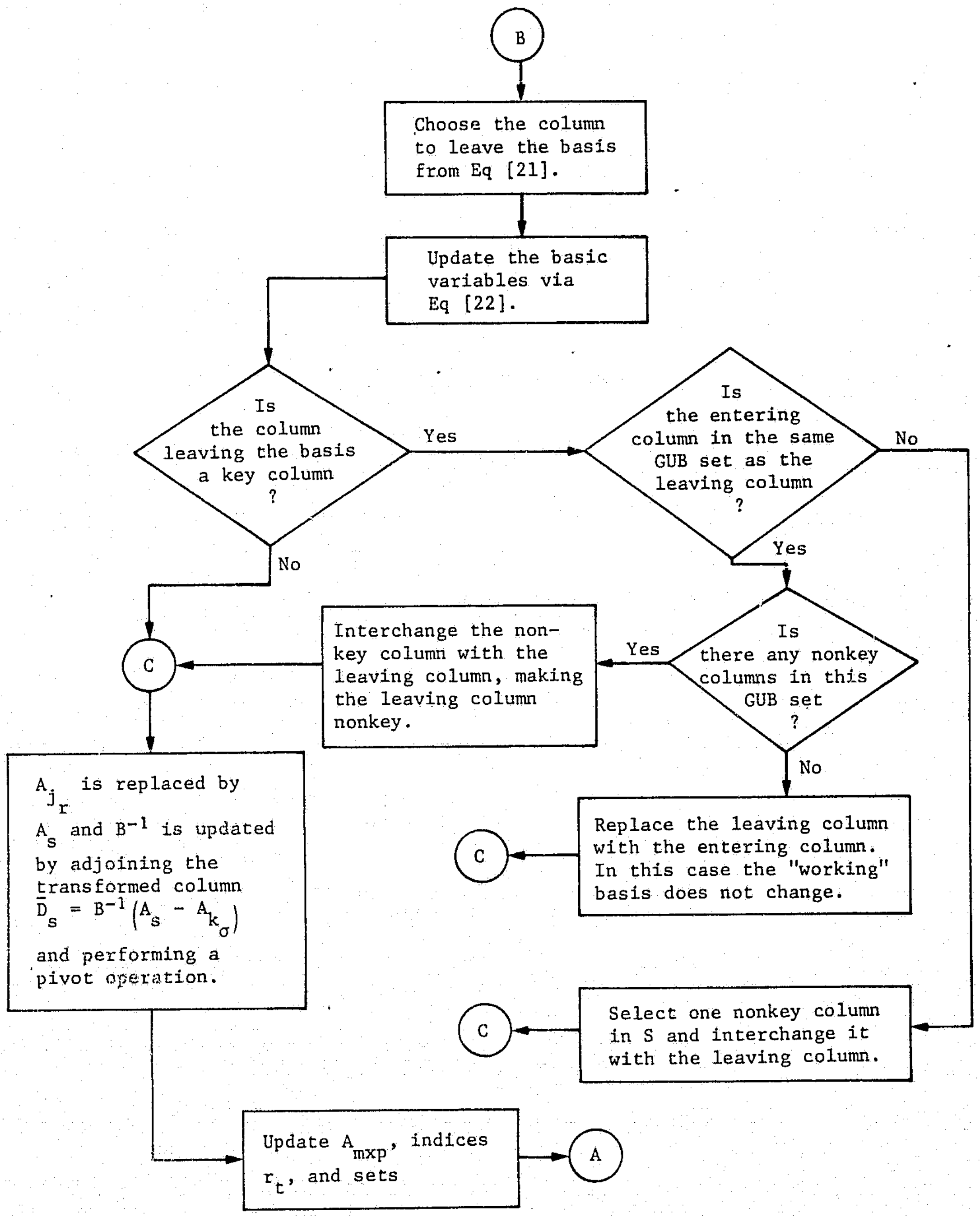
2) A_{j_r} and A_s are from the same GUB set (i.e., $\sigma = \rho$)

   a) if there exists at least one nonkey column that is a member of $\mathcal{J}_\rho = \mathcal{J}_\sigma$ select one and interchange it with A_{j_r}, and apply the operations described in Case 1.

b) If no nonkey column is from $\mathcal{J}_\rho$ then A_s replaces A_{j_r}. However, since the last m columns of B do not contain elements of $\mathcal{J}_\rho$, the working basis does not change. The upper left partition of B does, however, change when A_s replaces A_{j_r}.

2.4.35.6 Functional Block Diagram





ORIGINAL PAGE IS
OF POOR QUALITY

2.4.35.7 Typical Applications

As mentioned earlier, GUB structure arises naturally in many scheduling problem formulations. For example, the PWW project scheduling formulation, the simplified activity scheduling formulation, multi-item scheduling, and resource allocation problems all contain GUB structure. This does not mean, however, that GUB is the best algorithm for all of these formulations. Other fundamental considerations (such as, integrality) and numerous problem-dependent factors (such as, the ratio of P to m) determine which algorithm should be used. However, independent of any of these other considerations, a GUB structured LP can be very efficiently solved with this type module.

2.4.35.8 Implementation Considerations

The advantage of GUB is that it efficiently solves a special class of LP problems. However, this tool must be used on relatively large problems ($m \geq 100$, $p \geq 100$, $n \geq 1000$) before the computational savings are of any real consequence. On small problems, the difference between GUB and revised simplex is measured in fractions of seconds and would, in our opinion, not justify the GUB development costs.

For the solution of extremely larger GUB problems ($m \geq 1000$, $p \geq 1000$, $n \geq 10,000$), a production code similar to that contained in MPSX is recommended. For most problems a less sophisticated GUB algorithm, like the one specified here, would be adequate. The effort required to develop the level of a GUB algorithm would be slightly higher than that of a versatile primal simplex code.

This means that it would take basically two to three man-months of development to completely formulate, code, checkout, and document this module.

2.4.35.9 References

Dantzig, G. B., and Wolfe, P.: "The Decomposition Algorithm for Linear Programming." *Econometrica*, 9, No. 4, 1961. *Operations Research*, 8, January - February, 1960.

Lasdon: "Optimization Theory for Large Systems." *MacMillan Series in Operations Research*. 1970.

Orchard-Hays, W.: *Advanced Linear-Programming Computing Techniques*. McGraw-Hill Book Company, New York, New York, 1968.

2.4.36 MIXED_INTEGER_PROGRAM

2.4.36 MIXED_INTEGER_PROGRAM

2.4.36.1 Purpose and Scope

This module can be used to optimize any system whose performance can be mathematically modeled as a linear function in a set of decision variables that are subject to both linear algebraic and integrality constraints. Symbolically, the problem of determining the optimal system configuration must reduce to a mathematical problem of the form

Minimize: $z = C_2X \div C_1Y$

Subject to: $A_2X + A_1Y \leq b$

X integer

where:

X is an nxl integer column vector of unknowns,

Y is a pxl continuous column vector of unknowns,

C is a lxn continuous row vector of cost coefficients for the integer variables,

e is a lxp continuous row vector of cost coefficients for the continuous variables,

A is an mxn continuous matrix of constraint coefficients for the binary variables,

D is an mxp continuous matrix of constraint coefficients for the continuous variables,

b is an mxl continuous column vector of constraint limits.

2.4.36.2 Modules Called

This module requires a special purpose Geoffrion code that solves a mixed integer program containing a single continuous variable.

2.4.36.3 <u>Module Input</u>

1) The objective function coefficients c and e.

2) The constraint matrices A and D

e) The RHS vector b.

2.4.36.4 <u>Module Output</u>

1) Value of the decision variables

2) Final objective function value

3) Iteration Summary

2.4.36.5 <u>Functional Description</u>

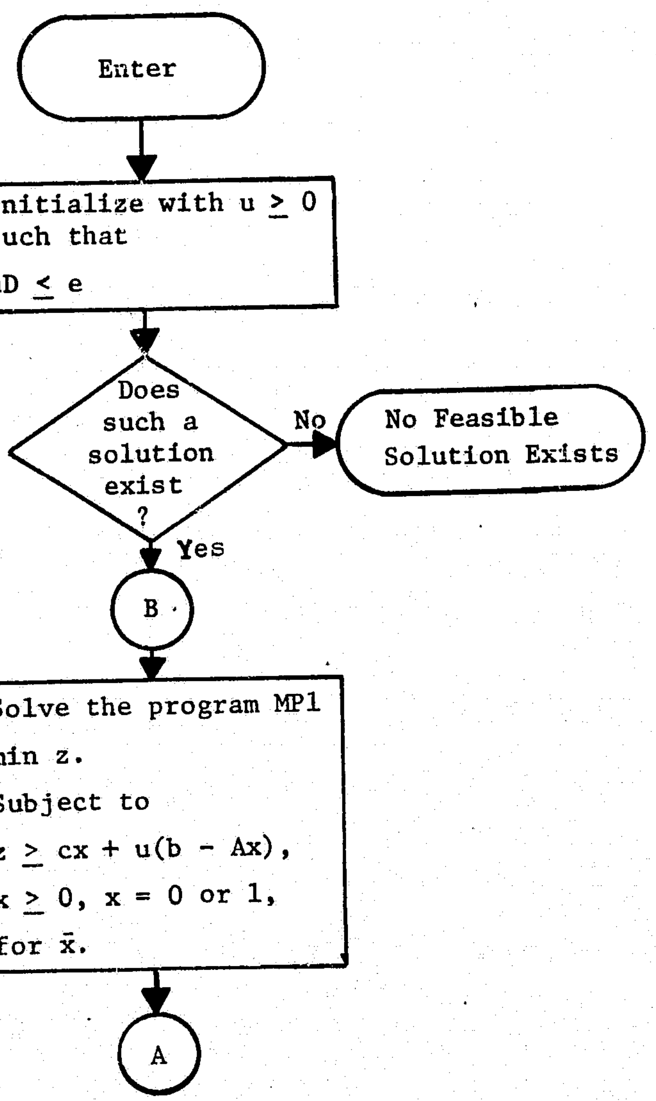
Branch and bound type algorithms have classically been applied to mixed-integer programs. Unfortunately, B&B methods can be inefficient if the number of integer variables is large. This is because of the branching rule that merely dichotomizes the continuous solution for each integer variable.

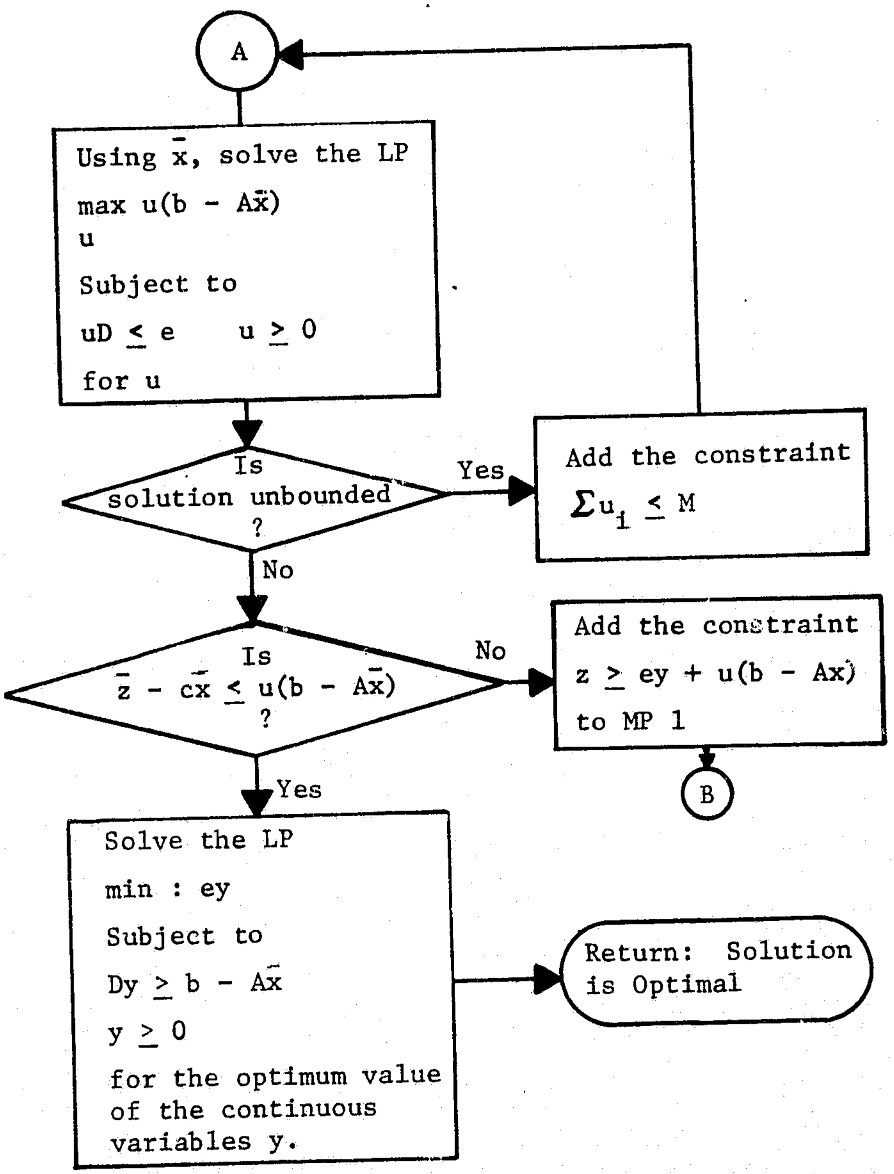
For problems of practical size, this procedure produces so many LP subproblems that even very fast simplex codes do not make this approach feasible. For problems with only a few integer variables, B&B methods are adequate and are, in fact, recommended.

When the number of integer variables is large, the Bender decomposition algorithm is the most effecient. Heuristically, this statement can be motivated by the observation that Bender's algorithm exploits the integer properties of the problem, which in this case dominate the solution process. As a consequence, Bender's method requires a fast 0-1 code in contrast to the B&B methods which require a fast LP code. Fortunately, several fast 0-1 codes exist, e.g., Geoffrion's extension of the Balas algorithm.

Bender's algorithm makes use of the fact that for given values of x, the problem reduces to an LP whose dual is independent of any particular choice of x. This enables an equivalent program with only one continuous variable to be formulated that can be solved as a subproblem to yield the overall integer solution. A brief description of this approach follows.

2.4.36.6 Functional Block Diagram




A
Using x̄, solve the LP
max u(b - Ax̄)
u
Subject to
uD ≤ e   u ≥ 0
for u
Is solution unbounded ?
Yes
Add the constraint
Σu_i ≤ M
No
Is z̄ - cx̄ ≤ u(b - Ax̄) ?
No
Add the constraint
z ≥ ey + u(b - Ax)
to MP 1
B
Yes
Solve the LP
min : ey
Subject to
Dy ≥ b - Ax̄
y ≥ 0
for the optimum value of the continuous variables y.
Return: Solution is Optimal

2.4.36.7 Typical Applications

This module can be applied to a wide variety of small OR type problems, e.g., capital budgetary, project selection, and alternative resource allocation. The modeling is, of course, restricted by the structure of the resulting mathematical program.

2.4.36.8 Implementation Considerations

Since this module has already been implemented, the numerous implementation considerations are described in the program documentation.

2.4.36.9 Reference

Benders, J. F.: "Partitioning Procedures for Solving Mixed-Variable Programming Problems, Numerische Mathematika," Vol 4, 1962.

2.4.37 PRIMAL_SIMPLEX

2.4.37 PRIMAL_SIMPLEX

2.4.37.1 Purpose and Scope

Primal simplex is an iterative algorithm for solving the general class of problems referred to as linear programming. Briefly stated a linear program (LP) is: given a set of m linear inequalities or equations in n variables, determine the nonnegative values of these variables that satisfy the constraints and optimize some linear function of the variables. Mathematically stated: determine the values x^j, $j = 1,2,\ldots,n$ that minimize:

[1] $$z = \sum_{j=1}^{n} c_j x^j$$

Subject to:

$$\sum_j a_j^i x^j - b^i \geq 0 \quad i = 1,\ldots,m$$

$$x^j \geq 0$$

The capability to solve the LP defined by Equation [1] is fundamental to any mathematical programming system. As a consequence, a large number of high performance codes exist for solving this basic class of problems (Ref 5, 6, 7). The principal components common to all of these operational codes are a modification of the basic revised simplex method by using the product form of the inverse, multiple pricing, and a composite approach for phase I. The reason that so much emphasis has been placed on speed and efficiency of the primal algorithm is that it is used repetitively by many other algorithms. For example, mixed integer and nonlinear separable programs both require an extremely fast primal algorithm to solve the numerous LP subproblems (typically several hundred) that arise in the solution of the master programs.

Current operational systems, such as those described in References 6, 7, and 8, are capable of solving problems with 10 000 constraints and essential unlimited number of decision variables. Problems with 1000 constraints are considered to be medium sized. The solution of problems of this size requires that the programming of the algorithm be done in assembly language to allow maximum computational efficiency. For problems of this magnitude, efficiency is synonymous with feasibility.

2.4.37.2 <u>Modules Called</u>

None

2.4.37.3 <u>Module Input</u>

The cost effective solution of large LP problems requires extensive data handling capabilities. These capabilities are needed to give the user flexibility and freedom in storing and operating upon the large masses of problem input data. In fact, the majority of the routines in any standard mathematical programming system are related to the data handling efforts.

The essential portions of the LP input are the large problems; these data are seldom input directly by the user, but rather generated or obtained from data previously written on mass storage devices. In any event, the data required are always essentially the same:

1) The objective function coefficients, c_j, $j = 1,\ldots,n$.

2) The constraint matrix, a_m^i, $i = 1,\ldots,m$, $j = 1,\ldots,n$.

3) The RHS vector, b^i, $i = 1,\ldots,m$.

4) The constraint specification type, ($=$, $\leq$, $\geq$).

2.4.37.4 Module Output

The primal simplex routine is not only capable of solving the LP, but it can also provide auxiliary information that is often as useful as the answer itself. Apart from a complete report writing capability, which is in itself a major issue, the minimum output of the primal simplex code should consist of:

1) Summary of certain key input information (costs, bounds, etc);

2) Status of each variable in the final solution, i.e., basic or nonbasic, feasible or nonfeasible, bounded or slack, etc;

3) Value of each basic variable;

4) Final objective function value;

5) Shadow prices (Lagrange or simplex multipliers);

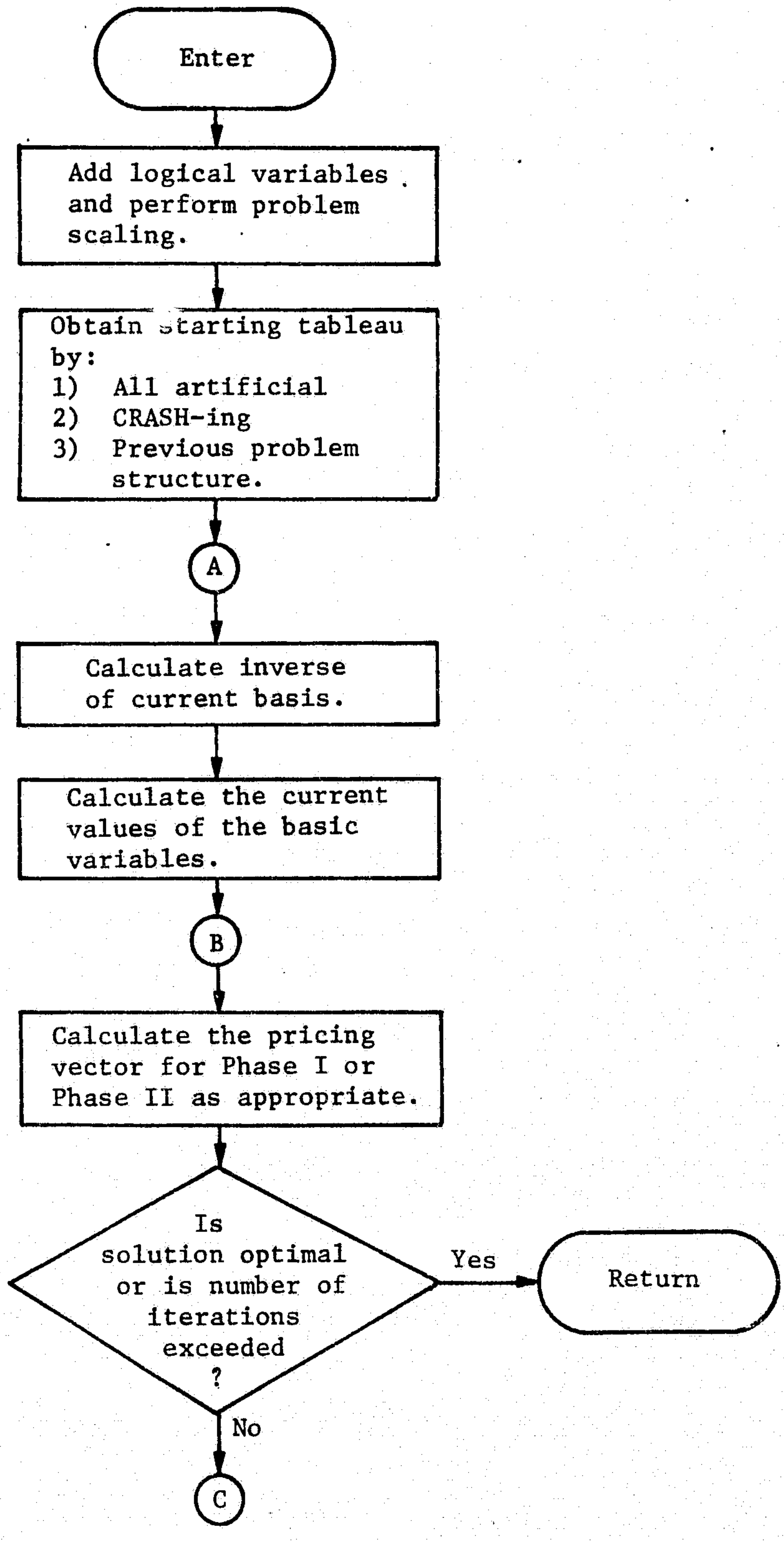
6) Iteration history summary, i.e., iteration counter, indices of basic variables, objective function value, etc.

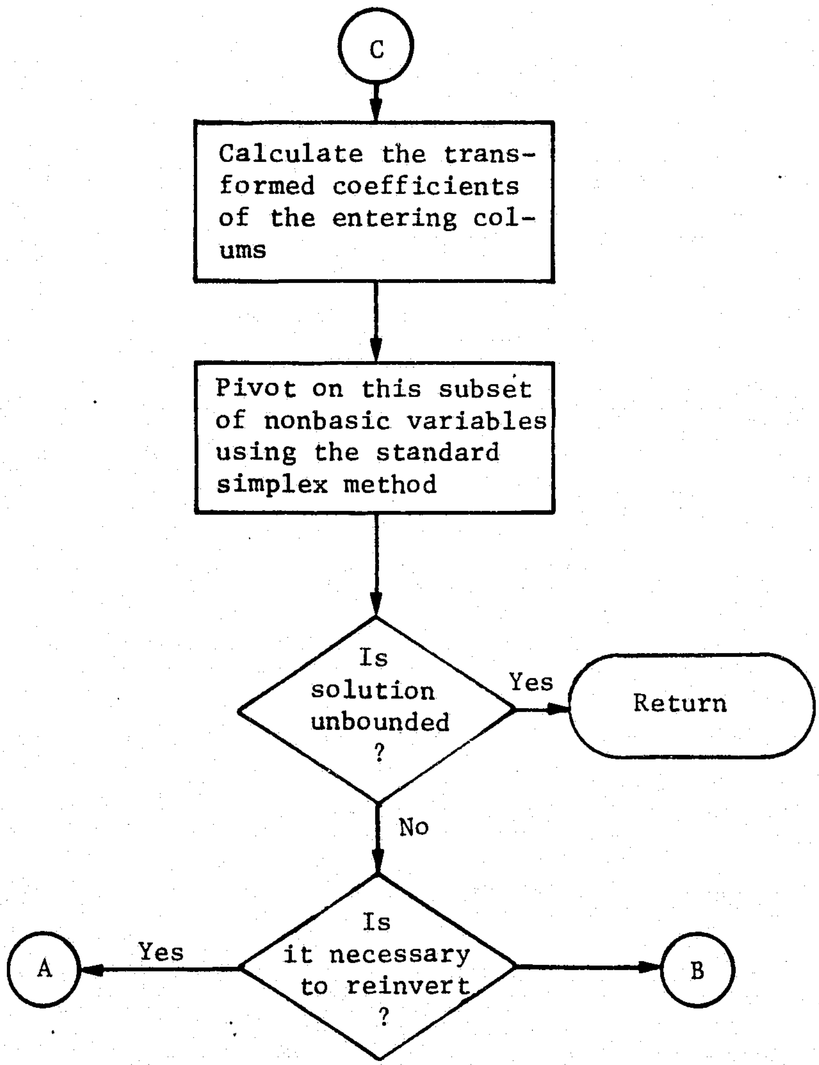
In some cases the algorithm may fail to converge and a diagnostic printout is required. The required printout is somewhat problem dependent; however, at a minimum it should include the (1) relative cost coefficients, (2) basic inverse, and (3) indices of the entering and leaving columns.

2.4.37.5 Functional Description

A description of the primal algorithm can be found in numerous references, and hence, it is not presented in detail here. For reference purposes, the code described in Orchard-Hays (Ref 2) is one of the most efficient. Reference 2 also gives a wealth of implementation techniques that should be reviewed before implementation. Since implementation of an efficient primal code is not straightforward, a general functional block diagram is presented. It is recognized, however, that many strategy variations exist in primal codes. Thus, this functional block diagram should only serve as a guideline during the development process.

2.4.37.6 Functional Block Diagram




C
Calculate the transformed coefficients of the entering colums
Pivot on this subset of nonbasic variables using the standard simplex method
Is solution unbounded ?
Yes
Return
No
Is it necessary to reinvert ?
Yes
A
B

2.4.37.7 Typical Applications

In general, there are few scheduling problems that can be modeled directly as linear programs. However, as mentioned previously, there are many situations in which this module is required as a basic computational routine in other algorithms that do apply directly to scheduling. In addition to these support functions, this module can be used directly to solve several categories of advanced planning and resource allocation problems. Simplified problems in transportation and distribution, production and inventory, and macro economics, can be modeled and solved with this type of a tool.

2.4.37.8 Implementation Considerations

The most important observation that can be made in this section is that the development of an efficient primal code is totally a problem of implementation not theory. As a result, every possible effort should be made to ensure that this algorithm is computationally efficient. This can be accomplished by careful development of the four principal areas that impact the efficiency of this module: (1) starting solution, (2) inversion, (3) pricing, and (4) pivot selection. All of these critical topics have been thoroughly discussed in the literature. A brief summary of the conclusions regarding these areas is presented in Table 2.4.37.8-1. Before the implementation of this module, it is recommended that the references presented in Table 2.4.37 -1 be completely reviewed.

Table 2.4.37 -1 Summary of Implementation Recommendation

Category	Principal Recommendation	Key References
Starting Solution (Phase I)	- Avoid starting from all artificial bases. - Use problem structure wherever possible. - CRASHing can have significant impact on the solvability of a particular LP.	2,3
Inversion	- Necessary to maintain control over numerical round-off problems - Reduces the size of the expanding representation of the basic inverse. - Reduces the total number of elementary operations by minimizing the number of nonzero elements in the representation of the inverse	8,9
Pricing and Pivoting	- Multiple pricing should be used. - Weighting the relative cost coefficients before selection has often significantly reduced the number of iterations. - The standard pivot selection rule used in single pricing algorithms appears to be best even in the multiple pricing environment.	10,11

2.4.37.9 References

1. Dantzig, G. B., and Wolfe, P.: "The Decomposition Algorithm for Linear Programming." *Econometrica*, 9, No. 4, 1961. *Operations Research*, 8, January - February, 1960.

2. Orchard-Hays, W.: *Advanced Linear-Programming Computing Technique*. McGraw-Hill Book Company, New York, N.Y., 1968.

3. Hadley, G: *Linear Programming*, Addison-Wesley Publishing Company, Inc, Reading, Mass, 1963.

4. Lasdon, Leon: "Optimization Theory for Large Systems." *MacMillan Series in Operations Research*. 1970.

5. MPSX-Benichou, M.; Gauthier, J. M.; Girodet, P.; Hentegis, G.; Ribrere, G.; and Vincent, O.: "Experiments in Mixed Integer Linear Programming," presented at the Seventh International Mathematical Programming Symposium, 1970, The Hague, Holland.

6. OPHELIE MIXED--Roy, B.; Benayoun, R.; and Tergny, J.: "From S.E.P. Procedure to the Mixed Ophelie Program," in J. Abodie (ed), Integer and Nonlinear Programming, North-Holland, Amsterdam, 1970.

7. UMPIRE--Tomlin, J. A.; "Branch and Bound Methods for Integer and Non-Convex Programming," in J. Abodie (ed) Integer and Nonlinear Programming, North-Holland, Amsterdam, 1970.

8. Hellerman, E., and D. Rarick; "Reinversion with the Preassigned Pivot Procedure," *Mathematical Programming*, 1, 2, (1971), p 195-216.

9. Larson, L.: "A Modified Inversion Procedure for Product Form of Inverse in Linear Programming Codes," Comm ACM Vol 5, 1962, pp 382-383.

10. Harris, P.M.J.: "Pivot Selection Methods of the DEVEX L.P. Code," British Petroleum Company, London, England, 1972.

11. Smith, D. M. and Orchard-Hays, W.: "Computational Efficiency in Product Form L.P. Codes," *Recent Advances in Mathematical Programming*, McGraw-Hill, New York, N.Y.

12. Lemke, C. E. and K. Spielberg, "Direct Search Algorithms for Zero-one and Mixed-Integer Programming," Operations Research, Vol 15, No. 5, 1967.

13. Balas, E., "An Additive Algorithm for Solving Linear Programs with Zero-one Variables," Operations Research, Vol 13, No. 4, 1965.

2.4.38 DUAL_SIMPLEX

2.4.38 DUAL_SIMPLEX

2.4.38 DUAL_SIMPLEX

2.4.38.1 Purpose and Scope

The dual simplex algorithm is a special purpose routine designed to solve the dual (of the primal) problem using the standard primal tableau. In essence, it is the primal simplex algorithm applied to the dual of the primal problem, where the dual of the LP

[1] Minimize $c'x$ Subject to $Ax \geq b$, $x \geq 0$ — Primal

$$\left.\begin{array}{l}\text{Minimize } c'x \\ \text{Subject to} \\ Ax \geq b \\ x \geq 0\end{array}\right\} \quad \text{Primal}$$

[2] Maximize $\pi'b$ Subject to $A'\pi \leq c$, $-\pi \leq 0$ — Dual

$$\left.\begin{array}{l}\text{Maximize } \pi'b \\ \text{Subject to} \\ A'\pi \leq c \\ -\pi \leq 0\end{array}\right\} \quad \text{Dual}$$

The dual simplex algorithm can be derived in a straightforward fashion by applying the primal simplex rules to the dual problem.

The most elementary application of the dual algorithm is when the number of rows in the primal is much larger than the number of columns. In this situation the primal algorithm would have to maintain an m-dimensional basis inverse while the dual algorithm would require only an n-dimensional basis. The resulting reduction in storage and computations can be significant. The dual algorithm is useful if:

1) Additional constraint rows are to be added to an LP whose optimal solution is known;

2) The RHS vector b^i is to be changed.

The advantages of the dual simplex in the above situations is that a new optimal, basic feasible solution can be easily constructed from the augmented dual problem. This is because the addition of a constraint or the alteration of the RHS vector in the primal simplex does not change the dual variable constraints. Hence, the dual solution corresponding to the optimal primal is also a basic feasible solution for the augmented dual simplex.

The dual algorithm can also be used in certain situations to eliminate the need of a Phase 1 in the primal algorithm. However, it is often difficult to find a basic feasible solution to the dual algorithm (i.e., a basic feasible solution to the primal with all positive relative cost coefficients). In the worst case, artifical variables may have to be added to the dual algorithm. This would require a Phase 1 in the dual algorithm that could be considerably more time-consuming than the direct application of the two-phase method to the primal algorithm. As a consequence, utilization of the dual simplex algorithm is only recommended in those situations where a dual basic feasible solution is readily obtained from the inherent problem structure.

2.4.38.2 <u>Modules Called</u>

None

2.4.38.3 <u>Module Input</u>

1) The number of structural variables, n

2) The number of constraints, m

3) The primal cost coefficients, c_j; $j = 1,2,\ldots,n$

4) The RHS vector, b^i; $i = 1,2,\ldots,m$

5) The constraint matrix, a_j^i; $j = 1,2,\ldots,n$; $i = 1,2,\ldots,m$

2.4.38.4 Module Output

1) Output option indicator

2) Iteration Summary

3) Solution Summary

2.4.38.5 Functional Description

The dual simplex algorithm can be motivated by the concept of complementary slackness. In essence, complementary slackness implies that if the relative cost coefficients of the primal algorithm are nonnegative, then the corresponding dual variables are dual feasible. In general, not every basic solution with $\bar{c}_j \geq 0$ will be feasible. However, when such solutions are feasible, then they are also optimal. Suppose we had a basic, but infeasible, solution that had all $\bar{c}_j \geq 0$ and this solution was updated by changing one column (row in the primal) at a time while maintaining $\bar{c}_j \geq 0$ for each update. An optimal, if one existed, could surely be found in this manner. This is precisely what the dual simplex algorithm does. However, the various simplex rules are slightly different. For example, the dual simplex algorithm computes the vector to leave the basis and then the vector to enter. This is the reverse of primal simplex. A summary of the key dual simplex operations follows. These operations can be motivated by studying the structure of the dual of the primal in canonical form

[3] $$\begin{bmatrix} \pi_1 & & & & & \\ & \ddots & & & & \\ & & \pi_m & & & \\ & & & \bar{c}_{m+1} & & \\ & & & & \ddots & \\ & & & & & \bar{c}_n \end{bmatrix} + \begin{bmatrix} (B^{-1})' \\ ---- \\ -(B^{-1}\bar{B})' \end{bmatrix} \begin{bmatrix} \bar{c}_1 \\ \vdots \\ \bar{c}_m \end{bmatrix} = \begin{bmatrix} (c_B B^{-1})' \\ ---- \\ c_{m+1} - \pi P_{m+1} \\ \vdots \\ c_n - \pi P_n \end{bmatrix}$$

$$-\bar{b}_1\bar{c}_1 - \dots -\bar{b}_m\bar{c}_m - v = -c_B B^{-1}$$

Applying the simplex rules to this canonical form yields the dual simplex algorithm.

1) Calculate the pivot column (pivot row in the primal tableau) via

$$\bar{b}_r = \min_i \left\{\bar{b}_i\right\} < 0$$

2) Calculate the pivot row (pivot column in the primal tableau)

$$c_s/-\bar{a}_{rs} = \min_{\bar{a}_{rj}<0} \left\{c_j/-\bar{a}_{rj}\right\},$$

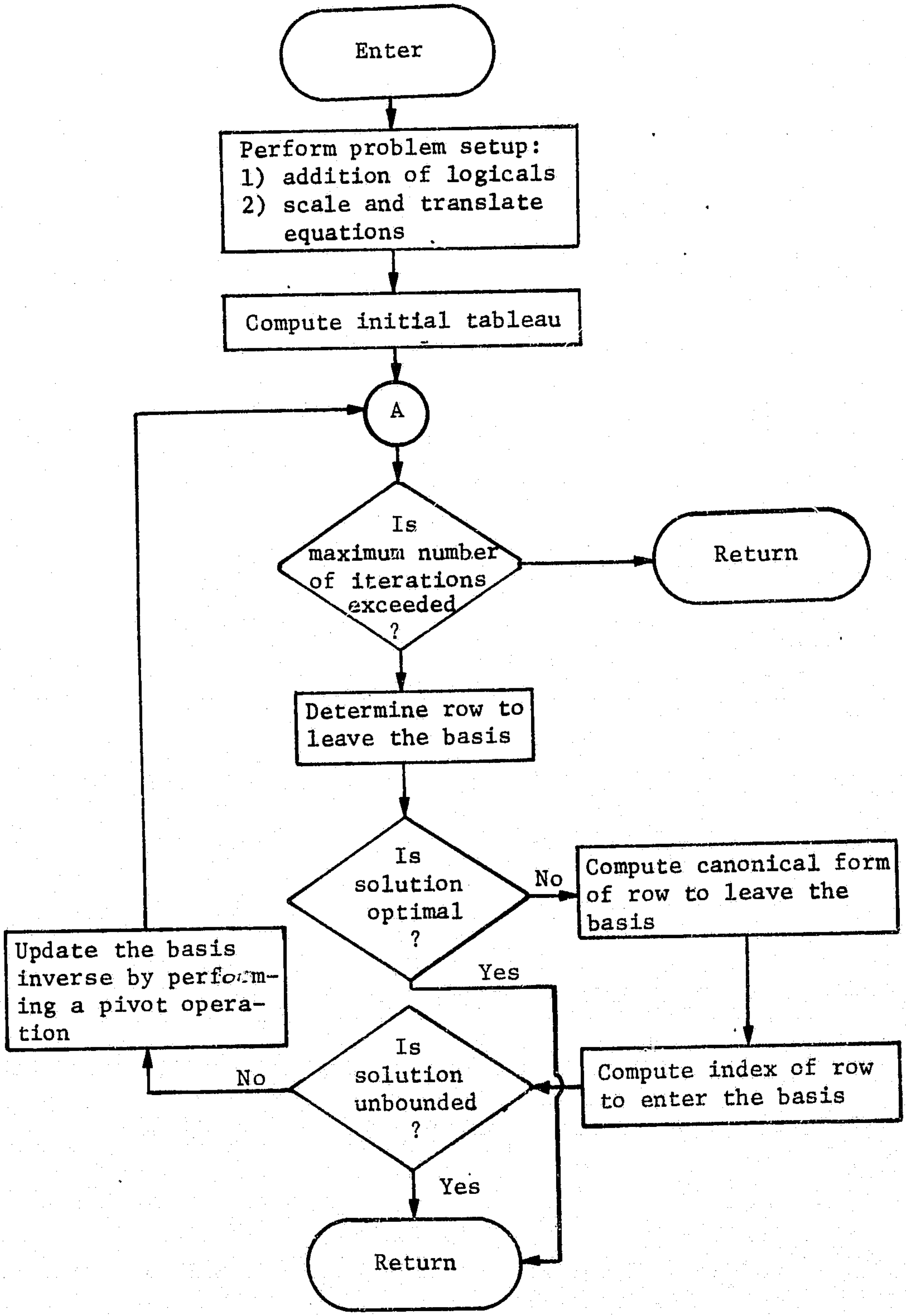
if all $\bar{a}_{rj} \geq 0$, then the primal is infeasible.

3) Pivot on $\bar{a}_{rs}$

4) If $\bar{b}_i \geq 0$, for all $i = 1,2,\dots,m$,

stop, the current solution is optimal; otherwise go to step 1.

2.4.38.6 Functional Block Diagram



2.4.38.7 Typical Applications

Dual simplex is generally used as a submodule in other algorithms where the highly specialized advantages of the dual structure can be exploited. For example, dual simplex is used internally in the Benders' decomposition algorithm to solve for the extreme points and rays of the primal problem for a fixed value of the integer variables. The dual is used in this situation because then the constraint set is independent of any particular choice of the integer variables. (For more details, see the description of the Bender decomposition algorithm.) Dual simplex is also used in the Geofferion zero-one algorithm to solve for the strongest surrogate constraint. In both of these examples, dual simplex was used because in the process of solving the master program a subproblem was created that was particularly compatible with the dual algorithm. This is very typical of the situations in which the dual simplex module would be used.

2.4.38.8 Implementation Considerations

A more general dual algorithm could be developed, similar to that described in Ref 3 which handles type 1 variables directly. In this more general setting, the dual algorithm is not the same as the primal simplex applied to the dual problem.

2.4.38.9 References

Lemke, C. E. and Spielberg, K: "Direct Search Algorithms for Zero-One and Mixed-Integer Programming; *Operations Research*, Vol 15, No. 5, 1967.

Lasdon, Leon: "Optimization Theory for Large Systems." *MacMillan Series in Operations Research*. 1970.

Orchard-Hays, W.: *Advanced Linear Programming Computer Techniques*, McGraw-Hill Book Company, New York, New York, 1968.

2.4.39 INTEGER_PROGRAM

2.4.39 INTEGER_PROGRAM

2.4.39.1 Purpose and Scope

This module can be used to optimize any system whose performance can be mathematically modeled as a linear function in a set of bounded integral decision variables that are subject to linear algebraic constraints. The difference between the integer and mixed-integer programs is that in the former all of the variables are constrained to be integral, while in the latter some may be continuous. Symbolically, the problem of determining the optimal system configuration must reduce to the following mathematical program

Minimize: $z = cx$

Subject to: $Ax + b \geq 0$

x integer

where

x is an nxl integer column vector of unknowns,

c is a lxn continuous nonnegative row vector of cost coefficients for the integer variables,

A is an mxn continuous matrix of constraint coefficients for the integer variables,

b is an mxl continuous column vector of constraint limits,

The implemented FORTRAN codes ZOSCA efficiently solves this program subject to certain restrictions. First, the decision vector x must be binary valued; that is, each of its components must be either zero or one. Theoretically, this requirement is not a restriction. As long as each integer decision variables is bounded above and below, it can be represented by a set of binary variables. Suppose

[1] $$\underline{x}^j \le x^j \le \bar{x}^j$$

where $\underline{x}^j$ and $\bar{x}^j$ are integers. Define k to be the unique smallest nonnegative integer such that

[2] $$\bar{x}^j - \underline{x}^j \le 2^{k+1}-1$$

Then the substitution

[3] $$x^j = \underline{x}^j + \sum_{i=0}^{k} 2^i y_j^i$$

replaces the single integer variable x^j with the k+1 binary variables y_j^i. In practice, however, such changes of variables can soon increase the dimensionality of a problem to the point where it is no longer tractable with existing computer codes. The second restriction is that the number of constraints, m, not exceed 50. The third and final restriction is that the number of decision variables, n, not exceed 90.

2.4.39.2 Modules Called

None

2.4.39.3 Module Input

1) The objective function coefficients, c

2) The constraint matrix, A

3) The constraint limits, b

4) Module control flags

2.4.39.4 Module Output

1) Final value of the decision variables

2) Final value of the objective function

3) Final value of the constraint feasibility

4) Optional levels of iteration diagnostics and decision histories.

2.4.39.5 Functional Description

The Geoffrion implicit enumeration algorithm incorporates two significant computational improvements to the standard Balas algorithm. These improvements are: (1) a flexible and economical version of the "back-track" procedure for exhaustive search in combinatorial problems; (2) augmentation of the "strongest" surrogate constraint to the infeasibility tests. The latter of these improvements is a major contribution in that it reduces the sensitivity of the solution time to the number of integer variables. In fact, results suggest that use of the imbedded linear program to calculate the "strongest" surrogate constraint reduces the solution-time dependence on the number of variables from an exponential to a low-order polynomial.

A functional description of the Geoffrion algorithm is presented in the functional block diagram in terms of the following nomenclature:

$\mathcal{S}$ - set containing the index of the elements in the partial solution, e.g., $\mathcal{S} = \{-5,4\}$ implies that $x_5 = 0$ and $x_4 = 1$ in the partial solution;

x^s - solution vector obtained from the trivial completion of $\mathcal{S}$;

$\bar{z}$ - incumbent cost function;

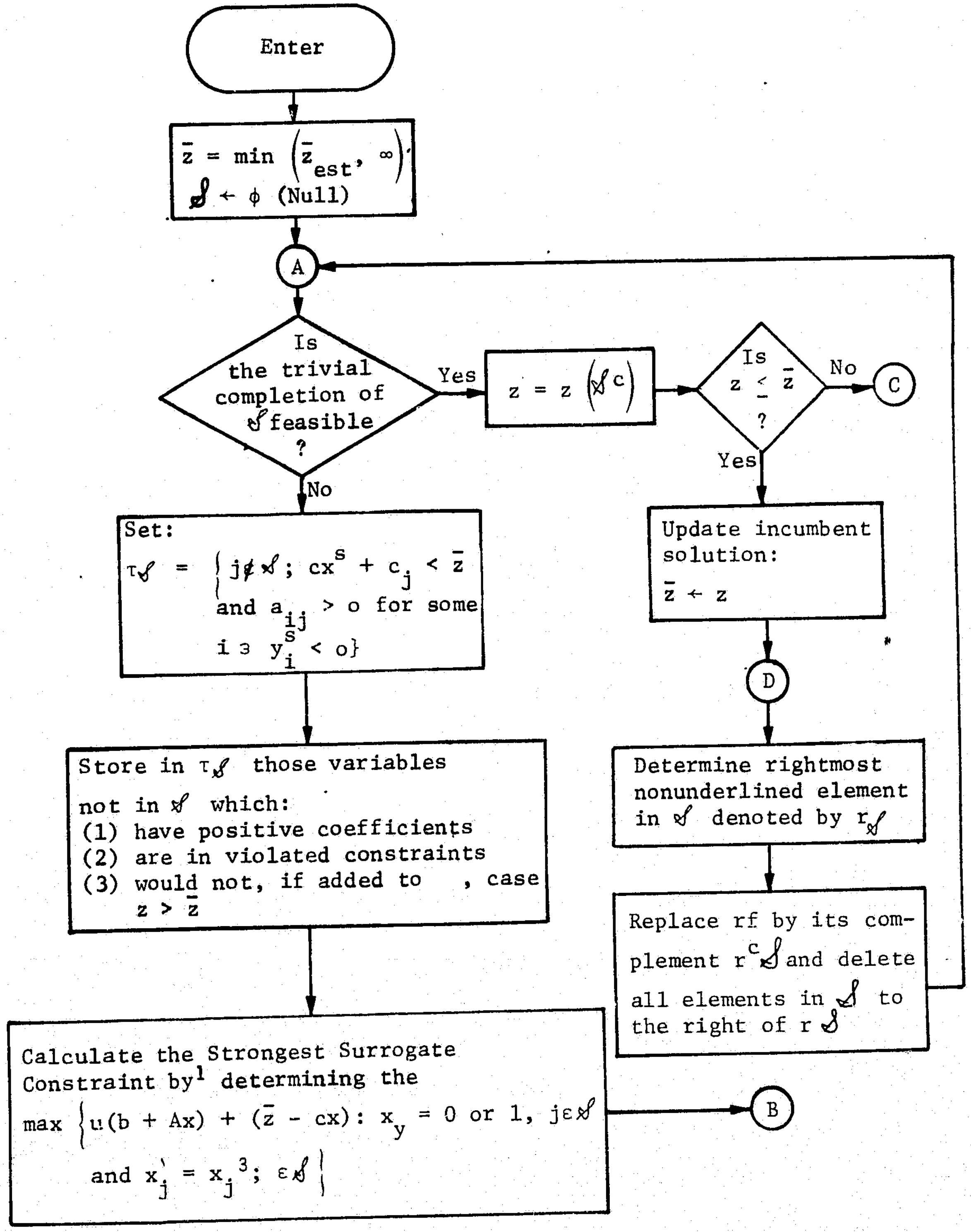
$\bar{x}$ - solution associated with $\bar{z}$, i.e., $\bar{z} = z(\bar{x}) = c'\bar{x}$;

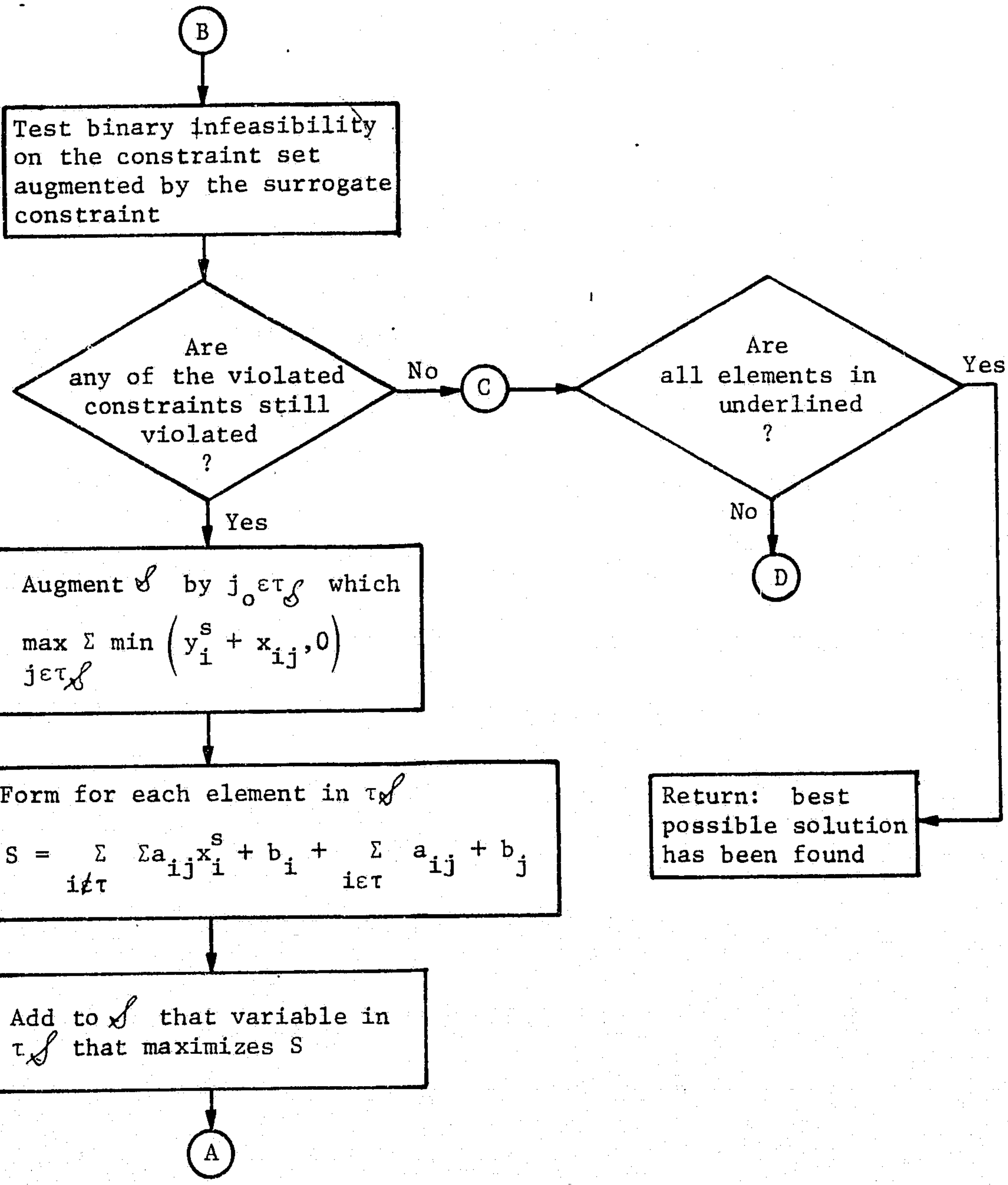
y^s - the constraints evaluated as x^z;

$T_{\mathcal{S}}$ - variables not in $\mathcal{S}$, which when elevated to 1 might eliminate infeasibility;

z_{est} - Input estimate of $\bar{z}$.

2.4.39.6 Functional Block Diagram




B
Test binary infeasibility on the constraint set augmented by the surrogate constraint
Are any of the violated constraints still violated ?
No
C
Are all elements in underlined ?
Yes
No
D
Yes
Augment $\mathcal{S}$ by $j_o \varepsilon \tau_{\mathcal{S}}$ which
$\max_{j \varepsilon \tau_{\mathcal{S}}} \Sigma \min \left(y_i^s + x_{ij}, 0 \right)$
Form for each element in $\tau_{\mathcal{S}}$
$S = \sum_{i \notin \tau} \Sigma a_{ij} x_i^s + b_i + \sum_{i \varepsilon \tau} a_{ij} + b_j$
Return: best possible solution has been found
Add to $\mathcal{S}$ that variable in $\tau_{\mathcal{S}}$ that maximizes S
A

2.4.39.7 Typical Applications

This module can be applied to any linear binary decision problem. For example, project scheduling can be formulated as a 0-1 program, and can be solved with this algorithm. [See sample problems for ZOSCA (zero-one surrogate constraint algorithm).] Experience in solving scheduling problems with ZOSCA indicates that only a small number of activities can be optimally scheduled in this manner. The principal limiting factor in this approach is the total slack in the activities to be scheduled. The more slack the harder the problem becomes. If there is little job slack, this approach becomes computationally feasible but with small expected payoff due to the highly constrained situation.

2.4.39.8 References

Geoffrion, A. M.: "Integer Programming Algorithms: A Framework and State-of-the-Art Survey." *Management Science*, Vol 18, No. 9, May 1972.

Balas, E.: "An Additive Algorithm for Solving Linear Programs with Zero-One Variables." *Operations Research*, Vol 13, No. 4, 1965.

2.4.40 REQUIREMENT_GROUP_GENERATOR

2.4.40 REQUIREMENT_GROUP_GENERATOR

2.4.40.1 Purpose and Scope

Scheduling problems that involve the allocation of distinguishable specific items to jobs may be solved via a model decomposition in which the individual identities of resources are initially relaxed leaving only resource pools. Project scheduling techniques can then be applied to generate start times for all jobs and to create the corresponding resource profiles. To complete the solution of the decomposed problem, specific allocations must then be made that are compatible with the timeline output by the project scheduling algorithms. These allocations must satisfy the requirements of specific jobs for resources with appropriate descriptors (e.g. a CREWMAN for the job 'LAUNCH' might require descriptor 'TRAINED' or a truck for the job 'LOAD' might require a descriptor 'EMPTY'). The allocations must also preserve resource continuity constraints between jobs. For example, CHECKOUT_PAYLOAD and LAUNCH_PAYLOAD both require a payload, and in fact this may be the same payload. Thus only one allocation is required for the two jobs. The problem description must contain the information that the two jobs are related through the requirements for the same specific resource, even though the identity of this resource is not provided.

The purpose of this module is to identify the jobs which require common resources. Its intended use is to bring requirements together into a group which is satisfied by the selection of a specific resource. The resource allocations for these jobs must be made on the basis of a group requirement and not on a job-by-job basis. Thus, a resource requirement group may be thought of as a set of requirements against which a single independent allocation decision must be made.

2.4.20.2 Modules Called

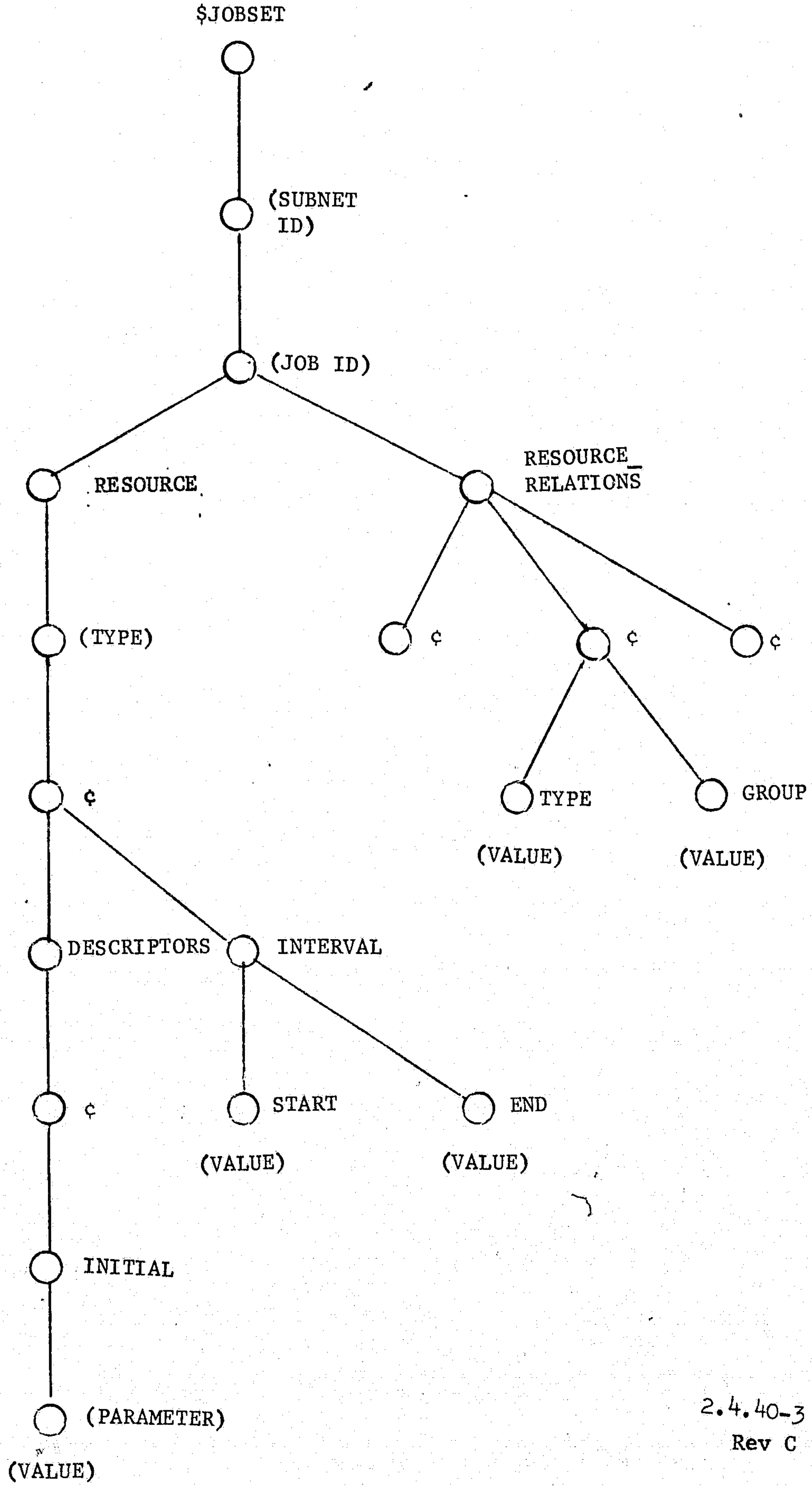
None

2.4.40.3 Module Input

The input to this module will be the descriptions of the resource relations between jobs, i.e., $JOBSET (see below) with the RESOURCE_RELATIONS substructure. This structure allows the problem formulator to specify that any of the resource items allocated to its parent process or opseq must be the same as those for any other process or opseq in the same OPSEQ with the same value of the 'GROUP' node. As separate occurrences of a process are identified as jobs, the GROUP value must be set with a value that is the same for the jobs within the opsequence but different from that for a different occurrence of the opsequence. For example, if an opsequence contains two processes that require the same resource, and the opsequence is to be repeated three times, the GROUP values that appear in $JOBSET might be 1 under each of the two jobs in the first occurrence, 8 for each of the jobs in the second occurrence, and 15 for each of the jobs in the third occurrence. Thus the resource selected must be the same within the same occurrence of an opsequence but could be different for different occurences of that opsequence. The 'TYPE' value specifies the resource type required by the job.

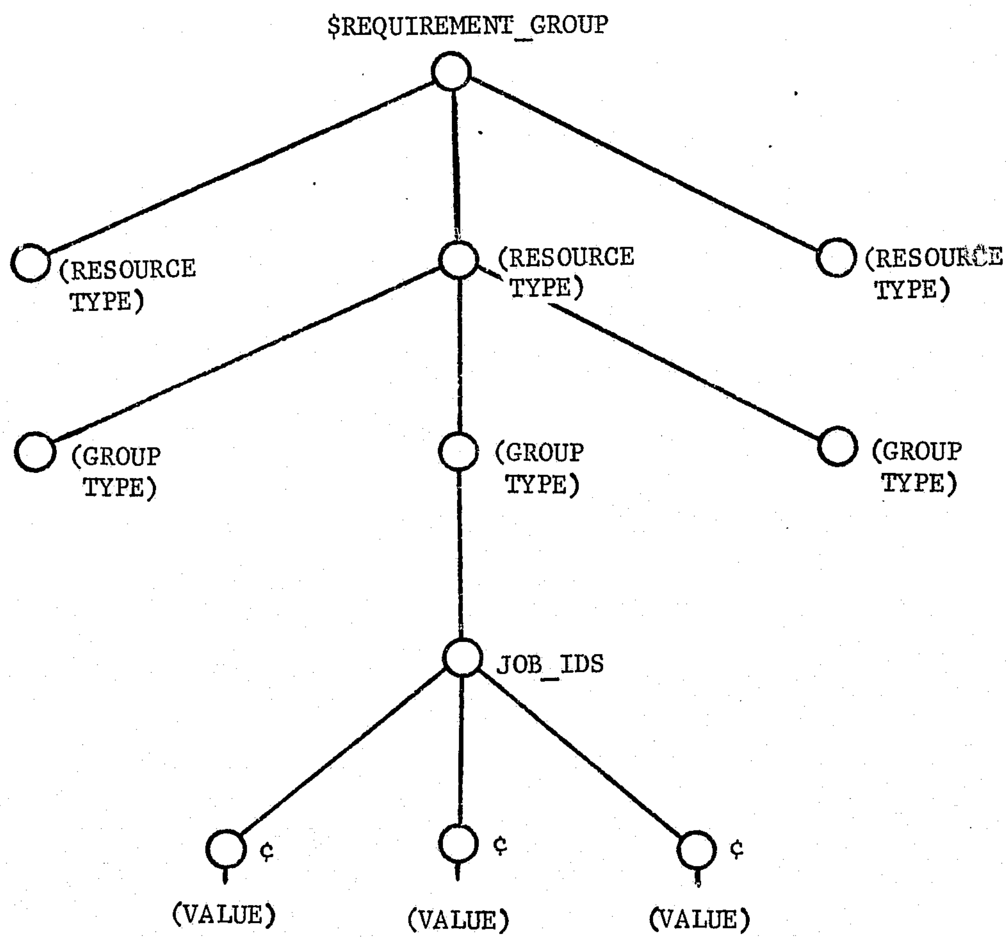
2.4.40.3 Module Input

The input to the module should be $JOBSET with the following minimum structure:

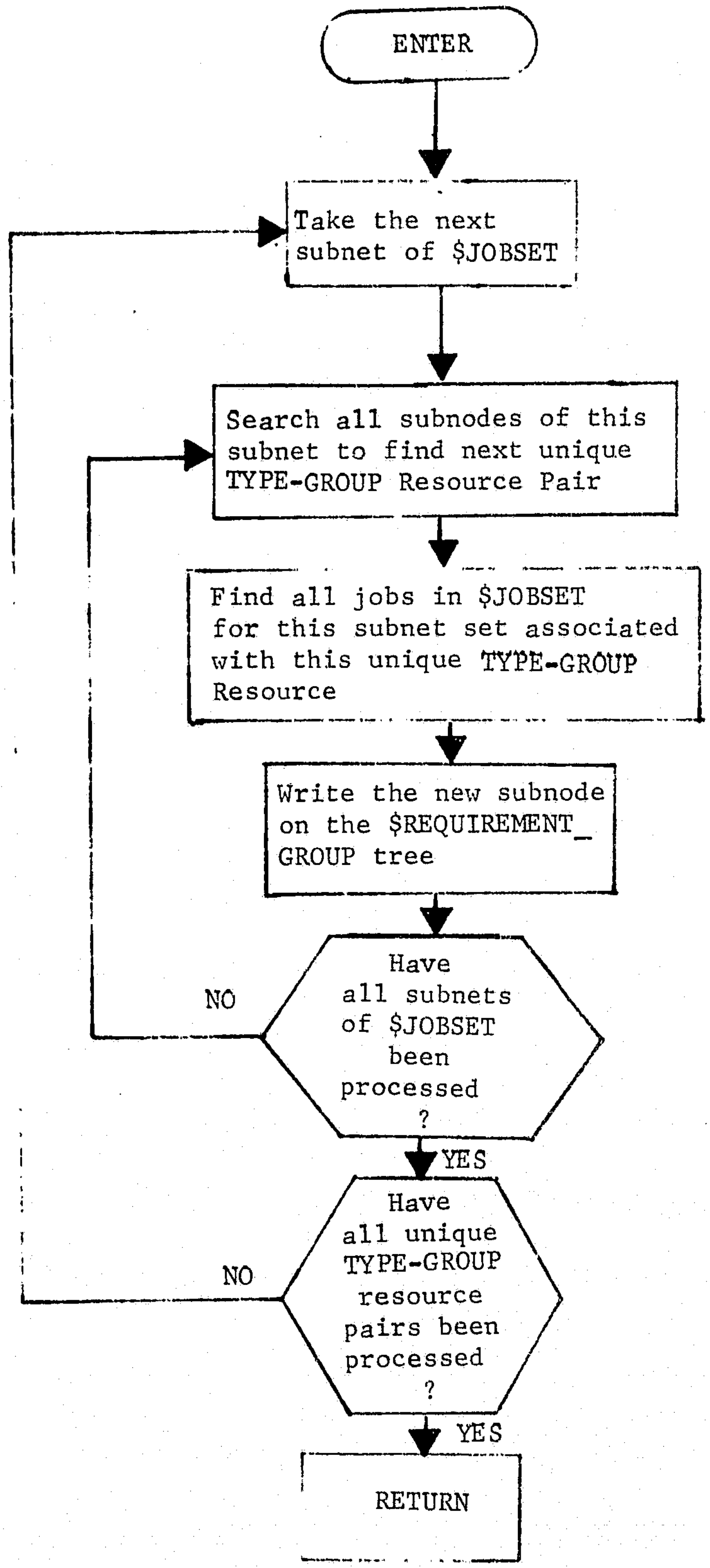


2.4.40.4 Module Output

The output of the module will be structured in a tree as shown below:



2.4.40.5 Functional Block Diagram



2.4.40.6 Typical Applications

This module provides a mechanism for explicitly allocating resources by collecting information about resource relations between jobs.

2.4.40.7 Detailed Design

The RESOURCE_RELATION node of each job of each subnet of $JOBSET is searched to find the next unique resource type-group pair. Then all jobs within that subnet are examined to see if they belong to the requirement group defined by the resource type-group pair. The job id of each job belonging to the requirement group is recorded.

2.4.40.8 Internal Variable and Tree Name Definitions

$SUBNET_ID	-	A pointer, specifying the particular subnet of $JOBSET currently working on.
$JOBNUM	-	A pointer, indicating the job id which is being evaluated for resource relations
$RELATED	-	A pointer, indicating the node of the RESOURCE_ RELATION structure currently being examined
$SAVE	-	A tree which has the type and group values of the resource relation currently being examined
$EXAMINED	-	A tree containing all resource relations examined thus far. It is pruned after each subnet of $JOBSET has been examined
$TYPE_NAME	-	Has the resource type name of the requirement group being generated

$GROUP_NAME	-	Contains the 'GROUP' value of the requirement group being generated
$JOBID	-	A pointer, within the internal procedure FIND_ ALL_JOB_IDS which points at all jobs in $JOBSET for the given subnet
$RELATIONS	-	A pointer, pointing at the RESOURCE_RELATION subnodes of $JOBID
$REQUIREMENT_GROUP	-	The resultant tree specifying resource relations between jobs

2.4.40.8 <u>COMMENTED CODE</u>

```
REQUIREMENT_GROUP_GENERATOR:   PROCEDURE($JOBSET, $REQUIREMENT_GROUP)
                OPTIONS(EXTERNAL);
/*********************************************************************/
/*                                                                   */
/*   SEARCH ALL SUBNODES OF $JOBSET TO FIND THE NEXT UNIQUE TYPE     */
/*    AND GROUP PAIR UNDER RESOURCE_RELATIONS                        */
/*                                                                   */
/*********************************************************************/
DECLARE $TYPE_NAME,$GROUP_NAME,$SUBNET_ID,$JOBNUM,$RELATED,$SAVE,
                    $EXAMINED    LOCAL;
   DO FOR ALL SUBNODES OF $JOBSET USING $SUBNET_ID;
      DO FOR ALL SUBNODES OF $SUBNET_ID USING $JOBNUM;
         DO FOR ALL SUBNODES OF $JOBNUM.RESOURCE_RELATION  USING
                                                  $RELATED;

           PRUNE $SAVE;
           LABEL($SAVE(FIRST)(NEXT))=$RELATED.TYPE;
           LABEL($SAVE(FIRST)(NEXT))=$RELATED.GROUP;
           IF $SAVE(FIRST) ELEMENT OF $EXAMINED      THEN;
           ELSE DO;
            GRAFT $SAVE(FIRST) AT $EXAMINED(NEXT);
            $TYPE_NAME=$RELATED.TYPE;
            $GROUP_NAME=$RELATED.GROUP;
            CALL FIND_ALL_JOB_IDS;
           END;
            END;
         END;
      PRUNE $EXAMINED;
      END;
 /*********************************************************************/
 /*                                                                   */
 /*    FIND_ALL_JOB_IDS SEARCHES THROUGH ALL SUBNODES OF $JOBSET      */
 /*    TO FIND ALL JOB_IDS ASSOCIATED WITH THE UNIQUE TYPE-GROUP      */
 /*    RESOURCE PAIR ALREADY DETERMINED.                              */
 /*                                                                   */
 /*********************************************************************/
 FIND_ALL_JOB_IDS:  PROCEDURE;
 DECLARE K,STIME,ETIME,KTR,POINTER,$TEMP,$SAVE,$INTERVAL_NUMBERS,
       $SUBIDS,$JOBID,$RELATIONS,$JOBNAME,$JOB_DESCRIPTION,
       $DESCRIPTION,$TEMP_TREE,$DESCRIPTOR   LOCAL;
   DO FOR ALL SUBNODES OF $JOBSET.#LABEL($SUBNET_ID) USING $JOBID;
         DO FOR ALL SUBNODES OF $JOBID.RESOURCE_RELATION  USING
                                                  $RELATIONS;

            IF $RELATIONS.TYPE IDENTICAL TO $TYPE_NAME &
               $RELATIONS.GROUP IDENTICAL TO $GROUP_NAME
               THEN IF LABEL($JOBID) ¬ELEMENT OF $REQUIREMENT_GROUP.
                  #LABEL($SUBNET_ID).
                  #($TYPE_NAME).#($GROUP_NAME).JOBIDS
               THEN DO;
  $REQUIREMENT_GROUP.#LABEL($SUBNET_ID).#($TYPE_NAME).#($GROUP_NAME).
                               JOB_IDS(NEXT)=LABEL($JOBID);

               END;
            END;
         END;
       END FIND_ALL_JOB_IDS;
  END REQUIREMENT_GROUP_GENERATOR;
```

APPENDIX: USER GUIDE TO THE TRANSLATOR WRITING SYSTEM

1.0 BASIC SYSTEM DESCRIPTION

The Martin Marietta Aerospace Translator-Writing System (TWS) is a set of procedures and software tools which provide a powerful method for rapid implementation of computer programming language translators. Aside from its speed, the TWS approach to translator implementation offers the even more important advantages of great flexibility and modifiability, and it helps to insure that the resulting translator is rigorously defined.

Figure 1 illustrates the translator implementation process using TWS. Manual steps are indicated by boxes with darkened corners. There are two manual inputs which are mandatory; the formal language definition and the token definition. These operations are defined in detail later in this document. Briefly, the language definition is a formal description of the mapping from the source language to the desired object language. This mapping is expressed in the form of a grammar (syntactic definition) of the source language which contains embedded semantic information expressed in terms of the object language. The source language is thus defined in terms of the object language. Such a grammar, augmented by semantic information, will be referred to as an "augmented grammar".

The token definition is a description of the basic elements of the source language in terms of the characters which comprise them. This input contains definitions of the formats of identifiers, numeric constants, etc. The definition takes the form of a state transition matrix. Occasionally, it may be necessary to modify

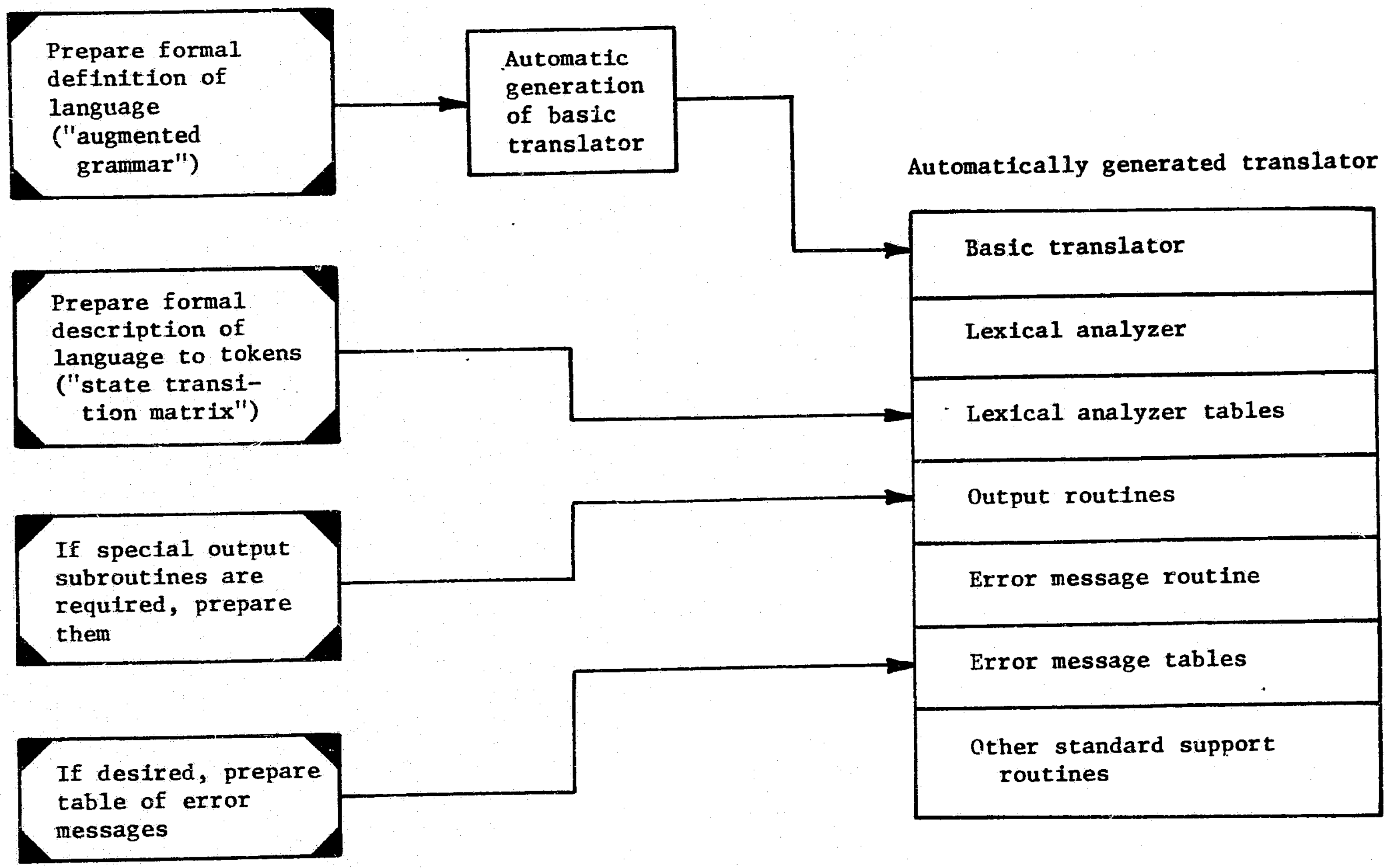


Figure 1. The Translator Implementation Process

the lexical analyzer (subroutine) itself, but a tabular input is sufficient for most stream-oriented source languages.

Depending on the object language, and possibly on the logical properties of the translation process (the source-object mapping), it may occasionally be necessary to modify the output routines. Separate output routines are, in any case, required for object languages with basically different format (e.g. FORTRAN's card-image versus PL/I's stream format). Given that the output routines exist for the specific object language selected, modification is usually not required.

Finally, if specialized error messaging is desired (rather than a constant error message for all error types), a table of error messages must be prepared.

Section 2 discusses in detail the use of the augmented grammar for language definition. Section 3 discusses the other (supporting) components of the generated translator.

2.0 THE TRANSLATOR DEFINITION METALANGUAGE

2.1 Basic Description

The TWS is based upon the use of the "augmented grammar" method of language definition. The source language is defined by an ordinary grammar which contains additional information pertaining to its meaning. This additional semantic information is used to specify what elements of the object language are to be generated to correspond to particular source language elements. The augmented grammar thus defines the mapping from source to object language. The augmented grammar is almost a complete description of the entire translator.

Let us consider, first, a simple example containing only syntactic information. The method used to specify language syntax will be that typically used in modern linguistics to describe context-free languages by means of a phrase-structure grammar. Specifically, a notation based on the Backus-Naur Form (Naur, 1960) will be used to define the syntax of a very simple language. Figure 2 shows the syntactic definition of the sample language.

```
SENTENCE  :=  NOUN_PHRASE  VERB_PHRASE ;
NOUN_PHRASE  :=  ARTICLE  NOUN ;
VERB_PHRASE  :=  VERB  NOUN_PHRASE ;
ARTICLE  :=  "THE" ;
NOUN  :=  "BOY" | "GIRL" | "DOG" | "CAT" ;
VERB  :=  "LOVES" | "HATES" | "BIT" | "SAW" ;
```

Figure 2 A Simple Grammar

This particular language is sufficiently simple that its definition can also be expressed in English. By comparing the formal definition in Figure 2 with the natural-language description following, the reader should easily acquire a feeling for the metalanguage used in such formal definitions. The definition says:

1) A sentence consists of a noun phrase followed by a verb phrase.
2) A noun phrase consists of an article followed by a noun.
3) A verb phrase consists of a verb followed by a noun phrase.
4) An article consists of the word "THE".
5) A noun consists of the word "BOY", or the word "GIRL", or the word "DOG", or the word "CAT".
6) A verb consists of the word "LOVES", or the word "HATES", or the word "BIT" or the word "SAW".

Such a grammar can be viewed in two ways other than as a simple definition. First, it can be viewed as a generative grammar. This particular grammar is capable of generating 64 sentences, such as, "THE BOY HATES THE CAT", "THE GIRL SAW THE DOG", etc. The generation of such sentences is accomplished simply by starting with the goal (in this case SENTENCE) and substituting its definition (NOUN_PHRASE VERB_PHRASE). Definitions are successively substituted for each variable that occurs until no variables remain. The resulting string is an instance of the goal (in this case, a sentence). This process can be viewed as generating a tree such as that shown in Figure 3 which describes the entire derivation of a sentence, or the phrase structure of the sentence.

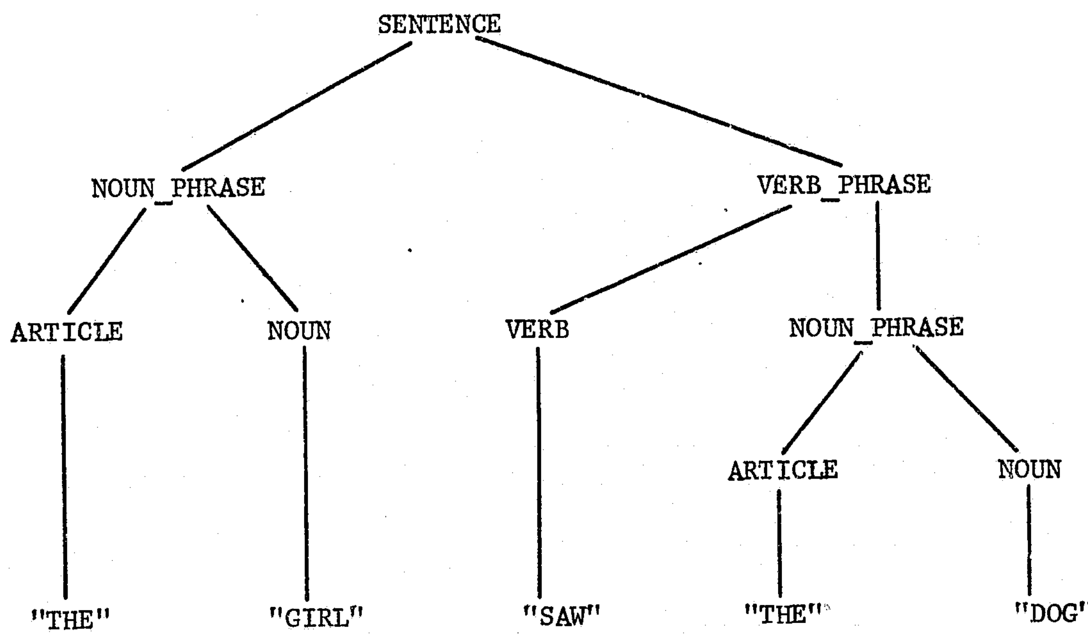


Fig. 3 A Phrase-Structure Tree

The other way of viewing such a grammar, and the relevant one here, is as the definition of a parsing process. Parsing is the inverse of the generative process. Parsing starts with, in this case, a sentence like "THE GIRL SAW THE DOG" and derives the underlying structure. Figure 3 can, therefore, be regarded as a parse tree, in which case the string at the bottom is the initial information and, with the help of the grammar, the structure is derived.

Parsing is the first step of the translation process. Once the input string has been recognized and its structure ascertained, the question of its meaning (the translation object language) can be addressed. This is not to suggest that parsing must be completed before object code can be generated, but parsing is logically first.

The augmented grammar translation method involves placement of information about object code (or meaning) directly in the grammar. Viewed as a translation process, the result is that object code is generated as parsing occurs. How this is accomplished will be seen in detail in succeeding sections. A point to be noted is that the structure and meaning of a language defined in this way is dependent for its definition on both the augmented grammar and the parsing properties that the translator is assumed to have.

2.2 Parsing Assumptions

As mentioned in the previous section, the assumed characteristics of the translator are significant when considering the meaning of a language definition expressed in augmented grammar form. The meaning of the individual symbols of the augmented grammar is not necessarily sufficient to fully define the language. Under certain circumstances, it is also necessary to know the parsing method employed. Consider, for example, the following partial grammar for a programming language.

```
STATEMENT  :=
     CONDITIONAL_STATEMENT
   | UNCONDITIONAL_STATEMENT ;
CONDITIONAL_STATEMENT  :=
     "IF"  CONDITION  "THEN"  STATEMENT
     ( "ELSE" STATEMENT | .EMPTY ) ;
```

To avoid the necessity for complex definitions of CONDITION and UNCONDITIONAL_STATEMENT, the following additional dummy rules will be assumed.

C-7

```
CONDITION  ::=  "CONDITION" ;
UNCONDITIONAL_STATEMENT  ::=  "STATEMENT" ;
```

It can be seen that this grammar provides for nested conditional statements such as

IF CONDITION THEN IF CONDITION THEN STATEMENT ELSE STATEMENT.

The statement is ambiguous, however, because the grammar is ambiguous. Does the "ELSE"-clause go with the first or second "IF"? Considered from a purely grammatical point of view, it may not matter. However, if the statement is to have meaning, the ambiguity must be resolved since the two syntactic structures may not mean the same thing.

There are two approaches available for the resolution of such ambiguities. The first, and undoubtedly the most aesthetically pleasing, is to rewrite the grammar in unambiguous form. The grammar for conditional statements can, in fact, be rewritten so that any "ELSE"-clause will be associated with the innermost "IF" that doesn't yet have an "ELSE"-clause. Unfortunately, the resulting grammar is considerably more cumbersome than that just presented. It is unambiguous, however, and that is an absolute requirement.

The second approach is the one adopted here, both to keep the grammar simple and to allow some more powerful specification "tricks". This approach is to assume a particular parsing method. Specifically, top-down deterministic parsing has been assumed. This assumption allows otherwise ambiguous grammars (such as that just presented for conditional statements) to be unambiguous.

A brief natural-English description of the parsing process for a single statement will be given. Unfortunately, space does not allow a detailed discussion of parsing methods in this document. Readers desiring further information should consult a standard reference, such as Aho & Ullman (1972) or Gries (1971). The basic properties of top-down deterministic parsing are: (1) once the first syntactic element in an expression has been recognized in the input string, the parser is committed to that expression and will not consider any alternative; (2) alternatives are considered in order; (3) if the first syntactic element of an alternative is not found, the next alternative is considered; (4) reiterating point (1), <u>no backup ever occurs</u> once the first syntactic element of an expression has been recognized.

Let us consider how top-down deterministic parsing affects the parsing of the statement

IF CONDITION THEN IF CONDITION THEN STATEMENT ELSE STATEMENT

using the previous grammar for conditional statements. The goal is a STATEMENT. Therefore, the input string is examined for concorddance with the definition of STATEMENT. The first alternative, CONDITIONAL_STATEMENT, is considered and its first syntactic element, "IF", is recognized in the input string. "CONDITION" is readily identified as an instance of the required metavariable, CONDITION, and "THEN" is recognized. The next required element in the CONDITIONAL_STATEMENT definition is a STATEMENT. This makes STATEMENT the goal again, but the remainder of the input string is now

IF CONDITION THEN STATEMENT ELSE STATEMENT.

This is readily recognized as another conditional statement and the first three symbols are processed by the same mechanism as before. This returns us again to STATEMENT as a goal, but with only STATEMENT ELSE STATEMENT remaining in the input string. Since the next symbol in the input string is not "IF", the CONDITIONAL_STATEMENT alternative is rejected. "STATEMENT" is recognized as an UNCONDITIONAL_STATEMENT and removed from the input string.

Now consider the current status of the parser. It has just recognized the STATEMENT element required by the definition of the inside CONDITIONAL_STATEMENT. It is still in the process of identifying the STATEMENT element of the outside CONDITIONAL_STATEMENT. What does it look for next? The next element it seeks is the symbol "ELSE" as a part of the <u>inside</u> CONDITIONAL_STATEMENT. Since the input string now looks like

ELSE STATEMENT

the requirement is satisfied, and the "ELSE"-clause is always associated with the innermost CONDITIONAL_STATEMENT. Hence, an ambiguous grammar has been rendered unambiguous in application.

Since the TWS uses top-down deterministic parsing, it is theoretically limited to a particular class of languages capable of being parsed in this manner. There are, however, several specific provisions of the TWS which serve to increase the parsing power of the system. Furthermore, the relevant class of languages encompasses most well-formed programming languages, so this limitation is not serious.

2.3 Metalanguage Primitives

This section describes all the primitive symbols ordinarily found in the augmented grammar definitions used by the TWS. These primitives are readily separable into several classes, which will be discussed in the following order:

- 2.3.1 Grammar and translator structure
- 2.3.2 Terminal symbols and symbol classes
- 2.3.3 Internal symbols
- 2.3.4 Stack Manipulation
- 2.3.5 Output
- 2.3.6 Symbol table operations
- 2.3.7 Artificial control
- 2.3.8 Error recovery and messages

First, a brief description of the basic grammar structure is in order. Every grammar begins with a start (".AUG_GRAM") statement and ends with ".END". Between these statements, the grammar contains one or more rules. Each rule consists of the rule name (which must have identifier syntax) followed by the string ":=", which has the meaning "consists of". The body of the rule then consists of one or more alternatives, separated by the <u>or</u> symbol (|). Within an alternative there must appear at least one syntactic element, and there may appear any number of additional syntactic and semantic elements. Several types of syntactic elements, as well as semantic elements, are defined later in this section. An additional syntactic element not mentioned later is the name of a rule. By writing the name of a rule within the body of another rule, the user causes the named rule to be applied during the parsing process.

A few additional conventions require explanation. If it is desired to combine several alternatives, so that the satisfaction of any one of them will constitute satisfaction of a syntactic requirement, the alternatives may be separated by or symbols and grouped in parentheses, as

(ALTERNATIVE_1 | ALTERNATIVE_2 | ALTERNATIVE_3).

There is also an iteration operator, the dollar sign ($), which is read "zero or more occurrences of . . .". Thus, a list of items separated by commas might be represented by the rule

LIST := ITEM $("," ITEM);

The final semicolon, incidentally, serves to terminate the rule.

2.3.1 Grammar and Translator Structure

The semantic elements listed below are basic to the structure of both the augmented grammar and the resultant generated translator.

- .AUG_GRAM - This must be the first element in any augmented grammar since it causes initialization of the generated translator. .AUG_GRAM must be immediately followed by the name of the goal rule. The goal rule name may optionally be followed by the following two parameters enclosed in parentheses:

  .INITIAL_CODE - specifies a dataset that contains initialization code (e.g., declarations) to be included at the beginning of the generated translator.

  .FINAL_CODE - specifies a dataset that contains wrapup code (e.g., statistical output statements) to be included at the end of the generated translator. Any such code will be executed upon satisfactory completion of parsing by the new translator. The datasets specified by these two parameters must be members of a user-defined library file. If a user wished to write a .AUG_GRAM statement for a grammar called GRAMMAR_37 and had the appropriate initial and final code in datasets START_UP and WRAP_UP, the statement would look like this:

.AUG_GRAM GRAMMAR_37 (.INITIAL_CODE=START_UP,.FINAL_CODE=WRAP_UP)

- .END - this must be the last element in a grammar. It simply indicates that there are no more rules in the grammar.

2.3.2 Terminal Symbols and Symbol Classes

Two different mechanisms exist which allow the TWS user to specify that a particular element must appear in the input string at a particular point in the parsing process. The first of these is concerned with specific symbols, and was illustrated in the sample language of Figure 2. If it is desired to test for a specific character string, that string is enclosed in quotation marks ("), as "READ", "WRITE", etc.

Frequently, the specific character string is irrelevant, but it is necessary to test for the presence of an element of a terminal symbol class. Several such classes are in common use, and the TWS provides for them. New classes can be readily added. The establishment of a class is accomplished by providing for it in the TWS. The specific properties which that class possesses, however, are determined by the lexical analyzer and may be varied from language to language. A typical set of these "tokens" is briefly described below.

.ID - An identifier consists of a string of alphanumeric and break (underbar) characters up to 31 in length. The first character must be alphabetic.

.LABEL - A label consists of an identifier followed immediately by a colon. (e.g. THIS_IS_A_LABEL:)

.NUM - A number is defined to be any of the conventional representations for a unsigned fixed or floating point value. (e.g. 84, 5.274E-19, 61.8)

.STRING - A string is enclosed in single quotes (') and may be up to 255 characters in length. Single quotes may appear within the string only in adjacent pairs.

2.3.3 Internal Symbols

The TWS provides two kinds of internal symbol. The first of these, called a "symbolic name for internal reference" (SNIR), provides a mechanism for automatic generation, by the translator, of dummy variable names, labels, etc. This mechanism is used whenever it is necessary to output a name not contained in the source program, if that name is required to vary with repeated outputs. The simplest example is the generation of labels. If the translator is required to generate labels, they must obviously all be different. Yet the grammar which generates them cannot refer to all possible labels; it must be able to cause output of "the next" label by referring to some symbolic name. This capability is provided by the

SNIR, as is the capability to generate groups of temporary (or reuseable) names for use as dummy variables. A SNIR is referred to as a string followed by an asterisk and a number, as "LABEL*01". LABEL*01 is a reference to the first translator-generated label which was available at the time of entry into the current rule.

The second internal symbol type is the switch, or flag, which can be set and tested at parsing time as directed by the augmented grammar. This capability allows a rule to be used for slightly varying purposes without rewriting the rule in multiple copies. It also allows memory of simple information throughout part or all of the parsing process, and thus allows the grammar to behave somewhat like a parametric grammar.

- .SNIR_USE - This element is used to reserve SNIR's (symbolic names for internal reference) for use by the grammar rule in which they appears. The number of each SNIR type to be reserved is specified in a parenthesized list that immediately follows .SNIR_USE. For example, if a user wished to reserve one SNIR of type "LABEL" and three of type "NUMBER", the first element on the right-hand-side of the appropriate rule would be: .SNIR_USE(LABEL*01,NUMBER*03). It is convenient to think of each SNIR type as a list of actual variable names with an associated availability pointer. .SNIR_USE reserves a given number of the actual variable names (so that they can be referenced symbolically in the rule) and also advances the availability pointer by that number. It is by this approach that .SNIR_USE allows nested rules to generate and use their own symbolic variables without mutual interference.

- .RESET- This element is used to logically perform the inverse function of .SNIR_USE by allowing the user to move the availability pointer for one or more SNIR types. The appropriate symbolic name is used to refer to the variable to which the pointer is to be moved. .RESET can be used to allow different rules to access the same variable and to make dummy variables available for reuse.
- .SET - This element provides a mechanism for the storage of temporary state information by the generated translator during the parsing process. It allows the use of status switches so that similar rules can be combined to avoid redundancy. The variable and the value it is to be set to are specified in a typical assignment statement format enclosed in parentheses immediately following .SET. For example, if a user wants to set OPTION_SWITCH to a value of 7, he would specify it as follows: .SET(OPTION_SWITCH=7). The left hand side of the "assignment statement" must be an identifier and the right hand side can be a numeric value or a character string.
- .TEST - Of course .SET would be of no benefit if there were not a means to test the variables that it sets. .TEST provides this capability. .TEST is actually a syntactic element that sets the parser's true-false indicator based on whether the specified condition was true or false. That is, if the condition specified in .TEST is false, it will have the same effect as if the attempted recognition of an element had failed.

.TEST can be used to test for a not-equal ("¬=") condition as well as an equal ("=") condition.

2.3.4 Stack Manipulation

The next three elements are used primarily to accomplish re-ordering of the output items (e.g., as in translation from infix operator notation to suffix notation). To facilitate operations of this sort the generated translator is provided with a push-down stack. Output items or groups of output items can be placed on the stack and saved there temporarily. Later, they can be popped from the stack for output or simply discarded.

In the following discussion three different parameters will appear frequently. Since their meaning is constant, they are defined here to avoid unnecessary repetition.

1) # - the pound sign always refers to the item on the top of the translator stack.

2) * - an asterisk always refers to the last terminal symbol recognized in the parsing process.

3) ** - double asterisks always refer to the variable currently indicated by the symbol table pointer.

- .SAV - This element places one or more items on the stack, saving them for future reference. The items are enclosed in parentheses and may be any one of the following types: 1) *; 2) **; 3) a SNIR; or 4) a character string.
- .CAT - This element places items on the stack by catenating them with the item currently on the top of the stack. That is, it differs

from .SAV in that it does not cause the stack to be "pushed down". This makes it possible to add one or more items to the item on top of the stack. The result can then be treated as a single unit that can be popped off the stack by a single translator command. .CAT operates on any of the four items that can be specified with .SAV.

- .POP - The pop command is used to simply throw away the top item on the stack. Since only the top item on the stack can be accessed, .POP is always used with a pound sign, i.e. .POP(#).

2.3.5 Output

The next two elements provide the output mechanism for the generated translator.

- .OUT - This element can be used for normal output from the translator stack. When used with a pound sign it removes the top item from the stack and causes it to be written out.

  .OUT is the primary output element and is sufficiently flexible to handle all other types of output except labels. .OUT is followed by a list of output items separated by commas and enclosed in parentheses. An output item may be any of the following five types: 1)#; 2)*; 3)**; 4)a SNIR: or 5) a character string enclosed in double quotes. The last of these types is the most common. These character strings will usually be parts of program statements in the object language of the generated translator.

- .LAB - This element allows the generated translator to create and output statement labels. The label name may be generated from a SNIR or from the last recognized terminal symbol. For example, if the last symbol was LABEL4, .LAB(*) would result in the output, "LABEL4:". .LAB simply appends a colon to the appropriate character string and then outputs it.

2.3.6 Symbol Table Operations

Since the generated translator must collect information about the identifiers and data aggregates encountered in the source program, it is provided with a "built-in" symbol table. The symbol table is an essential part of the translator and is used for storage of symbol names and attributes. The ten elements discussed below can be employed by the user to build, interrogate, and modify this table.

In order to allow and encourage structured programming techniques, the symbol table has intentionally been designed to easily accommodate block structured languages. It consists logically of a two-dimensional table and a pointer that can be moved to any table location. Each entry in the table consists of a symbol name and a character string specifying the symbol attributes. Each row in the table corresponds directly to a source program "block" of code.

- .BLKENTER and
- .BLKEXIT - These two elements are used to move the symbol table pointer vertically. .BLKENTER moves the pointer down a row and effectively clears from the row all previous symbol entries at that level. .BLKEXIT simply moves the pointer up a row. These elements can be used to implement a stack symbol table. Such an implementation is especially useful in a one-pass translator whose source language is block structured.

- .SEARCH_BLOCK and
- .SEARCH_ALL - Before a new symbol is entered into the symbol table it is almost always necessary to perform a search of all or part of the table in order to insure that the symbol has not already been entered. These two elements provide both the search and entry functions. .SEARCH_BLOCK searches only the last row of the symbol table, attempting to match each entry with last terminal symbol recognized (normally referenced with an asterisk). If the symbol is not found, it is entered as the next symbol in that row. .SEARCH_ALL operates in the same way except that it searches the entire symbol table from the current block up. Symbols that it does not find are entered at the highest level (i.e., the first row).
- .IF_NEW - This element is often used immediately after .SEARCH_BLOCK and .SEARCH_ALL. It is the primary mechanism by which symbol attributes are entered in the symbol table and associated with a given symbol name. .IF_NEW checks to see if the symbol currently indicated by the pointer is a new entry; if so, its associated attribute character string is changed to user-supplied specifications. Three arguments must be supplied. The first two are integers that specify the position and length of the portion of the attribute character string to be modified. The character string to be entered is the third argument, as .IF_NEW(1,2,"PD").
- .ENTER - The usage and operation of .ENTER are the same as those of .IF_NEW with one difference. As might have been guessed, .ENTER operates unconditionally (i.e. the attribute entry is always made,

regardless of whether or not the current symbol table entry is new.)

- .TABLE_TEST - An attribute character string is specified with this element in the same way as with .IF_NEW and .ENTER. .TABLE_TEST attempts to match the specified string with the attributes of the symbol currently indicated by the symbol table pointer. It then sets the translator's true-false indicator based on the success or failure of this attempted match. This element is syntactic.
- .INIT_BLOCK - This element simply moves the symbol table pointer to the beginning of the latest symbol table row, which contains all the symbols encountered thus far in the translation of the current source program block.
- .FIND_NEXT - This element is used to find the next symbol table entry in the current row that has a given set of attributes. It begins its search with the symbol <u>after</u> the one indicated by the symbol table pointer. If a symbol with the given attributes is found, the symbol table pointer is updated to indicate the appropriate entry. The attribute character string to be found is specified with standard three-argument format.

  This element is often preceded by an .INIT_BLOCK and then invoked iteratively to find all symbols of a given type at the current block level.
- .SEARCH_PROCEDURE - This element is not in any way related to the symbol table but is functionally similar to .SEARCH_BLOCK. It searches a separate list of names for a character string specified

immediately after it and enclosed in parenthesis. If a name is found in the list that matches the specified string, its associated count is incremented by one. Otherwise, the new name is entered into the list and its count is initialized to one. This mechanism is available to the user as a general tool and can be used for a variety of purposes (e.g. statistics keeping, data area from which wrapup code can be generated, etc.)

2.3.7 Artificial Control

The syntactic and semantic elements discussed so far provide the bulk of the functional capabilities generally required by a translator. The additional elements described below can be used to furnish some degree of artifical control over the parsing process. This makes the generated translator more flexible by allowing some variation from its normal top-down deterministic operation.

- .EMPTY - This element is equivalent to a reference to the null character string. Of course, this syntactic element is always matched during the parsing process. This fact makes .EMPTY useful for forcing the top-down deterministic parser to commit itself to a given grammar rule.

.EMPTY is also useful for specifying optional elements. For example, one could define a number to be preceded by an optional plus or minus sign with the following rule:

```
NUMBER: = ("+" | "-" | .EMPTY)  .NUM;
```

This rule specifies that either "+", "-", or nothing at all may precede the number itself.

- .NEG - This element unconditionally sets the parser's true-false indicator to false. Of course this normally occurs only when there is a failure to recognize a syntactic element.
- .RETURN - This element causes immediate return from the currently active rule to the rule that invoked it, with no change to the setting of the true-false indicator. Positive or negative returns may be caused by ".EMPTY .RETURN" or ".NEG .RETURN", respectively.
- .PEEK - This element allows the translator to look ahead at the next source symbol without removing it from the input string. The string to be "peeked" for follows .PEEK and is enclosed in parenthesis. The parser's true-false indicator is set based on whether or not the next symbol matches this string. It is also possible to peek for a terminal symbol class and to peek for a list of items, as .PEEK(";" | .ID).
- .DO - This element can be used to cause the immediate execution of a statement written in the host language of the translator. The statement must appear as a character string enclosed in parentheses, immediately following the .DO, e.g. .DO("THIS IS A TRANSLATOR STATEMENT;"). .DO causes the statement to be output in-line with the code of the generated translator.

Since it is rarely the case that the needed translator logic cannot be implemented using the other elements, the use of .DO generally should be avoided, if possible. However, .DO is a useful tool for generating extra logic unrelated to the translation process (e.g. output formatting, statistics keeping, etc.).

2.3.8 Error Recovery and Messages

A sophisticated translator should be able to detect and recover from errors in the source program it is parsing. The detection of errors allows appropriate messages to be output to the programmer. Recovery from errors allows the translator to continue parsing, thus detecting all, or at least most, syntax errors in one pass. The remaining elements provide both the recovery and message output mechanisms required.

- .ERR - This element is used to conditionally output error messages and also provides needed error recovery capabilities. The error message can be specified directly as a character string or indirectly by an integer. If an integer is used, it is assumed to be the index of a message contained in a user-supplied array called "@ERROR_MESSAGE".

  The two elements described below are used as arguments by .ERR to scan past extraneous information before normal operation of the parser is resumed.

  .SCANTO - causes the translator to scan to the first occurrence of the character string specified as its argument.

  .SCANBY - causes the translator to scan to the character <u>after</u> the first occurrence of the specified character string.

  The argument of .SCANTO or .SCANBY can be the character string to be scanned for or it can be "MATCHING_PAREN". The latter causes the translator to find the next unmatched right parenthesis.

  It is important to note that .ERR takes no action at all unless the parser's true-false indicator has a value of false. This makes it easy to place .ERR in-line with the expected syntactic elements (e.g. "END" .ERR("ERROR---MISSING END STATEMENT,.SCANTO(";"))).

- .MESSAGE - This element is similar to .ERR, except that its action is unconditional and it allows no scanning options. It sets the parser's true-false indicator to false.

2.4 A Simple Example

To aid the reader to become familiar with this method of combined syntactic/semantic specification, a simple arithmetic assignment language is fully defined in this section, and the definition is applied to a brief program in the language. It is perhaps easiest to consider the program first, since it provides an insight into the nature of the language to be defined. The program reads the diameter of a sphere and calculates and writes the sphere's volume by a sufficiently roundabout method to demonstrate the language concept.

```
PI = 3.14159 $
READ DIAMETER $
RADIUS = DIAMETER / 2 $
VOLUME = 4 / 3 * PI * RADIUS * RADIUS * RADIUS $
WRITE VOLUME $
END $
```

For the sake of simplicity, no branching or conditional statements are included. Since there is only one program flow in which all statements are executed in sequence, the "END" statement serves double duty as a "STOP" statement.

This section will illustrate the translation of this language into instructions for an artificial assembler-language-like "pseudomachine", and into PL/I.

Consider a simple machine capable of performing functions necessitated by this language. The machine has been designed for conceptual simplicity. It does not actually exist in the simple (but inefficient) form given here, and it will therefore be referred to as a pseudomachine. This is not to suggest that such a machine could not be built; it is not difficult to build a software machine (i.e., an emulator) with the characteristics specified here. The 11 operations of this simple machine are in fact a subset of the operations of the PLANS pseudomachine, for which we have built such an emulator.

The most basic characteristic of the pseudomachine is that, in addition to ordinary memory for randomly accessible storage of variable values, it has a push-down stack (last-in-first-out queue) which serves as the basic storage medium for its central processor. All data operations, including in particular all memory access and update operations, are done through the stack, which replaces all the registers (except the instruction address register) of a typical simple computer.

The pseudomachine is a single-address machine that is programmed in a language very similar to ordinary assembler languages. For purposes of language definition, it is assumed that this "symbolic" language is the actual language of the machine, with no translation step involved. This is purely a simplifying assumption, however, with no implications for the use of the pseudomachine as a semantic definition tool only.

The 11 operations which can be performed by this simple pseudomachine are defined in Fig. 4.

Consider now the sequence of pseudomachine commands,

```
LDA  RADIUS
LD   DIAMETER
LDL  2
DIV
STO
```

```
LDL   OPERAND                          (LOAD LITERAL)
      PUSH THE LITERAL OPERAND ONTO THE STACK.
      E.G.,  LDL PAYLOADS   RESULTS IN THE TRANSFORMATION:
               XXXXXX             PAYLOADS
               YYYYYY     -->     XXXXXX
                                  YYYYYY

LDA   OPERAND                          (LOAD ADDRESS)
      PUSH THE ADDRESS CONTAINED IN THE OPERAND FIELD (I.E., THE
      ADDRESS CORRESPONDING TO THE VARIABLE NAME IN THE OPERAND)
      ONTO THE STACK.
      E.G., IF THE CORE ADDRESS 01430 CORRESPONDS TO THE VARIABLE
      NAME 'XVAR', THEN   LDA XVAR   RESULTS IN THE TRANSFORMATION:
               XXXXXX             01430
               YYYYYY     -->     XXXXXX
                                  YYYYYY

LD    OPERAND                          (LOAD)
      PUSH THE CONTENT OF THE ADDRESS (I.E., VARIABLE NAME) IN THE
      OPERAND FIELD ONTO THE STACK.
      E.G., IF THE CONTENT OF CORE ADDRESS 01430, REFERENCED ABOVE,
      IS -316.25, THEN   LD XVAR   RESULTS IN THE TRANSFORMATION:
               XXXXXX             -316.25
               YYYYYY     -->     XXXXXX
                                  YYYYYY

STO                                    (STORE)
      STORE THE CONTENT OF POSITION 1 AT THE ADDRESS CONTAINED IN
      POSITION 2. POP 2 POSITIONS.
      E.G.,   STO   RESULTS IN THE TRANSFORMATION:
               27.2               XXXXXX
               01430      -->     YYYYYY
               XXXXXX
               YYYYYY
      WHILE REPLACING THE PREVIOUS VALUE OF XVAR, -316.25, BY 27.2.

IN                                     (INPUT)
      READ A NUMERICAL VALUE IN STANDARD EXTERNAL FORMAT FROM A
      PUNCHED CARD. PLACE THE VALUE IN THE ADDRESS IN POSITION 1.
      POP 1 POSITION.
      E.G., IF VARIABLE 'X' IS LOCATED AT ADDRESS 130 AND IF THE
      NEXT PUNCHED CARD TO BE READ CONTAINS THE STRING 22.7,
      THEN THE STATEMENT   IN   RESULTS IN THE STACK TRANSFORMATION:
               130                XXXXXX
               XXXXXX     -->     YYYYYY
               YYYYYY
      WHILE REPLACING THE CURRENT VALUE OF X BY THE NEW VALUE 22.7.

OUT                                    (OUTPUT)
      PRINT THE NUMERICAL VALUE CONTAINED AT THE ADDRESS IN POSITION 1.
      POP 1 POSITION.
      E.G., IF THE VARIABLE 'X' IS LOCATED AT ADDRESS 130 AND HAS THE
      CURRENT VALUE 22.7, THEN THE STATEMENT   OUT   RESULTS IN THE
      STACK TRANSFORMATION:
               130                XXXXXX
               XXXXXX     -->     YYYYYY
               YYYYYY
      WHILE PRINTING THE NUMERICAL VALUE 22.7.

ADD                                    (ADD)
      ADD THE CONTENTS OF POSITIONS 1 AND 2, REPLACE THE CONTENT
      OF POSITION 2 BY THE RESULT, POP 1 POSITION.
      E.G.,   ADD   RESULTS IN THE TRANSFORMATION:
               23                 29
               6          -->     XXXXXX
               XXXXXX             YYYYYY
               YYYYYY

SUB                                    (SUBTRACT)
      SUBTRACT THE CONTENT OF POSITION 1 FROM THAT OF POSITION 2,
      REPLACE THE CONTENT OF POSITION 2 BY THE RESULT, POP 1 POSITION.
      E.G.,   SUB   RESULTS IN THE TRANSFORMATION:
               23                 -17
               6          -->     XXXXXX
               XXXXXX             YYYYYY
               YYYYYY

MULT                                   (MULTIPLY)
      MULTIPLY THE CONTENTS OF POSITIONS 1 AND 2, REPLACE THE CONTENT
      OF POSITION 2 BY THE RESULT, POP 1 POSITION.
      E.G.,   MULT   RESULTS IN THE TRANSFORMATION:
               12                 36
               3          -->     XXXXXX
               XXXXXX             YYYYYY
               YYYYYY

DIV                                    (DIVIDE)
      DIVIDE THE CONTENT OF POSITION 1 INTO THE CONTENT OF POSITION 2,
      REPLACE THE CONTENT OF POSITION 2 BY THE RESULT, POP 1 POSITION.
      E.G.,   DIV   RESULTS IN THE TRANSFORMATION:
               12                 .25
               3          -->     XXXXXX
               XXXXXX             YYYYYY
               YYYYYY

STOP                                   (STOP)
      STOP.
```


Fig. 4 Operations of a Simple Pseudomachine which will shortly be seen to correspond to the statement

RADIUS = DIAMETER / 2;

Assuming that the variable names 'RADIUS' and 'DIAMETER' correspond to memory locations 1012 and 1014, respectively, consider the effect of executing these pseudomachine commands.

The statement

LDA RADIUS

pushes the address 1012 onto the (assumed empty) stack, leaving the stack in the state

1012

The statement

LD   DIAMETER

pushes the value of diameter (say, 7) onto the stack, with the result

7
1012.

The statement

LDL 2

pushes the value 2 onto the stack, yielding

2
7
1012.

The statement

DIV

divides 7 by 2, throws away those numbers (i.e., "pops" them off the stack) and places the quotient on the stack, with the result

3.5
1012.

The statement

STO

places the value 3.5 in memory location 1012 in place of the previous value of the variable 'RADIUS'.

Now that a pseudomachine is fully defined, the final step in language definition is the specification of the correspondence between the source language and the speudomachine commands. For the language in question, the language definition of Fig. 5 is appropriate.

Using this augmented grammar, let us now consider the parsing and translation of the statement

RADIUS = DIAMETER / 2;

Fig. 6 shows a parse tree for this statement. In addition, the object statements (the translator output) are shown encircled. Just as the input stream was parsed left to right, the output statements were generated left to right, with one exception. In order to translate from infix binary operator notation in the source language to suffix notation in the object language, it was necessary to save (in the grammar, .SAV) the binary operator "DIV" until after the output "LDL 2" had been generated, and then to place the saved operator in the output stream (in the grammar, .OUT(#)). Since this was accomplished with two instructions that were evaluated in parse order, this isn't really an exception after all.

```
.AUG_GRAM  ARITH

ARITH  :=
     $( STATEMENT  ";" )  "END"  .OUT(STOP)  ";" ;

STATEMENT  :=
     .PEEK("END")  .NEG  .RETURN
   | "READ"  .ID  .OUT(LDA,*/IN)
   | "WRITE"  .ID  .OUT(LDA,*/OUT)
   | .ID  .OUT(LDA,*)  "="  EXPRESSION  .OUT(STO) ;

EXPRESSION  :=
     TERM  $( ADD_OP  TERM  .OUT(#) ) ;

TERM  :=
     FACTOR  $( MULT_OP  FACTOR  .OUT(#) ) ;

FACTOR  :=
     .NUM  .OUT(LDL,*)
   | .ID  .OUT(LD,*)
   | "("  EXPRESSION  ")" ;

ADD_OP  :=
     "+"  .SAV(ADD)
   | "-"  .SAV(SUB) ;

MULT_OP  :=
     "*"  .SAV(MULT)
   | "/"  .SAV(DIV) ;

.END
```

Fig. 5 Complete Definition of a Simple Language

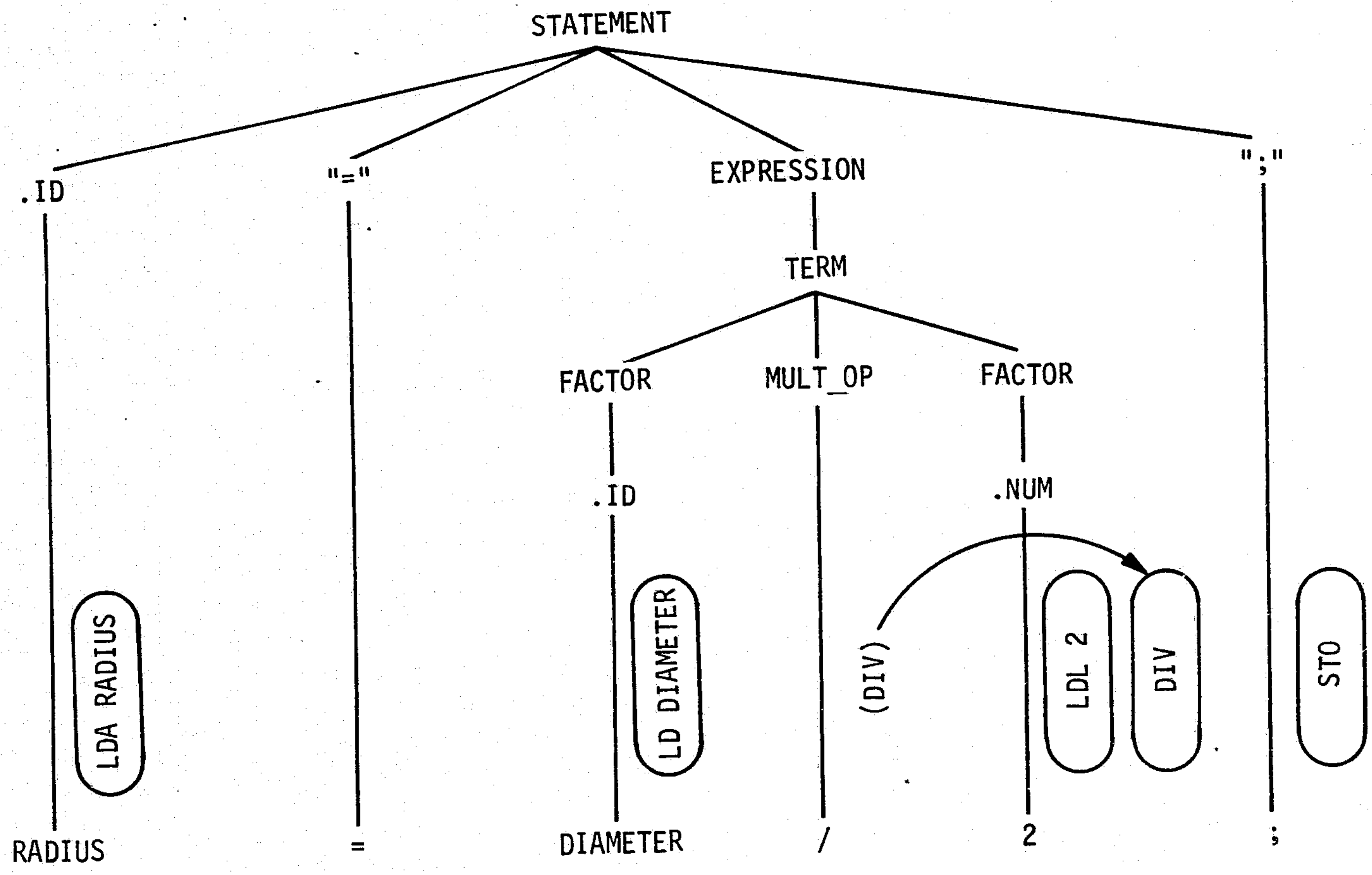


Fig. 6 *Translation Diagram for the Statement* "RADIUS = DIAMETER / 2 ;".

The reader should verify that the given language definition translates our original sample program

```
PI = 3.14159 ;
READ DIAMETER ;
RADIUS = DIAMETER / 2 ;
VOLUME = 4 / 3 * PI * RADIUS * RADIUS * RADIUS ;
WRITE VOLUME ;
END ;
```

into the pseudomachine program,

```
LDA  PI
LDL  3.14159
STO
LDA  DIAMETER
IN
LDA  RADIUS
LD   DIAMETER
LDL  2
DIV
STO
LDA  VOLUME
LDL  4
LDL  3
DIV
LD   PI
MULT
LD   RADIUS
MULT
LD   RADIUS
MULT
LD   RADIUS
MULT
STO
LDA  VOLUME
OUT
STOP
```

and that the latter correctly expresses the original program's function.

Now let us consider the translation of the original program into another language of similar level, say PL/I. Because the source language is already in a PL/I-like syntax, this translation is almost trivial. As an exercise, though, it should provide additional insight. The augmented grammar is shown in Fig. 7.

```
.AUG_GRAM  ARITH

ARITH  :=
      .OUT("DUMMY: PROCEDURE OPTIONS (MAIN);")
      $( STATEMENT  ";"  .OUT(";") )
      "END"  ";"
      .OUT("RETURN;", "END;")  ;

STATEMENT  :=
      .PEEK("END")  .NEG  .RETURN
   |  "READ"  .ID
      .OUT("GET EDIT (", *, ") (COL(1),E(20,0));")
   |  "WRITE"  .ID
      .OUT("PUT SKIP EDIT (", *, ") (E(15,8));")
   |  .ID  .OUT(*)
      "="  .OUT(*)
      EXPRESSION  ;

EXPRESSION  :=
      TERM  $( ADD_OP  TERM )  ;

TERM  :=
      FACTOR  $( MULT_OP  FACTOR )  ;

FACTOR  :=
      .NUM  .OUT(*)
   |  .ID  .OUT(*)
   |  "("  .OUT(*)
      EXPRESSION
      ")"  .OUT(*)  ;

ADD_OP  :=
      "+"  .OUT(*)
   |  "-"  .OUT(*)  ;

MULT_OP  :=
      "*"  .OUT(*)
   |  "/"  .OUT(*)  ;

.END
```

Fig. 7 Augmented Grammar for ARITH-to-PL/I Translation

References (Appendix)

1. Aho, A.V., & Ullman, J.D., The Theory of Parsing, Translation, and Compiling. Vol. 1: Parsing. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

2. Gries, D., Compiler Construction for Digital Computers. New York: Wiley, 1971.

3. Naur, P. (Ed.) Report on the Algorithmic Language ALGOLGO. Communications of the ACM, 1960, 3, 299-314.

3.0 COMPONENTS OF THE GENERATED TRANSLATOR

3.1 Lexical Analyzer

In addition to the automatically generated translator and several standard support routines and declarations which are simply included, as is, in the PL/I translator to be compiled, several routines and declarations require user intervention, as shown in Figure 1. The first of these is the lexical analyzer, which recognizes the basic symbols and symbol classes (or tokens) of the source language.

The lexical analyzer consists of a subroutine, usually not requiring modification, and a set of declarations which constitute a table by which the subroutine is controlled. This section discusses the procedure whereby such a table is constructed. It will use, as an example, an early TWS lexical analyzer.

The first step in the process is the verbal definition of the tokens of the language. For our example, these definitions are as follows.

.ID - Identifier. One to 31 characters, all alphanumeric or underbar (_), first character strictly alphabetic.

.STRING - Character string literal. Logically unlimited in length, first and last characters double quotation marks ("). The contained characters are unrestricted, except that no quotation marks may occur except in adjacent pairs.

.COMMENT - Comment. Unlimited in length, first, second characters must be a slash asterisk (/*) and the comment must be ended by an asterisk slash (*/).

.SNIR - Symbolic name for translator generated symbols. Identifier followed by an asterisk (*).

.POINTSTR- Point string. Period (.) followed by an identifier.

.NUM - Number. Includes all integer digit strings only.

In addition to the catagories mentioned, the following are single and double character tokens:

:=

;

()

$

|

#

,

/

*

**

The next step in the process is to convert the definition to a state transition diagram which defines the character-by-character scanning process to be performed by the lexical analyzer. The state transition diagram will be converted to a matrix, and it is certainly possible to skip the diagram and go directly to the matrix. However, fewer errors appear to result if the recommended approach is used.

Fig. 8 shows the state transition diagram for our example. The lexical analyzer always starts in state zero (q_o). One state trans-

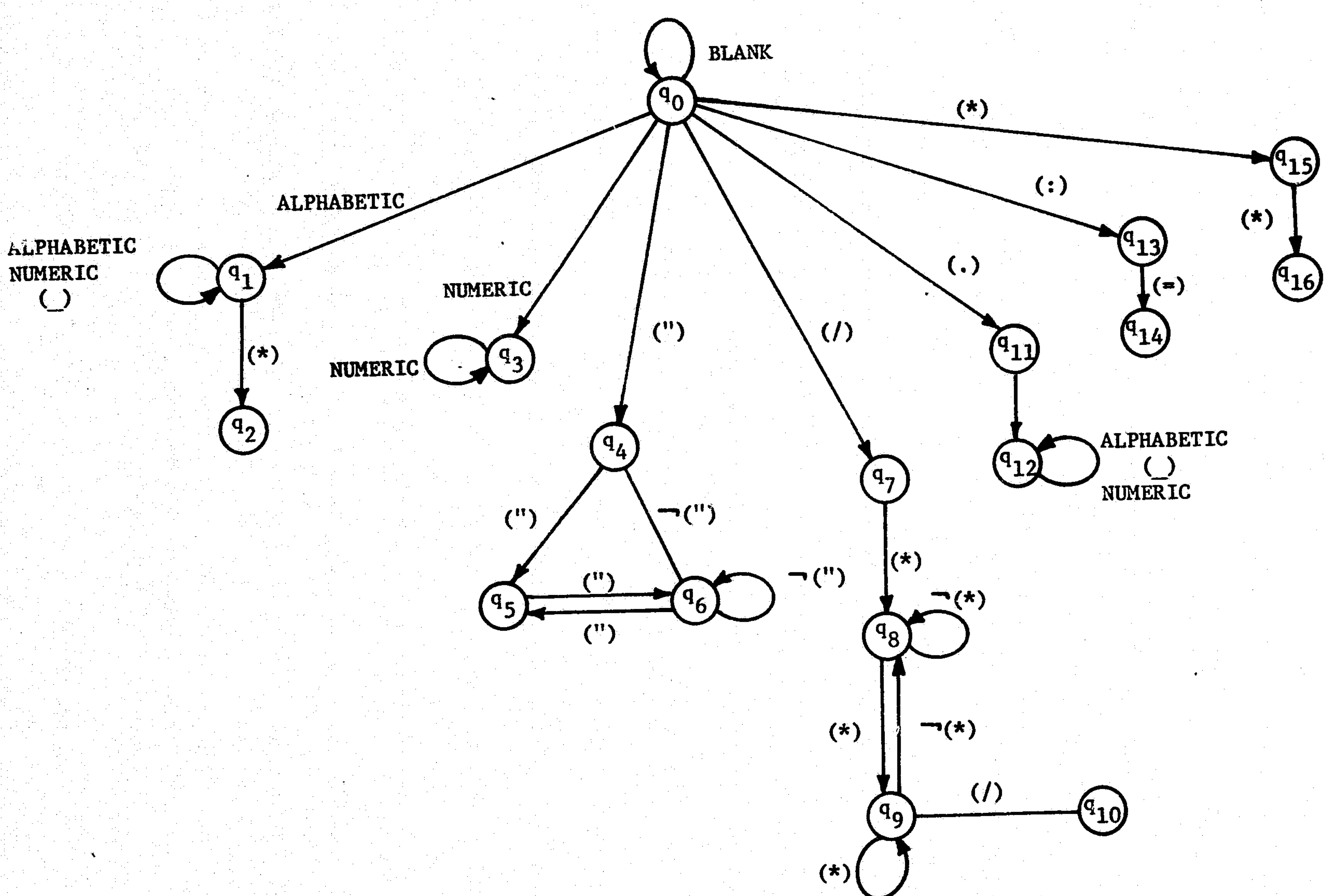


Fig 8 State Transition Diagram for Sample Lexical Analyzer

ition is made for each character removed from the input string. Whenever a purely final state is reached, a complete token has been scanned. For example, q_2 is a final state since no state transitions from q_2 are indicated. When the current state is an intermediate state, another character must be scanned. If that character causes a legal state transition, the process continues. If not, then either a token has been processed (if the current state is also a possible final state) or an error has occurred.

Figure 9 shows the state transition matrix which corresponds to Figure 8. To this table has been added information about state types and information identifying the special terminal symbol classes (ID, etc.). Expressed in terms of the matrix, the scanning algorithm is as follows. Starting at state zero or row zero in the table, find the current character. From row zero and the column corresponding to the current character the entry defines the next row. This process is applied until no transition is possible to another row. If the state type corresponding to the current row is final a complete token has been parsed. An error has occurred if no transition is possible and the corresponding state is start (S) or intermediate (I).

1) EXAMPLE: ABC =

Starting at state zero with the letter A and the row, column value is 1. Now with letter B, using the new row 1 the column corresponding to B contains a 1. Repeating the process with letter C, using row 1, the corresponding column also points to 1. Finally,

STATE \ CHAR		.	(	|	$	*	)	;	/	,	_	:	#	=	"	(b)	A-Z	0-9	STATE TYPE	TOKEN TYPE
0		11	15	15	15	14	15	15	7	15		13	15	15	4	0	1	3	S	
1						2					1						1	1	IF	ID
2																			F	SNR
3																		3	IF	NUM
4		6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6	6	I	
5															6				IF	STR
6		6	6	6	6	6	6	6	6	6	6	6	6	6	5	6	6	6	I	
7						8													IF	
8		8	8	8	8	9	8	8	8	8	8	8	8	8	8	8	8	8	I	
9		8	8	8	8	9	8	8	10	8	8	8	8	8	8	8	8	8	I	
10																			F	CMT
11																	12		I	
12											12						12	12	IF	
13														15					I	
14						15													F	
15																			F	

STATE TYPES

S - START

I - INTERMEDIATE

F - FINAL

Fig. 9 State Transition Matrix for Sample Lexical Analyzer when the character (=) is looked up in row 1, no transition is possible.Since row 1 is a final state, the parse is complete. The token ABC is an identifier since the final state was state/row 1.

2) EXAMPLE: /*A*/

Starting at state zero with a slash (/) the row, column value is 7, with the next character an asterisk (*). Using row 7, the corresponding column value is 8. The next character (A), using row 8, gives a value 8. Continuing with an asterisk (*) and row 8 gives a value 9. Repeating the process with a slash (/) using row 9 the corresponding value is a 10. Finally, when the character blank is looked up in row 10 no transition is possible. Row 10 is a final state, so the parse is complete, and the token /*A*/ is a comment since the final state was state/row 10.

3.2 Output Routines

The augmented grammar translator output package for PL/I object code consists of two parts. The first part is the direct output routine, @OUT; the second part consists of the routines using the code stack.

The direct output routine is called when a reference is made to the augmented grammar element .OUT. The direct output routine @OUT has a single argument, which contains the next portion of output to be written. The routine @OUT saves the code in a code stack until a complete line of code is generated. A line of code is output depending on the following conditions:

1) the last character in the current parameter argument contains a semicolon (;). This causes the current code saved in the code buffer to be output, starting in column 5.

2) the last character in the current parameter argument contains a cent sign (¢). The cent sign character is stripped off, and the current code saved in the code buffer is output starting in column 5.

3) the last character in the current parameter argument contains a question mark (?). The question mark is stripped off and the remainder is added to the code buffer.

4) the last character in the current parameter argument contains a colon (:). This causes the current code saved in the code buffer to be output, starting in column 2. (Labels and continuation lines start in column 2; all other lines start in column 5.)

5) with any other character as the last character, the parameter argument is added to the code buffer to be output on a subsequent call to @OUT.

The second group of routines allows the user to save code on a stack and output the code at a later time. @OUT may be used in the meantime to output code ahead of the code saved by these routines. These routines are written as a last-in-first-out stack and are referenced by using the elements .SAV and .CAT in the grammar. The routines are:

@SAV(arg) - Calls are generated by a reference to element .SAV; the argument is added to the top of the stack as a separate item.

@CAT(arg) - Calls are generated by a reference to element .CAT; the argument is concatenated to the top item of the stack.

Items are removed from the stack by references in the grammar to .OUT(#). This generates a call to @CODE_STK_OUT which in turn references @POP_CODE_STACK finally calling @OUT to generate the proper output files.

Ordinarily, no modification of these routines is required for PL/I object code generation. However, certain language properties may require their revision. For example, in the translation of PLANS to PL/I it was necessary to be able to output statements ahead of the statement which has already been partially output. This was accomplished by modifying @OUT so that it contained a stack of partially completed lines of output. Provision was made for adding characters to the current incomplete line, outputting the current line, pushing a new line onto the stack, and popping lines off.

3.3 ERROR MESSAGING

For simple, nonproduction languages, the user may simply wish to avail himself of the default error messaging capability of the TWS. Anytime a required syntactic element is missing in a source program and no .ERR specification immediately follows that element in the augmented grammar, a system error message will result.

If the user desires to write his own specialized error messages, he may do so in either of two ways. The error message may be written as part of the .ERR specification, as .ERR("S:MISSING ARITHMETIC OPERATOR"). Alternatively, the user may provide a declaration of the form

```
DECLARE @ERROR_MESSAGE (5) CHAR(60) VARYING STATIC INIT(
'N:THIS IS A NOTE',
'W:THIS IS A WARNING',
'S:THIS IS A SEVERE ERROR',
'F:THIS IS A FATAL ERROR',
'S:MISSING ARITHMETIC OPERATOR');
```

With this declaration, the augmented grammar element .ERR(5) would result in the error message "MISSING ARITHMETIC OPERATOR" any time the syntactic element preceding the .ERR specification is not found in the source program.

The example above also illustrates all four error message severity levels. The first character of an error message is assumed to indicate the severity of the error. The symbols used are "N", "W", "S", and "F", for note, warning, severe error, and fatal error, respectively. The error message routine automatically sets an appropriate system condition code for the most severe error encountered. In addition, a fatal error immediately halts the translator, while a severe error terminates code generation, but allows continued parsing to detect any other errors.