

MCR-74-314
NAS9-13616

Phase 1
Final
Report

NASA CR-
140282
September 1974

Volume II

Use of the Basic
Language and
Module Library

**Scheduling
Language and
Algorithm
Development Study**

(NASA-CR-140282) SCHEDULING LANGUAGE AND
ALGORITHM DEVELOPMENT STUDY. VOLUME 2:
USE OF THE BASIC LANGUAGE AND MODULE
LIBRARY Final Report (Martin Marietta
Corp.) 193 p HC \$12.75

N74-33705

Unclas
50474

CSCL 09B G3/08

227

PRICES SUBJECT TO CHANGE

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151

MARTIN MARIETTA

MCR-74-314
NAS9-13616

Phase 1
Final
Report

Volume II

September 1974

Use of the Basic
Language and
Module Library

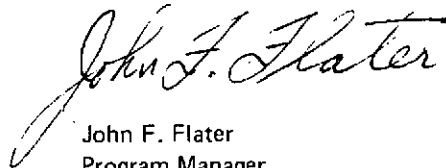
SCHEDULING LANGUAGE
AND ALGORITHM
DEVELOPMENT STUDY

*DRL T-890
Line Item 3*

Prepared by:

R. A. Chamberlain
D. E. Cornick
J. F. Flater
R. J. O'Doherty
F. M. Peterson
H. R. Ramsey
J. K. Willoughby

Approved


John F. Flater
Program Manager

MARTIN MARIETTA CORPORATION
P.O. Box 179
Denver, Colorado 80201

FOREWORD

This is the Phase 1 Final Report of the Scheduling Language and Algorithm Development Study performed by Martin Marietta Corporation, Denver Division, under Contract NAS9-13616. The purpose of this study was to conceive and specify a high-level computer programming language and a program library to be used in writing programs for scheduling complex systems such as the Space Transportation System. This report is presented in three volumes plus an appendix:

Volume I - Study Summary and Overview

Volume II - Use of the Basic Language and Module Library

Volume III - Detailed Functional Specification for the Basic
Language and the Module Library

Appendix - Study Approach and Activity Summary

Volume I summarizes the objectives and requirements of the study and discusses the "why" behind the objectives and requirements. Unique results achieved during the study or unique features of the specified language and program library are then described and related to the "why" of the objectives and requirements. Finally, a description of the significance of study results, in terms of expected benefits, is provided.

Volume II summarizes the capabilities of the specified scheduling language and the program module library. It is written with the potential user in mind and, therefore, provides maximum insight on how the capabilities will be helpful in writing scheduling

programs. Simple examples and illustrations are provided in Volume II to assist the potential user in applying the capabilities of his problem.

The detailed functional specifications presented in Volume III are the formal product of Phase 1. These specifications are written as requirements for software implementation of the language and the program modules, and are aimed at a specific audience.

A separate Appendix summarizes the analyses, describes the approach used to identify and specify the capabilities required in the basic language, and presents results of the algorithm and problem modeling analyses used to define specifications for the scheduling module library. The appendix is directed toward the reader who is interested in how the study conclusions and results were reached.

CONTENTS

		<u>Page</u>
1.0	INTRODUCTION	1
2.0	USE OF PLANS LANGUAGE	3
2.1	Basic Language Structure and Statements	3
2.2	Labeled Trees	8
2.3	PLANS Tree Access Mechanisms	13
2.4	PLANS Tree Update Mechanisms	24
2.5	Special Statements	46
2.6	A Simple Example	51
3.0	SYSTEMS OPERATIONS AND PROBLEM MODELING	61
3.1	A Generic Scheduling Operations Model	61
3.2	Problem Types Accommodated by the Operations Model	69
3.3	PLANS Library Modules	71
3.4	Problem Description Using the Operations Model	79
3.4.1	Simple Predecessor Networks	81
3.4.2	Networks with Job Duration	85
3.4.3	Simple Resource Pools	87
3.4.4	Events in Job Networks	91
3.4.5	Job Splittability	95
3.4.6	Variable Resource Requirements	97
3.4.7	Item-Specific Resources	101
3.4.8	Resources Alternatives	105
3.4.9	Process Alternatives	107
3.4.10	General Temporal Relations	111
3.4.11	Item-Specific Resources with Explicit Descriptors	115
3.4.12	Pooled Resources with Explicit Descriptors	119
3.5	Operations Model Data Structure	123
3.5.1	\$RESOURCE	127
3.5.2	\$PROCESS	131
3.5.3	\$OPSEQ	137
3.5.4	\$OBJECTIVES	143
3.5.5	\$JOBSET	147
3.5.6	\$SCHEDULE	149
3.5.7	ASSIGNMENT Subnode of \$RESOURCE	151
3.6	Heuristic and Mathematical Programming Solution Techniques	155
4.0	ILLUSTRATIVE EXAMPLES	163
4.1	Ordering of a Precedence Network	163
4.2	Payload Grouping	167
4.3	Project Scheduling	173
4.4	Flight Assignment	182
		thru 186

Figure

2.1-1	PLANS (and PL/I) Hierarchic Block Program Structure . . .	4
2.2-1	A Labeled Tree	10
2.2-2	The Tree of Fig. 2.2-1 in Textual Format	11
2.3-1	Basic Tree Access Mechanisms	15
2.3-2	Conditional Access Using Qualifier .FIRST	19
2.3-3	Use of Label Qualifier ".ALL"	20
2.3-4	Conditional Access Using Qualifier ".ALL"	22
2.3-5	Indirect Referencing	24
2.4-1	Results of a Sequence of Tree Assignment Statements . . .	26
2.4-2	Type Conversion in Tree Assignment Statements	29
2.4-3	Label Assignment Statements	32
2.4-4	Treatment of Base-Node Labels	33
2.4-5	Value-Substructure Exclusivity	36
2.4-6	Assignments to Nonexistent Nodes	38
2.4-7	GRAFT Statements	40
2.4-8	INSERT Statements	41
2.4-9	GRAFT INSERT Statements	42
2.4-10	PRUNE Statements	45
2.5-1	ORDER Statements	48
2.5-2	Automatic Generation of Combinations	49
2.6-1	ORDER_BY_PREDECESSORS: Iteration 1	53
2.6-2	ORDER_BY_PREDECESSORS: Iteration 2	57
2.6-3	ORDER_BY_PREDECESSORS: Iteration 5	58
3.1-1	Scheduling Operations Model Fundamental Concepts	65
3.1-2	Sample Network Flow Diagram for System Operations Description, Shuttle Mission	66
3.1-3	Operations Model/Solution Algorithm Interface Example with OSARS Annotations	68
3.4-1	Problem Description Using the Operations Model	80
3.4.1-1	A Predecessor Network Example	82
3.4.2-1	Scheduling with a Predecessor Network That Includes Job Durations	86
3.4.3-1	Feasible Project Scheduling Example	87
3.4.3-2	Association of Requirements for Pooled Resources with Network Jobs	88
3.4.4-1	Use of Events to Interface Between Networks	92
3.4.5-1	Illustration of a Splittable Job	96
3.4.6-1	Variable-Level Resource Requirements	98
3.4.6-2	Resolving Usage Conflicts with Piecewise Constant Required Resources	99
3.4.6-3	Typical Pool Level Profile	99
3.4.7-1	Illustration of Item-Specific Resources Required by Jobs	102
3.4.8-1	Illustration of Resource Alternatives	106
3.4.9-1	Conversion from Resource Alternatives to Process Alternatives	107
3.4.9-2	Illustration of Process Alternatives	108
3.4.10-1	Illustration of a General Temporal Relations	112

3.4.11-1	Illustration of an Item-Specific Resource with an Explicit Descriptor	116
3.4.12-1	Partitioning of a Pooled Resource with Explicit Descriptors	120
3.5.1-1	\$RESOURCE Standard Data Structure	128
3.5.1-2	\$RESOURCE Data Tree for a Shuttle Application	130
3.5.2-1	\$PROCESS Standard Data Structure	132
3.5.2-2	Use of \$PROCESS ⁷ for a Shuttle Application	136
3.5.3-1	\$OPSEQ Standard Data Structure	138
3.5.3-2	General Substructure for TEMPORAL_RELATIONS	139
3.5.3-3	Use of \$OPSEQ for a Shuttle Application	141
3.5.4-1	\$OBJECTIVES Standard Data Structure	144
3.5.5-1	\$JOBSET Standard Data Structure	148
3.5.6-1	\$SCHEDULE Standard Data Structure	150
3.6-1	Project Schedule System within the PLANS Module Library	157
3.6-2	Problem Models for PLANS Project Scheduling Modules	158
3.6-3	Example of Nonconstant Resource Demand Profile	160
3.6-4	Problem Characteristics Amenable to Mathematical Programming	162
4.1-1	Data Structures Illustrating ORDER_BY_PREDECESSORS	164
4.1-2	PLANS Subroutine for Ordering Jobs by Predecessors	165
4.3-1	Network Design for Project Scheduling	174
4.3-2	Flow Logic for Time Transcendent Project Scheduling Algorithm	180
4.3-3	PLANS Code for Time Transcendent Project Scheduling Algorithm	181
4.4-1	Examples of PLANS Code for FLIGHT ASSIGNMENT Algorithm	184

Table

3.2-1	Use of PLANS Generic Operations Model for Describing Typical Problem Classes	70
3.3-1	Summary of PLANS Module Library Contents	74
3.4.1-1	Basic Information of a Predecessor Network	81
3.4.1-2	Ordering Network Jobs by Predecessors	84
3.4.7-1	Examples of Pooled and Item-Specific Resources	103
3.5-1	Operations Model Data Structures	124

1.0 Introduction

1.0 INTRODUCTION

Volume II presents both PLANS (Programming Language for Allocation and Network Scheduling) and the programming library modules from the user's viewpoint. It describes capabilities and gives many specific examples to provide insight to the potential usefulness of the language for building scheduling and resource allocation software.

This volume is intended to provide a basic understanding of the nature and use of PLANS. It is not a detailed user guide such as might be used by a programmer for actual coding. While this document is meant to serve that function on an interim basis, a detailed user guide will be produced as a part of the PLANS implementation (Phase II) effort. The discussion in this volume is not meant to serve as a functional specification of PLANS. The detailed functional specifications form a part of Volume III of this report.

Section 2.0 of this volume describes the characteristics of the PLANS language itself. Section 3.0 discusses the characteristics of scheduling problems and how they can be modeled. The framework described is referred to here as a generic scheduling operations model. The application of various library modules including solution algorithms is discussed in the context of this model. Illustrative examples are included in Section 4.0 where both modeling and coding are presented.

2.0 Use of the PLANS Language

2.0 USE OF THE PLANS LANGUAGE

PLANS (Programming Language for Allocation and Network Scheduling) is a computer programming language designed for use in the expression of procedures for solving schedule construction and resource allocation problems. It is a high-level language intended to allow easy, direct expression of the kinds of functions frequently found in scheduling programs and algorithms. The close correspondence between basic functional operations and PLANS statements is intended to allow the scheduling program designer to easily accomplish for himself both initial programming and program modification. A detailed discussion of these considerations, together with the analytical results which caused PLANS to take its current form, can be found in the appendix to this report.

2.1 BASIC LANGUAGE STRUCTURE AND STATEMENTS

For reasons outlined in the appendix, the basic program structure and syntax of PLANS are similar to those of PL/I. A brief description of their basic characteristics is given below. For a more detailed introduction, see *A Guide to PL/I for FORTRAN Users*, IBM Document SC20-1637-3. For detailed general information, see *IBM System/360 Operating System PL/I (F) Language Reference Manual*, IBM Document GC28-8201-4, and the corresponding *Programmer's Guide*, GC28-6594, or other similar manuals.

The basic structure of PLANS is a hierarchic block structure. Groups of statements may be combined into logical blocks called PROCEDURE blocks, BEGIN blocks, and DO-groups. These blocks are preceded by PROCEDURE, BEGIN, or DO statements and followed by END. These blocks may then be nested at will, yielding a hierarchic structure such as that shown in Fig. 2.1-1.

Note in Fig. 2.1-1 that statements are terminated by semicolons and that statement labels are indicated by colons. PLANS, like PL/I, is a free-format language with no card column restrictions for specific types of information. A statement may be several cards long or, conversely, several statements may appear on a card. Indentation has no significance to PLANS and can be used as desired to improve program structure visibility.

```
A: PROCEDURE;  
  statement - a1  
  statement - a2  
  B: BEGIN;  
    statement - b1  
    statement - b2  
    C: PROCEDURE;  
      statement - c1  
      statement - c2  
      D: DO;  
        statement - d1  
        statement - d2  
        E: BEGIN;  
          statement - e1  
          statement - e2  
          END;  
        statement d3  
        END;  
      statement - c3  
      END;  
    statement - b3  
    END;  
  statement - a3  
  END;
```

*Fig. 2.1-1
PLANS (and PL/I) Hierarchic Block
Program Structure*

Procedure blocks correspond to main programs or subroutines. If a procedure appears within another procedure (as PROCEDURE C is within PROCEDURE A, for instance), it is executable only by means of a CALL statement. Thus, if statement-b2 is not a CALL or other transfer-of-control statement, it will be followed logically by statement-b3, with transfer of control skipping around PROCEDURE C. BEGIN blocks and DO-groups, on the other hand, are executed in line and are for many practical purposes equivalent to single statements.

Aside from the fact that block-structured languages simplify program structure and may improve program readability considerably, they also tend to increase the power of the language by providing a natural mechanism for treating a whole block of statements as a logical entity. Thus, by way of simple example, the statements

```
SUM = 0;  
IF K < 10  
  THEN DO J = 1 TO K;  
    SUM = SUM + J;  
  END;
```

sum the first K integers if, and only if, $K < 10$. Otherwise $SUM = 0$.

It is a basic property of block-structured languages that variables have global (rather than local) scope unless specified otherwise. Thus, where a FORTRAN subroutine is written as a separate entity from the calling program and uses variables whose names are meaningful only within the subroutine (i.e., names that

are local), the procedures (subroutines) of a hierarchic language are usually nested within the calling program, and have access to all its variables unless explicitly excluded.

When using a procedure as a subroutine, it is usually desirable to pass a parameter list as part of the procedure call and return. This can be explicitly accomplished in very much the same way as in most languages, using statements of the form

```
CALL EVAL_W(X1, Y1, Z1, W);
```

and

```
EVAL_W: PROCEDURE(X, Y, Z, W);
```

where the names used for the parameters are free to vary between the two statements. Incidentally, note use of the underline () symbol. This, combined with a maximum name length of 31 characters for most purposes, allows one to use meaningful and readable labels, variable names, etc (e.g., THIS_IS_A_READABLE_NAME).

This has been a rather cursory treatment of the PLANS (PL/I) program structure. While this structure is easy to use successfully, its more sophisticated ramifications clearly exceed the scope of this volume. The reader is again referred to the PL/I documents mentioned at the beginning of this section for further details.

PLANS incorporates a fairly complete set of ordinary arithmetic, transfer-of-control, and conditional and iterative statements that are treated in essentially the same way they are used in PL/I. Illustrative examples of these statements follow with brief descriptions but without rigorous definition.

PLANS uses conventional arithmetic assignment statements such as

```
XVAR = (Y*3.0)-((Z**2)/26)*Y;
```

Conventional operator priorities and left-to-right performance rules apply for the most part, so that the statement

```
XVAR = Y * 3.0 - Z**2/26*Y;
```

has the same meaning as the statement above.

Aside from the CALL statement, which has already been discussed, the only transfer-of-control capability required for most purposes is the simple GO TO statement. This statement has the form

```
GO TO A_STATEMENT_LABEL;
```

where A_STATEMENT_LABEL is a statement label, as

```
A_STATEMENT_LABEL: XVAR = Y*3.0-Z**2/26*Y; .
```

Conditional statements are similar to those of PL/I and therefore are a good deal more powerful than those familiar to FORTRAN programmers. The IF...THEN...ELSE... syntax is used (IF...THEN... is legal), and IF statements can be nested. No parentheses are required around the conditional expression (after "IF"). Thus, the following program segment is legal and behaves in a way that should be familiar.

```
IF YIELD SIGN = 1
  THEN IF TRAFFIC COMING = 1
    THEN GO TO STOP;
    ELSE GO TO GO;
  ELSE IF STOP SIGN = 1
    THEN GO TO STOP;
    ELSE GO TO GO; .
```

In addition to the noniterative DO-group shown in Fig. 2.1-1 and

the two special iteration statements that are peculiar to PLANS, PLANS also uses two common forms of iterative DO statements.

The first of these is the usual delimited form

```
DO I = 1 TO 10;
```

or

```
DO I = 2 TO 20 BY 2;
```

or

```
DO I = (K-1)**2-14 TO A*B*C;
```

while the second is an open-ended form

```
DO WHILE (A*X**2<27); .
```

In each of these forms, the condition is tested at the beginning of the DO-group, rather than at the end, so that a DO-group starting

```
DO I = 1 TO 0;
```

or

```
A = 20;  
DO WHILE (A < 10);
```

would not be executed at all. Of course, the use of END to terminate the DO-group obviates the need to refer to a statement label in the DO statement.

2.2 LABELED TREES

The principal difference between PLANS and most other programming languages is that PLANS is oriented primarily toward the manipulation of ordered, labeled tree structures. In preparation for a discussion of the PLANS tree operations, this section will define a labeled tree and establish both graphical and textual conventions for representing such trees.

Figure 2.2-1 illustrates a labeled tree, as well as our graphical format for trees. The tree is, of course, a hierarchical data structure. The branching points are called *nodes*, and the *root node* is shown at the top. The nodes are represented graphically by circles. The *name* of the tree, in this case \$PAYLOAD, is shown above the root node. The dollar sign is a special character used to identify tree names so the PLANS compiler can discriminate them from variable names. Each node has a *label* (the character string to the right of the node), but the label may be *null*. In the figure, the only node shown with a null label is the root node, but any node can conceivably have a null label.

The nodes exactly one level below a given node are called its *descendants*. The root node of the tree in Fig. 2.2-1 has four descendants. The node labeled LIFESCIENCE has two descendants, which, in turn, are labeled WEIGHT and WINDOW. Nodes that have descendants are called *nonterminal* nodes; nodes without descendants are called *terminal* nodes. There are 11 terminal and 9 non-terminal nodes in the figure. In addition to a label, a terminal node has a value, which may be either a *character string* or a *numeric value*. Values are shown below their terminal nodes. Like labels, values may be null.

While the graphical format is convenient for displaying conceptual tree structures and for demonstrating the effect of specific PLANS statements, it is too cumbersome and rigid for convenient use in the display of specific tree structures, especially large ones. For this purpose, the textual format is used.

Fig. 2.2-1

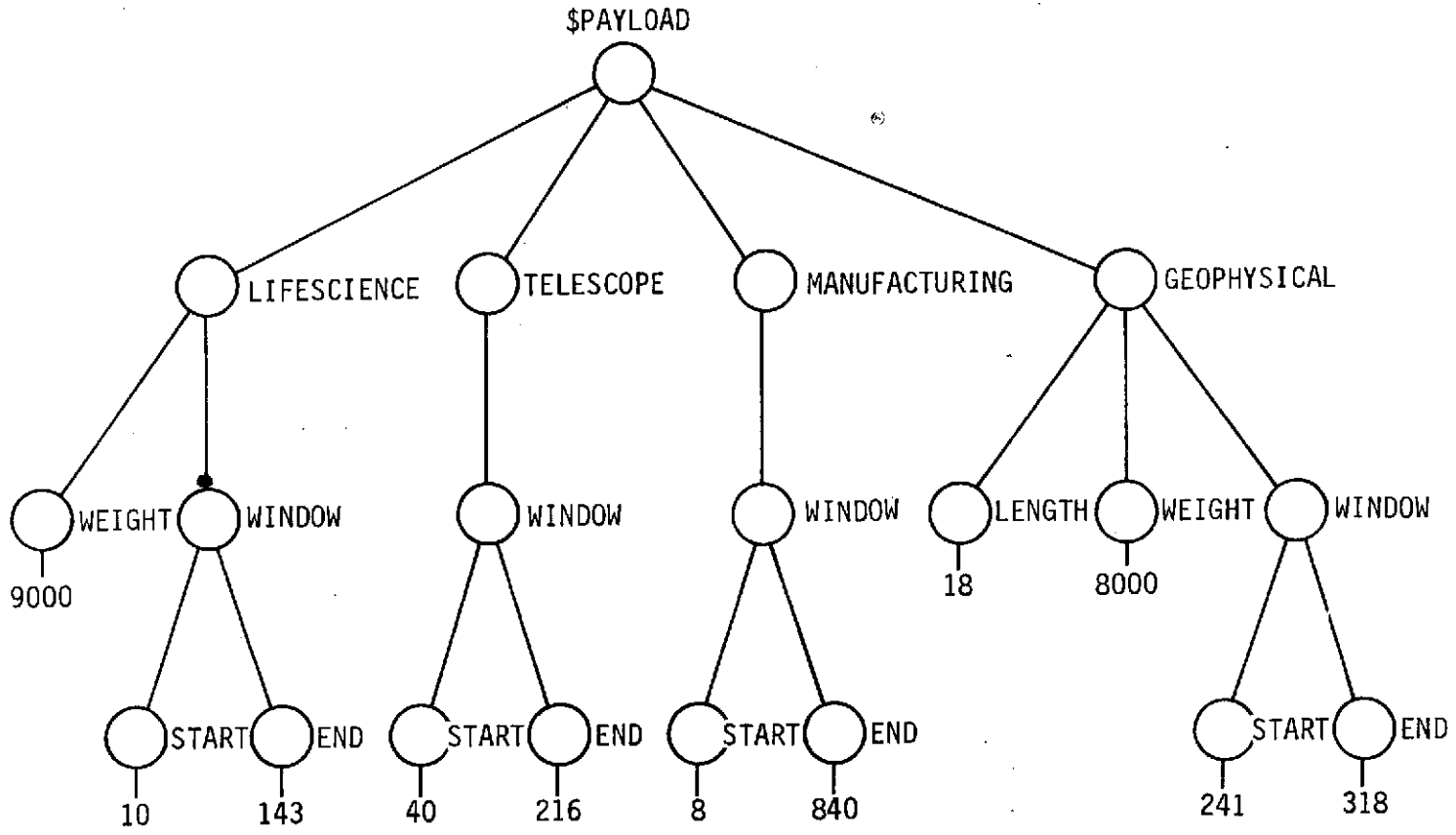


Fig. 2.2-1 A Labeled Tree

The tree of Fig. 2.2-1 is expressed in the form shown in Fig. 2.2-2. In this case, the structure is defined by the indentation pattern, rather than by node-connecting lines. Each line of text represents a node. The information occurring first on a line is the node label, while the values of terminal nodes are separated from the corresponding labels by a hyphen (-) character that is surrounded by blanks. In order to allow rigorous definition of tree structures in which some nodes have null labels, it is necessary to employ a special convention for representing them. Null

```
$PAYLOAD
  LIFESCIENCE
    WEIGHT - 9000
    WINDOW
      START - 10
      END - 143
  TELESCOPE
    WINDOW
      START - 40
      END - 216
  MANUFACTURING
    WINDOW
      START - 8
      END - 840
  GEOPHYSICAL
    LENGTH - 18
    WEIGHT - 8000
    WINDOW
      START - 241
      END - 318
```

Fig. 2.2-2 The Tree of Fig. 2.2-1 in Textual Format

labels are represented by a cent sign (¢). This convention is occasionally employed in the graphical format, although it is unnecessary there.

It can be seen in Fig. 2.2-2, that while numeric values are given for terminal payload descriptor nodes, physical units are not; i.e. 9000 is given for the weight and 10 for the window start time of the lifescience payload, but no indication is given whether these are pounds, kilograms, seconds, etc. The option exists to specify these physical units in the application program logic, in which case the information in the example is adequate. The alternative is to enter physical unit values as character string input data in the tree structure textual format. For readability, a shorthand format has been adopted to present the value of physical units, thus

```
$PAYLOAD
  LIFESCIENCE
    WEIGHT - 9000
          - LBS
    WINDOW
      START - 10
            - HRS
      END - 143
          - HRS .
```

This is a shorthand form in the sense that the numeric and character string data are properly different values for two different null labeled nodes, that is

```
$PAYLOAD
  LIFESCIENCE
    WEIGHT
      ¢ - 9000
      ¢ - LBS .
```

Entry of multiple values for a labeled node, as in this example, would require the inclusion of logic within the application program, to recognize the character string node values and associate them correctly with the numeric node values.

An additional convention, which has been adopted, is the use of parenthesized labels and values to represent variable data in the definition of a particular tree application. If a label or value occurs without parentheses, it is assumed that the character string shown is literally present in the tree. For example, the tree

```
$PAYLOAD
  LIFESCIENCE
    WEIGHT - 9000
    LENGTH - 27
```

contains only actual values and labels. But if one wanted to show only the nature of the information contained in this tree, the following form might be used.

```
$PAYLOAD
  (PAYLOAD NAME)
    (CHARACTERISTIC) - (VALUE)
    (CHARACTERISTIC) - (VALUE)
    .
    .
  (PAYLOAD NAME)
    .
    .
```

2.3 PLANS TREE ACCESS MECHANISMS

PLANS provides the programmer with a number of simple and powerful means of accessing and updating the information contained in the labeled tree structures discussed in the previous

section. These methods are based on the notion that the programmer can "point" to a particular tree node by specifying which tree it is in, which descendant of the root node it is under, which descendant of *that* node it is under, etc. (Remember that the term "descendant," as used here, means *immediate* descendant.)

Suppose, for example that information about the telescope payload in Fig. 2.2-1 is desired. Because the name of the tree is \$PAYLOAD and the name of the payload in question is TELESCOPE, the programmer might write \$PAYLOAD.TELESCOPE to access this information. This is an example of qualification by label.

Alternatively, qualification can be done by position, using the familiar subscript notation. As was mentioned before, but not explained in detail, PLANS trees are *ordered* trees; that is, the ordering of the descendants of a node is significant. Unless action is taken to change or reorder a tree structure, the order remains constant. In the present example, since TELESCOPE is the second payload, information about that payload can be referred to as \$PAYLOAD(2), as well as \$PAYLOAD.TELESCOPE. Examples of simple qualification by label and by subscript are shown in Fig. 2.3-1.

Figure 2.3-1 also illustrates the use of one of two keywords, LAST and NEXT, which have special meaning when used as subscripts in PLANS tree access and update statements. LAST allows the programmer to refer to the last information appended below a given node without knowing either the label or the index of the node he wants. Thus, for example \$PAYLOAD(LAST) refers to the geophysical payload in the figure. Similarly, NEXT refers to the next subnode

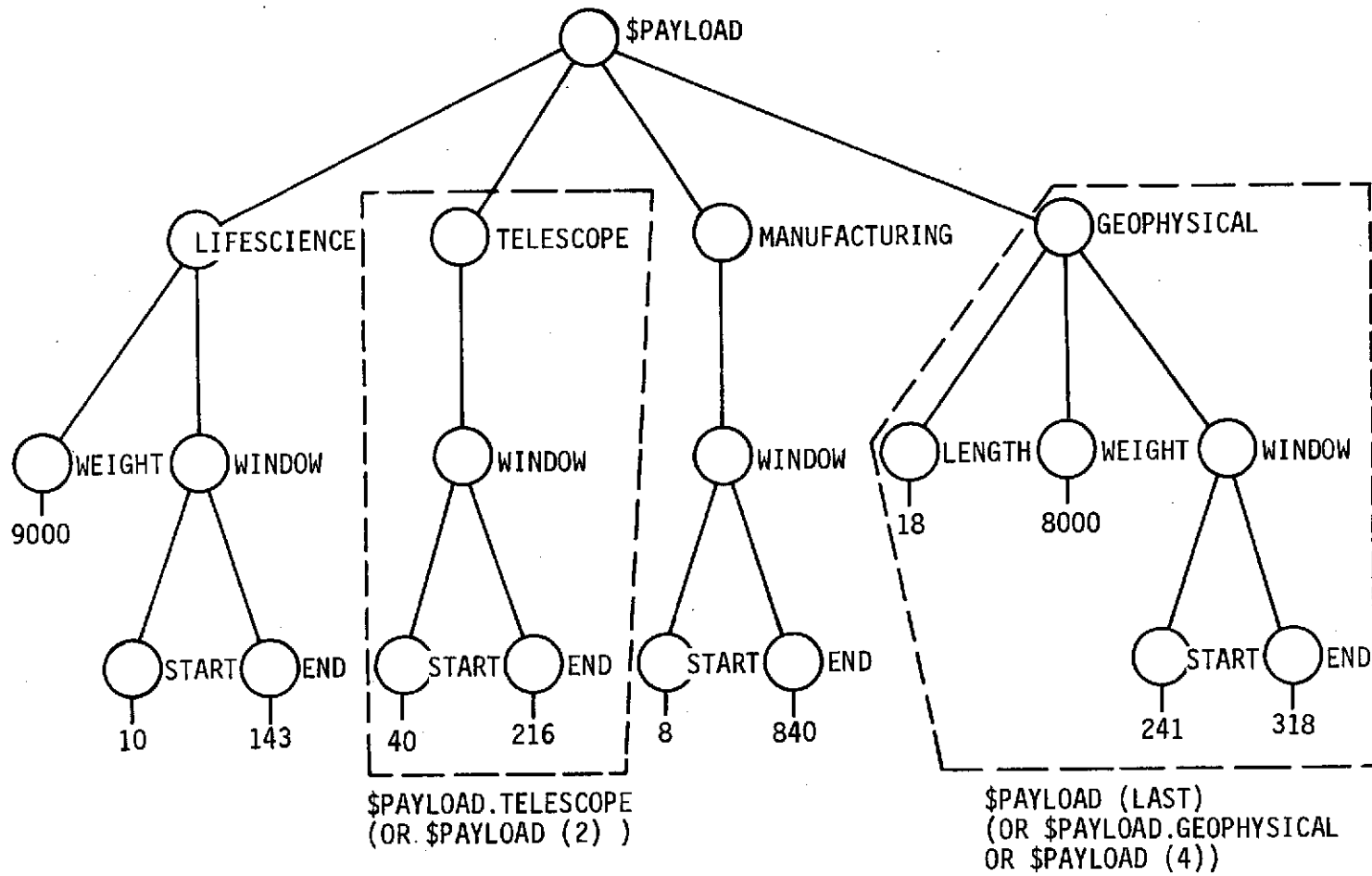


Fig. 2.3-1 Basic Tree Access Mechanisms

after the last one appended below a given node. The reference is therefore to a node that does not yet exist. NEXT is meaningful only in the context of updates, but is mentioned here because of its association with LAST.

Access qualified by label or by subscript can be continued to any desired depth in the tree, and labels and subscripts can be mixed at will. Consider for example, the node with value 216 in Fig. 2.3-1. Several ways of "pointing" to the node are:

```
$PAYLOAD.TELESCOPE.WINDOW.END  
$PAYLOAD.TELESCOPE.WINDOW(2)  
$PAYLOAD.TELESCOPE(1).END  
$PAYLOAD(2).WINDOW(2)  
$PAYLOAD.TELESCOPE(LAST)(LAST)  
$PAYLOAD(2)(1)(2) .
```

References to particular nodes may be intended to refer to a tree substructure (i.e., the node "pointed" to, including its label, and anything below that node in the tree) or to a value. The meaning of a tree reference depends on the context in which the reference occurs. Thus, a statement which commands that a node be "pruned" (i.e., deleted from its tree) is obviously a structure reference, while a statement like

```
WINDOW_DURATION=$PAYLOAD(2).WINDOW.END  
-$PAYLOAD(2).WINDOW.START;
```

refers, because of its arithmetic nature, to the values (216 and 40) of the two tree nodes used in the statement.

Tree relational expressions are a source of considerable power in PLANS. These are logical (Boolean) expressions that have a value of TRUE or FALSE. These expressions are analogous to

arithmetic relational expressions (e.g., XVAR < YVAR) and are usable in IF...THEN...ELSE statements.

PLANS tree relations include IDENTICAL TO, SUBSET OF, and SUPERSET OF. The expression \$TREE_A IDENTICAL TO \$TREE_B has the value TRUE if, and only if, the entire substructure of \$TREE_A (or the value of \$TREE_A, if it has no descendants) is identical to that of \$TREE_B with respect to structure and order, labels, and values. The expression \$TREE_A SUBSET OF \$TREE_B has the value TRUE if, and only if, for each first-order substructure of \$TREE_A, there exists an identical first-order substructure in \$TREE_B. The expression \$TREE_A SUPERSET OF \$TREE_B is equivalent to \$TREE_B SUBSET OF \$TREE_A.

In each of the sample expressions above, any node reference can be substituted for \$TREE_A and \$TREE_B, and these Boolean expressions can be combined with Boolean AND (&) and OR (|) operators in the usual fashion. Thus, one can write a statement like

```
IF $PAYLOAD(I)SUBSET OF $CANDIDATES
  & $PAYLOAD(I).WINDOW.END > LAUNCH_DATE
  THEN GO TO THIS_PAYLOAD_OK;
```

Note that this statement contains node references that are structural and a node reference that is arithmetical.

Tree relational expressions are most useful in conjunction with two keywords, FIRST and ALL, which have special meaning when used as label qualifiers in PLANS tree node references. \$PAYLOAD.FIRST, when used by itself, is a legal expression with the same meaning as \$PAYLOAD(1). The important usage of FIRST

is in an expression like \$PAYLOAD.FIRST:(CONDITION). This expression means "find the first descendant of the node \$PAYLOAD that satisfies the condition in parentheses," where the condition is a Boolean expression. An example is shown in Fig. 2.3-2. The operation desired by the programmer might be stated, "find the first descendant of \$PAYLOAD whose launch window duration is greater than 150 days." Expressed in more procedural terms, the operation might be, "Consider each *element* (i.e., each descendant of \$PAYLOAD) in turn. Calculate the launch window duration of the *element* being considered. If greater than 150, proceed as if it had been referenced by name." Note the correspondence of the use of the word ELEMENT in this procedural description and in the corresponding tree node reference, \$PAYLOAD.FIRST:(ELEMENT.WINDOW.END - ELEMENT.WINDOW.START > 150) .

The label qualifier ALL works in a similar manner, but represents a reference to more than one descendant. Thus, if \$PAYLOAD refers to the tree structure of Fig. 2.3-1, then \$PAYLOAD.ALL refers to the portion of that structure indicated in Fig. 2.3-3. Writing \$PAYLOAD.ALL is equivalent to "pointing" to \$PAYLOAD(1), \$PAYLOAD (2),..., \$PAYLOAD(LAST), each in turn. \$TREE.ALL is therefore a way of referring to the *substructure* of a node without including the node itself.

It should be noted that the position of a label or subscript qualifier in a PLANS tree node reference always corresponds to the level of the node in the structure itself. (A sophisticated reader who has looked at a PLANS program that uses

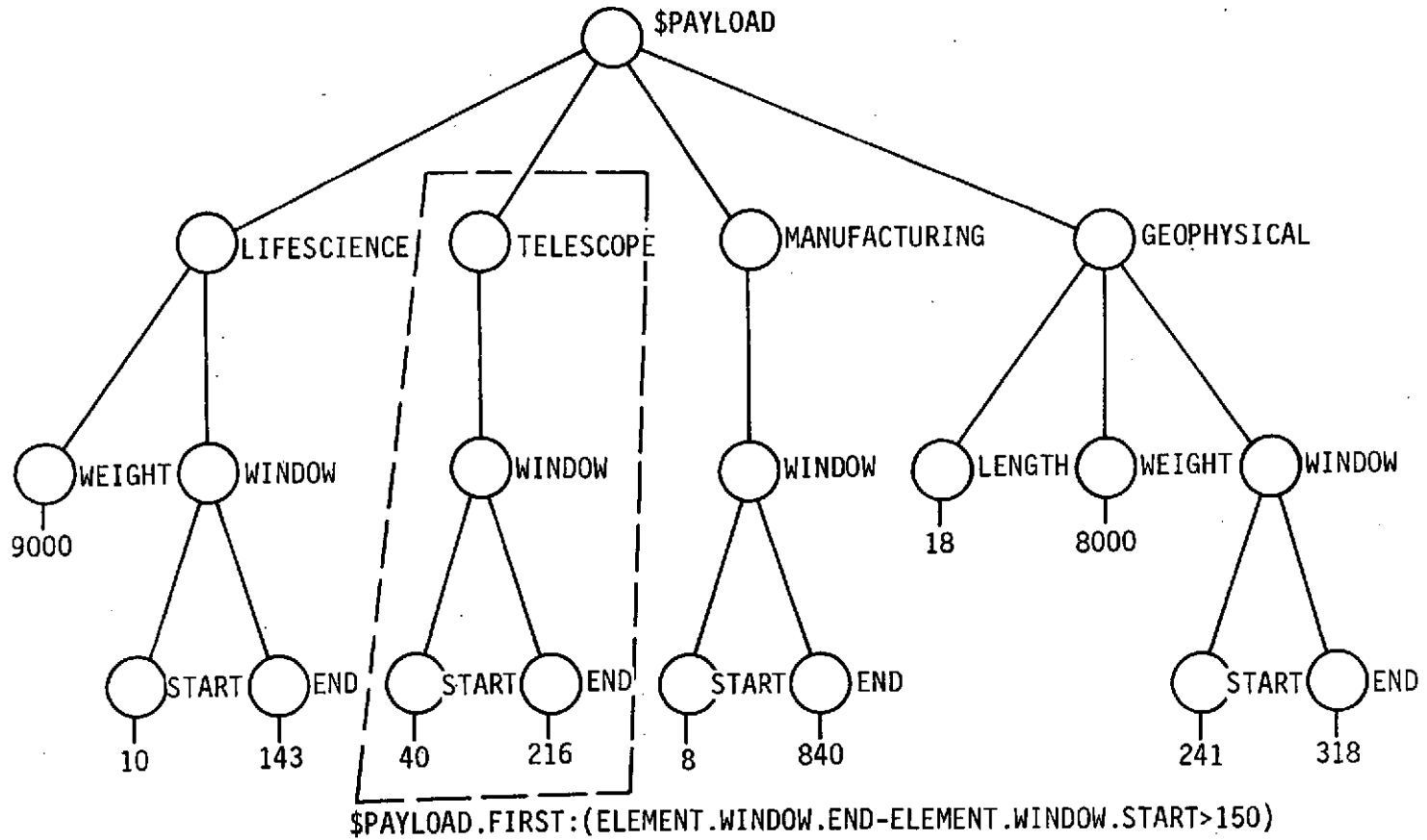


Fig. 2.3-2 Conditional Access Using Qualifier ".FIRST"

Fig. 2.3-3

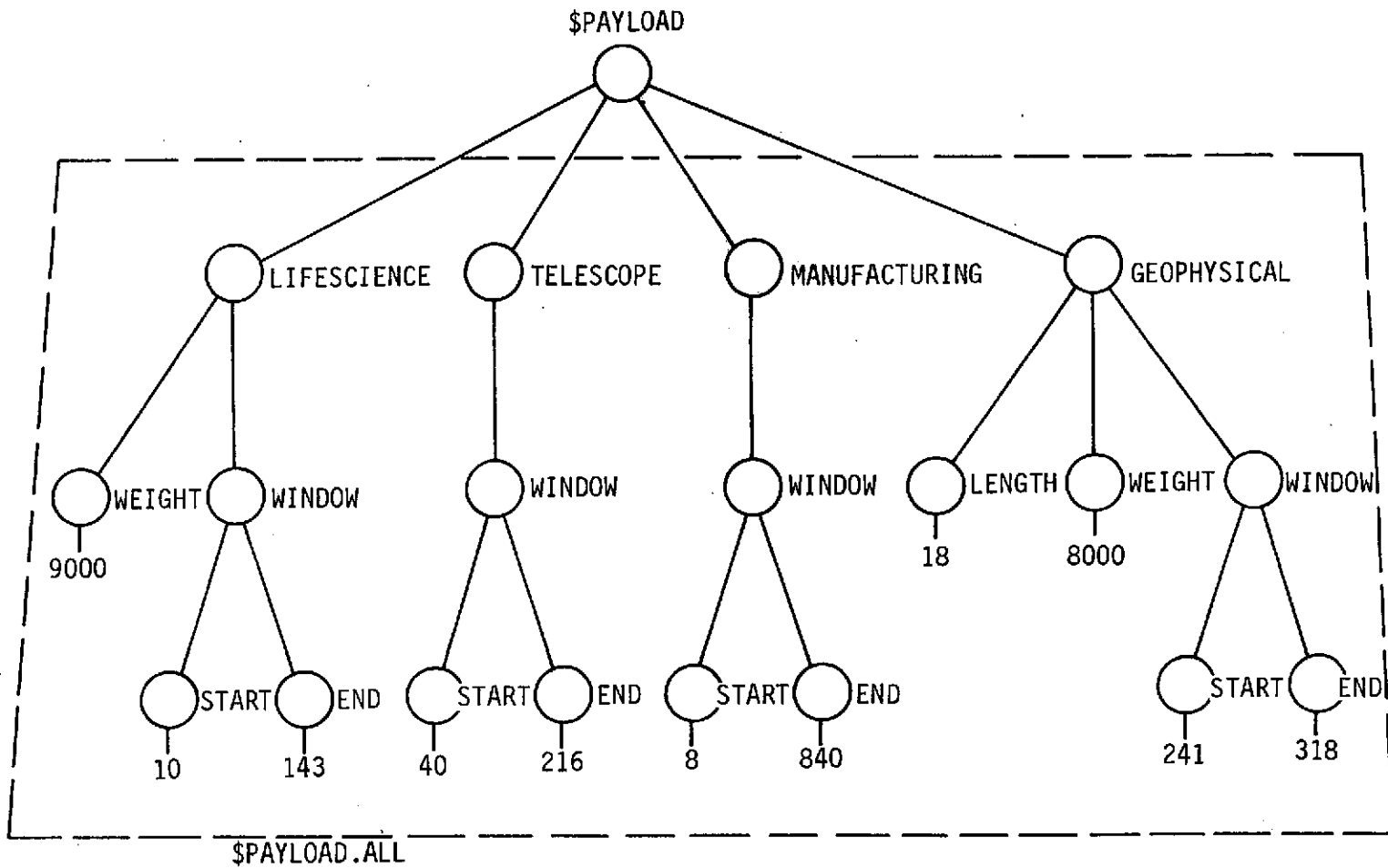


Fig. 2.3-3 Use of Label Qualifier ".ALL"

subroutines may detect an apparent exception to this statement. Please be assured that it is not an exception, but merely a question of tree name scope.) While this correspondence of position is clear in a tree node reference like \$PAYLOAD(2).WINDOW.END, it will be helpful to the user to keep in mind that it also applies to the use of FIRST and ALL. Thus \$PAYLOAD.ALL is a reference to one or more nodes that are one level *below* \$PAYLOAD .

As in the case of FIRST, ALL is most usefully employed in expressions like \$PAYLOAD.ALL:(condition), where the condition is a Boolean expression. In this case, instead of a single node, the reference is to *all* the subnodes of a particular node *that satisfy the stated condition*. An example is shown in Fig. 2.3-4. This "all such that" capability is very powerful as a means of filtering sets of elements for a particular set of characteristics.

Sometimes the programmer needs to refer to the label on a node rather than to its value or the structure it represents. For this purpose, PLANS provides the special function LABEL. For example, the reference LABEL(\$PAYLOAD(3)) applied to the tree of Fig. 2.3-1 yields the character string MANUFACTURING.

A second special function of PLANS is NUMBER. This function returns the number of descendants possessed by a given node. Thus, the expression NUMBER(\$PAYLOAD) applied to the tree of Fig. 2.3-1 yields the numerical value 4.

A particularly important tree access feature is indirect referencing. Unless the programmer resorts to very expensive iterative tree searching there is no way, without indirect

Fig. 2.3-4

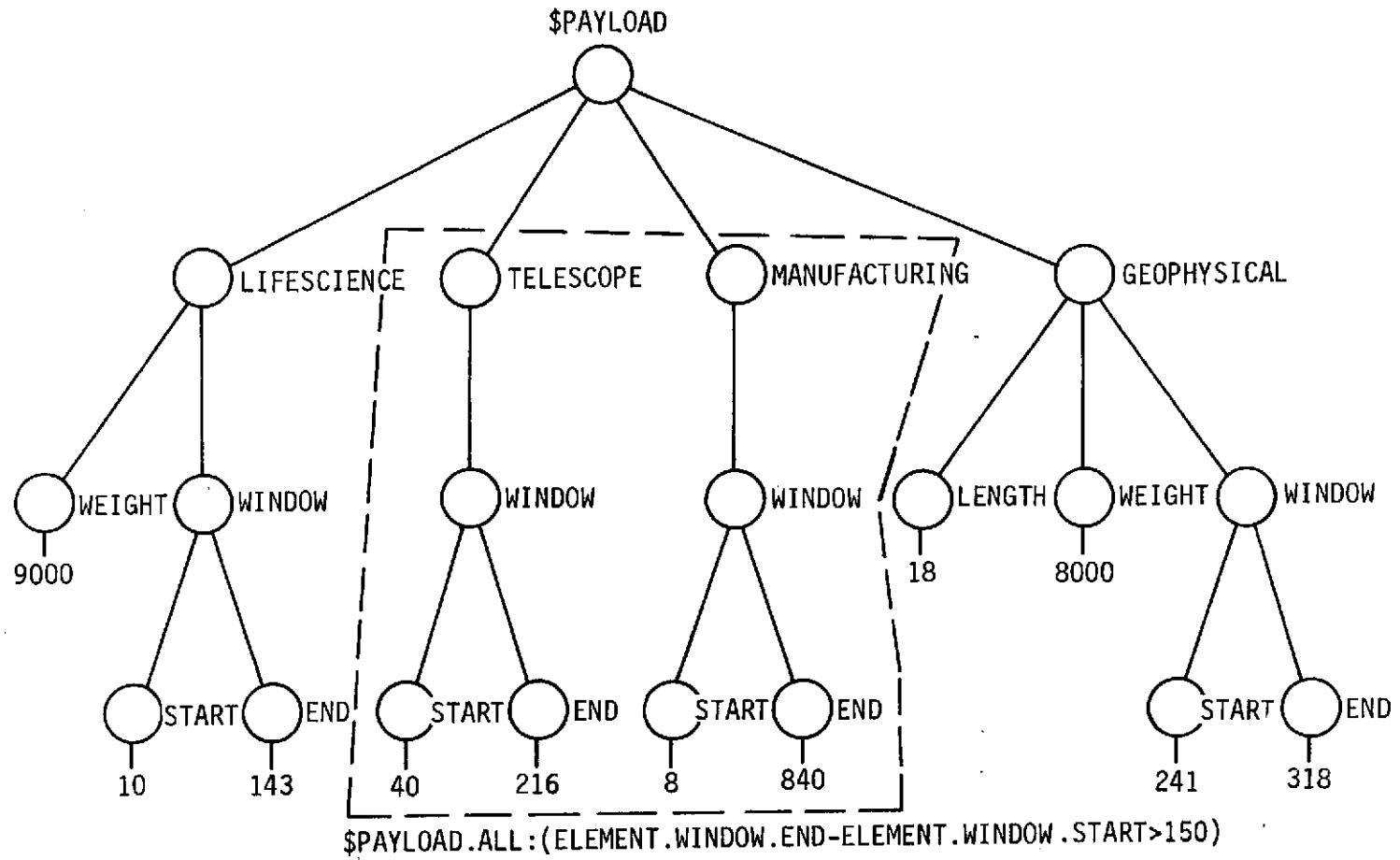


Fig. 2.3-4 Conditional Access Using Qualifier ".ALL"

referencing, that he can write a program to schedule shuttle flights that do not contain words like PAYLOAD, ORBITER, etc. In order to access information about these resources, he wants to use them as labels for qualified access or, conceivably, as tree names. What is needed is a capability that allows the characteristics of a problem to reside in the data, rather than the program. Only in this way can a program that schedules shuttle flights also schedule machine shop operations. What the programmer needs is the capability to read in, as data, the labels he will use to access particular tree nodes. Accordingly, PLANS allows the kind of indirect referencing illustrated in Fig. 2.3-5. What the programmer is attempting to do in this illustration is to access information about the resource types named in a tree called \$RESOURCE_REQUIREMENTS . He therefore writes the tree node expression \$RESOURCE_INFO.#(\$RESOURCE_REQUIREMENTS(1)) to access information about the first such resource type. This expression might be read, "the descendant of the node \$RESOURCE_INFO whose label is the character string found as the current value of the node \$RESOURCE_REQUIREMENTS(1)". The programmer is in effect saying, "Behave as if I had written \$RESOURCE_INFO.ORBITER, but allow me the freedom to use some other label than ORBITER by changing the data, without changing the program.

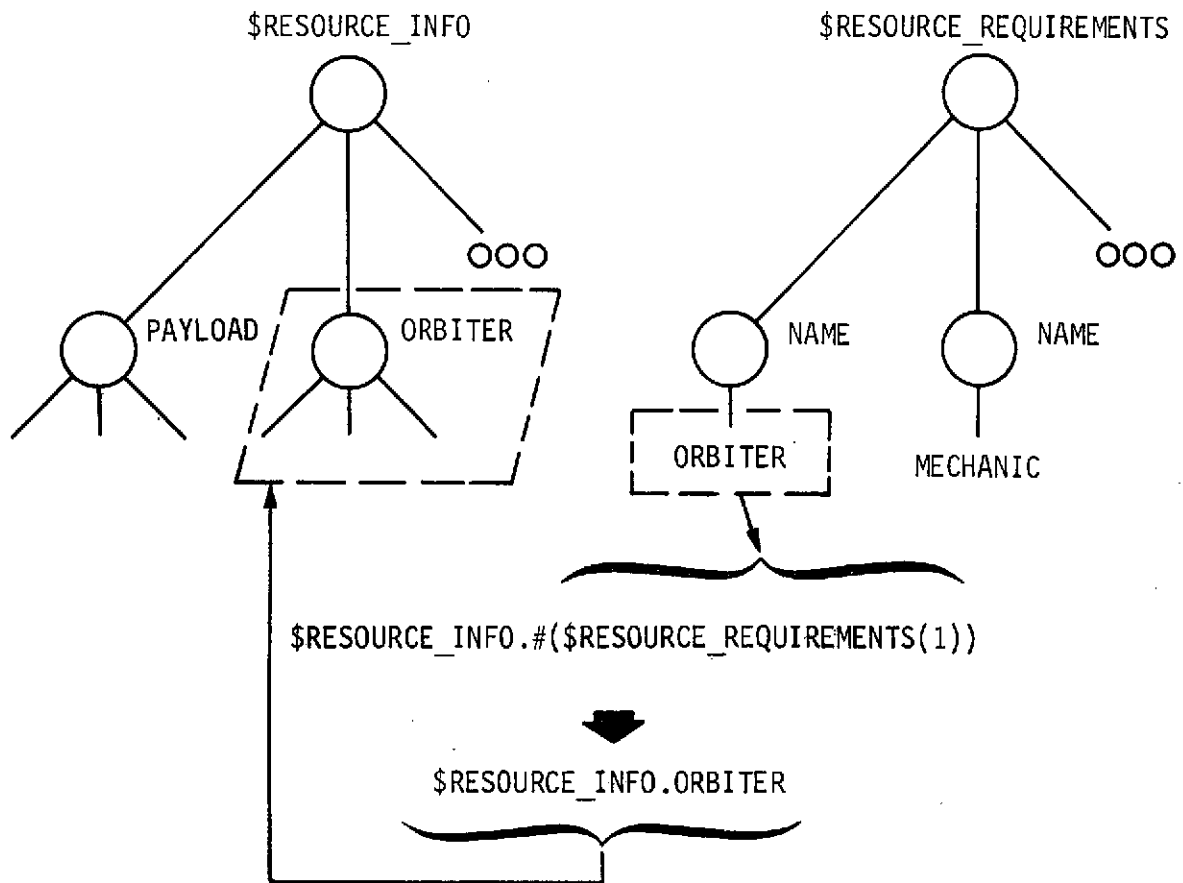


Fig. 2.3-5 Indirect Referencing

2.4 PLANS TREE UPDATE MECHANISMS

The basic PLANS tree update mechanism is the tree assignment statement. The reader is undoubtedly familiar with the properties of such ordinary arithmetic assignment statements as

$$XVAR = YVAR; .$$

The function of such a statement might be described algorithmically as: (1) destroy the current "contents" of the variable XVAR, (2) make an exact copy of the current "contents" of the variable YVAR without modifying YVAR, and (3) place the copy "in" XVAR. While the execution of such a statement is more direct than this

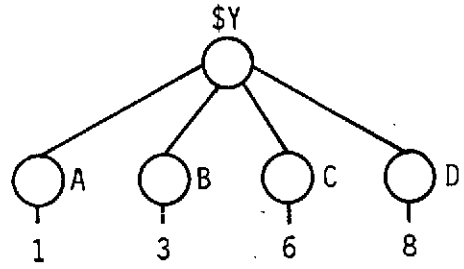
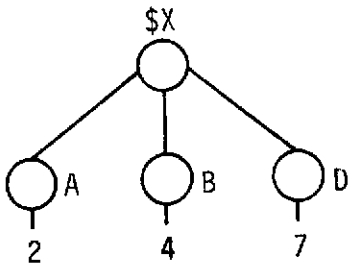
algorithm would indicate, the analogy with tree assignment statements should be clear if the reader will think in terms of this algorithm while considering the statement

```
$XTREE = $YTREE; .
```

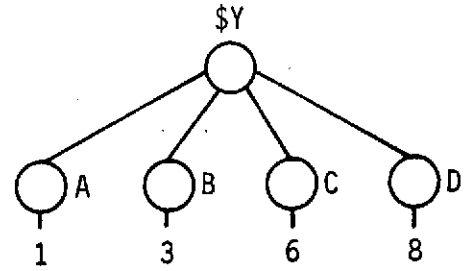
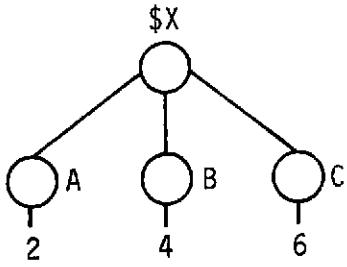
This statement (1) destroys the current value or substructure of \$XTREE, (2) creates a copy of the node and value or substructure of \$YTREE, and (3) places the resulting structure at \$XTREE. The net result, as with the arithmetic assignment statement, is one of *replacement* or *assignment* of a new value.

Of course, the tree node expressions in a tree assignment statement may be more complex than simple tree names. Several examples are shown in Fig. 2.4-1, and should be considered in detail. Figure 2.4-1 (a) shows the initial condition of two trees, \$X and \$Y, which will be successively modified by the execution of a series of tree assignment statements. The first such statement, \$X(3) = \$Y.C, modifies the tree \$X, as shown in (b). Note that the original third subnode of \$X has been deleted and replaced with a *copy* of the node \$Y.C, and that the tree \$Y has not been altered at all.

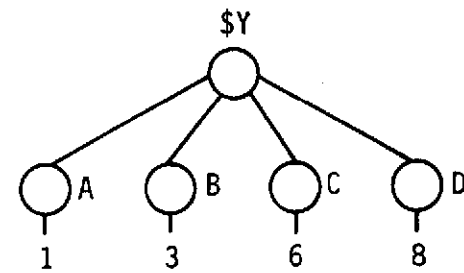
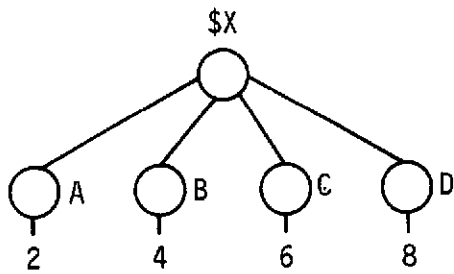
Beginning with the trees in (b), the statement \$X.D = \$Y(LAST) results in the modified \$X shown in (c). If \$X had had a node labeled "D", it would have been replaced as in the previous example. Because the left-hand side of the tree assignment statement referred to a node not yet in existence, *a new subnode of \$X was created*. The new subnode is always added at the right. Thus,



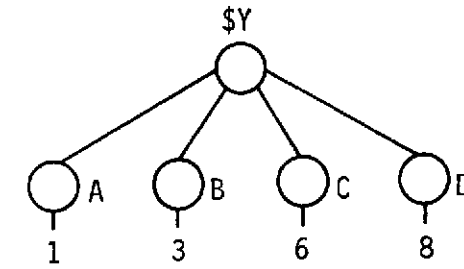
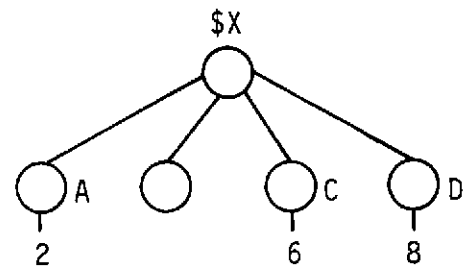
(a) Original Trees



(b) Trees after $\$X(3) = \$Y.C$



(c) Trees after $\$X.D = \$Y(LAST)$



(d) Trees after $\$X(2) = \$Y.E$

Fig. 2.4-1 Results of a Sequence of Tree Assignment Statements

in the absence of a node $\$X.D$, the statement $\$X.D = \$Y(LAST)$ behaves like the statement $\$X(NEXT) = \$Y(LAST)$.

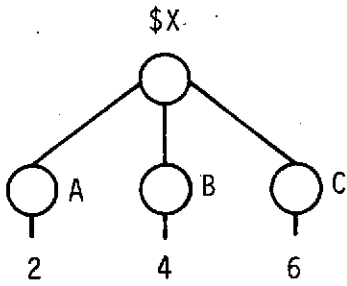
What if the right-hand side represents a nonexistent node? This case is illustrated in (d). In (c), there is no node $\$Y.E$. Therefore, the statement $\$X(2) = \$Y.E$ (1) deletes the contents of $\$X(2)$, (2) makes a copy (null) of $\$Y.E$, and (3) replaces $\$X(2)$ with the copy. That is, $\$X(2)$ is replaced by a null node under these circumstances. This convention is consistent with the execution of the statement when the node in question exists, and has the advantage that it allows the programmer to test explicitly for a null node ("IF $\$X(2) = \$NULL$ THEN...") if he is in doubt about the existence of the node referred to on the right-hand side of the tree assignment statement. This same behavior occurs when a conditional tree access is used in which the condition is not satisfied. Suppose, for example, that the programmer had wanted to replace $\$X(2)$ in the example by a copy of the first descendant of $\$Y$ that had a substructure. He might have written $\$X(2) = \$Y.FIRST:(NUMBER(ELEMENT) > 0)$. Because none of the descendants of $\$Y$ satisfies the condition, the result would have been identical to that resulting from $\$X(2) = \$Y.E$. Both statements yield the same result as $\$X(2) = \$NULL$.

Section 2.3 gave several illustrations of automatic conversion in which PLANS tree node references occurring in an arithmetic context were implicitly evaluated (i.e., the *value* of the referenced node was used, rather than the node itself, which

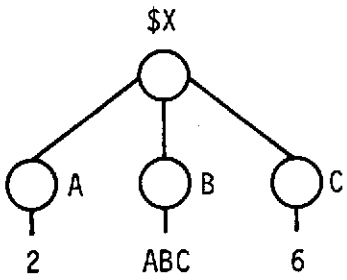
is a structure). Analogously, character strings and arithmetic expressions may appear in tree assignment statements, as illustrated in Fig. 2.4-2.

Figure 2.4-2(a) shows the initial condition of the tree \$X. Figure 2.4-2(b) shows \$X as modified after the execution of the statement \$X.B = 'ABC'. Described algorithmically, here is what has happened: (1) the value or substructure of \$X.B has been deleted, because \$X.B occurs on the left-hand side of a tree assignment statement; (2) the right-hand side of the statement has been evaluated *as a tree expression*, because that is what is called for by the tree assignment statement, and (3) a copy of the tree structure referred to on the right-hand side has replaced the value or substructure of \$X.B. By this description, then, this tree assignment statement operated like any other. But how is something, evaluated *as a tree expression* when it is in fact a character string?

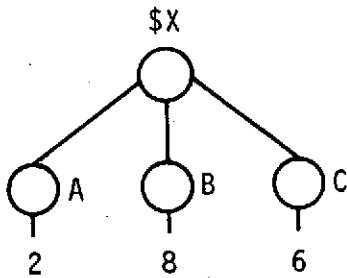
Any time a character string or arithmetic expression occurs, when the context clearly calls for a tree expression, a dummy tree is created. This dummy tree has only a single node, the root node, which has a null label. The *value* of the node is the string or arithmetic value specified in the PLANS expression, in this case the string 'ABC'. The dummy tree is then used just as if the programmer had explicitly created the tree and placed the tree's name in the program. In the case of the example, the result is the same as if the programmer had written \$X.B = \$DUMMY, where \$DUMMY is a tree with one node, no label, and the string value 'ABC'.



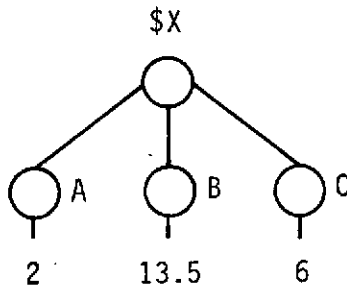
(a) Original Tree



(b) Tree after $\$X.B = 'ABC'$



(c) Tree of (a) after $\$X.B = 2*4$



(d) Tree of (a) after $\$X.B = \$X.C + 7.5$

Fig. 2.4-2 Type Conversion in Tree Assignment Statements

As suggested in the explanation above, the same mechanism applies when an arithmetic expression appears in a context that requires a tree node reference. Thus, application of the statement $\$X.B = 2*4$ to the tree of 2.4-2(a) yields the result shown in (c). The value of the arithmetic expression, in this case 8, is calculated, placed on a dummy node, and becomes the value of $\$X.B$. It may occur to the reader that the same behavior could as well be described as replacement of the value (or substructure) on the left by the value of the expression on the right. As the discussion of Fig. 2.4-4 will show, this is not always true. It will be helpful, therefore, to think in terms of the generation of a dummy node when considering statements of this type.

Figure 2.4-2(d) shows a statement of the same basic sort as that of (c). In this case, the arithmetic expression on the right-hand side involves a tree node reference. Because $\$X.C$ occurs within an arithmetic expression, it has the value 6, just as if $\$X.C$ were an arithmetic variable name. Therefore, the statement $\$X.B = \$X.C + 7.5$ results in substitution of the numeric value 13.5 at $\$X.B$.

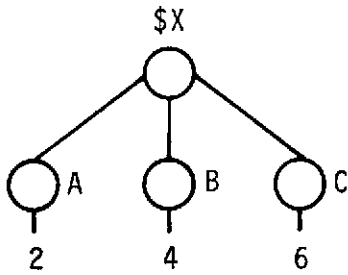
The previous discussion has shown a mechanism whereby the value or substructure of a node can be replaced. Sometimes it is the node *label* that requires modification. In this case, the label assignment statement is used. LABEL is a special PLANS function that takes as its argument a tree node reference. LABEL($\$X(1)$) is a reference not to the node $\$X(1)$ and its

substructure, but to its label alone. The LABEL function can appear anywhere a character string can appear in a PLANS program. In addition, it can appear on the left-hand side of an equal sign, as Fig. 2.4-3 shows. Such a statement is a command to replace the current label of the specified node with the new string, which is obtained by evaluating the expression to the right of the equal sign.

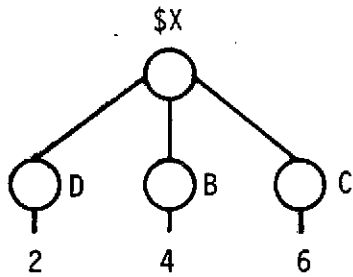
Consider Fig. 2.4-3; (a) shows the initial state of the tree \$X. Figure 2.4-3 (b) illustrates the effect of the label assignment statement LABEL(\$X(1)) = 'D'; , which simply replaces the current label of the node \$X(1), "A", with a new string, "D". The label assignment statement only replaces labels, having no structural effect if the referenced node already exists. If the indicated node does not exist, it is established, with a null value and the indicated label.

The statement illustrated in (c) has exactly the same effect as that of (b). It makes no difference whether the node is referenced by label (\$X.A) or by subscript (\$X(1)). The effect of the statement is the same. As the discussion of Fig. 2.4-4 will show, this is not true of all tree statements.

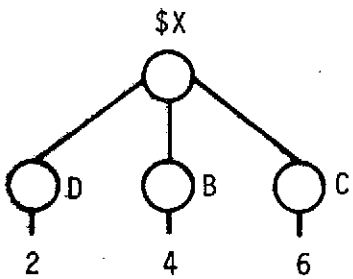
Figure 2.4-3 (d) is another illustration of automatic conversion. The right-hand side of the statement LABEL(\$X(1)) = \$X.C is a tree expression, but the context calls for a string or numerical value. The value of \$X.C is therefore obtained, and replaces the label of \$X(1). An additional concept is illustrated here: labels can be numerical values. In fact, anything that can be a



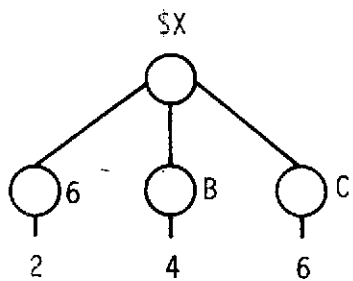
(a) Original Tree



(b) Tree after LABEL (\$X(1)) = 'D'



(c) Tree of (a) after LABEL(\$X.A) = 'D'



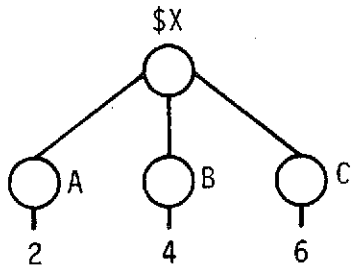
(d) Tree of (a) after LABEL(\$X(1)) = \$X.C

Fig. 2.4-3 Label Assignment Statements

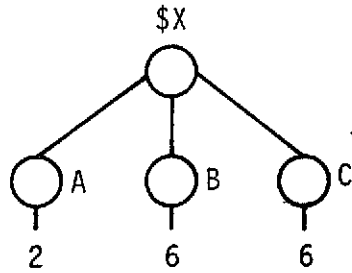
value can be a label, and vice versa. However, nodes that have numerical values (or strings not having identifier syntax) cannot be accessed by label in a PLANS program. Thus, $\$X.6$ is not a legal expression. On the other hand, $\$X(1)$ is still a legitimate way to refer to this node. This property can be used to advantage in some numerical applications.

Figure 2.4-4 illustrates an important property of PLANS tree operations, the treatment of the label on the *base* node of the operation. The "base node" is the node at which a modification occurs. In the case of a tree assignment statement, the node named on the left-hand side is the base node. The question is, When does the existing label on the base node remain, and when is it replaced? It is unlikely that any decision rule that might be incorporated into PLANS would successfully anticipate the desires of the programmer in all cases. Therefore, a simple decision rule has been incorporated that should be right most of the time. If the base node was accessed by label, the label remains; if by subscript, the label is replaced.

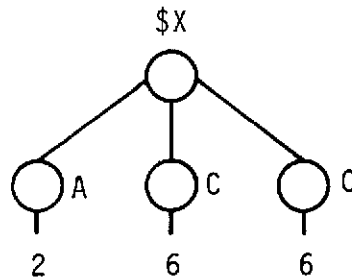
In Fig. 2.4-4(b), for example, the statement $\$X.B = \$X.C$ has been executed on the tree of (a). The base node, $\$X.B$, retains its original label, "B", because the programmer specified the node by that label. Thus, while the value or substructure of $\$X.C$ will appear on the node $\$X.B$ after execution of this statement, the base node label will be left unchanged. In (c), on the other hand, the base node label is replaced. In this case, the statement $\$X(2) = \$X.C$ involves a base node access by subscript.



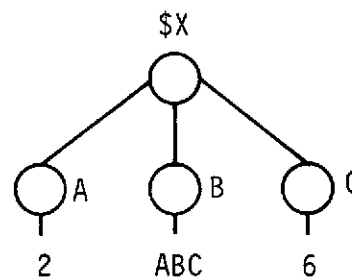
(a) Original Tree



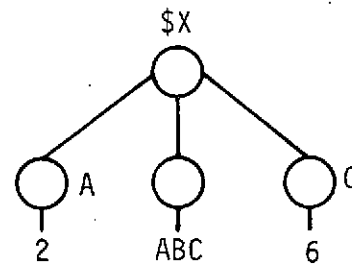
(b) Tree after $\$X.B = \$X.C$



(c) Tree of (a) after $\$X(2) = \$X.C$



(d) Tree of (a) after $\$X.B = 'ABC'$



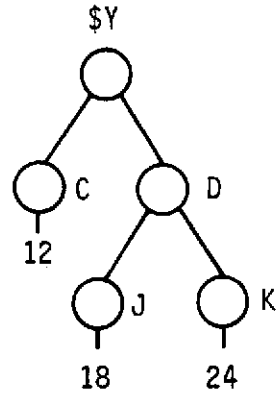
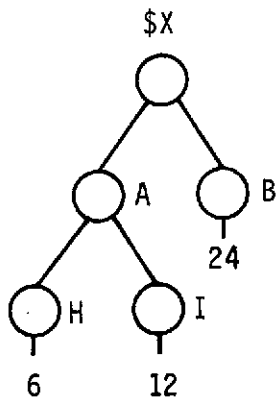
(e) Tree of (a) after $\$X(2) = 'ABC'$

Fig. 2.4-4 Treatment of Base-Node Labels

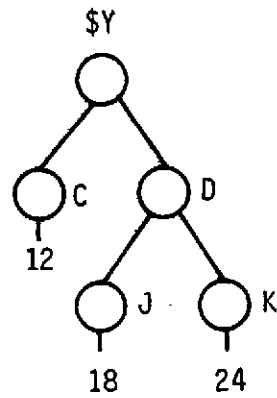
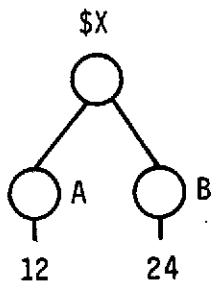
Notice that the new label, "C," is copied from the structure referred to on the right-hand side.

Figure 2.4-4 (d) shows a situation similar to (b). Again, by specifying the base node by label, the programmer has made an assertion about which label is to appear on the node after execution of the statement. The fact that the expression on the right-hand side is a string expression has no special result here. Figure 2.4-4(e) on the other hand, deserves special attention. Because the node $\$X(2)$ was not specified by label, the existing label on this node is deleted. But the DUMMY node defined in the right-side expression has a null label. Therefore the apparent result of a statement like $\$X(2) = 'ABC'$ is to delete the existing base node label. As mentioned previously, recognition of those cases in which a dummy structure is created will assist the programmer in assuring that the desired operations are achieved.

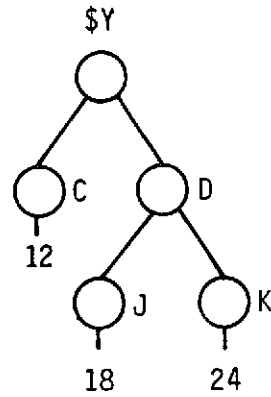
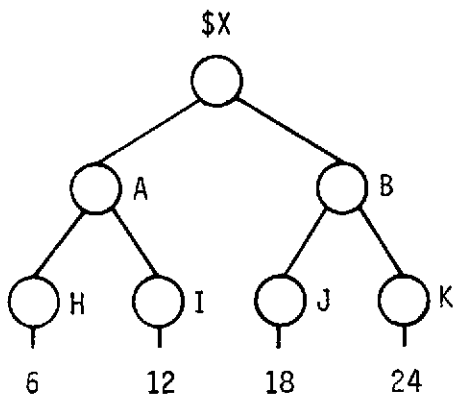
Another property of PLANS tree operations that should be well understood is the exclusivity of values and substructures. A node may have a null value or it may possess either a value or a substructure, but it may never have both a value and a substructure. Figure 2.4-5 illustrates this concept. In (b), execution of $\$X.A = \$Y(1)$ places a new value on the node $\$X.A$, with the result that the previous *substructure* of $\$X.A$ is deleted. Figure 2.4-5 shows the converse case in which placement of a new substructure on the node $\$X.B$ deletes the previous *value* of that node.



(a) Original Trees



(b) Trees after $\$X.A = \$Y(1)$



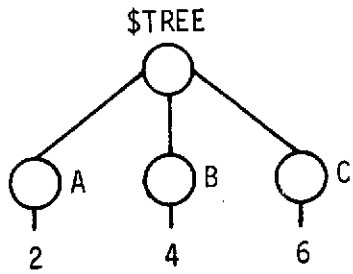
(c) Trees of (a) after $\$X.B = \$Y(2)$

Fig. 2.4-5 Value-Substructure Exclusivity

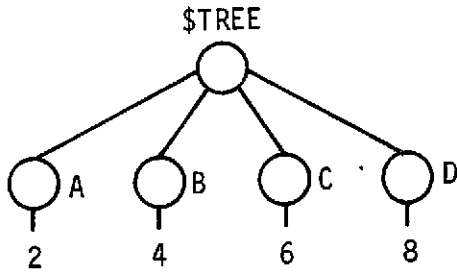
Figure 2.4-6 illustrates some tree assignments to previously nonexistent nodes. Figure 2.4-6(b) shows the tree of (a) as modified by the statement $\$X.D = 8$. It should be noted that this statement has assigned a value *and* a label to the new node. Any time the referenced node does not exist, it is caused to exist as specified. If it was specified by label, this means the indicated label must be placed on the new node.

Figure 2.4-6(c) shows the result of a statement in which a nonexistent node was specified by subscript. Because no label was used to indicate the node and the expression on the right has no label (it is a dummy node), the resulting node has a value, but no label. Figure 2.4-6(d) involves a new node with a label, but no value. In the figure this result was achieved by the statement $\text{LABEL}(\$X(\text{NEXT})) = 'D'$. However, because two prime (quote) marks together refer to the null character string, the same result would be observed after execution of the statement $\$X.D = ''$. This statement places a null string on the node as a value, but that is completely equivalent to no value at all.

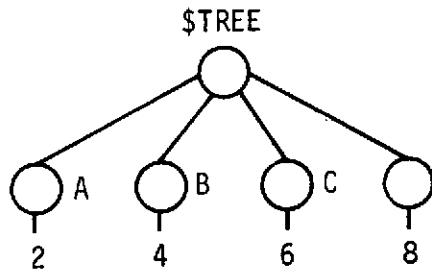
Figure 2.4-6(e) shows what happens when an assignment is made to a node specified by a subscript that is too large. (It is, of course, only *too* large if the programmer *did not want* the result shown in the figure.) The programmer has stated that the fifth subnode of $\$X$ is to acquire the value 8. But this can only occur if, after execution of the statement, $\$X$ has at least five descendants. Because there were only three descendants before the statement was executed, *two* new nodes will be created. Only



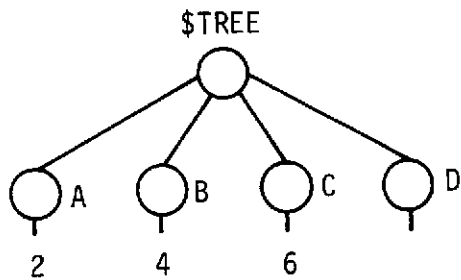
(a) Original Tree



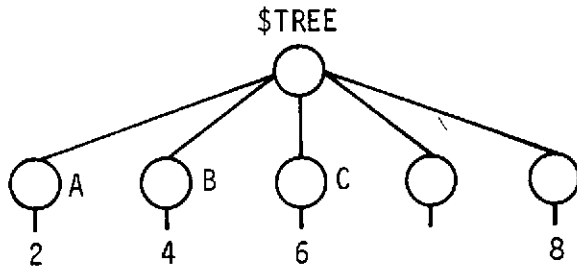
(b) Tree after $\$X.D = 8$



(c) Tree of (a) after $\$X(NEXT) = 8$



(d) Tree of (a) after LABEL ($\$X(NEXT)$) = 'D'



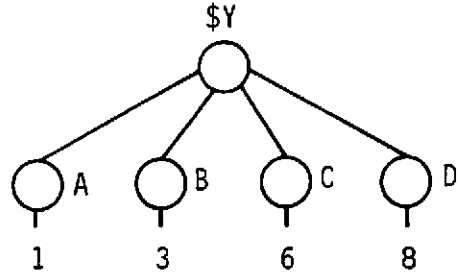
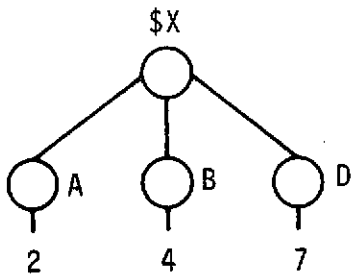
(e) Tree of (a) after $\$X(5) = 8$

Fig. 2.4-6 Assignments to Nonexistent Nodes

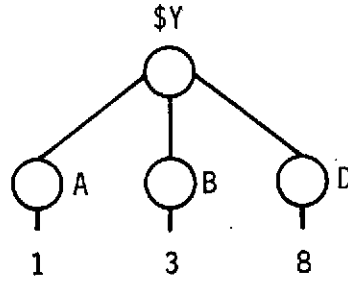
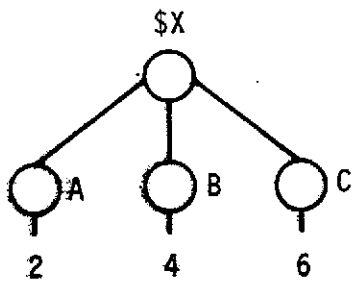
the latest of these newly created nodes is involved in a tree assignment statement; therefore, only the last node can acquire a label or a value. The remaining new node(s), in this case $\$X(4)$, will be null.

In addition to the tree assignment statement, which usually *replaces* the current contents of a specified node with a *copy* of the contents of another node, there are three other basic tree manipulation statements that perform somewhat similar functions. These are the GRAFT, INSERT, and GRAFT INSERT statements. Instead of simple replacement, the INSERT and GRAFT INSERT statements result in an insertion, with no deletion of information from the target tree. Instead of *copying* the information to be added to the target tree, the GRAFT and GRAFT INSERT statements *remove* the specified structure from its original location and *move* it to the target tree. Examples of these statements are shown in Fig. 2.4-7 (GRAFT), 2.4-8 (INSERT), and 2.4-9 (GRAFT INSERT). In each case, the results of the statements should be compared with one another (they are parallel cases) and with the corresponding tree assignment statements in Fig. 2.4-1.

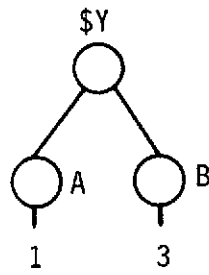
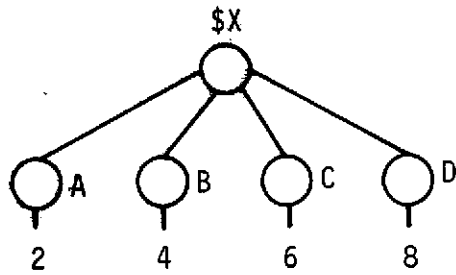
Figures 2.4-1(b), 2.4-7(b), 2.4-8(b), and 2.4-9(b) show the effects of the statements $\$X(3) = \$Y.C$; GRAFT $\$Y.C$ AT $\$X(3)$; INSERT $\$Y.C$ AT $\$X(3)$; and GRAFT INSERT $\$Y.C$ AT $\$X(3)$; respectively. These statements all perform parallel operations, differing only by virtue of the special properties of the four statement types. It should be noted in particular that: (1) the tree assignment and GRAFT statements have the same effect (replacement) on $\$X$,



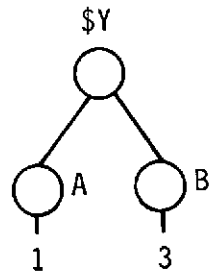
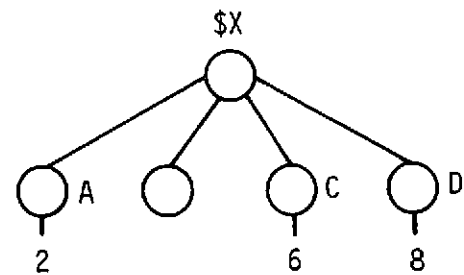
(a) Original Trees



(b) Trees after GRAFT \$Y.C AT \$X(3)

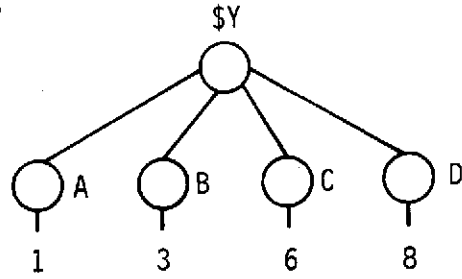
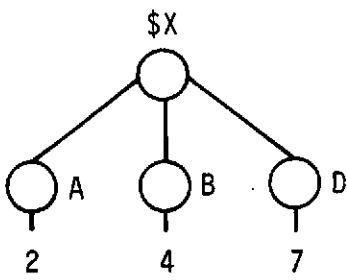


(c) Trees of (b) after GRAFT \$Y(LAST) AT \$X.D

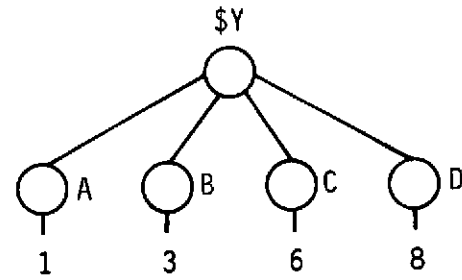
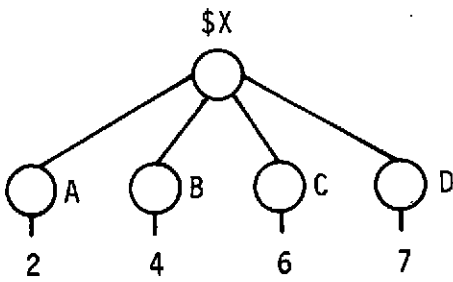


(d) Trees of (c) after GRAFT \$Y.E AT \$X(2)

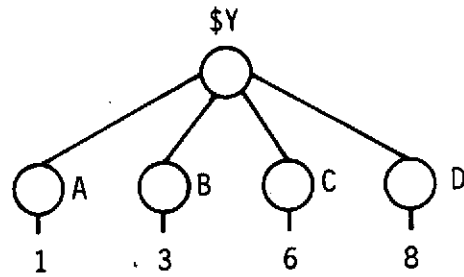
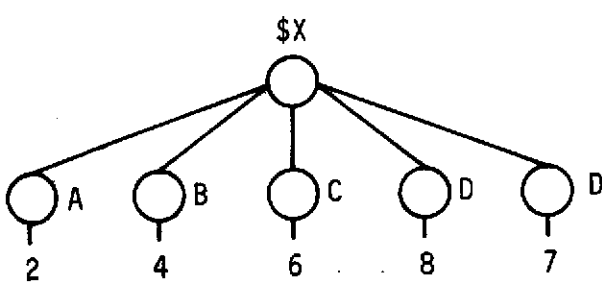
Fig. 2.4-7 GRAFT Statements



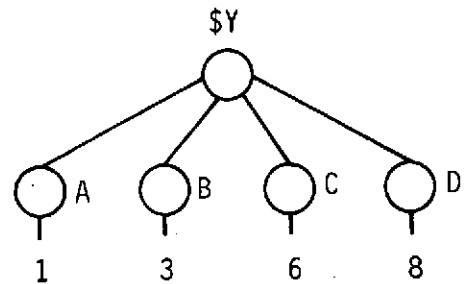
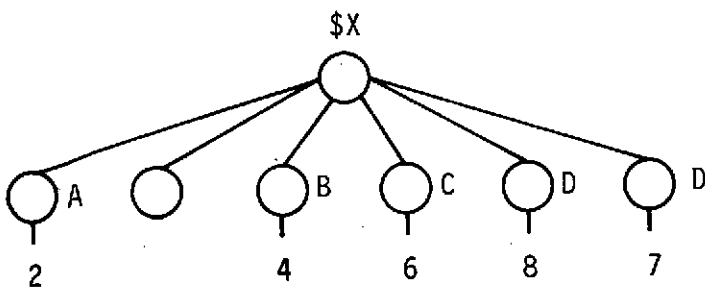
(a) Original Trees



(b) Trees after INSERT \$Y.C AT \$X(3)

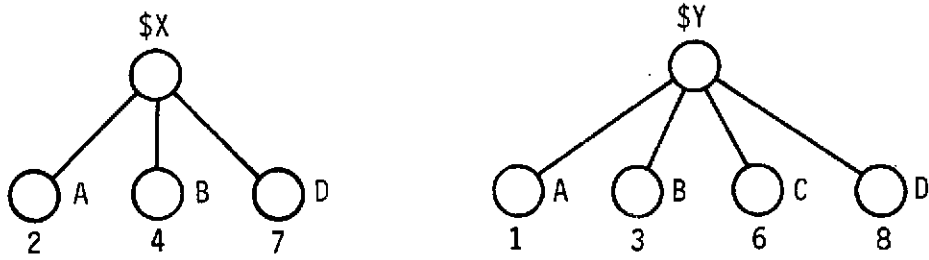


(c) Trees of (b) after INSERT \$Y(LAST) AT \$X.D

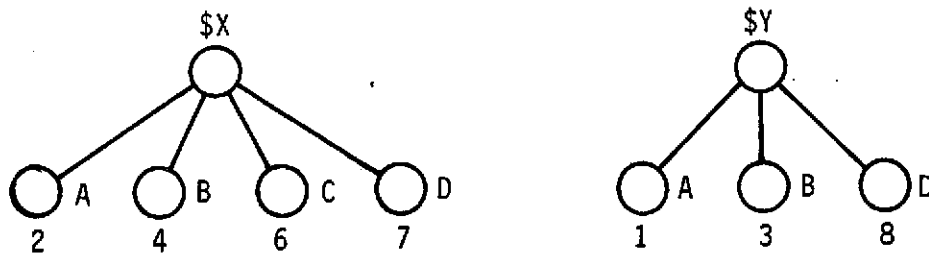


(d) Trees of (c) after INSERT \$Y.E AT \$X(2)

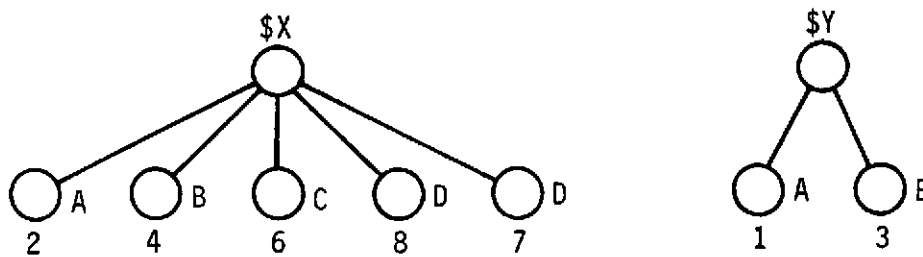
Fig. 2.4-8 INSERT Statements



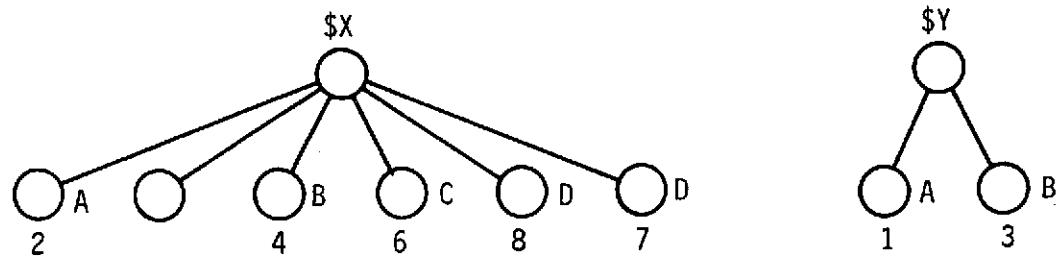
(a) Original Trees



(b) Trees after GRAFT INSERT \$Y.C\$ AT \$X(3)\$



(c) Trees of (b) after GRAFT INSERT \$Y(LAST)\$ AT \$X.D\$



(d) Trees of (c) after GRAFT INSERT \$Y.E\$ AT \$X(2)\$

Fig. 2.4-9 GRAFT INSERT Statements

the target tree; (2) INSERT and GRAFT INSERT have the same effect (insertion) on \$X; (3) the tree assignment and INSERT statements have no effect on \$Y; and (4) GRAFT and GRAFT INSERT have the same effect (deletion) on \$Y. It should also be noted that insertions are made *at* (or, if you prefer, *before*) the named node. The named node, and all others to the right, are "moved" one node to the right.

Parts (c) of the four figures show additional parallel operations of these four types. It should be observed that the INSERT and GRAFT INSERT operations of (c) result in two subnodes of \$X that possess the same label. This is quite allowable, but the programmer should be aware that, if this occurs, the subsequent reference \$X.D is a reference to only the *first* such node. Either node can still be referenced by subscript, however, and a reference of the form \$X.ALL:(LABEL(ELEMENT = 'D')) would access *all* such nodes in one operation.

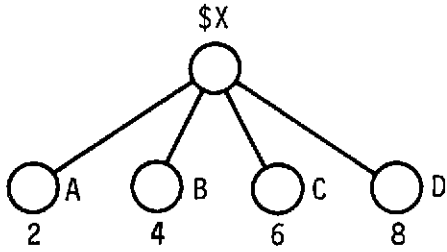
Parts (d) of the four figures are included to make it clear that in all cases an update to the target tree is performed. If the operation calls for a nonexistent node to be inserted or placed into a tree, a null node will result. The programmer can then test for the presence of a null node and, if desired, remove it.

A final note on these four statements is particularly important. The GRAFT, INSERT, and GRAFT INSERT statements give the appearance of being more complex than the tree assignment statement, and the programmer may naturally assume that the latter is

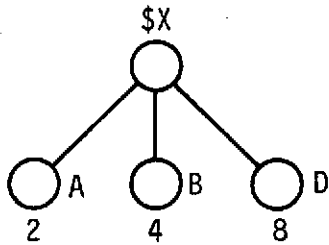
more efficient and should be given preference whenever there is a choice. A little reflection on the underlying structural operations will show that this is not true. The tree assignment and INSERT statements require the generation of a complete *copy* of an existing structure. The execution cost of these statements (and the storage space required) is largely a function of the size of the structure that must be copied. The GRAFT and GRAFT INSERT statements, on the other hand, require only the alteration of a few pointers so that an existing structure can be moved, completely intact, to another tree location. The execution cost of these statements is minimal, no additional storage is involved, and the cost is entirely independent of the size of the structure that is moved. It cannot be overemphasized that GRAFT and GRAFT INSERT operations are not only very powerful, but are also very efficient!

It is frequently necessary to remove a structure from a tree without placing it anywhere else. This simple deletion operation is accomplished by means of the PRUNE statement, illustrated in Fig. 2.4-10. The programmer simply specifies the node (or nodes) that, together with the associated substructure, is to be removed. This operation allows the removal of undesired information from a tree. It may also be used, particularly as in (d), to release computer storage that is no longer needed. It should be kept in mind while programming in PLANS that the programmer is really doing his own dynamic storage allocation (although PLANS handles all the details for him). When information is no longer needed,

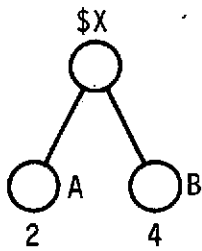
(a) Original Tree



(b) Tree after PRUNE \$X.C



(c) Tree of (a) after PRUNE \$X.C, \$X.D



(d) Tree of (a) after PRUNE \$X



Fig. 2.4-10 PRUNE Statements

its storage can be reused, but only if the programmer releases it by means of a PRUNE statement.

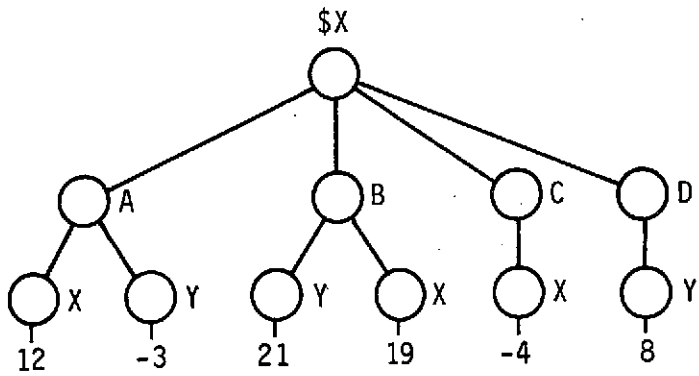
2.5 SPECIAL STATEMENTS

The previous sections have discussed the basic dynamic tree manipulation capabilities of PLANS. These capabilities have been provided because they fulfill the essential requirements of schedule development and optimization programming. It should be apparent, though, that the capabilities of PLANS have been kept quite general in nature. This provides the flexibility that is needed to span a problem space as broad as that of scheduling. There remain, however, a small number of operations that are complex, and well-defined, that occur frequently in scheduling operations, and are difficult to handle with basic PLANS. These include ordering (sorting) and the generation of the combinations or permutations of a set of elements. Special statements have been provided in PLANS for the performance of these slightly more specialized functions.

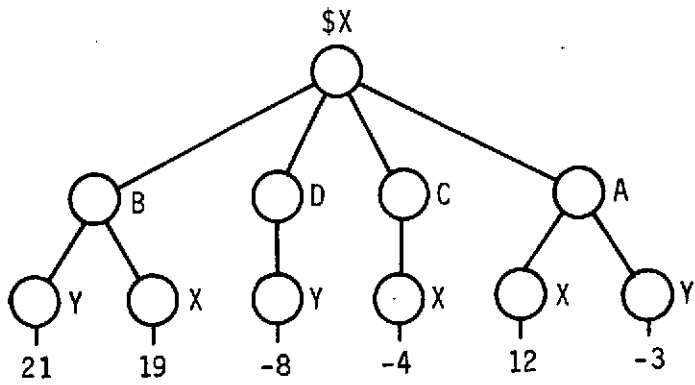
The ORDER statement is used to place the subnodes of a particular node in ascending or descending order by a particular property they possess. If, for example, it is desired to order a group of payloads by weight, heaviest first, one might write ORDER \$PAYLOADS BY WEIGHT; or if they were to be ordered by length, shortest first, with ties broken by width, narrowest first, a statement of the form ORDER \$PAYLOADS BY -LENGTH, -WIDTH; would be appropriate.

Figure 2.5-1 illustrates a few of the properties of the ORDER statement. It should be apparent that an ORDER statement refers to a node, which, in turn, has subnodes to be ordered. Each subnode has, at least potentially, the property or properties on which ordering is to occur. Ordering can be in either ascending or descending order. Figure 2.5-1(b) illustrates an ORDER statement in which the subnodes of the node \$X are sorted into descending order on the basis of property Y. Note that, where the property in question is not possessed by a particular subnode (e.g., \$X.C has no subnode labeled Y), a value of zero is assumed and the sort is performed accordingly. Note also that the normal ordering is descending; that is, the largest value occurs first. Thus, after execution of ORDER \$PAYLOADS BY WEIGHT, the heaviest payload will be \$PAYLOAD(1). If ascending order is desired, the property name should be preceded by a minus sign (-), as in (c).

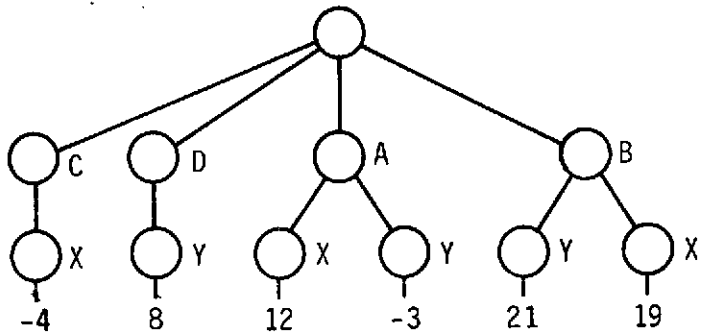
PLANS provides a special DO statement for the generation of all the combinations or permutations of a set, taken a specified number at a time. Thus, for example, one can write DO FOR ALL COMBINATIONS OF \$X TAKEN 2 AT A TIME, with the result that the DO-END group that begins with this statement will be executed once for each 2-element combination of the subnodes of \$X. The particular combination that is relevant during an iteration of this DO-END group may be referred to within the DO-END group by the reserved tree name \$COMBINATION (or \$PERMUTATION in the case of a DO FOR ALL PERMUTATIONS... statement). Figure 2.5-2 shows an example. \$X has three elements with labels A, B, and C.



(a) Original Tree



(b) Tree after ORDER \$X BY Y



(c) Tree after ORDER \$X BY -X

Fig. 2.5-1 ORDER Statements

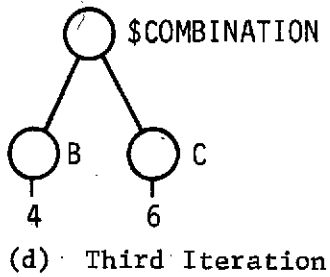
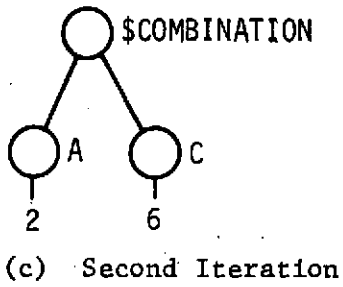
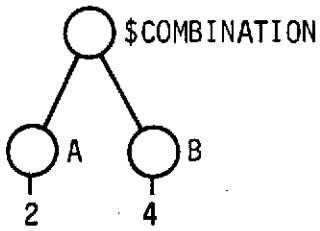
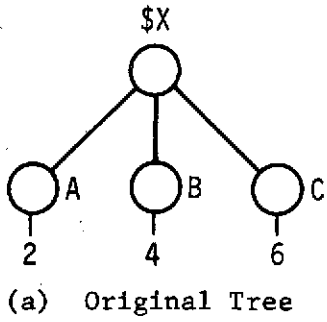


Fig. 2.5-2 Automatic Generation of Combinations

The combinations of these elements taken 2 at a time are {A, B}, {A, C}, and {B, C}. It will be observed that the tree \$COMBINATION contains the node (and, in fact, the entire substructure) of \$X.A and \$X.B during the first iteration of the DO-END group, \$X.A and \$X.C during iteration two, and \$X.B and \$X.C during iteration three. The DO-END group is automatically exited before iteration four, just like the usual DO I = 1 TO 3 statement. The combinations (or permutations) are generated in standard (lexicographic) order. They, therefore, provide a mechanism for automatic generation of combinations in standard sequence, and if a complete list of combinations of all sizes is needed, nested DO-END groups of the form

```
DO I = 1 TO NUMBER($X);
  DO FOR ALL COMBINATIONS OF $X TAKEN I AT A TIME;
  .
  .
  .
  END;
END;
```

can be used.

In order to provide efficient execution of these commands, the tree \$COMBINATION (or \$PERMUTATION) is not actually generated. The existing tree (in the example, \$X) from which the combinations are generated is actually used. \$COMBINATION is merely a convenient way of referencing only those subnodes that are involved in the current combination. Thus, although \$X(1) is the element A, \$X(2) is B, and \$X(3) is C, during the second iteration \$COMBINATION(1) is the element A, \$COMBINATION(2) is the element C, and there is no \$COMBINATION (3). The important point to

understand is that modifications made to \$COMBINATION are actually being made to \$X. Furthermore, no structural modifications are allowed at the base node level. In the example, one could change the value of \$COMBINATION(2) or add a substructure to it, but no immediate descendants of either \$X or \$COMBINATION may be added, deleted, or reordered inside the DO-END group.

2.6 A SIMPLE EXAMPLE

To assist the reader in gaining greater intuitive feeling for PLANS dynamic tree operations, a simple (but useful) PLANS program will now be considered in some detail. The program is called ORDER_BY_PREDECESSORS, and its function is to place a list of jobs, any one of which may have any of the others as a required predecessor, into an order such that the predecessors, if any, of each job occur earlier in the list than does the job itself. This function, fairly difficult in most programming languages, is very simple and straightforward in PLANS. While there are many functionally equivalent ways to write this program, one of the simplest and most efficient is as follows.

```
1 ORDER_BY_PREDECESSORS: PROCEDURE ($JOBLIST, $ORDERED_LIST) ;
2   DECLARE $TEMP, $NAME_LIST LOCAL ;
3 LOOP:
4   GRAFT $JOBLIST.FIRST:(ELEMENT.PREDECESSOR SUBSET OF $NAME_LIST)
5     AT $TEMP ;
6   IF $TEMP IDENTICAL TO $NULL THEN RETURN ;
7   $NAME_LIST(NEXT) = LABEL($TEMP) ;
8   GRAFT $TEMP AT $ORDERED_LIST(NEXT) ;
9   GO TO LOOP ;
END ORDER_BY_PREDECESSORS ;
```

Let us consider the execution of this program for a simple data case.

Notice, first (line 1), that the program is a called procedure with the explicit parameters \$JOBLIST and \$ORDERED_LIST. The calling program will initialize these trees as desired. Rather than attempting to reorder \$JOBLIST, ORDER_BY_PREDECESSORS will move the jobs, one at a time, from \$JOBLIST to \$ORDERED_LIST; so that the \$ORDERED_LIST will become a correct ordering of the jobs that were originally in \$JOBLIST. \$JOBLIST, on the other hand, will be returned null, assuming all goes well. Upon return from ORDER_BY_PREDECESSORS, then, the calling program will use \$ORDERED_LIST where \$JOBLIST was used before (or will GRAFT \$ORDERED_LIST AT \$JOBLIST) after checking \$JOBLIST for a null condition.

In line 2, \$TEMP and \$NAME_LIST are declared to be local trees. This means two things: (1) any use of these tree names within this procedure is entirely local, and will not affect trees of the same name outside this procedure, and (2) each time ORDER_BY_PREDECESSORS is called, \$TEMP and \$NAME_LIST will be initially null, and any storage they use will be made available for reuse upon return without any other action on the programmer's part.

Let us assume the initial data shown in part (a) of Fig. 2.6-1. \$JOBLIST describes a predecessor network in which job B has no predecessor jobs, jobs C and D must each be preceded by job B, and job A must follow both C and D. The diagram shows only information

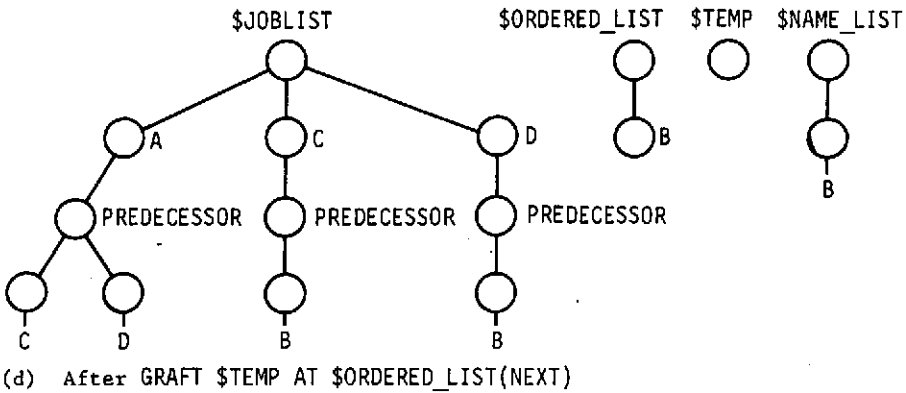
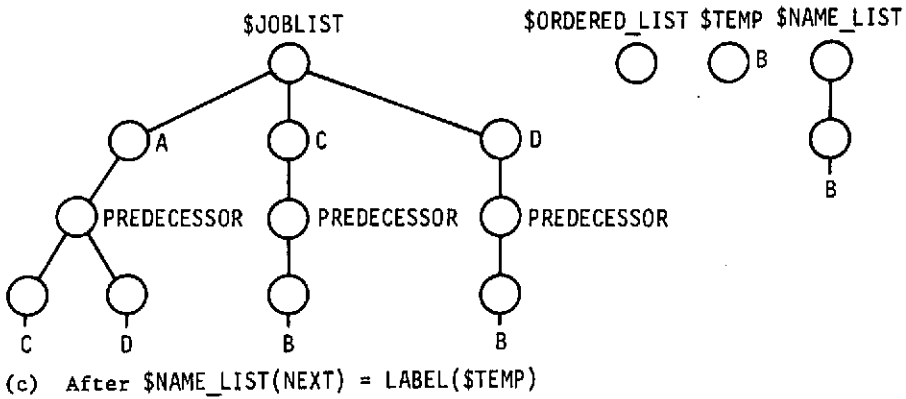
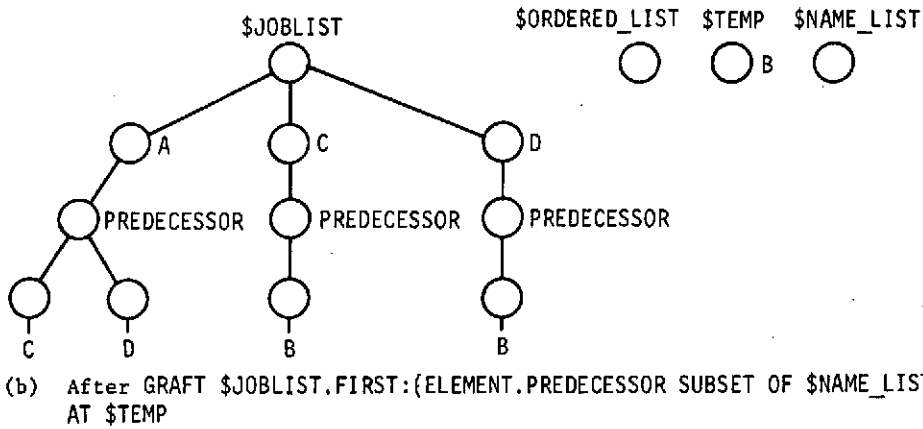
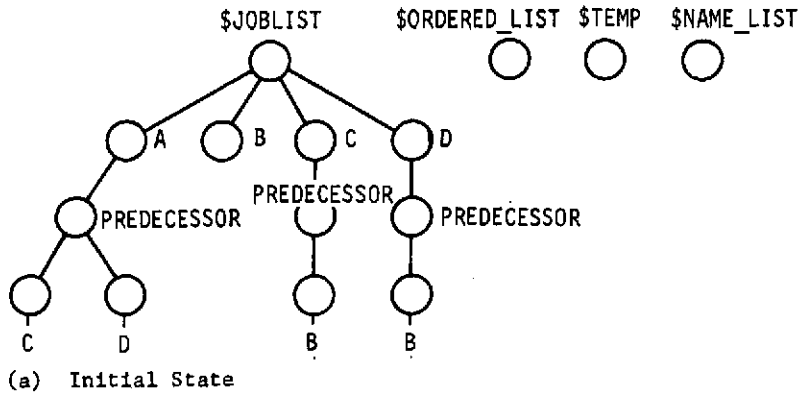


Fig. 2.6-1 ORDER_BY_PREDECESSORS: ITERATION 1

essential for present purposes. However, it is assumed that other information about each job (e.g., duration, resource requirements, etc) is also present. Because we can access predecessor information by label without regard for its ordinal position, any other information about these jobs is irrelevant, so long as the label PREDECESSOR is used only with the meaning assumed here. \$ORDERED_LIST is assumed to be null. Ordinarily this condition will be assured by the calling program. \$TEMP and \$NAME_LIST are automatically initialized to a null condition.

Consider now the effect of the GRAFT statement of line 4 on these trees. This statement specifies that a particular job is to be removed from \$JOBLIST and placed at \$TEMP. The job to be selected is to be the first job whose predecessor set is a subset of \$NAME_LIST. \$NAME_LIST will be used to collect the names of the jobs in \$ORDERED_LIST, so that the SUBSET OF relation can be used to automatically determine whether the predecessor requirement of a particular job is satisfied. Because \$NAME_LIST is presently null, the only job of \$JOBLIST that can possibly satisfy the conditional access is a job that has no predecessors. Note that job B fulfills this requirement, and that it is not necessary in this case that a node labeled PREDECESSOR even appear under job B because a nonexistent node has all the properties of a null node, including null subnode structure. Job B therefore satisfies the conditional access, and is removed from \$JOBLIST and placed at \$TEMP, as shown in Fig. 2.6-1(b). While the diagram includes no subnodes of the job B base node, that node *and all its*

substructure have replaced the previously null root node of the tree \$TEMP.

The statement of line 5 now tests for failure of the previous GRAFT statement. In the event that no subnode of \$JOBLIST satisfied the access condition, \$TEMP will now be null, and detection of this condition can be used to trigger return from ORDER_BY_PREDECESSORS. In the present case, however, \$TEMP is not null. A node is defined as null only if it either does not exist or has *both* a null value and a null label. Regardless of any substructure, the node \$TEMP now has the label "B" and is therefore not null, and no return occurs.

The statement of line 6 is therefore executed, placing the name of the job that was found into \$NAME_LIST. Several things should be noticed here. Since \$NAME_LIST is currently null, \$NAME_LIST(NEXT) is equivalent to \$NAME_LIST(1). LABEL(\$TEMP) is a character string. Therefore, a dummy node is established, with a null label, and placed at \$NAME_LIST(NEXT). The statement causes the job name, "B" to be a value of \$NAME_LIST(1), so that subsequent comparisons of \$NAME_LIST and PREDECESSOR nodes will find job names as values in both places.

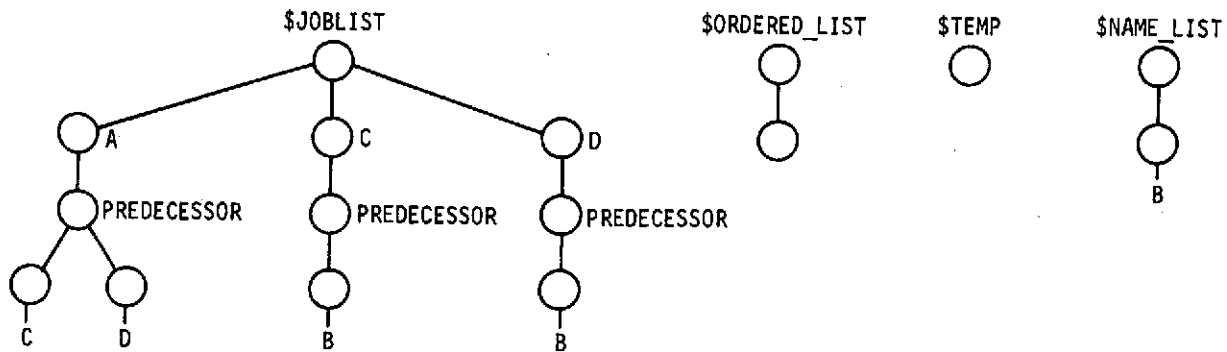
Finally, line 7 is executed, moving the found job, with all descriptive information, from \$TEMP to the next available position in \$ORDERED_LIST. Note that \$TEMP again reverts to a null condition. Trees always have root nodes, although they may be null. Thus, the removal of the node labeled "B" causes another (null) node to be placed at \$TEMP.

The program has now found the first job that can be executed and has moved it into `$ORDERED_LIST`. Another iteration is therefore initiated (line 8) by jumping back to `LOOP`.

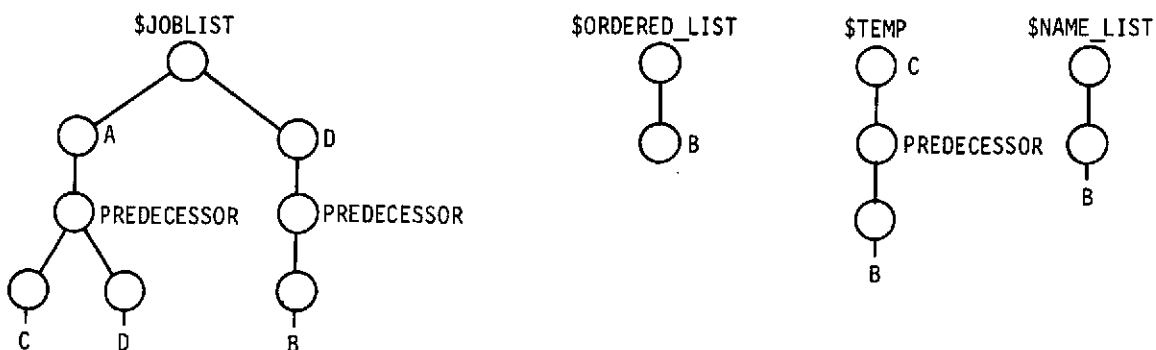
Figure 2.6-2 shows the tree manipulations that occur during iteration 2. The initial state, part (a), is the same as that at the end of iteration 1. The conditional `GRAFT` statement of line 4 again searches for a job whose predecessors, if any, are all named in `$NAME_LIST`. Since `$NAME_LIST` now contains the job name "B," either a job with no predecessors or a job with only the predecessor "B" will satisfy the access condition. The first such job now in `$JOBLIST` is job C, which is therefore grafted at `$TEMP` [Fig. 2.6-2(b)]. Because `$TEMP` is not null, no return is made at line 5.

The statement at line 6 places the name of the found job at the next available subnode of `$NAME_LIST`. As shown in Fig. 2.6-2(c), `$NAME_LIST` now contains the names of the two jobs (B and C) found so far. `$TEMP` is grafted (line 7) onto the next available position of `$ORDERED_LIST` [part (d)], which now contains all the information about jobs B and C (in that order) that was originally in `$JOBLIST`. Only the jobs not yet placed in `$ORDERED_LIST` still remain in `$JOBLIST`. Line 8 then causes another iteration to begin.

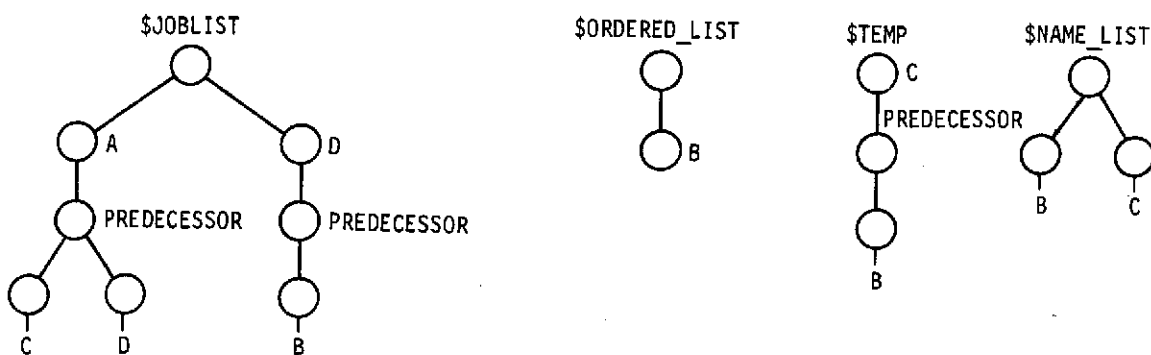
This process is repeated two more times, once for job D and once for job A, with the result shown in Fig. 2.6-3(a). All jobs have now been moved to `$ORDERED_LIST`. When the conditional `GRAFT` statement of line 4 is executed, no job will be found, and a null



(a) After Iteration 1



(b) After GRAFT \$JOBLIST.FIRST:(ELEMENT.PREDECESSOR SUBSET OF \$NAME_LIST) AT \$TEMP



(d) After GRAFT \$TEMP AT \$ORDERED_LIST(NEXT)

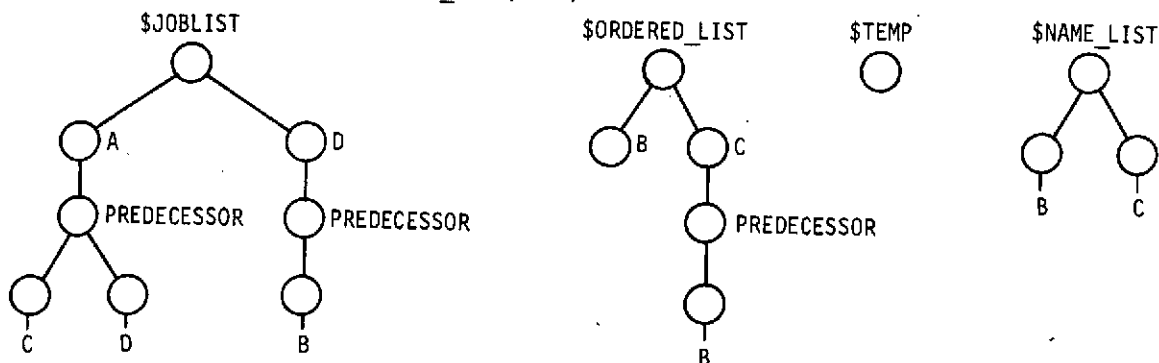
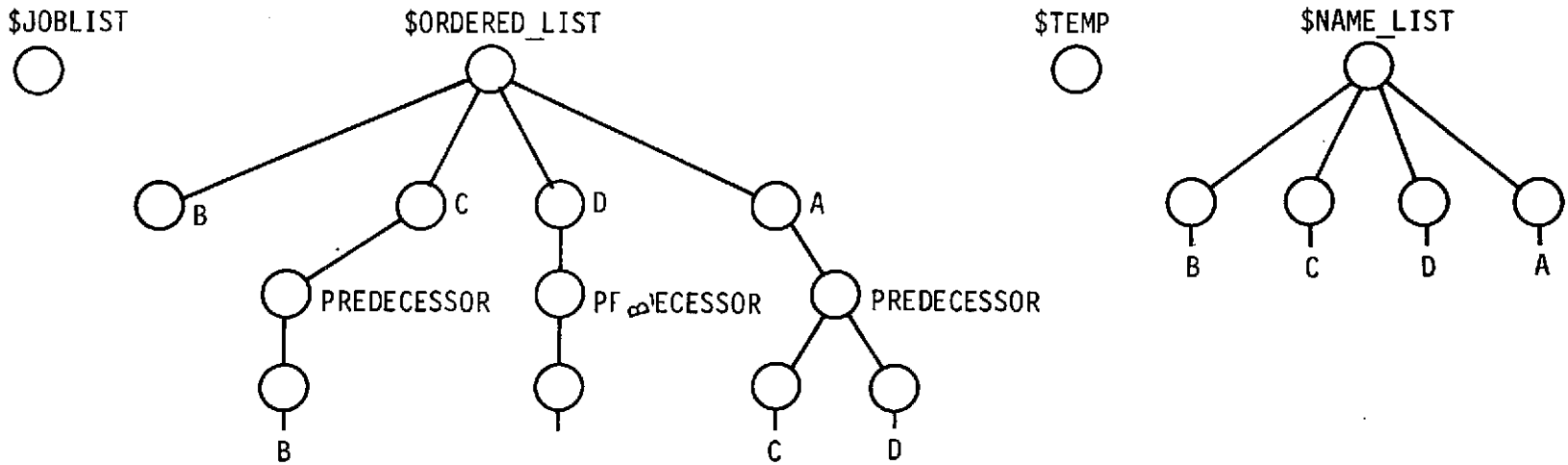
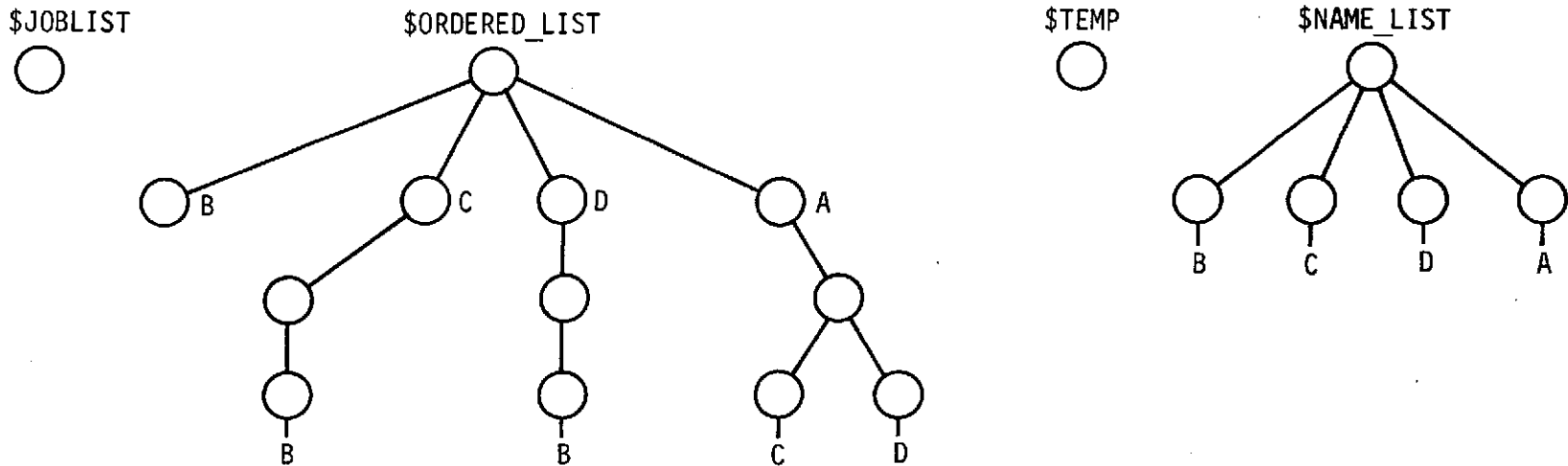


Fig. 2.6-2 ORDER_BY_PREDECESSORS: ITERATION 2

Fig. 2.6-3



(a) After Iteration 4



(b) After GRAFT \$JOBLIST.FIRST:(ELEMENT.PREDECESSOR SUBSET OF \$NAME_LIST) AT \$TEMP

Fig. 2.6-3 ORDER_BY_PREDECESSORS: ITERATION 5

node will be placed on \$TEMP (b). This condition allows the return to occur in line 5. \$JOBLIST and \$ORDERED_LIST will be returned to the calling program, while \$TEMP and \$NAME_LIST will be pruned automatically in order to free their storage.

It may occur to the reader to question the use of \$TEMP because a found job could be grafted (line 4) directly at \$ORDERED_LIST(NEXT). However, this would require two additional accesses to \$ORDERED_LIST(LAST), one to test for a null condition (line 5) and one to extract the label (line 6). In addition, before exit, the extra null node that would have been grafted onto \$ORDERED_LIST would have to be removed. It should always be borne in mind that node access time is a function of the number of sub-nodes that must be scanned (left to right) before the desired node is found. Thus, \$TEMP is more efficient to access than is \$ORDERED_LIST(LAST), and the difference is more pronounced as the \$ORDERED_LIST grows. Because GRAFT statements are very efficient, the use of \$TEMP is preferable here.

3.0 System Operations and Problem Modeling

-60-

3.0 SYSTEM OPERATIONS AND PROBLEM MODELING

The successful computer-aided solution of scheduling and resource allocation problems requires an integrated approach using three conceptual and analytical tools: (1) a computer programming language, (2) a descriptive model or representation of the system and its operations, and (3) problem solving approaches or decision making rules (algorithms). The scheduling language (PLANS) specified during this study supplies the first of these tools. The special features of PLANS that facilitate this effort have been described extensively in Chapter 2.0 of this volume as well as in Volumes I and III. A library of routines (modules) that perform operations model and algorithm functions is also specified in Volume III. This section discusses the use of the PLANS modules for problem description and solution.

3.1 A GENERIC SCHEDULING OPERATIONS MODEL

The scheduling modules like the language itself are applicable to complex systems in general and are not specific to the Space Shuttle. The reasons for, and expected benefits of, a generic approach are discussed in Volume I and are not repeated here. It is appropriate however to raise the question, "What are the consequences of using a generic modeling approach for the potential user of the PLANS programming system?"

3.1.1 Generic Modeling Consequences for the User

One price the PLANS user must pay to benefit from the general problem modeling approach developed by this study, is to learn a generic system description nomenclature. He must also learn to

recognize the functional elements, physical resources, and operational processes (activities) of his system and their interrelationships within the operations model framework. If he does this, he will gain, from the following pages, an understanding of how various techniques can be applied effectively using PLANS.

Before a generic approach to operational system and problem modeling is described, two important points should be stated.

- 1) There is no substitute for knowledge of the system that is to be modeled (and scheduled);
- 2) The *automation* of systems scheduling and resource allocation should not be accepted on an *a priori* basis without qualifications.

The significant increase in capabilities due to the power of the PLANS language and module library should not promote the misconception that the user will not need to think about his problem applications in depth.

Because a generic modeling approach was used to develop specifications, the PLANS module user will not find an input data structure including explicit labels such as, PAYLOAD NAME, PAYLOAD LENGTH, ORBITER SERIAL NUMBER, or ORBITER PAYLOAD BAY LENGTH. With the generic model, it is up to the user to know that these input items, when expressed generically, have the form RESOURCE: TYPE; NAME; PARAMETER; CLASS; etc. Similarly he does not find BRIEF CREW or CHECKOUT PAYLOAD, but finds instead PROCESS: NAME;

DURATION; TYPE; RESOURCES GENERATED; RESOURCES DELETED; and REQUIRED SOURCES; etc. All of these generic elements are described and properly related to each other in Section 3.4.

Note that nothing prevents the use of PLANS to program models and algorithms with explicit RESOURCE or PROCESS labels, if the user is willing to trade the flexibility and other benefits of the generic approach for the presumed advantages of explicit labels in the basic program structure and logic. However, it should be understood that although generic labels are used in a basic program code, the data for that code will contain the problem-specific data in explicit terms and thus will provide descriptive detail in a user-oriented format. This compensates for the use of generic labels in the program logic while preserving the flexibility of that logic.

Even more significant than the perception of system elements in terms of generic elements, is the choice of correct resources and processes for the model and the determination of the appropriate level for resource and process description. Selection of these items requires a knowledge of the system as well as a knowledge of the approach that will be used to obtain a solution.

3.1.2 Elements of the Generic Scheduling Operations Model

Drawing upon the Shuttle system as an example, it can be recognized that the description of Shuttle operations is characterized mainly by interrelated sets of activities or processes that integrate various physical system resources to achieve a

final launch configuration. This configuration must, of course, be capable of achieving specific objectives. Because scheduling is of primary concern, the parameters used to describe system operations consist of a set of *processes* that require specific *resources* for particular *time intervals*. The association of specific resources with a time interval constitutes a schedule unit, i.e., a basic element of a schedule. The concept of using processes to associate resources into schedule units is illustrated in Fig. 3.1-1.

The recognized technique to describe relationships between activities or events is the network or flow diagram. Figure 3.1-2 is a sample diagram that illustrates the flow of Shuttle mission operations on a top-level basis. Such diagrams visually depict one or more *operations sequences* in terms of predecessor-successor relationships and also contribute to the recognition of more general temporal relations.

A fundamental concept of the generic operations model is that for scheduling and resource allocation purposes an operational system can be described in terms of RESOURCES, PROCESSES and OPERATIONS SEQUENCES. This concept leads directly to the specification of standard PLANS data structures that contain descriptive information arranged with the same logical separation. The operations model data structures are described in detail in Section 3.5.

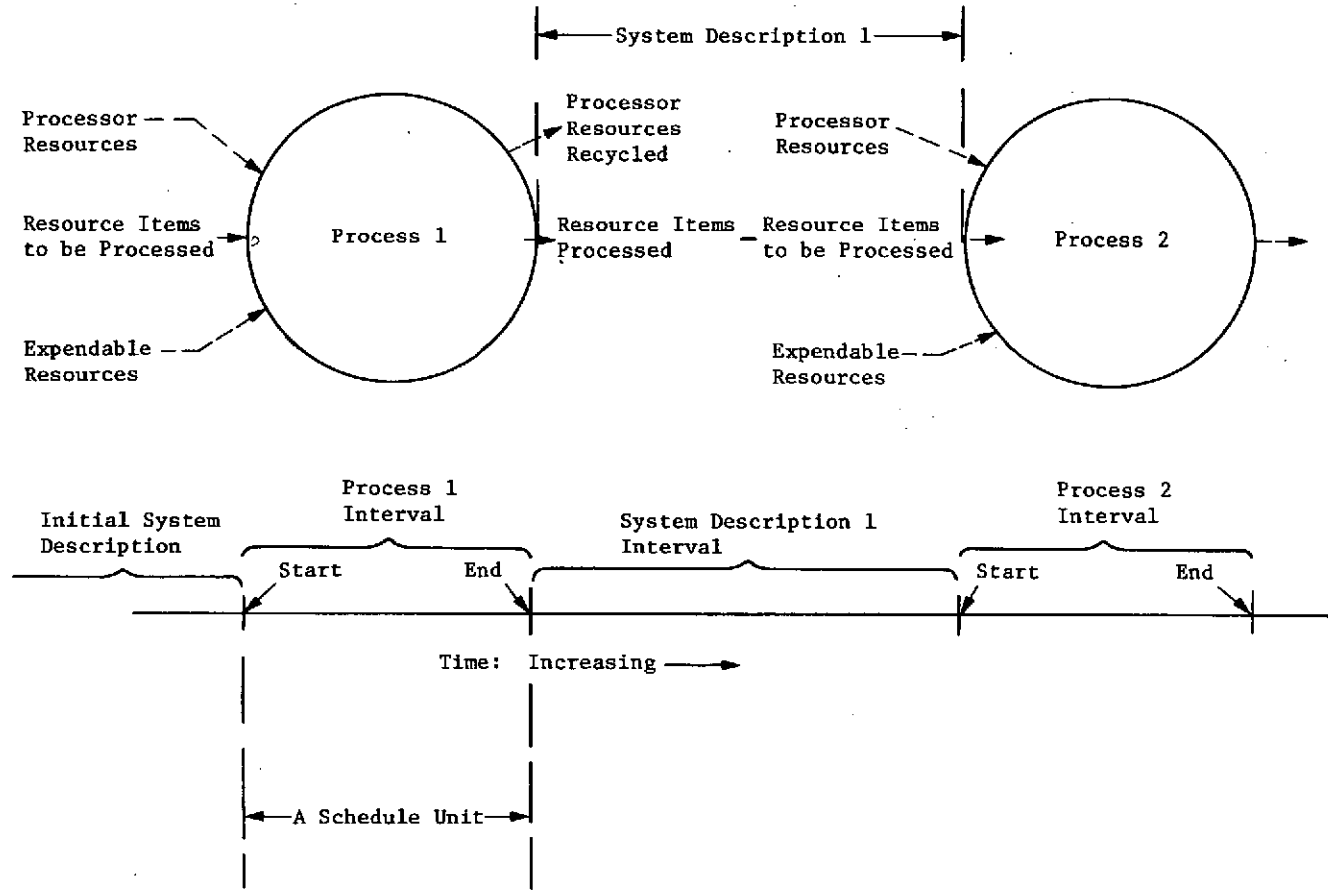


Fig. 3.1-1 Scheduling Operations Model Fundamental Concepts

Fig. 3.1-2

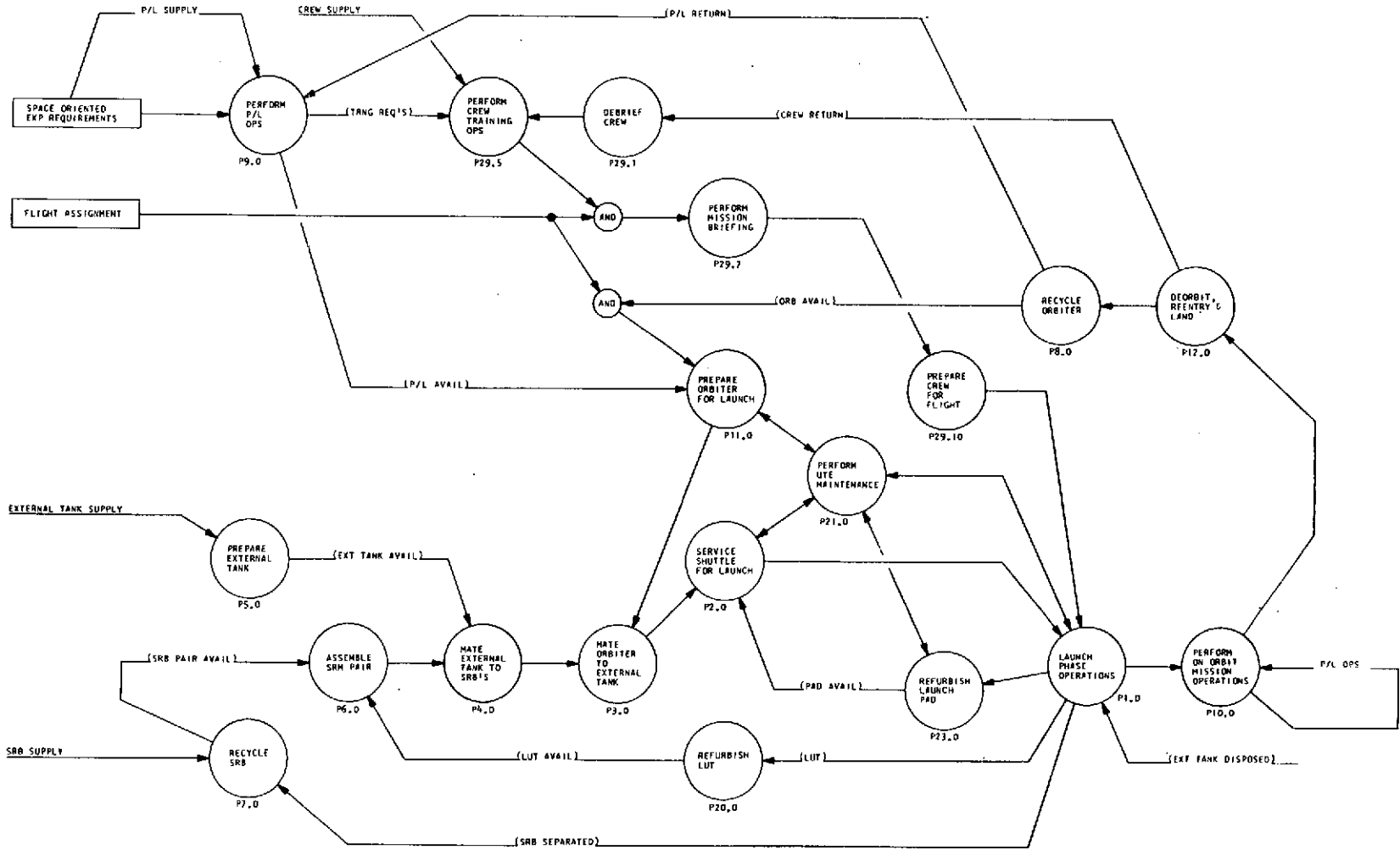


Fig. 3.1-2 Sample Network Flow Diagram for System Operations Description, Shuttle Mission

The use of the generic operations model concept requires not only an understanding of the conventions for describing the system to be scheduled, but also the conventions used to establish any procedural logic associated with the model. In particular, a conceptual framework is needed for understanding how descriptive information is communicated to decision logic algorithms to solve a scheduling problem. The roles of the model and the algorithm may be interpreted in terms of a dialog; the algorithms request information about a system and its operation on which to base a scheduling decision and the operations model supplies the data. Any functions that must be performed to supply the algorithms with appropriate model information can be regarded as operations model functions. A typical integration of operations model functions and algorithm functions is shown in Fig. 3.1-3. Annotations on the figure relate to the OSARS (NASA-MPAD) program currently used to assign resources to flights.

The logical separation between operations model functions and algorithm functions serves to define logic boundaries for the PLANS module library. The user who perceives this distinction and who uses the descriptive conventions of the generic operations model will find that the PLANS module library will provide many powerful capabilities that are easily incorporated into his PLANS program.

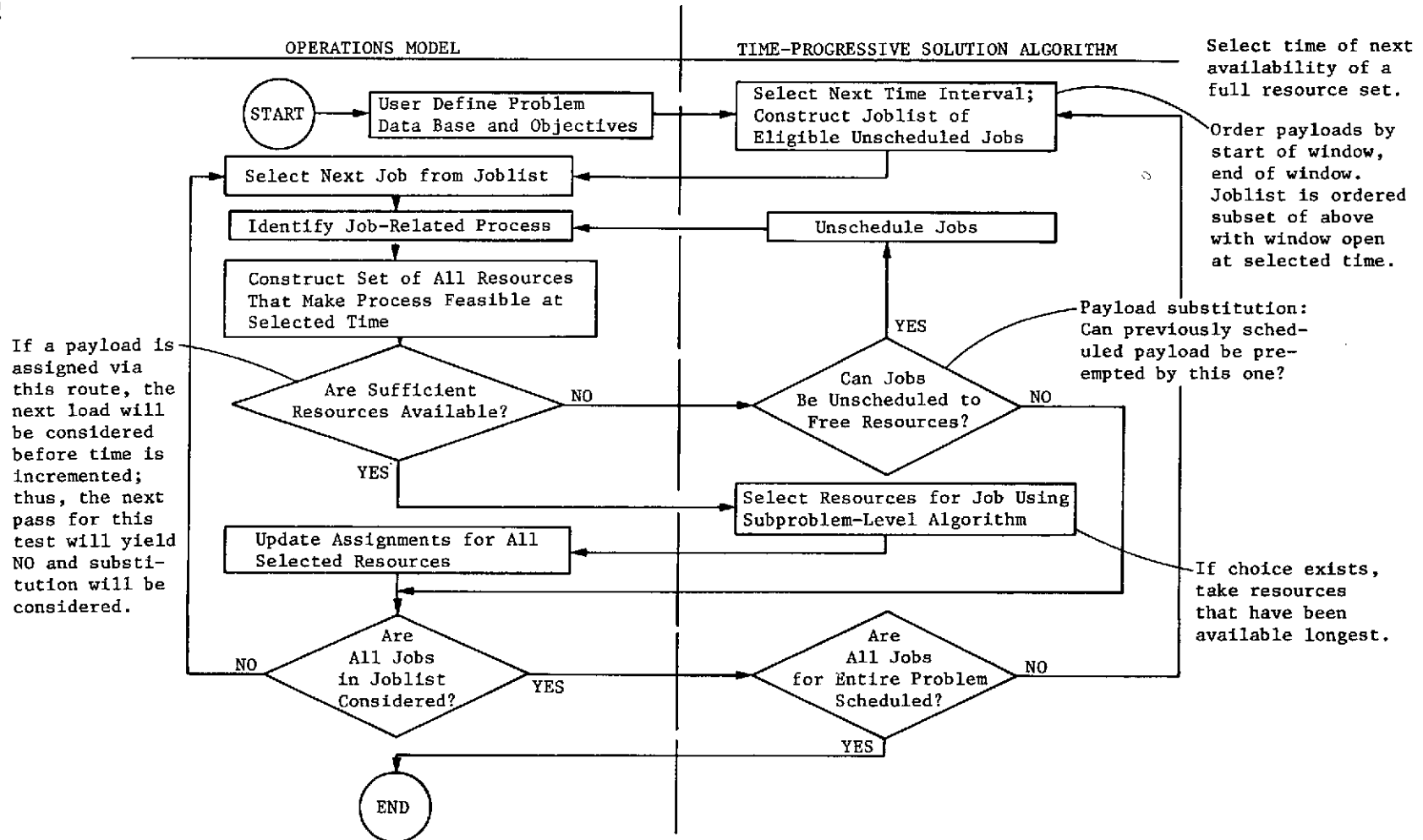


Fig. 3.1-3 Operations Model/Solution Algorithm Interface Example with OSARS Annotation

3.2 Problem Types Accommodated by the Operations Model

68-a

3.2 PROBLEM TYPES ACCOMMODATED BY THE OPERATIONS MODEL

To assist the reader whose interest is in a special problem or class of problems, this section states some typical problems without detail, characterizes those problems in terms of the operations model described in Section 3.4, and suggests the subsections where modeling details can be found. This information is provided in Table 3.2-1. Each problem type in Table 3.2-1 could require any or all of the descriptive generality contained in Section 3.4. The reader should not assume that the subsections referenced contain the only relevant material.

Table 3.2-1

Use of PLANS Generic Operations Model for Describing Typical Problem Classes

Problem Class	Typical Plans Operations Model Characterization	Sections of Volume II Giving More Detail	
		Paragraph	Page
Project Planning & Control	Predecessor networks, splittable jobs, pooled resources	3.4.1 thru 3.4.6 3.6	
Crew Activity Timelining	General temporal relations with item-specific resources	3.4.7 3.4.10 3.4.11	
Cargo Containerization	Item-specific resources, quasi-enumerative solution techniques	3.4.7 3.6	
Facility Utilization	Predecessor network with item-specific resources	3.4.1 3.4.2 3.4.7	
Transportation System Scheduling	General temporal relations with pooled resources	3.4.3 3.4.10	
Plant Production Scheduling	Predecessor networks with pooled resources	3.4.1 thru 3.4.6 3.6	
Computer Job Scheduling	Splittable jobs pooled resources	3.4.1 thru 3.4.5	
Payload Compatibility Grouping	Item-specific resources, quasi-enumerative solution techniques	3.4.7 3.6	
Personnel Resource Allocation	Pooled resources, explicit descriptors for pooled resources	3.4.3 3.4.12	
Experiment Selection & Equipment Allocation	Item-specific & pooled resources	3.4.3 3.4.7	

3.3 PLANS Library Module

70-a

3.3 PLANS LIBRARY MODULES

Functional specifications for a library of routines, called modules, have been developed in parallel with the specifications for the programming language (PLANS). The contents of such a library of modules have been integrated using the generic operations model concepts described in Section 3.1. Decisions on what level of detail to include are necessarily somewhat arbitrary and based on nonquantifiable judgments. However, the current modules have resulted from a functional analysis of many classes of scheduling problems using the following criteria:

- 1) Each module in the current specification should be limited to a single logical function. Although it is possible to group several of the specified modules together, based on high-level functional similarity, to do so either restricts flexibility or increases the computational inefficiency of the functions represented. Therefore, the modules specified for the program library perform a single separable logical function.
- 2) Each module specified performs a function that is common or likely to occur in developing typical scheduling software. Although this criteria seems self-evident, it is easy to conceptualize numerous modules that are applicable to only an infrequent special case or that are required only in unenlightened or highly encumbered approaches to a scheduling problem. In those cases where a function would be required

by one approach to a problem but not required by an alternative and clearly superior approach, a module to perform that function has not been specified.

- 3) Each module specified does not contain judgments or decision-making logic for which the criteria are open to opinion. For example, no modules assume specific economic models, queuing service policies, or criteria for resolving resource alternatives. There are no approximations of dependent variables by polynomials or piecewise-linear functions buried in any module logic. These judgmental matters have been considered too problem-dependent and inflexible for an initial library specification. Because of the criterion for functional simplicity and separability [criterion 1) above], the specified operations model modules perform elementary operations and generally return information on which decisions can be made rather than the decisions themselves. The modules that are specified as algorithms make simple decisions based on quantitative criteria that are easily perceived by the user. A clear distinction has been attempted between simple decision-making modules (i.e., algorithms) and information providing modules (the operations model) so that all of the latter are equally applicable whether exercised interactively by a user making real-time decisions or in a batched system design where algorithm modules make the scheduling decisions.

The criteria stated are appropriate for specifying the first modules to be placed in a program library because flexibility and commonality of application are prime considerations. It is not intended to imply, however, that future additions to the module library should be restricted by these criteria. Analyses are currently underway that will lead to the specification of higher-level modules. Such modules will combine the functions of many of the currently specified modules through special purpose executive logic. Special attention is being paid to methods for translating generalized problem formats into the more restrictive structures required by existing solution methodologies.

Table 3.3-1 contains a brief description of the modules specified in this study. Detailed functional descriptions for all modules are provided in the sections of Volume III indicated in the table. The reader who is interested in the use of a particular module should refer to the sections indicated in the table.

Table 3.3-1 Summary of PLANS Module Library Contents

Module Name	Brief Description of Intended Usage	Volume II References		Volume III Specifications
		Paragraph	Page	
DURATION	Calculates the duration of any standard (simple or multiple) interval			2.4.1
ENVELOPE	Calculates an interval that is the smallest cover of a given standard (simple or multiple) interval			2.4.2
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.			2.4.3
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.			
FIND_MAXIMUM	Finds the maximum value in a numeric set and all the elements that have that maximum value.			2.4.5
FIND_MINIMUM	Finds the minimum value in a numeric set and all the elements that have that minimum value.			2.4.6
CHECK_FOR_PROCESS_DEFINITION	Checks for input data consistency, i.e., checks that all operations sequences named in \$OBJECTIVES are defined in \$OPSEQUENCE and that all processes in those operations sequences or in \$OBJECTIVES are defined in \$PROCESS.			2.4.7
GENERATE_JOBSET	Creates individual jobs for each occurrence of a process specified explicitly or via an operations sequence in \$OBJECTIVES. Merges information contained in \$OBJECTIVE, \$OPSEQUENCE, and \$PROCESS into a tree called \$JOBSET. Jobs in \$JOBSET are ready for the decision algorithms to make explicit assignments.	4.3		2.4.8
CHECK_EXTERNAL_TEMP_RELATIONS	Identifies temporal constraint violations that would occur if two sets of job assignments were merged. Useful for checking if a potential assignment will be consistent with existing assignments.	3.4.10		2.4.9
CHECK_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.	3.4.10		2.4.10
CHECK_ELEMENTARY_TEMP_RELATION	Checks satisfaction of a single binary temporal relation given specific assignments for the two jobs named in the temporal relation.	3.4.10		2.4.11

Table 3.3-1 (cont)

Module Name	Brief Description of Intended Usage	Volume II References		Volume III Specifications
		Paragraph	Page	
NEXT_SET	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.	4.4		2.4.12
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.	4.3		2.4.13
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.	3.4.12		2.4.14
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.			2.4.15
UPDATE_RESOURCE	Records the scheduling of a schedule unit (job) by writing assignments in \$RESOURCE for all resources used in the schedule unit.	4.4		2.4.16
WRITE_ASSIGNMENT	Writes a single assignment for a resource and adds the assignment node in chronological order in \$RESOURCE.	4.3		2.4.17
UNSCHEDULE	Deletes assignments from \$RESOURCE for all resources associated with a specified job to be deleted.			
COMPATIBILITY_SET_ GENERATOR	Enumerates all compatible subsets of an input set using externally supplied compatibility criteria.	3.6		2.4.19
FEASIBLE_PARTITION_ GENERATOR	Generates all sets of integers with a given number of elements that sum to a given total. Useful in fathoming many branches in enumerative heuristics.	3.6		2.4.20
PROJECT_DECOMPOSER	Identifies at subprojects within a project description; i.e., finds subnetworks that contain all predecessors and successors of its member activities.	3.4.1 3.6		2.4.21
REDUNDANT_ PREDECESSOR_ CHECKER	Identifies and eliminates redundant specifications of predecessors in \$JOBSET; e.g., in A < B B < C A < C, the last specification is redundant.	3.4.1 3.6		2.4.22

Table 3.3-1 (cont)

Module Name	Brief Description of Intended Usage	Volume II References		Volume III Specifications
		Paragraph	Page	
CRITICAL_PATH_CALCULATOR	Calculates early and late start and finish times as well as total and free float in a network of jobs.	3.4.2, 3.4.4 3.6 4.3		2.4.23
PREDECESSOR_SET_INVERTER	Creates, from a set of jobs with predecessors, an equivalent set of jobs with successors. Used in critical path analyses.	3.4.4 3.6		2.4.24
NETWORK_CONDENSER	Eliminates activities (jobs) from a network leaving only events linked by critical delays as branches.	3.4.2		2.4.25
CONDENSED_NETWORK_MERGER	Merges two condensed networks into a single composite condensed network, and computes the critical path data for the composite network.	3.4.2		2.4.26
NETWORK_ASSEMBLER	Assembles a master network from sub-networks with interfacing events. The relations between the subnetworks may be more general than those describable by nesting operations sequences.	3.4.2 3.4.4 3.4.6 3.6		2.4.27
CRITICAL_PATH_PROCESSOR	Condenses, merges and computes critical path data for a master network. Performs executive function, which calls NETWORK_CONDENSER, CONDENSED_NETWORK_MERGER, and CRITICAL_PATH_CALCULATOR.	3.4.2 3.4.4 3.6		2.4.28
NETWORK_EDITOR	Identifies and eliminates both redundant predecessors and cycles specified in the specification of precedence networks. Performs executive function, which calls ORDER_BY_PREDECESSOR and REDUNDANT_PREDECESSOR_ELIMINATOR.	3.4.1		2.4.29
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.	3.4.11		2.4.30
ORDER_BY_PREDECESSOR	Produces a list of jobs where all jobs appear in the list only after all their predecessor have appeared; i.e., produces a nonunique technological ordering.	3.4.1 3.6 4.1		2.4.31
RESOURCE_ALLOCATOR	Allocates resources to jobs to satisfy all resource constraints and heuristically produce a minimum duration schedule. Uses project scheduling problem model.	3.4.6 3.6 4.3		2.4.32
RESOURCE_LEVELER	Reallocates resources to smooth the usage of resources while maintaining schedule constraints. Uses project scheduling problem model.	3.4.6 3.6		2.4.33

Table 3.3-1 (concl)

Module Name	Brief Description of Intended Usage	Volume II References		Volume III Specifications
		Paragraph	Page	
HEURISTIC_SCHEDULING_PROCESSOR	Performs both time-progressive resource allocations/job scheduling and resource leveling. Performs executive function for RESOURCE_ALLOCATOR and RESOURCE_LEVELER. Uses project scheduling problem model.	3.4.5 3.4.6 3.6		2.4.34
GUB_LP	Solves special-purpose linear programs that arise as simplified models of transportation, distribution, and multi-item scheduling problems. Uses generalized upper bounding LP format.	3.6		2.4.35
MIXED_INTEGER_PROGRAM	Solves linear programs that contain both continuous and integer-valued decision variables.	3.6		2.4.36
PRIMAL_SIMPLEX	Solves linear programs that arise in the process of solving scheduling and resource allocation problems with multi-level algorithms.	3.6		2.4.37
DUAL_SIMPLEX	Solves dual linear programs that arise as a result of the structure of multi-level scheduling and resource allocation algorithms. Uses primal simplex format.	3.6		2.4.38
INTEGER_PROGRAM	Solves the linear form of the binary decision making problem.	3.6		2.4.39

3.4 Problem Description Using the Operations Model

- 78 -

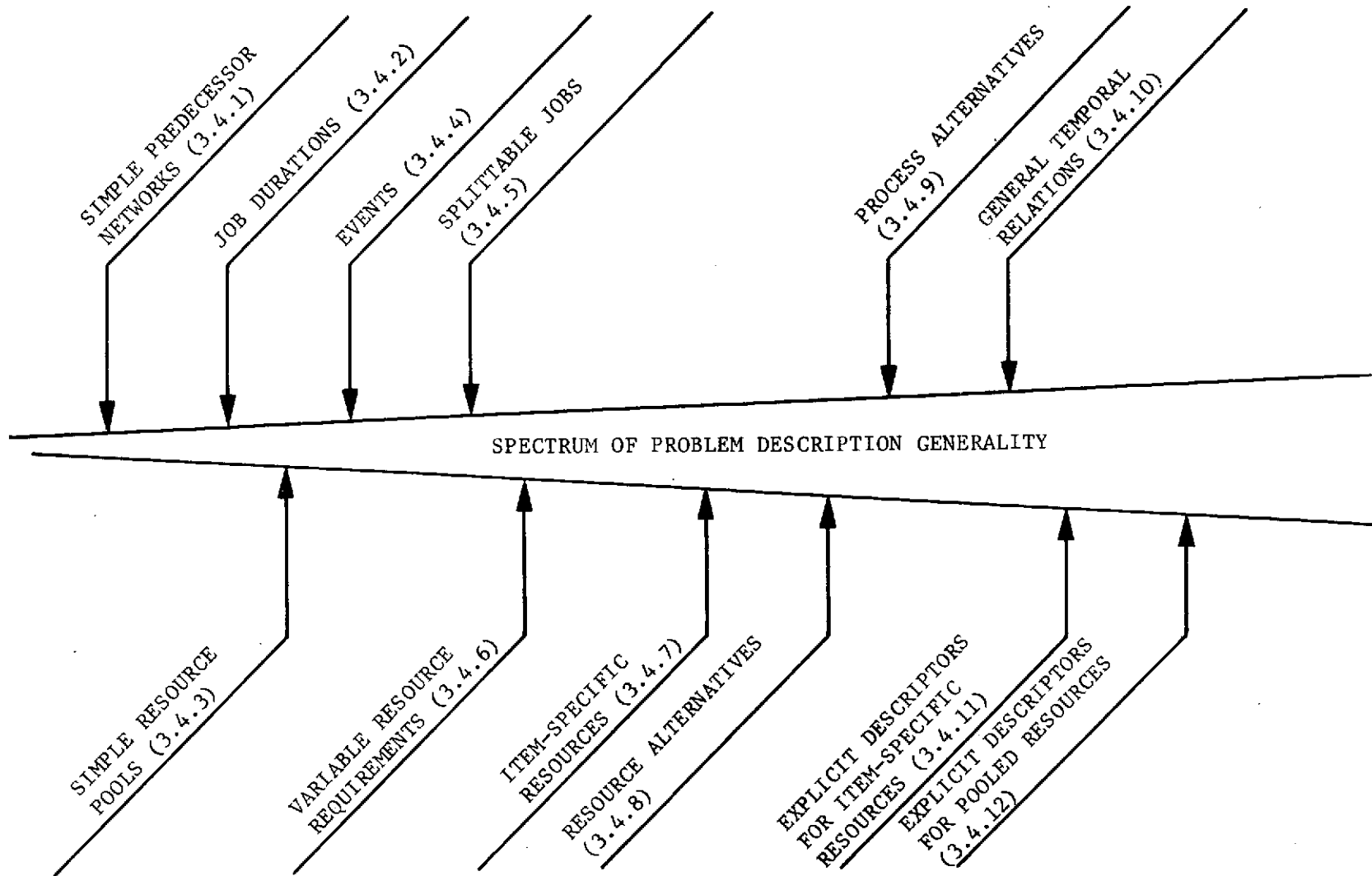
3.4 PROBLEM DESCRIPTION USING THE OPERATIONS MODEL

This section classifies scheduling problem descriptions. The material is presented in sequence from the simplest scheduling problem descriptions to the most complex as illustrated in Fig. 3.4-1. Each section briefly discusses an additional generality to a problem description that results in a more complex problem from the point of view of programming logic and/or solution method. The sequence of presentation is a logical one, but the section does not require beginning-to-end reading to enable the identification of descriptive characteristics appropriate for any particular problem. Furthermore, the presentation sequence does not imply that any specific problem description will include all generalities to the left of a particular point in Fig. 3.4-1.

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS

80

Fig. 3.4-1



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

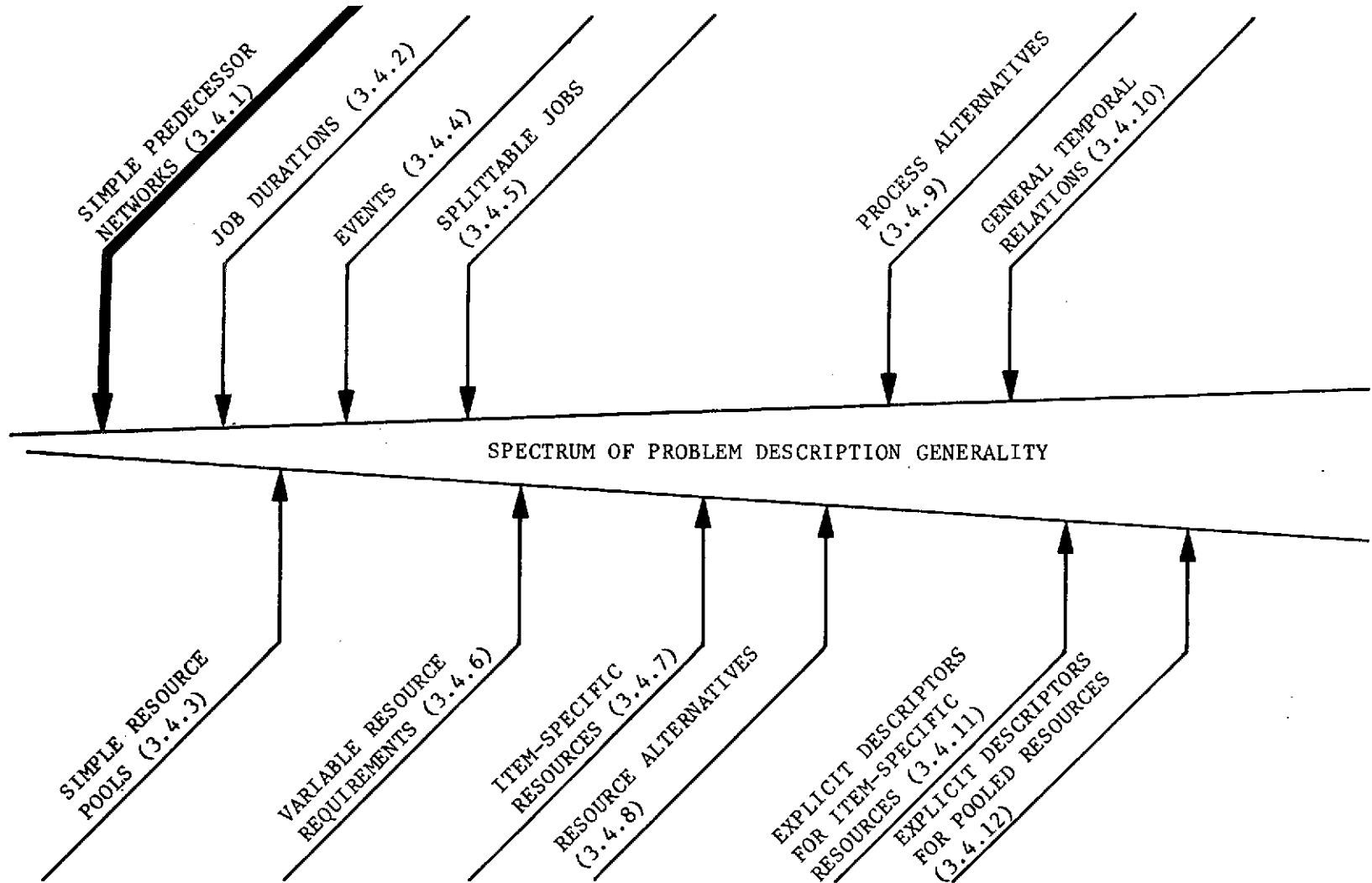
Fig. 3.4-1 Problem Description Using the Operations Model

3.4.1 Simple Predecessor Networks

80-a

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS

9-08



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.1 Simple Predecessor Networks

The simplest scheduling problem model permits only simple predecessor relationships between jobs. Job A is a predecessor of Job B if, and only if, Job A is completed at a time before or equal to the start of Job B. This simple relationship permits an entire precedence network to be completely defined by a list of jobs with an associated list of predecessors for each job. A simple precedence network is illustrated in Fig. 3.4.1-1. The information contained in the network diagram of Fig. 3.4-1 is shown in Table 3.4.1-1.

Table 3.4.1-1 Basic Information of a Predecessor Network

Job	Predecessors
Mate External Tank to SRBs	--
Mate Orbiter to External Tank	Mate External Tank to SRBs Perform Payload Operations
Refurbish Launch Pad	--
Perform Payload Operations	--
Perform Crew Training	Perform Payload Operations
Launch	Perform Crew Training Mate Orbiter to External Tank Refurbish Launch Pad
Refurbish Launch Pad	--

Fig. 3.4.1-1

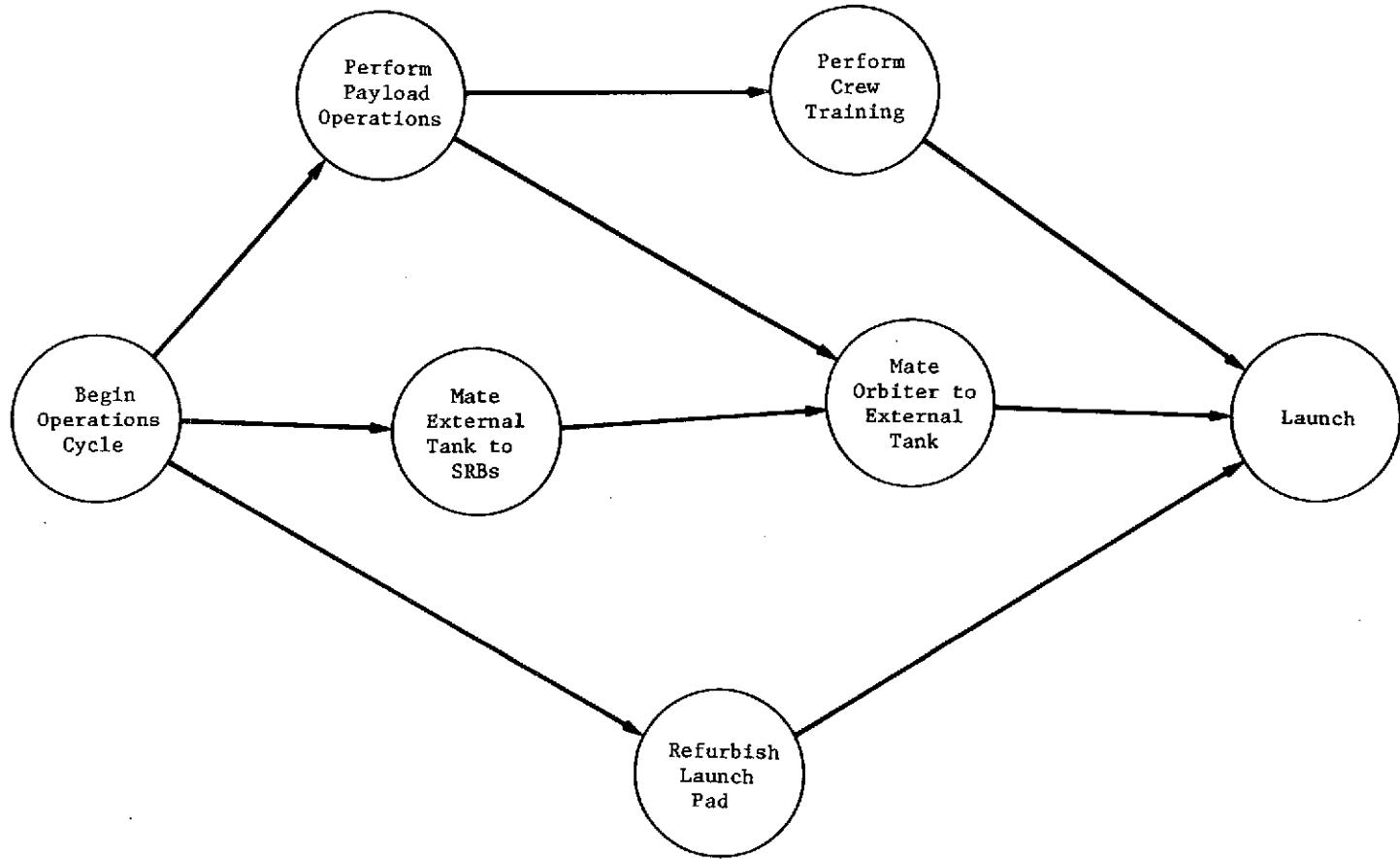


Fig. 3.4.1-1 A Predecessor Network Example

Elementary operations can be performed with the simple precedence network defined. If, in the example, "Perform Payload Operations" were given as a predecessor of "Launch," the specification is redundant as long as "Mate Orbiter to External Tank" is specified as a predecessor of "Launch." In more complex networks, redundant specifications are easily constructed. Therefore, the module library includes a module, REDUNDANT_PREDECESSOR_CHECKER, that will detect and remove a redundant specification.

It is also common to inadvertently specify a loop in a network. This is illustrated below:

```
JOB A
  PREDECESSOR
    JOB B
JOB B
  PREDECESSOR
    JOB C
JOB C
  PREDECESSOR
    JOB A
```

The module library contains a module NETWORK_EDITOR that detects and eliminates cycles or loops in a network.

A list of jobs and their associated predecessors that constitute a precedence network may be ordered so that each job appears on the list only after all its predecessors have appeared. The simple illustration network can be presented in such an ordering as shown in Table 3.4.1-2.

Table 3.4.1-2
Ordering Network Jobs by
Predecessors

Perform Payload Operations
Mate External Tank to SRBs
Refurbish Launch Pad
Perform Crew Training
Mate Orbiter to External Tank
Refurbish Launch Pad
Launch

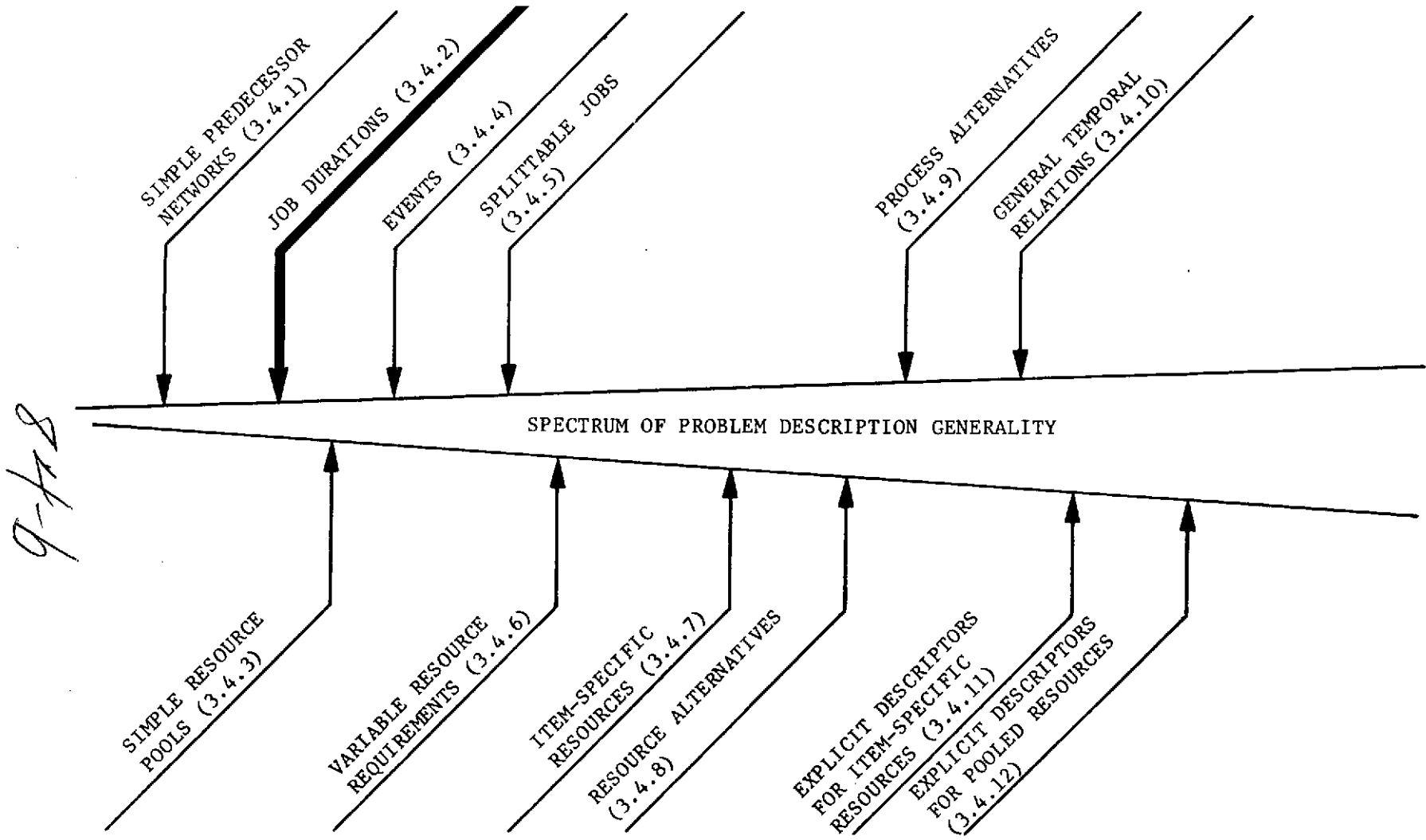
Obviously this ordering is not unique, it represents, however, a sequence in which jobs could be completed without violating any precedence constraints. The module `ORDER_BY_PREDECESSORS` produces a proper ordering. An ordering that produces the sequence in which all predecessor relationships are satisfied is called a *technological* ordering.

3.4.2 Networks with Job Durations

84-a

C2

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.2 Networks with Job Durations

If job durations are added to the network information, additional computations can be made. The minimum time schedule can be determined for this problem model. All paths in the network for which no unscheduled time is possible in the minimum time schedule are called *critical paths*. All other paths contain slack, i.e., time intervals associated with one or more jobs within which the start times may be altered without causing a delay to the minimum time schedule. A simple illustration of these definitions is shown in Fig. 3.4.2-1. All CPM (Critical Path Method) and PERT (Project Estimation and Review Technique) analyses are based on simple networks containing jobs with fixed durations and predecessor sets. The PLANS module library contains five modules that perform computations on CPM problem models. They are:

- 1) CRITICAL_PATH_PROCESSOR
- 2) CRITICAL_PATH_CALCULATOR
- 3) NETWORK_CONDENSER
- 4) CONDENSED_NETWORK_MERGER
- 5) NETWORK_ASSEMBLER

The first is an executive routine and the second calculates parameters for a simple network. The others provide computations associated with more general networks that are addressed in a subsequent section.

Fig. 3.4.2-1

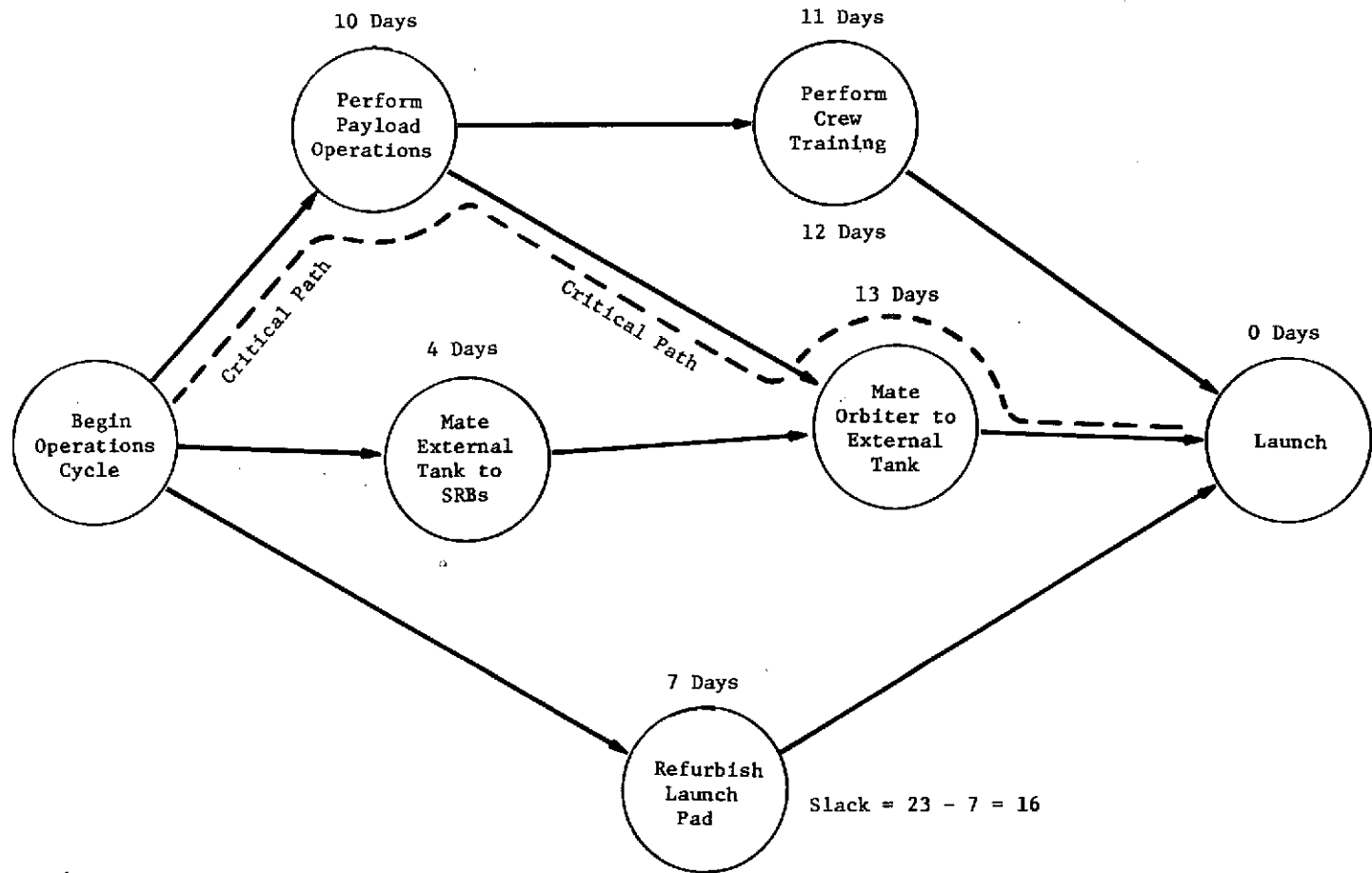
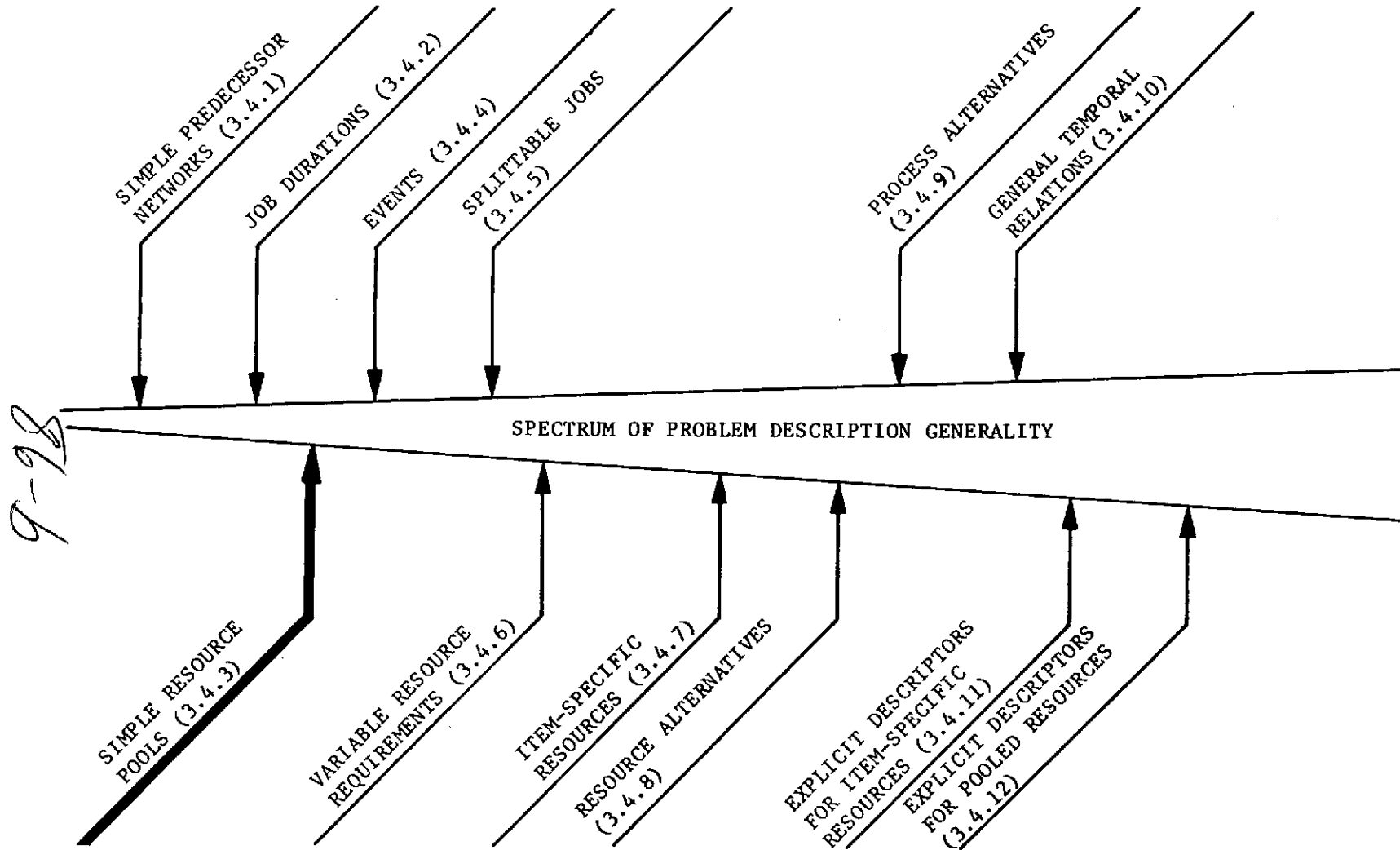


Fig. 3.4.2-1
Scheduling with a Predecessor Network That Includes Job Durations

3.4.3 Simple Resource Pools

86-a

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.3 Simple Resource Pools

If each job in the problem model requires a specified number of resource units as illustrated in Fig. 3.4.3-2, then the network analysis becomes a project scheduling problem. Stated simply, *project scheduling is network scheduling plus resource/job relationships*. In project scheduling, the relationships between each job and its required resources are quite simple. Each job requires a fixed number of resource units of one or more types. A *pool* exists for each type. A job may be scheduled if the pools contain the required number of resource units for the duration of the job. An illustration of a feasible project schedule is shown in Fig. 3.4.3-1 for two resource types and three jobs.

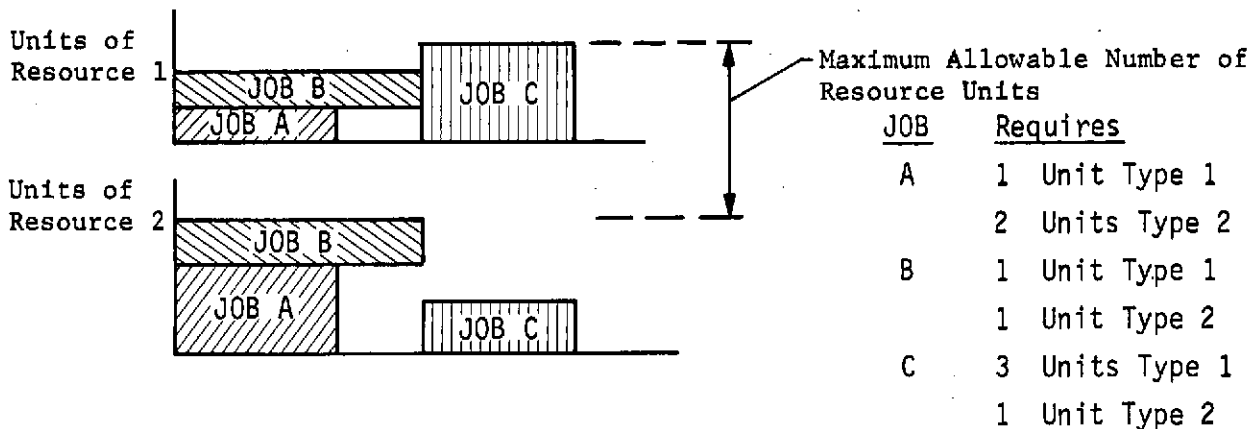


Fig. 3.4.3-1 Feasible Project Scheduling Example

Notice that if the total number of resources had not been constrained, all three jobs could have been scheduled concurrently producing a shorter schedule. Thus adding resource constraints to the problem model usually does alter the CPM (i.e., simple network with job durations) schedule.

Fig. 3.4.3-2

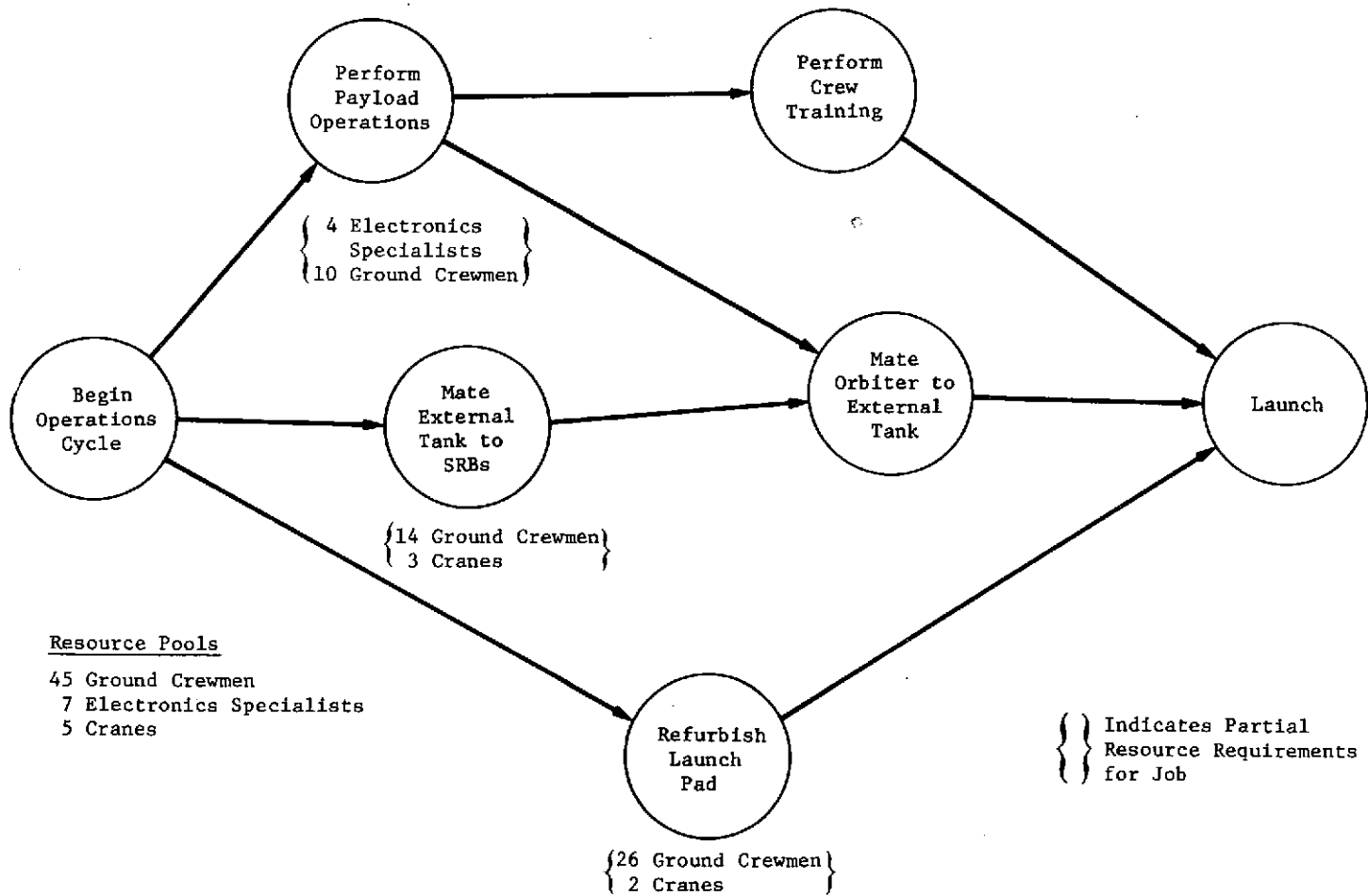


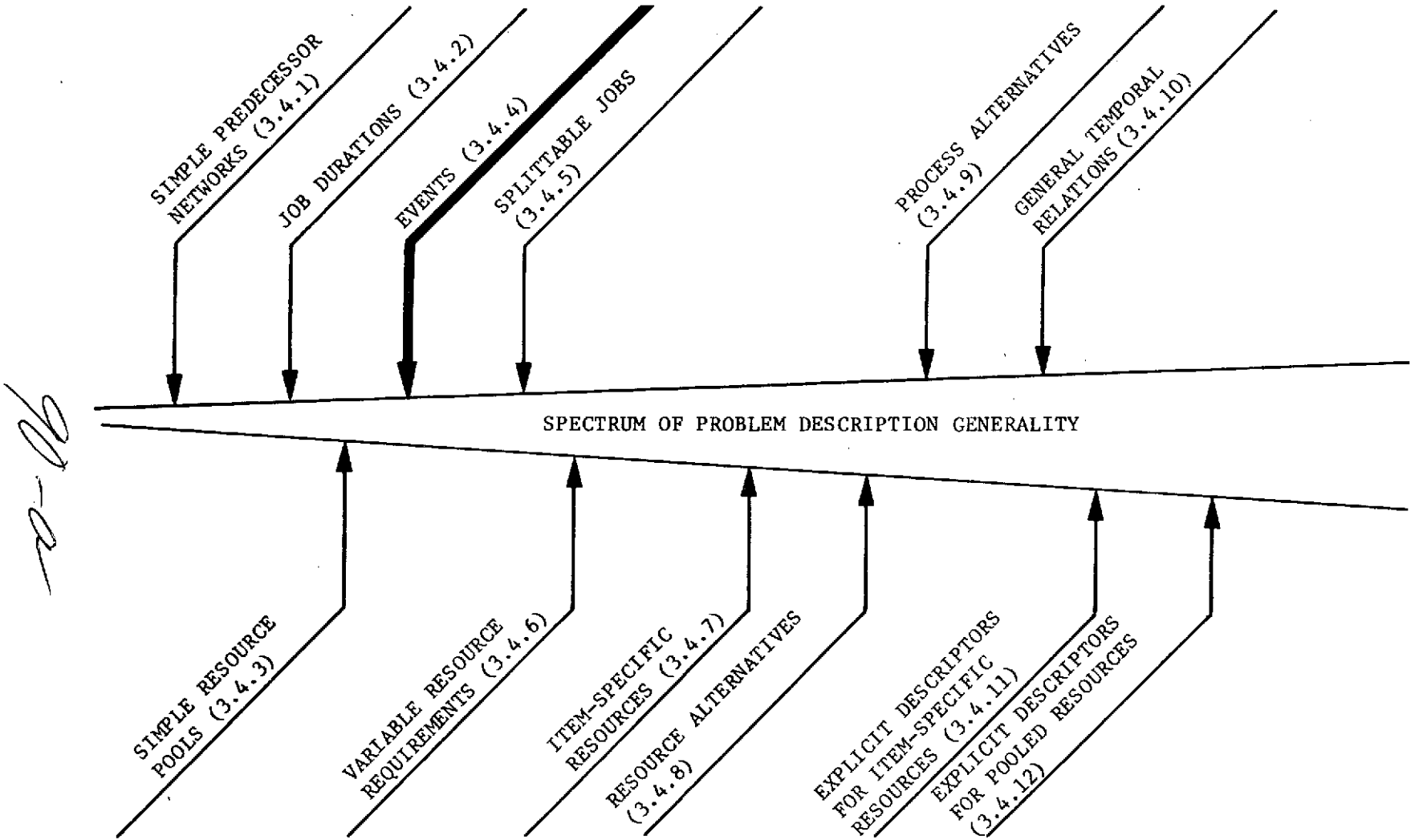
Fig. 3.4.3-2
 Association of Requirements for Pooled Resources with Network Jobs

It is obvious that the problem model for simple project scheduling utilizes pooled resources and that each job selects *indiscriminately* from the pools and *returns resources to the pools upon completion*. A job utilizes prescribed quantities from the pools. From the resource point of view, the job creates an assignment interval during which these quantities are described as "in process" until the job is completed, at which time they are described as "available." The descriptors "in process" and "available" are called *implicit descriptors* because they can be inferred from the existence of an assignment interval; i.e., if we know that Job A uses three units of Resource 1 and Job A is scheduled, then we know implicitly that three units of Resource 1 have the descriptor "in process" during the duration of Job A. *The resources used in the project scheduling problem model can, therefore, be described as pooled resources with, implicit descriptors only.* Although this description appears at this point to be a terminology overkill, it will be useful later in distinguishing the project scheduling model from more generalized models.

3.4.4 Events in Job Networks

90

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.4 Events in Job Networks

It is useful to be able to place events into a network in addition to jobs. *An event may be thought of as a zero duration job that requires no resources.* This function is illustrated in Fig. 3.4.4-1. Its basic usefulness is to permit the integration of two or more networks. A large network may be decomposed into smaller networks by defining the interfacing events between the smaller networks. This may permit network analysis to be accomplished on networks that are too large to be analyzed without decomposition.

The PLANS module library contains several routines that perform operations on generalized networks. The PROJECT_DECOMPOSER identifies networks within larger networks so that analyses can be performed independently on the smaller networks. The NETWORK_CONDENSER module eliminates the jobs in a network leaving only the events. The branches in such an *event node network* represent delay times. The condensation of a network permits a CPM analysis on a very large network using decomposition principles. The PREDECESSOR_SET_INVERTER module converts all predecessors in a network into equivalent successor sets. This inversion is necessary for performing a CPM analysis (the CRITICAL_PATH_PROCESSOR and CRITICAL_PATH_CALCULATOR modules are applicable) or for condensing a network into an event node network.

The permissibility of events in networks allows the problem analyst to describe complex networks in terms of subnetworks with interfacing events. A *subnetwork* is itself a network, which

Fig. 3.4.4-1

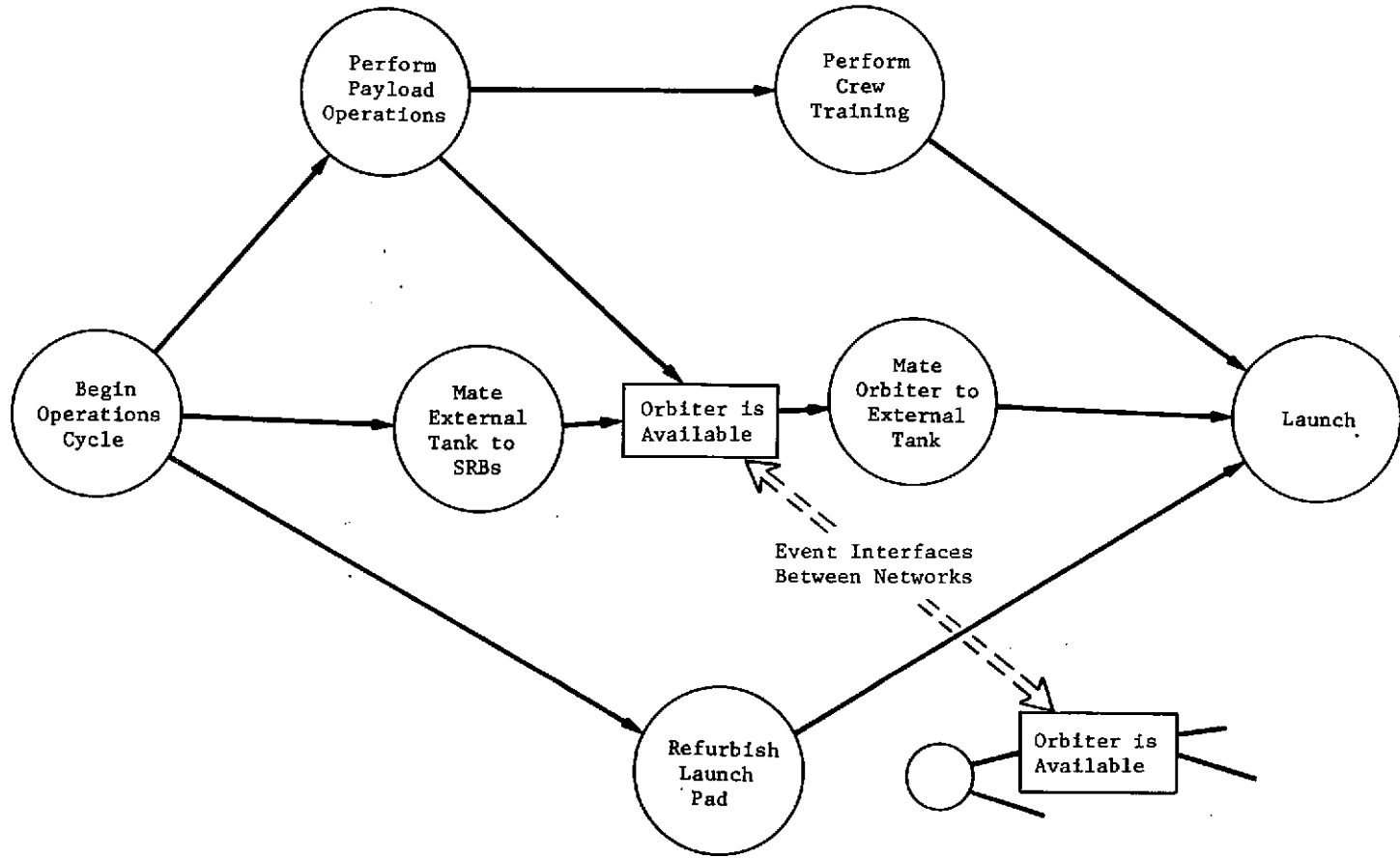


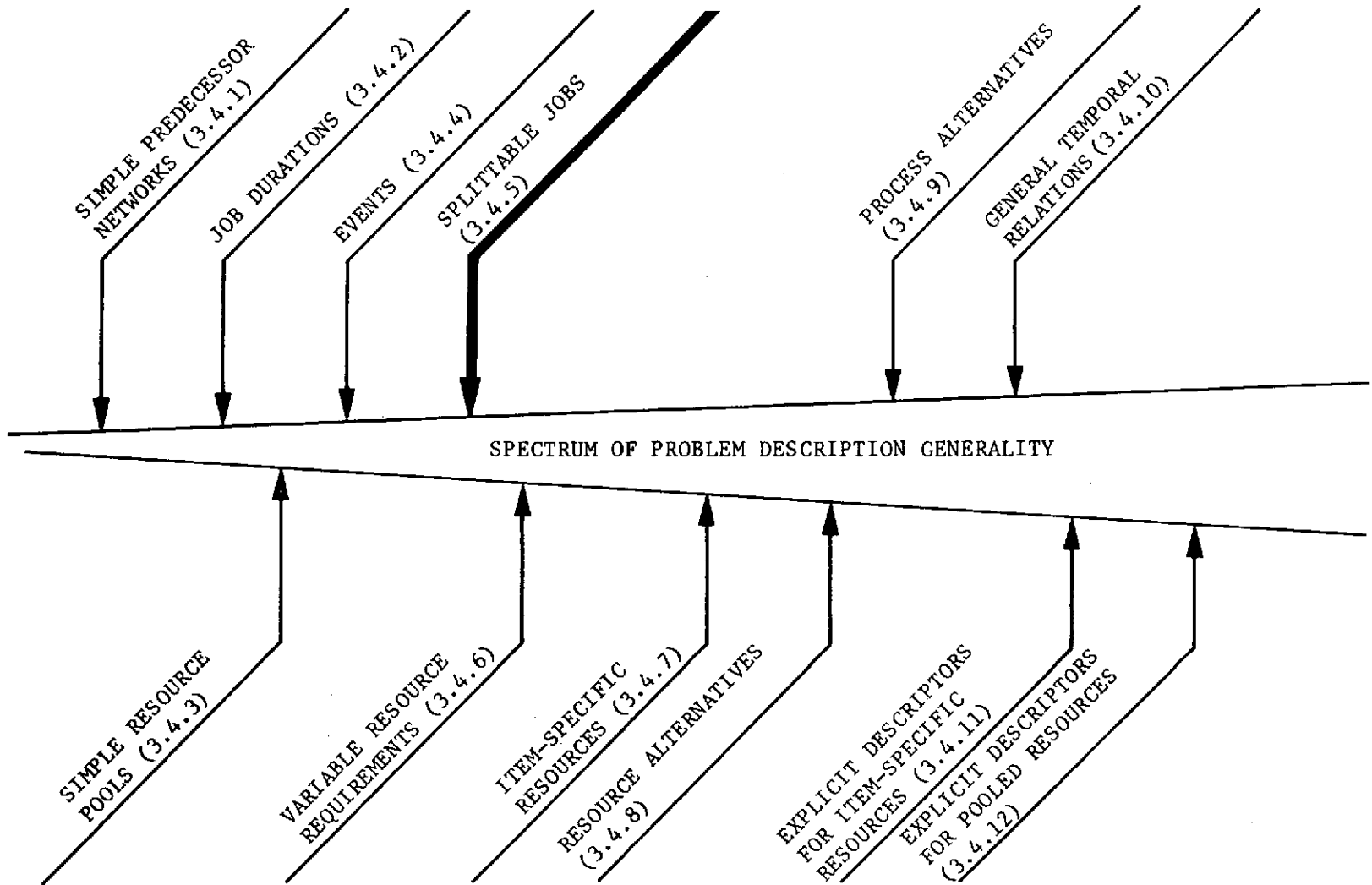
Fig. 3.4.4-1 Use of Events to Interface Between Networks

contains the immediate predecessors and successors of all its member jobs or events that are not interfacing events. Because any subnetwork may, itself, have one or more interfacing events to still other subnetworks, a heirarchical relationship between networks can be described. Scheduling with resource constraints may require a single master network. This capability is provided in the library by the NETWORK_ASSEMBLER routine.

3.4.5 Job Splittability

-94-

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.5 Job Splittability

The simple network can be generalized slightly by permitting *splittable jobs* as illustrated in Fig. 3.4.5-1. Splittable jobs are jobs that can be terminated before completion and restarted from the interrupt point at any later time. The sum of the durations of the job segments is equal to the duration of the original job. The HEURISTIC_SCHEDULING_PROCESSOR module and the three modules it calls, all permit the description of jobs that are splittable or nonsplittable.

Fig. 3.4.5-1

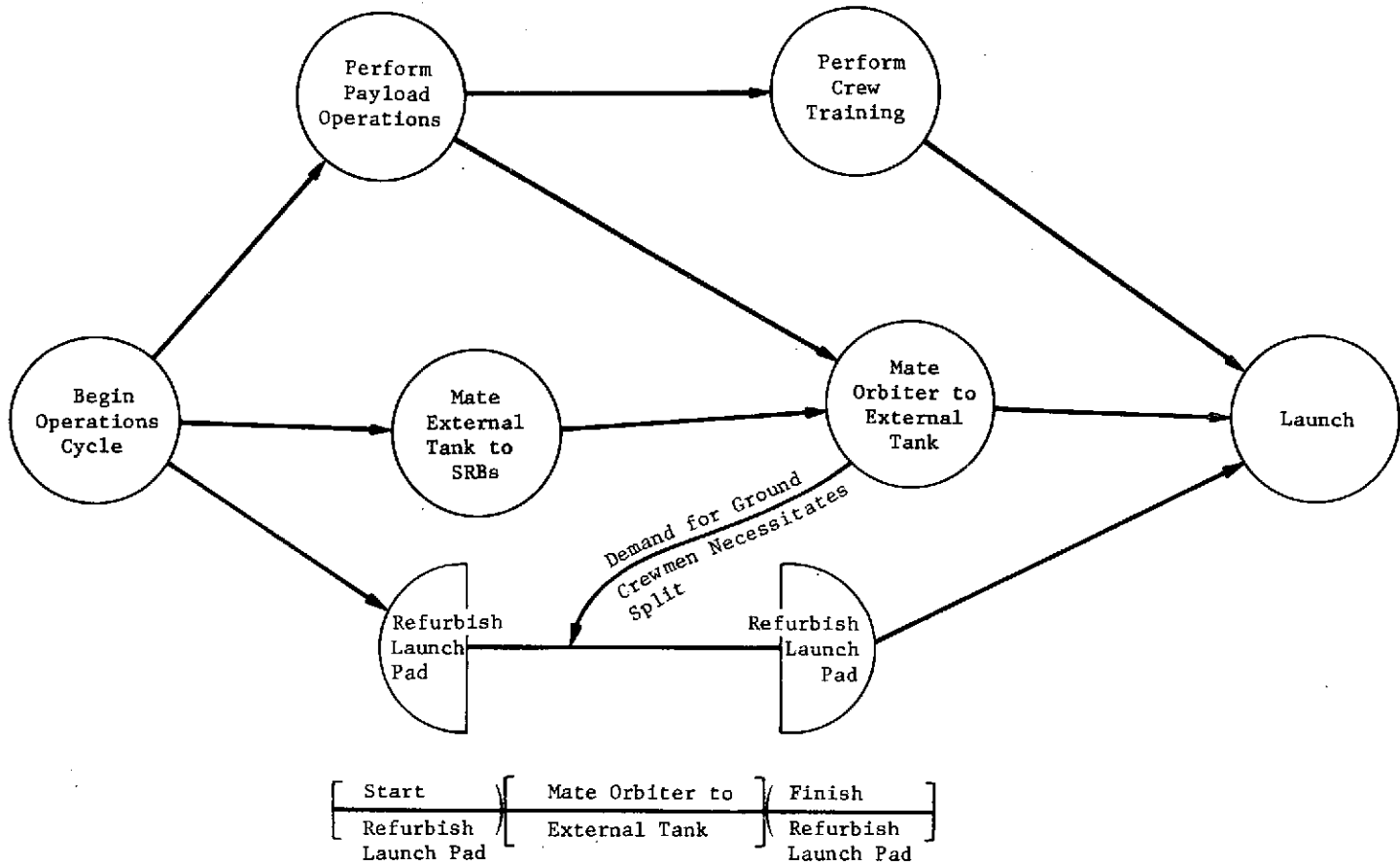
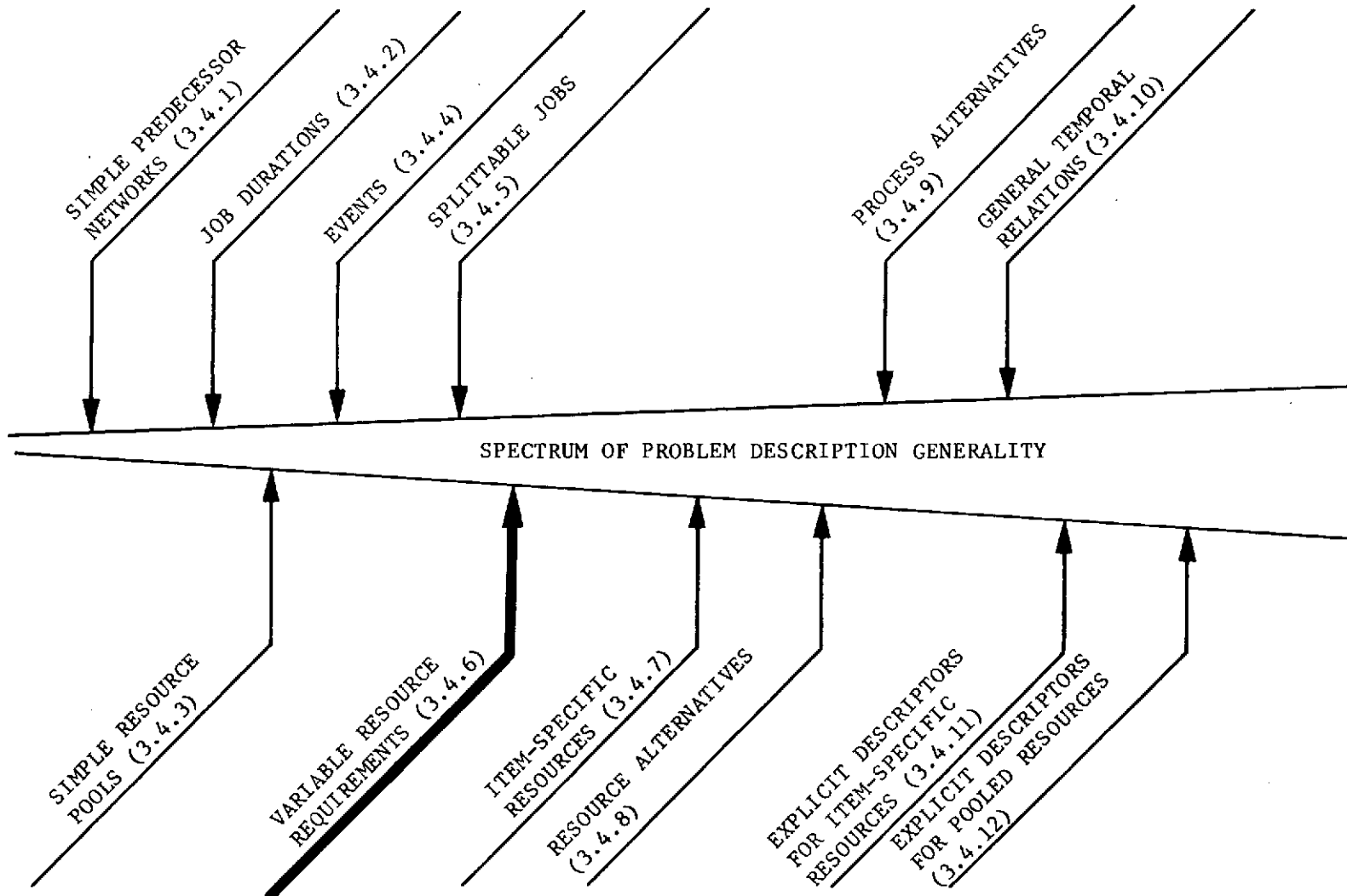


Fig. 3.4.5-1 Illustration of a Splittable Job

3.4.6 Variable Resource
Requirements

96-a

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.6 Variable Resource Requirements

A simple generalization of the problem model is possible while maintaining compatibility with the project scheduling module called the HEURISTIC_SCHEDULING_PROCESSOR. This generalization permits piecewise constant resource profiles such as that shown in Fig. 3.4.6-1.

The PLANS library modules that have been specified for solving project scheduling problems are all capable of handling piecewise constant resource profiles associated with any job. When many resource pools are included in the problem model and when these resources are shared between many jobs, the resolution of resource conflicts becomes a significant problem. This can be appreciated by examining Fig. 3.4.6-2, which illustrates only two resources associated with three jobs. The modules specified for the PLANS library will satisfy the complex resource constraints associated with a large number of resources and jobs. The HEURISTIC_SCHEDULING_PROCESSOR serves as the executive module, which calls the NETWORK_ASSEMBLER, the RESOURCE_ALLOCATOR, and the RESOURCE_LEVELER modules. See Section 3.6.

Just as the resource requirements for a single job may be variable, so can the total resource pool levels be variable with time. Figure 3.4.6-3 illustrates a profile that reflects the fact that fluctuations will occur in available manpower.

Fig. 3.4.6-1

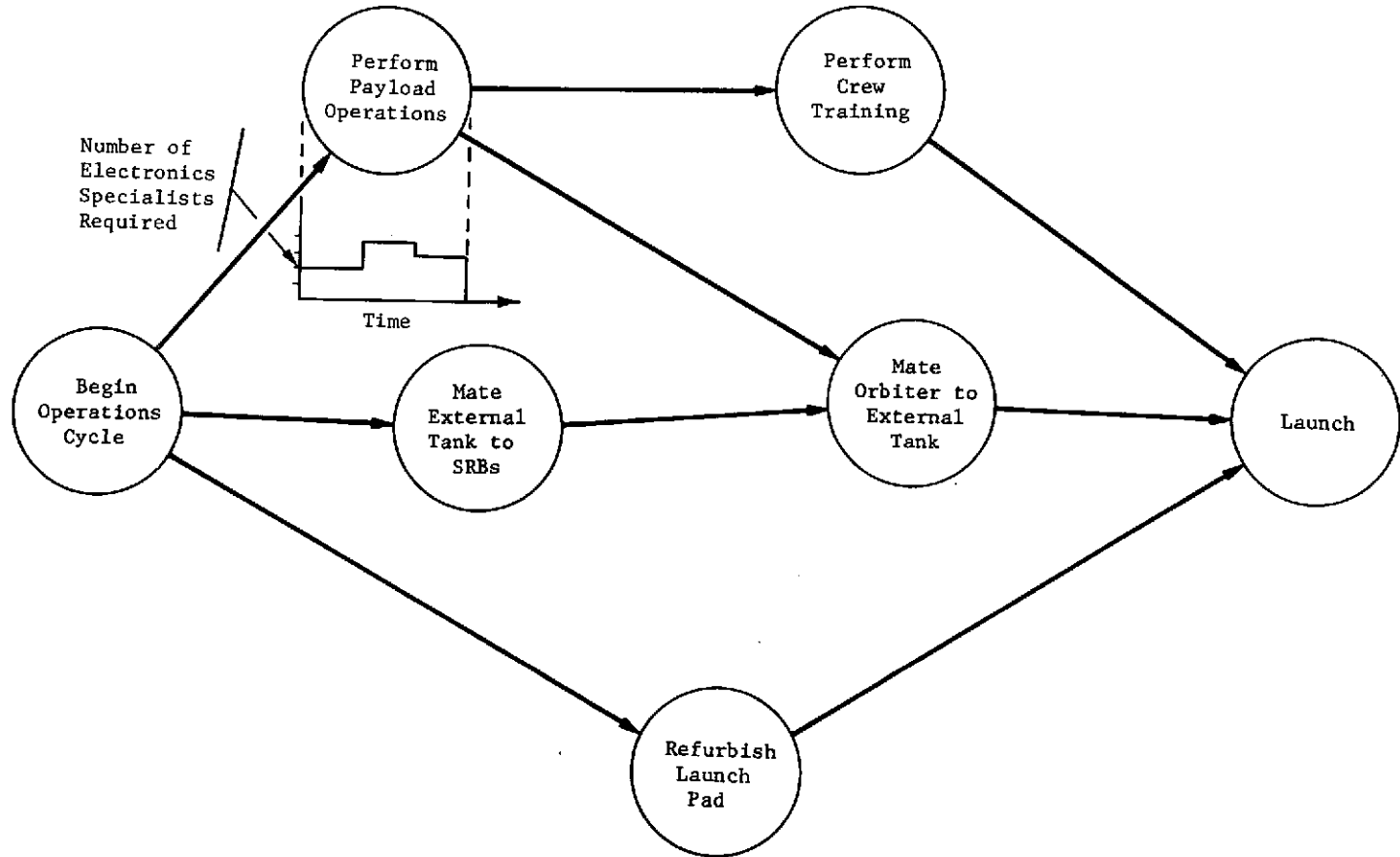


Fig. 3.4.6-1 Variable-Level Resource Requirements

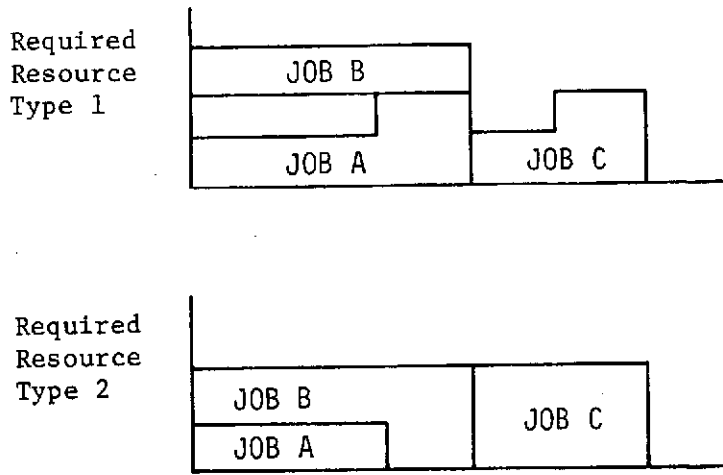


Fig. 3.4.6-2
 Resolving Usage Conflicts with Piecewise
 Constant Required Resources

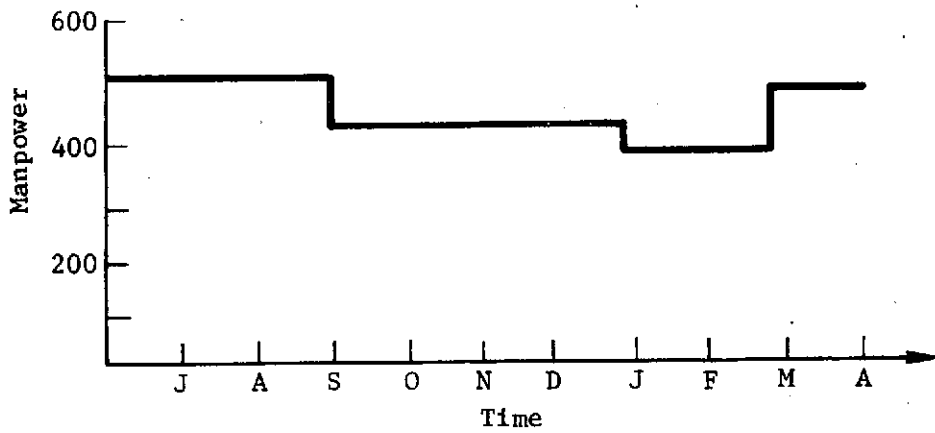
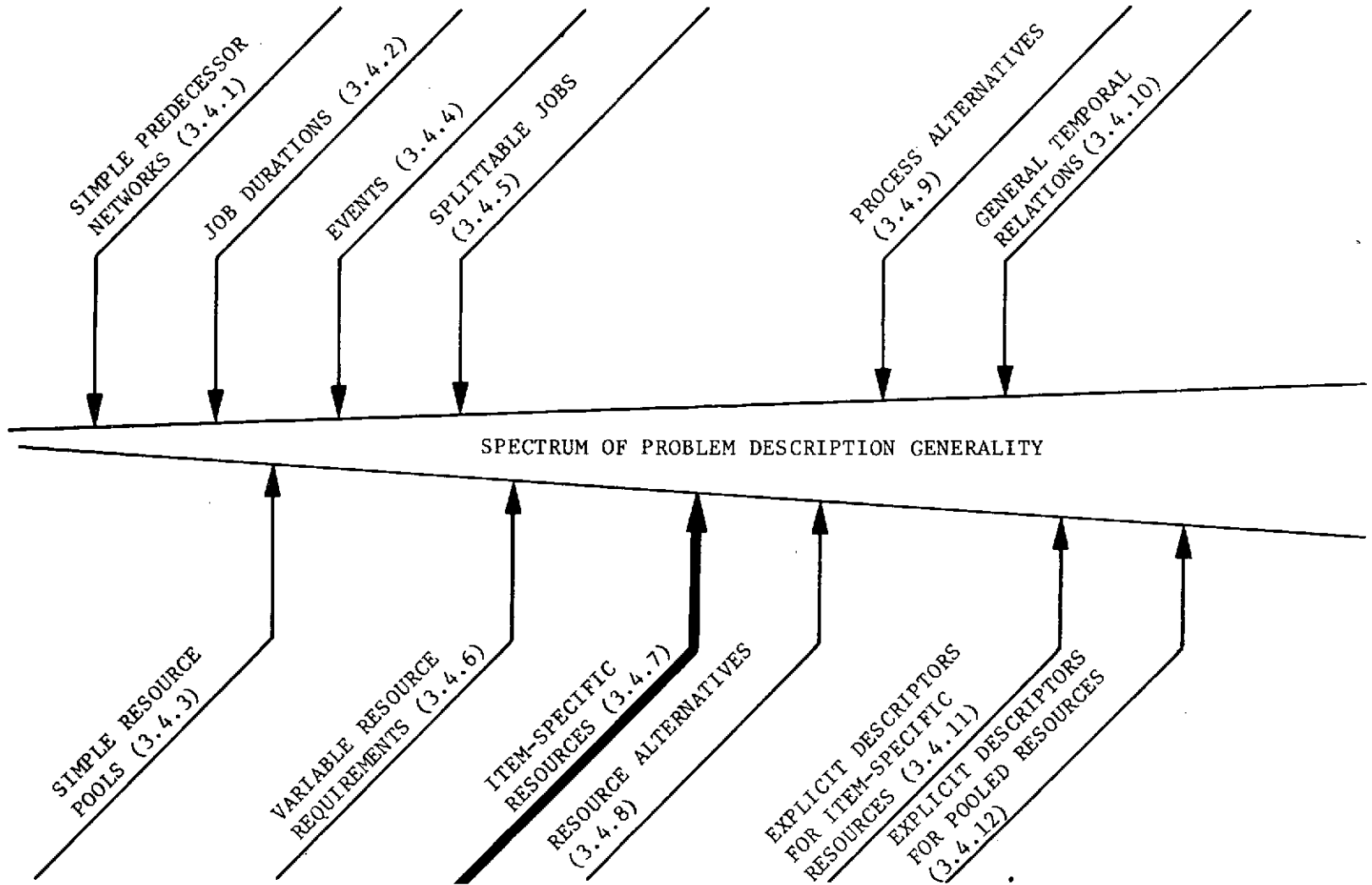


Fig. 3.4.6-3 Typical Pool Level Profile

3.4.7 Item-Specific Resources

-100-

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.7 Item-Specific Resources

Many scheduling problems cannot be adequately formulated using resource pools. This usually results from the necessity to keep track of the assignments for specific resource items. For example, it might not be acceptable to produce a schedule that provided only the information that one truck (from a pool) was used from 9:00 am to 9:12 am. It is only acceptable to specify that truck number 90526 was used. When specific traceable assignments are required, the problem model must include "item-specific resources." Some item-specific resources that might be required by the jobs in the illustration used in previous sections are shown in Fig. 3.4.7-1. The inclusion of item-specific resources in the operations model permits resource allocation to be coupled into scheduling. Problems using item-specific resources are those that require the determination of not only when a job is to be done and how many resources are to be used, but also *which* resources are to be used. Table 3.4.7-1 provides several examples of pooled and item-specific resources. Because the use of item-specific resource descriptions adds to the complexity of the problem model, solutions require more sophisticated techniques that are much more costly to execute. In addition, tracking all item-specific resources adds substantially to computer storage requirements. Thus, the use of item-specific resources should not be done unnecessarily.

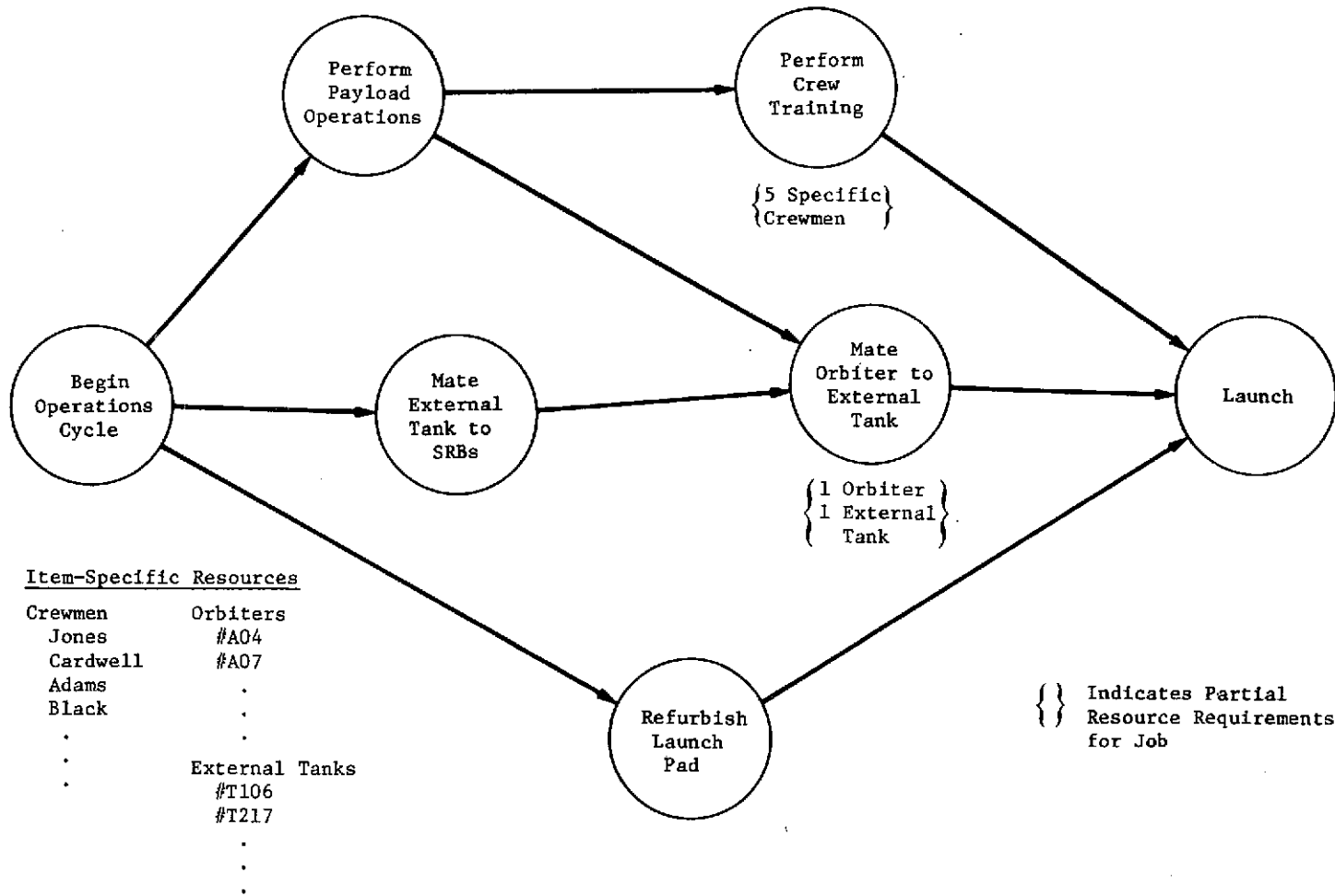


Fig. 3.4.7-1 Illustration of Item-Specific Resources Required by Jobs

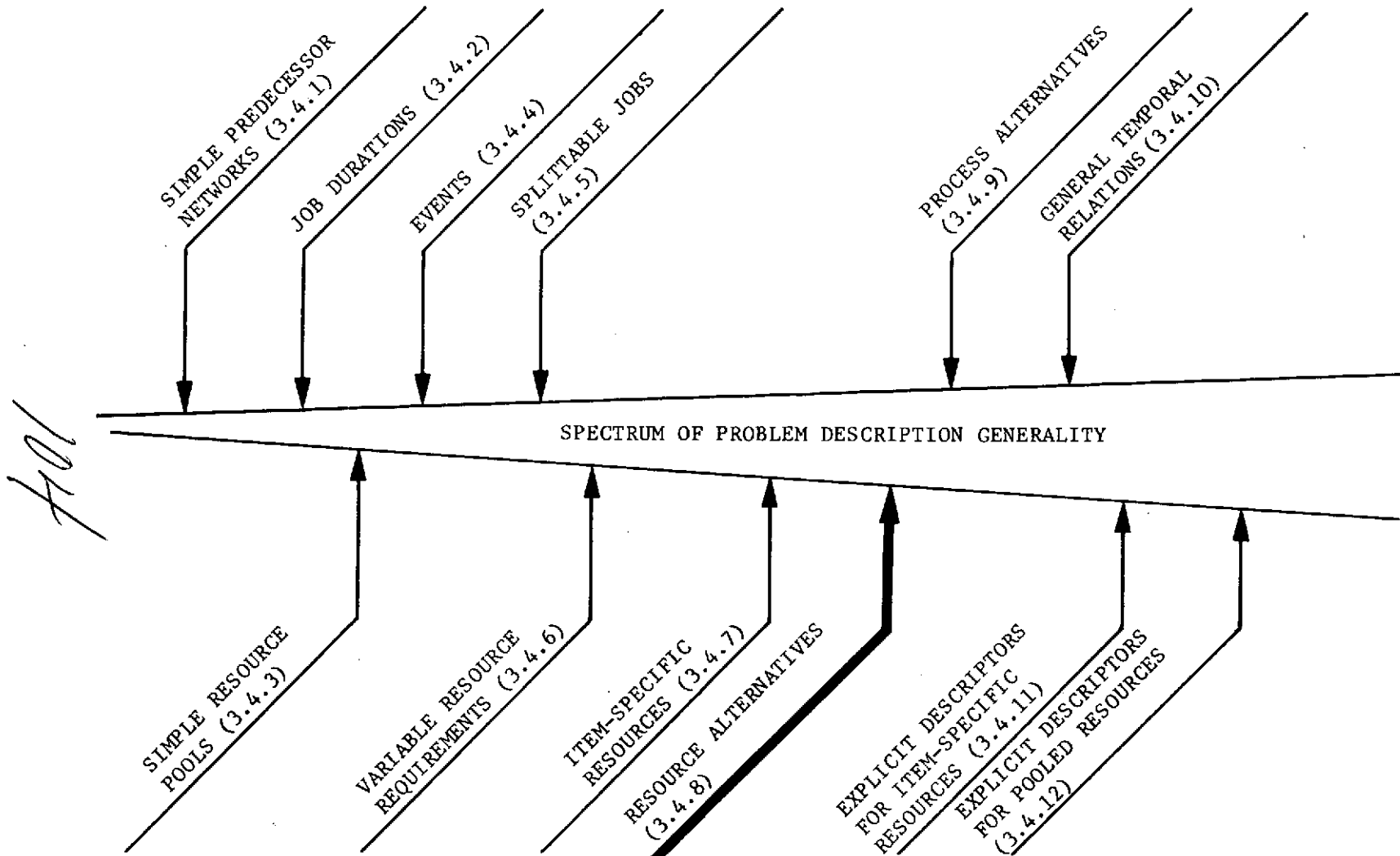
Table 3.4.7-1
 Examples of Pooled and
 Item-Specific Resources

Pool	Item-Specific
Crewmen	Jones Clayborn Betts Blackburn
Vehicle Seating Units	Bus 6 Shuttlebus Van . . .
Food Units	Meal 6 Supplement 2 Meal 4 . . .
Computer Storage Cells	Disc 2 Tape 6 Core Block 2 . . .
Machines	Lathe 1 Auto Lathe Lathe 6 . . .

3.4.8 Resource Alternatives

103-a

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.8 Resource Alternatives

Many problems require the consideration of alternative (substitutable) resources, as illustrated in Fig. 3.4.8-1. These alternatives may be directly substitutable (i.e., an adjustable wrench may be substituted for an open-end wrench) or only functionally comparable (i.e., a conveyer may be functionally equivalent to three laborers). The selection between alternatives may be based on a preset priority with the availability of the resource as the criterion. Other alternatives may influence some problem "figure of merit" such as duration, total cost, or resource utilization smoothness. In this case, an executive logic could be empowered to evaluate the effects of its selections. Typically a heuristic decision rule must be used to make the selections between resource alternatives during the algorithm operations.

Within the operations model, the required resources are defined in the definition of a process as a series of "and" and "or" resources. That is, a process requires each of the resources of the "and" portion plus one of the alternatives of each partition of the "or" section of the process definition. This approach readily identifies those resources in a process definition that have suitable alternatives.

Fig. 3.4.8-1

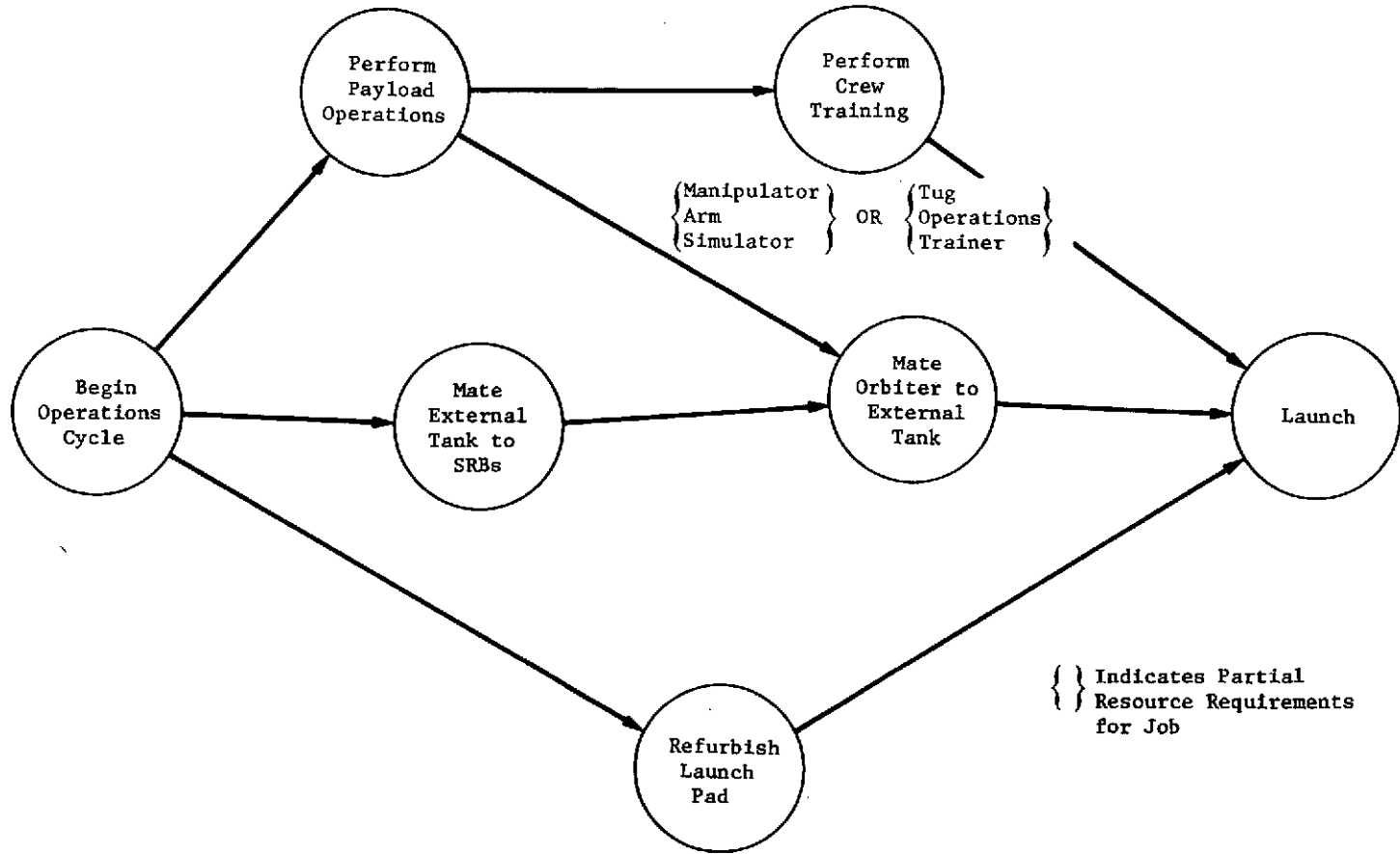
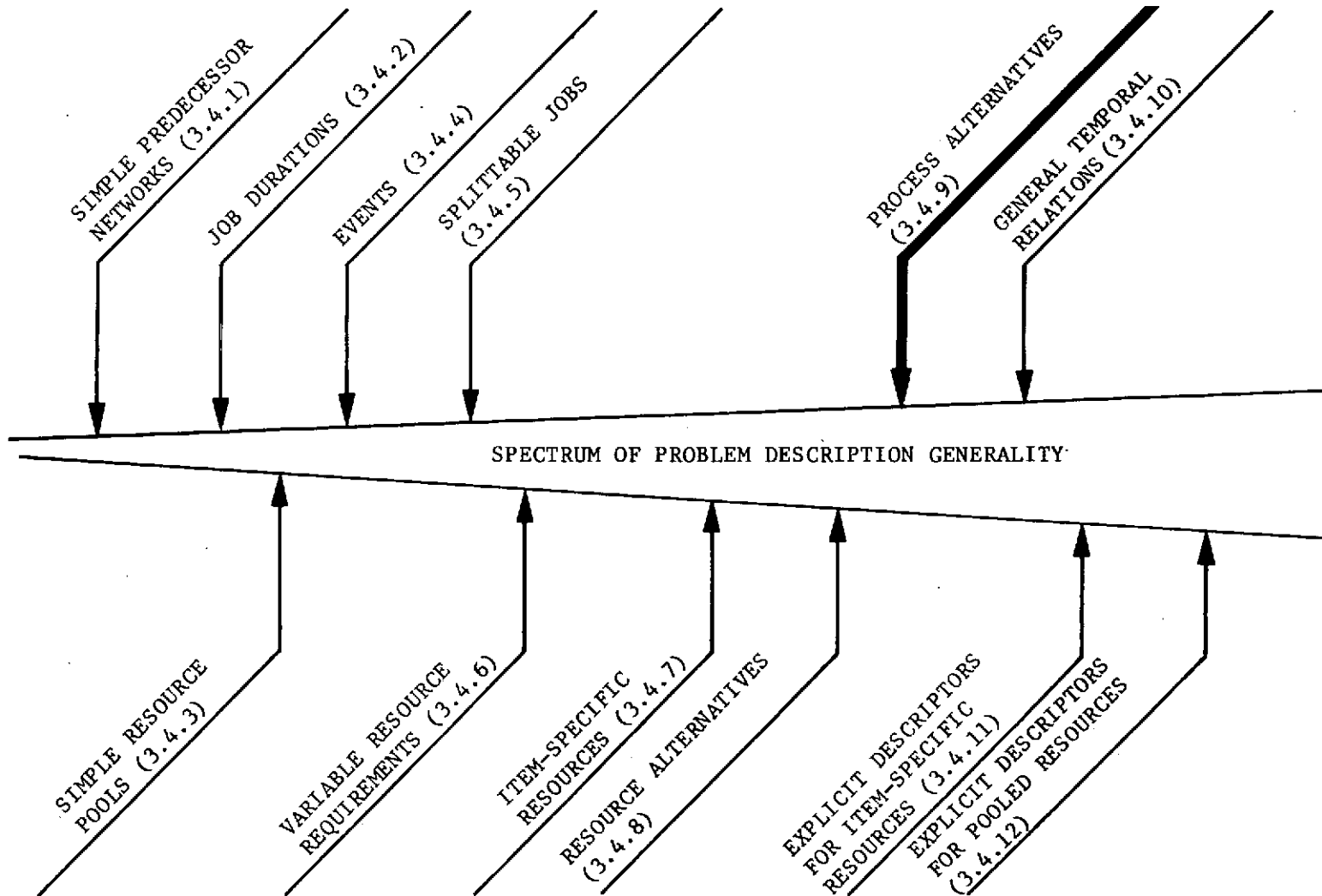


Fig. 3.4.8-1 Illustration of Resource Alternatives

3.4.9 Process Alternatives

106-a

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.9 Process Alternatives

Process alternatives such as the one shown in Figure 3.4.9-2 may arise in a problem description as a means of handling resource alternatives or because a processing option must be modeled. A conversion from resource alternatives to process alternatives results if each combination of resource alternatives is considered as a distinct process. (See Fig. 3.4.9-1.)

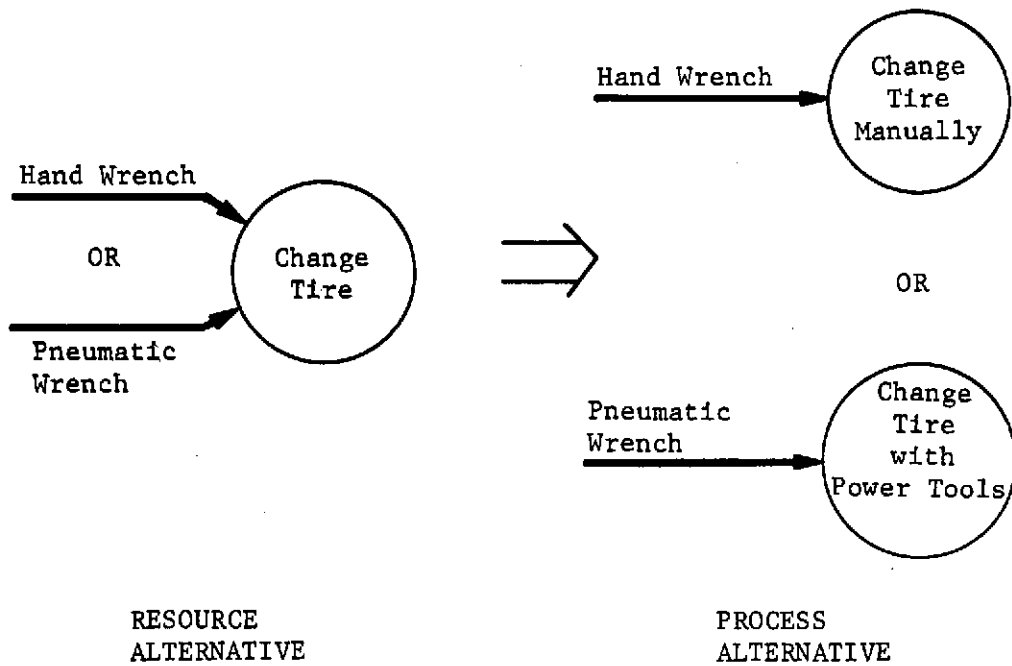


Fig. 3.4.9-1
Conversion from Resource Alternatives
to Process Alternatives

This approach becomes very cumbersome if many of the resources required for a process have alternatives because each combination of resource alternatives implies a unique process definition. A substitutable process is created either by alternative resource combinations or by the existence of functionally equivalent and

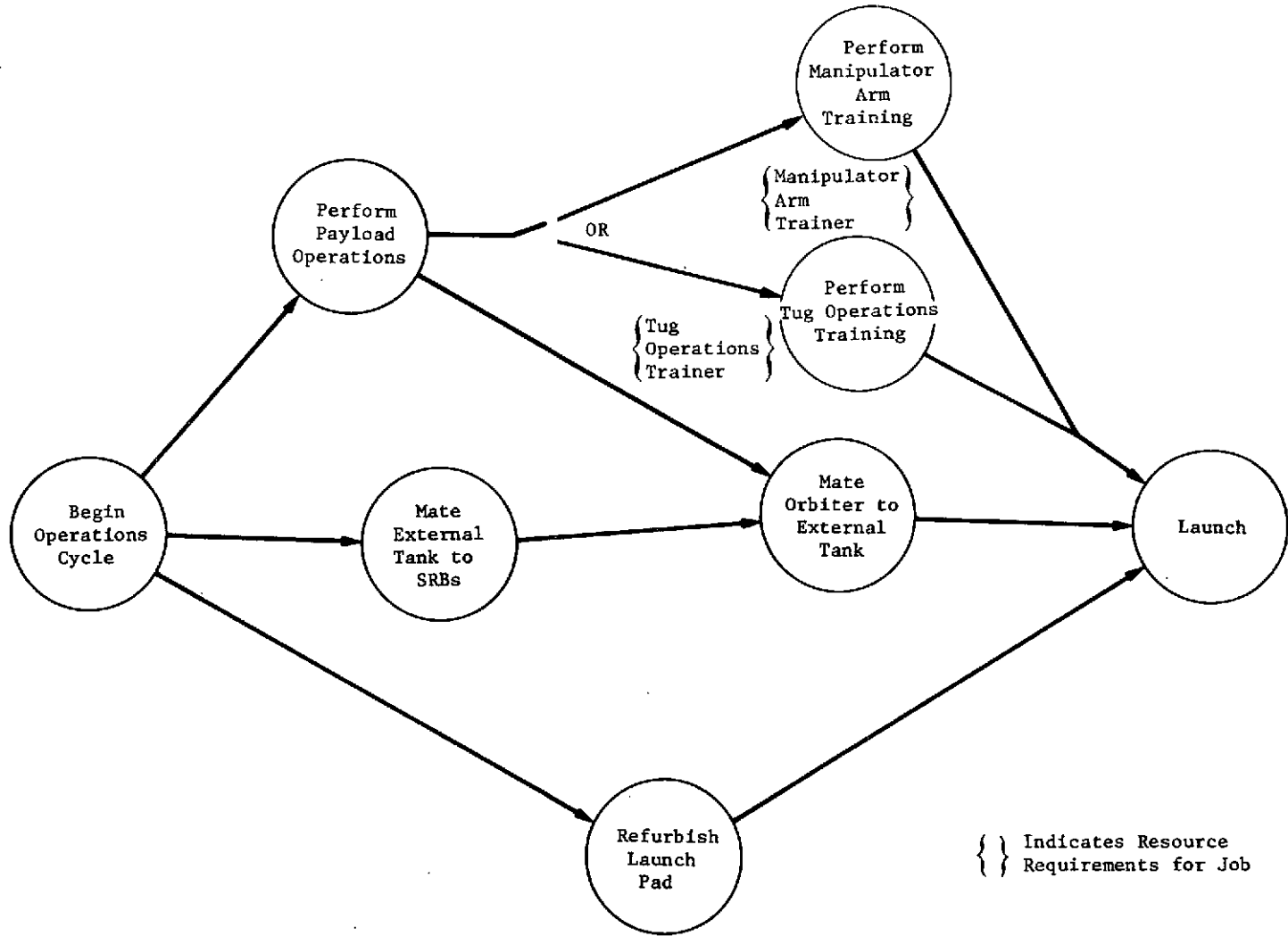
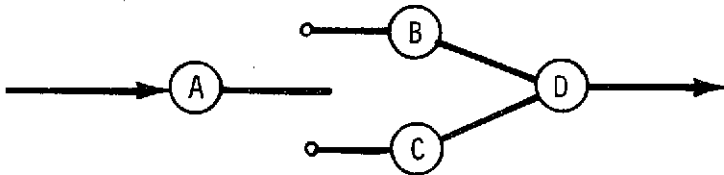
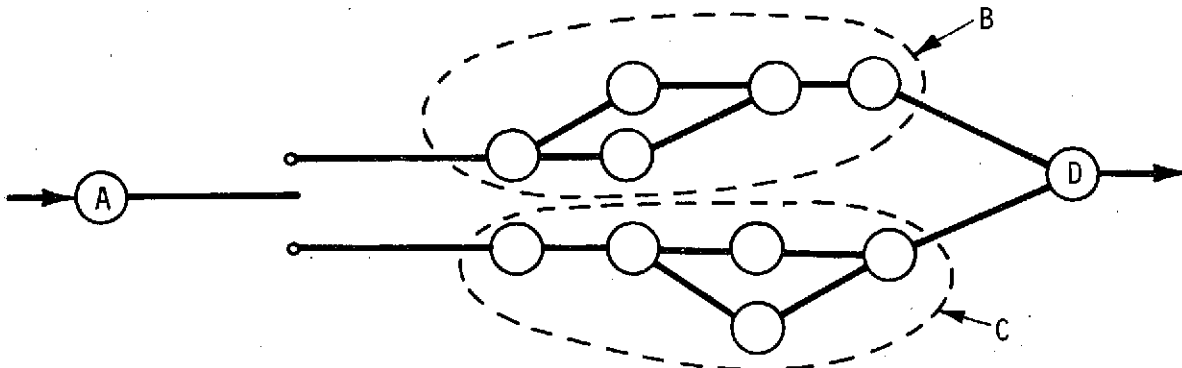


Fig. 3.4.9-2 Illustration of Process Alternatives

and redundant activities (i.e., any number of the required resources may be different). In network form, these process alternatives can be shown as in sketch A where process B and C represent



the alternatives. Similarly, alternative subnetworks can be specified if more than one process is involved in the alternative subnetworks. For example, the subnetworks B and C, as shown in sketch B, would define alternative subnetworks.



In subsequent discussions of the operation model, networks and subnetworks are referred to as operations sequences, because the latter term is more descriptive of the temporal relations between processes.

At this point it may be necessary to make a distinction between alternative operations sequences and conditional branches in formulating his solution strategy. For this purpose, consider the "alternative operations sequences" as functionally equivalent

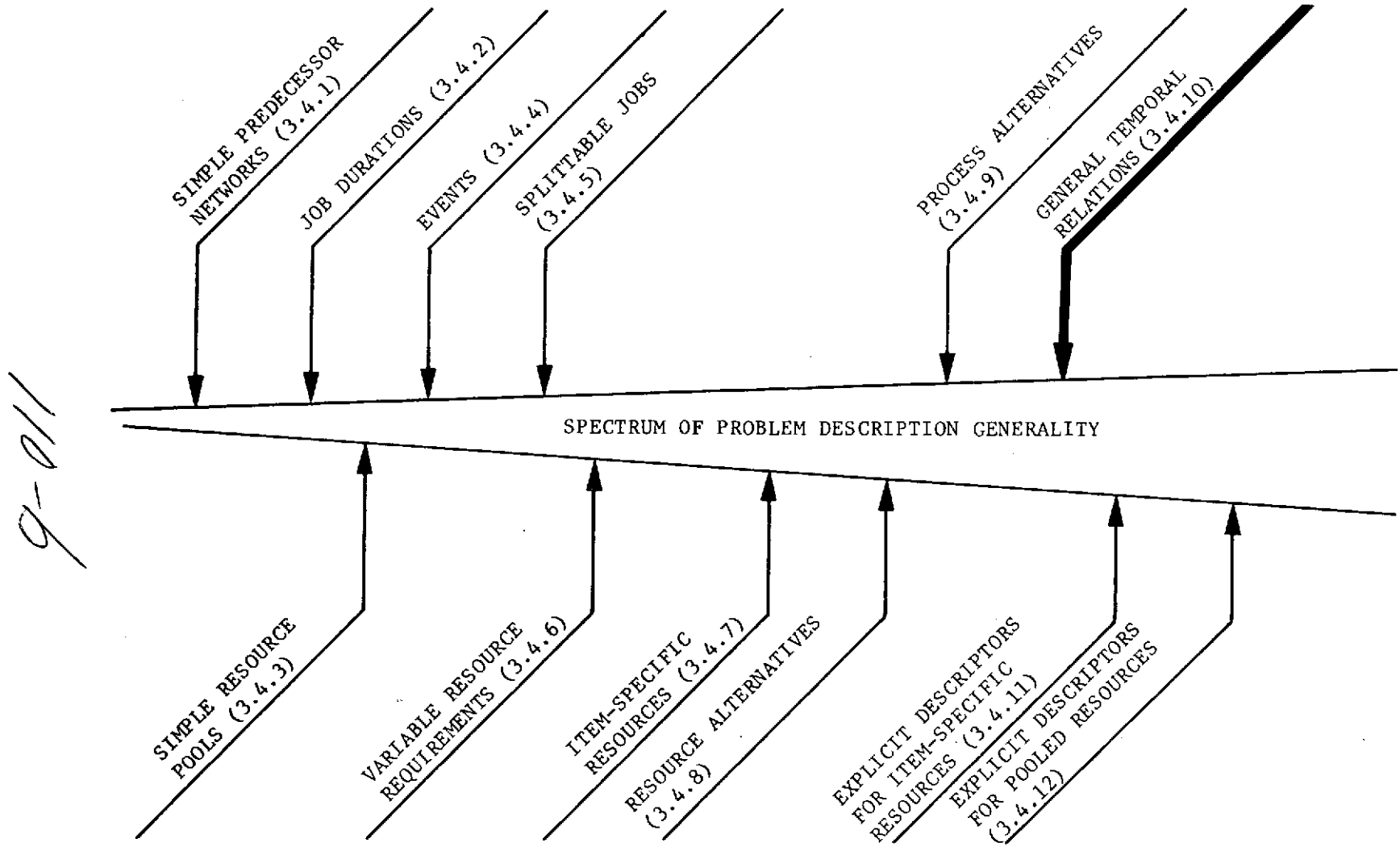
subnetworks for which the selection criterion is based on a problem objective such as least cost, minimum time, resource availability, etc. This decision would, therefore, have to depend on the contents of the subject subnetworks and a prediction or simulation of the impact of choosing each alternative. In contrast, consider a "conditional branch" as a switch between separate subnetworks, which may or may not be functionally the result of activities completed earlier in the network. This selection would be independent of the overall problem objective function and, therefore, independent of the contents of the subnetworks.

To illustrate this distinction, consider a machining activity in which a part may be produced by either mechanical milling or chemical milling. Each method could be modeled as a subnetwork in a larger shop model. Because the beginning and end states of the particular part are identical, the selection of method to be used could be based on comparative cost or duration, or possibly some other problem constraint such as least manpower required. Thus, these functionally equivalent subnetworks would be "alternative operations sequences." Conversely, a Shuttle operations model might contain two or more subnetworks that define checkout sequences for payloads. The selection criterion in this case would depend on the characteristics of the payload under consideration. Because the particular payload would be selected earlier in the operations "loop", the "conditional branch" selection would not be based on an objective function of the problem or contents of the subnetworks, but on operational constraints.

3.4.10 General Temporal Relations

110-a

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS

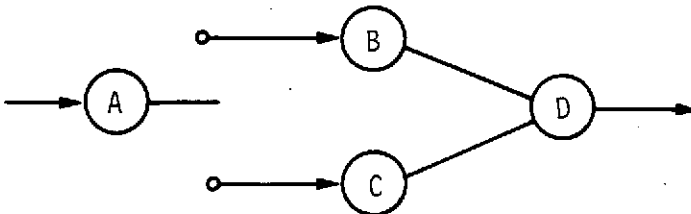


RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.10 General Temporal Relations

Occasionally, the sequences between processes must be described with relationships that are more general than the simple predecessor relationship (If the end of activity A must occur before the beginning of activity B, then A is simply a predecessor of B.) In a more general operations model, however, other temporal relationships may exist such as the one illustrated in Fig. 3.4.10-1.

If a set of "alternative operations sequences" exists in a network (as shown in the sketch), it would be incongruous to specify both alternatives as predecessors to the subsequent activity (D) because both predecessors cannot be completed. However, it is necessary to indicate the direction of flow regardless of the alternative selected. In this case, unambiguous specification would label the subsequent activity (D) as a "successor" to both alternatives.



Temporal relations that are more general than either predecessors or successors may be represented as

$$\begin{Bmatrix} s_i \\ f_i \end{Bmatrix} \begin{Bmatrix} < \\ < \\ = \\ > \\ > \end{Bmatrix} \begin{Bmatrix} s_j \\ f_j \end{Bmatrix} \begin{Bmatrix} + \\ - \end{Bmatrix} K$$

where i and j are any activities or events in the project and "s" denotes a start time while "f" signifies a finish time. The

Fig. 3.4.10-1

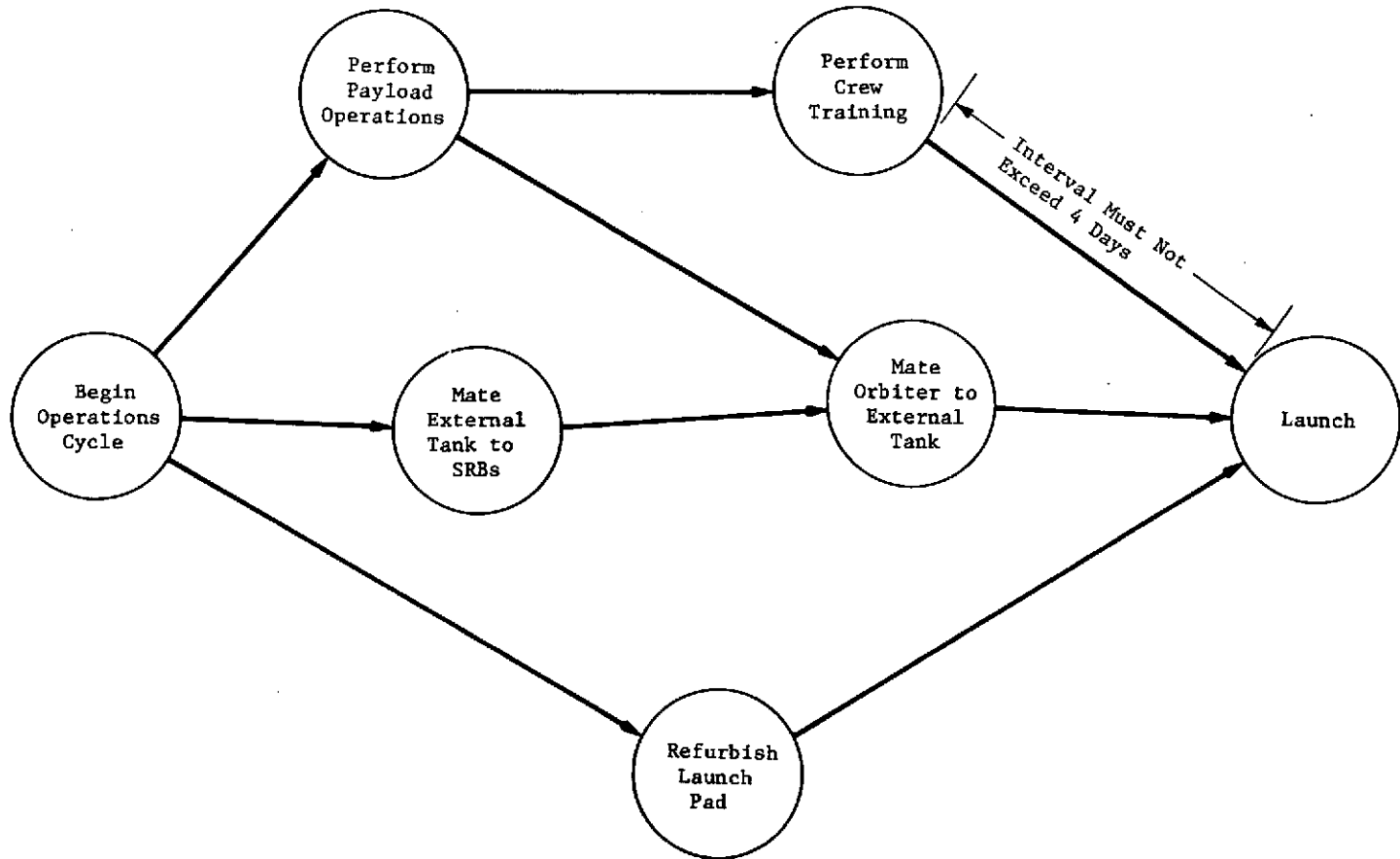


Fig. 3.4.10-1 Illustration of a General Temporal Relations

addition of fixed time intervals (K) or inequalities containing constant time intervals has many applications in system modeling. For example, a system involving the pouring of concrete may have a constraint specifying that the troweling activity must begin within 30 minutes of completion of the pouring. Similarly, an activity involving a sealing coat may be constrained not to start within 48 hours of completion of the pouring activity. The PLANS module library contains three modules that can be used with general temporal relationships. These are CHECK_EXTERNAL_TEMP_RELATIONS, CHECK_INTERNAL_TEMP_RELATIONS, and CHECK_ELEMENTARY_TEMP_RELATION. All perform checking for constraint satisfaction.

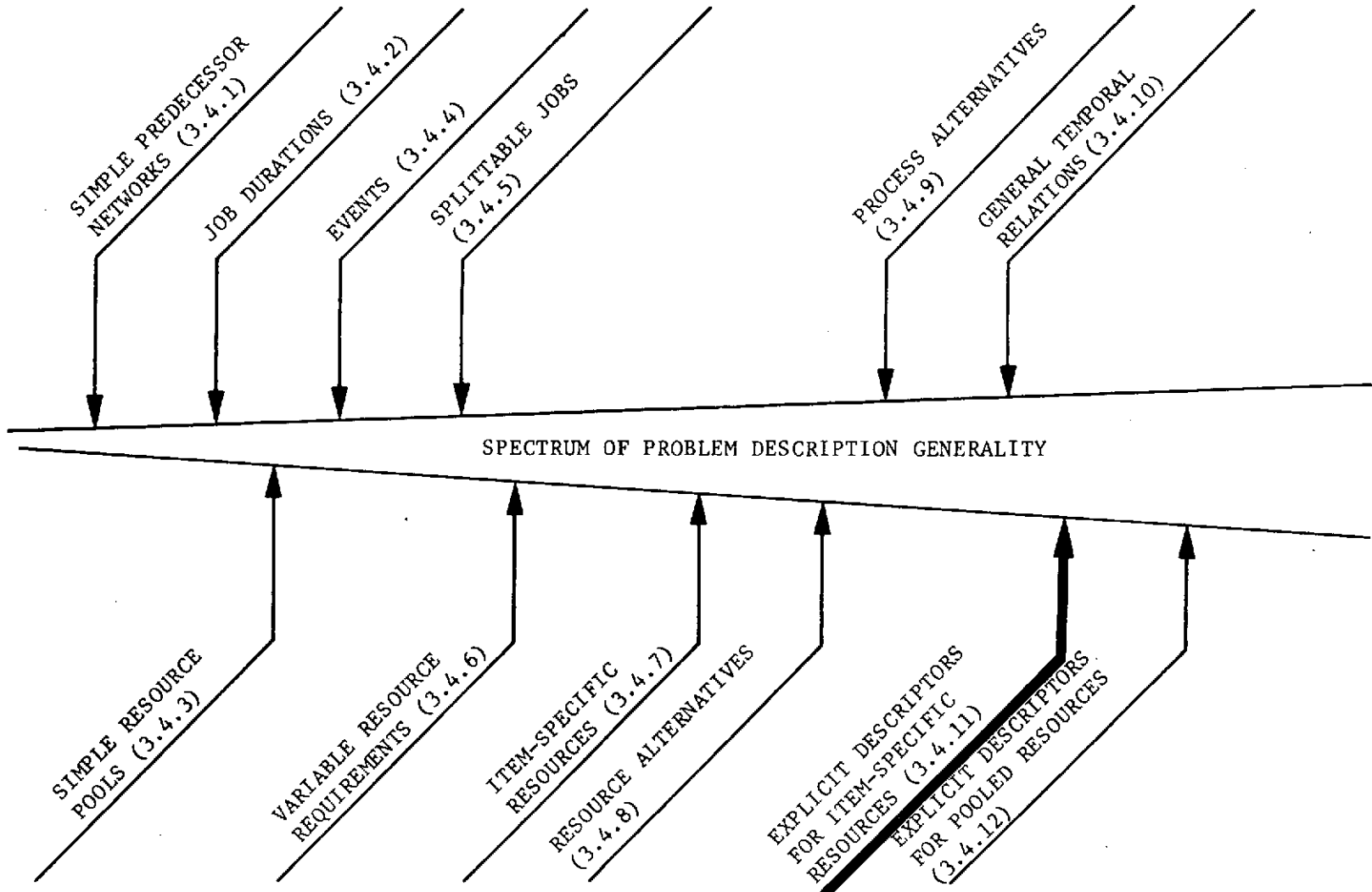
Our approach to handling the generalized temporal relationships is to reduce the network to closely continuous and ordinary predecessors or successors by introducing "dummy" activities. Dummy activities have finite durations, but differ from regular activities in that no resources may be required. Thus, they represent a span of time in a network or subnetwork. 'Closely continuous' implies that the end time of a preceding activity equals the start time of the successor activity. Although methods exist for handling general temporal relationships, they are substantially more complex than methods that handle only predecessors and successors. The modeler is, therefore, encouraged to use general temporal relationships only when simple predecessor/successor relationships have proven inadequate.

3.4.11 Item-Specific Resources
with Explicit Descriptors

113-a

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS

-111-



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.11 Item-Specific Resources with Explicit Descriptors

Processes require many resources that change their previous characteristics and thus must have additional *explicit* descriptors. For example, in Fig. 3.4.11-1, the job "Mate Orbiter to External Tank" is shown to require one orbiter with a status descriptor REFURBISHED. It is a property of explicit descriptors that they change only at the end of the process time and that the way they change must be specified explicitly as a part of the process definition. For example, if a given resource, Truck 87, had been assigned to two activities that first loaded the truck and then drove it to Los Angeles, it might have a descriptor LOAD_STATUS with a corresponding value LOADED, and a descriptor LOCATION with a value LOS_ANGELES. These data would be retained as part of the two assignments for the resource. If a subsequent activity moved the truck to San Francisco, the result would be to change the descriptor LOCATION. The descriptor LOAD_STATUS would be unaffected by the current process and any inquiry concerning Truck 87 at a subsequent time would find the loaded truck in San Francisco. (Because the driver is considered a separate resource, it should not be inferred that the driver was found loaded in San Francisco!)

It should be noted that explicit descriptors must have mutually exclusive values for any given descriptor. That is, if, in the example just discussed, the truck were moved to DOCK_23, an additional descriptor (i.e., DOCK_LOCATION) should be added if the location SAN_FRANCISCO were to be retained.

Fig. 3.4.11-1

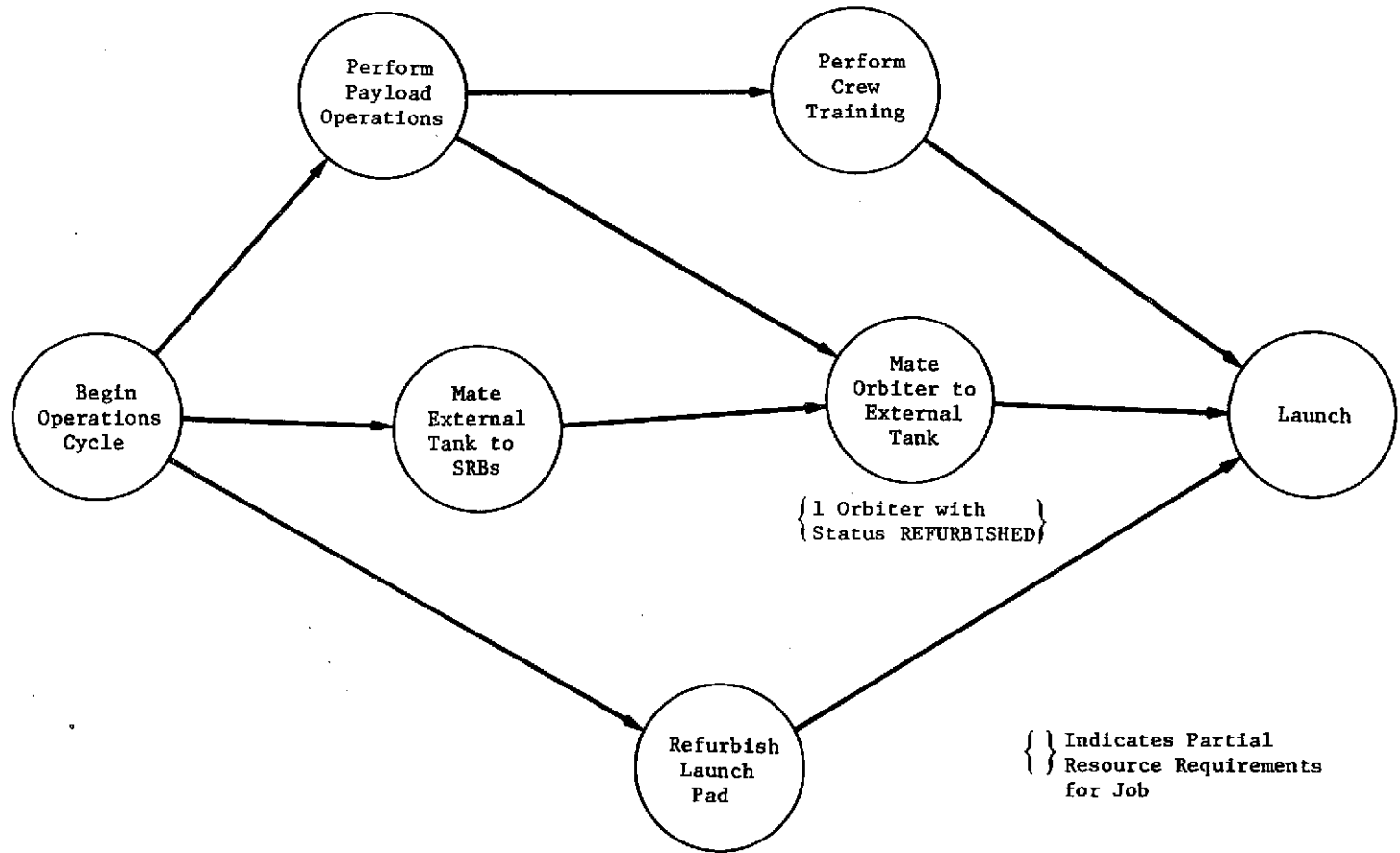


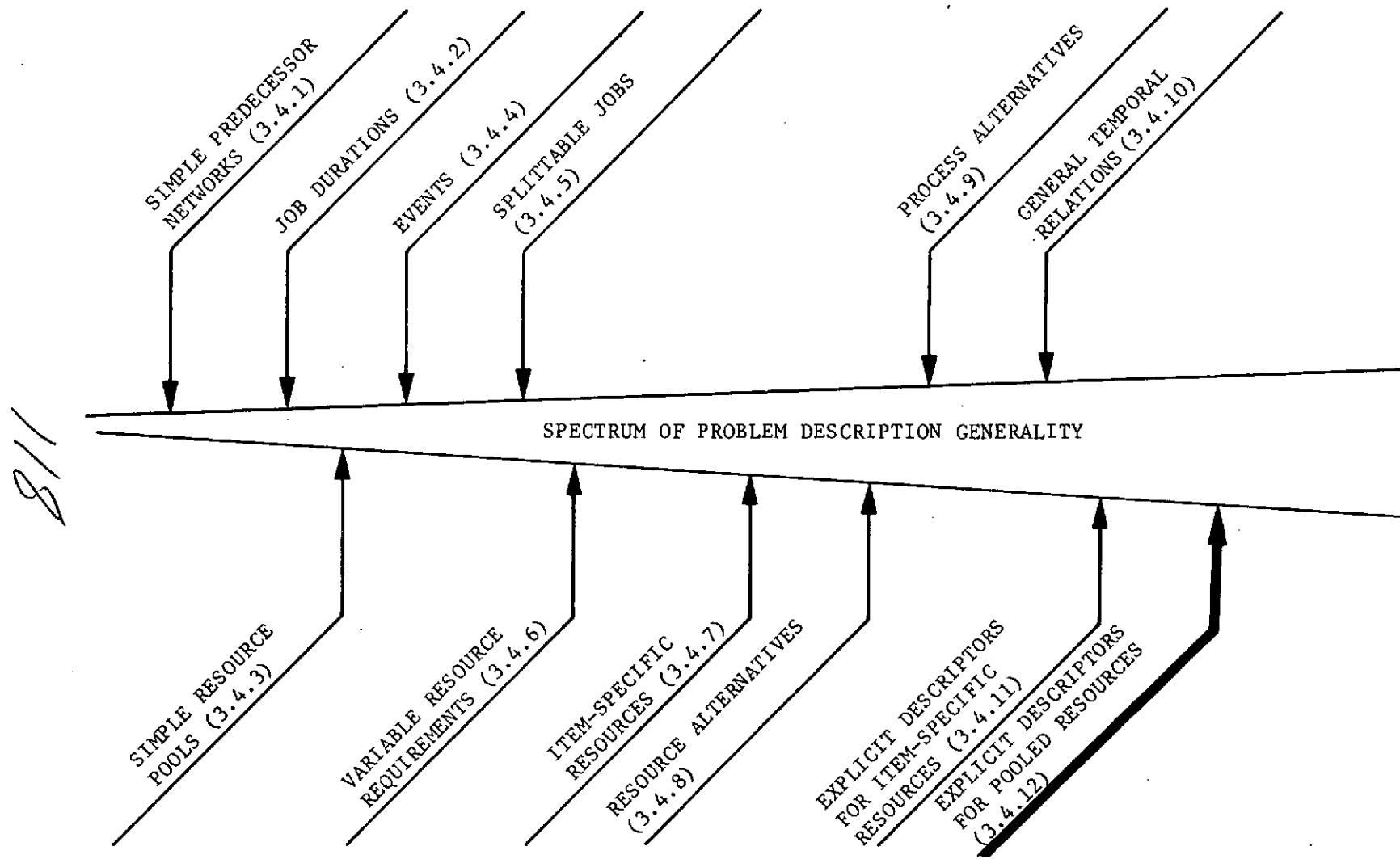
Fig. 3.4.11-1 Illustration of an Item-Specific Resource with an Explicit Descriptor

It should also be noted that if activities are being placed on a timeline ahead of some previously scheduled activities (time-transcendant scheduling), the resource profile for all times later than the new activity may be altered. The module library contains a routine called CHECK_DESCRIPTOR_COMPATIBILITY, which determines conflicts that would result in attempting to schedule a job requiring an item-specific resource with explicit descriptors.

3.4.12 Pooled Resources with
Explicit Descriptors

117-a

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



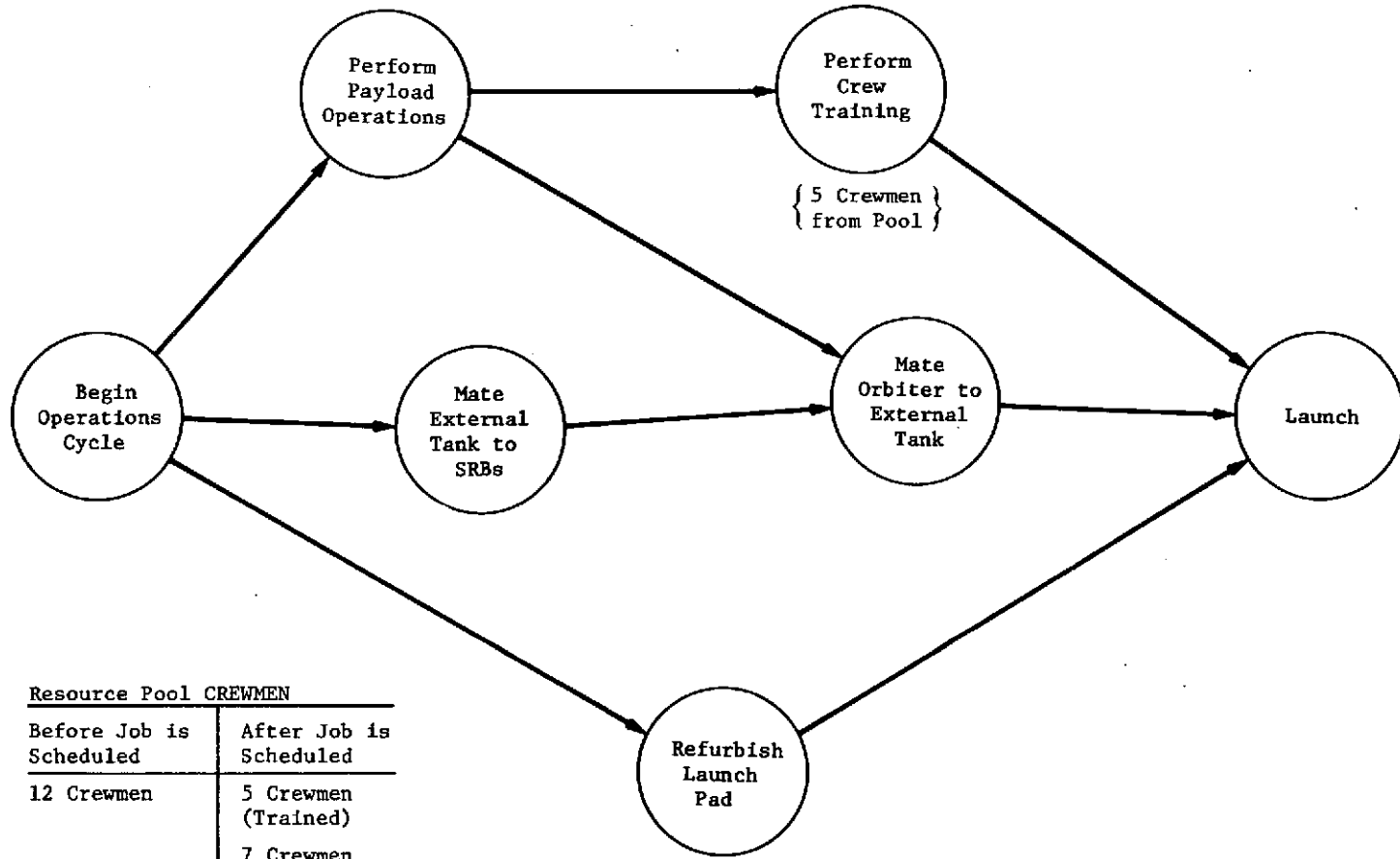
RESOURCES AND THEIR RELATIONSHIPS TO JOBS

3.4.12 Pooled Resources with Explicit Descriptors

Pooled resources represent a very difficult extension of the generalization capability of explicit descriptors. Because explicit descriptors alter a resource so that it does not revert to its original state at the end of the scheduled activity, a pool is continually repartitioned as resources are allocated to various activities. This repartitioning is illustrated in Fig. 3.4.12-1. Since it is necessary to know from which partition a set of resources is to be drawn, it becomes necessary to describe the partitioning of the entire pool for each assignment made. This obviously becomes very cumbersome as the size of the pool or the number of assignments grows. The problem is compounded if time-transcendent scheduling is attempted or if activities previously scheduled must be unscheduled. The module POOLED_DESCRIPTOR_COMPATIBILITY is designed to identify the descriptor conflicts that occur when attempting to insert a job on the timeline that requires pooled resources with explicit descriptors. When unscheduling, pooled resources for all activities occurring at a later time must be repartitioned to reflect the unscheduled resources.

Both of these techniques become extremely cumbersome and time-consuming as the size of the problem increases. If the explicit descriptors required to describe a given resource can be anticipated and the number of such descriptors is small, the various partitions of the pooled resource can, themselves, be considered a pooled resource. In this case, the pooled resource can be

Fig. 3.4.12-1



Resource Pool CREWMEN	
Before Job is Scheduled	After Job is Scheduled
12 Crewmen	5 Crewmen (Trained)
	7 Crewmen

Fig. 3.4.12-1 Partitioning of a Pooled Resource with Explicit Descriptors

treated as if it had only the implicit descriptors of IN PROCESS or AVAILABLE, thereby avoiding the complexity of modeling using pooled resources with explicit descriptors. Although data structures, conventions, and library modules are provided for scheduling that accommodate the most complex problem models, it is recommended that substantial effort be devoted to analyzing modeling alternatives that will permit greater use of well-developed technology and substantially less demanding logic design and checkout efforts. Alternatives include the definition of separate pools without explicit descriptors for each anticipated partition, and the substitution of item-specific resources with explicit descriptors for elements of the pool.

3.5 Operations Model Data Structures

122

3.5 OPERATIONS MODEL DATA STRUCTURES

This report has discussed the basic data structure of PLANS and presented the rationale for assuming a general hierarchical form. It should be noted that any "standardized" modules must make some assumptions as to the location of particular data within the overall structure. Therefore, this section defines some standard data trees for modeling an operational system. The trees discussed are summarized in Table 3.5-1. These standardized trees are an important part of the Operations Model, which is used as a framework for the library modules specified in Volume III. Obviously, these structures may be augmented to accommodate particular program peculiarities as the programmer deems necessary, or may be disregarded completely if no modules are used that assume the standardized structures. It should be recognized that the trees are defined from a "modeling" viewpoint and that few, if any, modules will use the entire content of any given tree. The "mandatory" contents assumed by each module for any given tree structure are defined in the functional specification of that module. No modules have been specified for the library that recognize the absence of required data and initiate logic to supply the missing data. PLANS coding can be used to build such routines, however. For example, logic that reads a node called DURATION and, finding no numeric value, calls a routine whose name appears as the value is easily written in PLANS. Nevertheless, no PLANS library modules contain such logic. Therefore,

it is the programmer/user's responsibility to ensure that required data are supplied to a called library module. In general, the order of nodes at a given level is insignificant.

Table 3.5-1 Operations Model Data Structures

The problem descriptions of Section 3.4 can be represented as hierarchical data trees compatible with the PLANS language.	
User-Defined Data Trees are	
\$RESOURCE	Describes the resources of the system (3.5.1)
\$PROCESS	Describes the activities of the system (3.5.2)
\$OPSEQ	Describes the operational sequences of the system (3.5.3)
\$OBJECTIVES	Describes the objectives and constraints of a problem
The problem descriptions of Section 3.4 can be used in PLANS programs to generate schedules and/or resource allocations. The information associated with problem solutions can also be represented in hierarchical data trees compatible with the PLANS language.	
Program-Created Data Trees are	
\$JOBSET	Describes each single occurrence of a process of a problem (3.5.5)
\$SCHEDULE	Describes each job and the associated resources that are assigned to a specific time interval (3.5.6)
ASSIGNMENT Subnode of \$RESOURCE	Describes the assignments made for each resource of a problem (3.5.7)

The Operations Model standard data trees can be classified either as user-defined or as program-created as shown in Table 3.5-1. User-defined data trees are structures built by a user to describe the system of interest. They provide the mechanism by which the user provides data to the program. Program-created

data trees are generated during the execution of the scheduling logic. These fundamental structures are created or used by many of the library modules. Therefore, they serve to integrate the modules and should form a basis for specific program executive logic.

3.5.1 \$RESOURCE

3.5.1 \$RESOURCE

125a

OPERATIONS MODEL DATA STRUCTURES

The problem descriptions of Section 3.4 can be represented as hierarchical data trees compatible with the PLANS language.

THE USER-DEFINED DATA TREES ARE:

\$RESOURCE	Describes the resources of the system
\$PROCESS	Describes the activities of the system
\$OPSEQ	Describes the operational sequences of the system
\$OBJECTIVES	Describes the objectives and constraints of a problem

The problem descriptions of Section 3.4 can be utilized in PLANS programs to generate schedules and/or resource allocations. The information associated with problem solutions can also be represented in hierarchical data trees compatible with the PLANS language.

THE PROGRAM-CREATED DATA TREES ARE:

\$JOBSET	Describes each single occurrence of a process of a problem
\$SCHEDULE	Describes each job and the associated resources that are assigned to a specific time interval
ASSIGNMENT subnode of \$RESOURCE	Describes the assignments made for each resource of a problem

3.5.1 \$RESOURCE

\$RESOURCE provides the structure to define all supplies, elements, or resources needed to perform any activities to be modeled in the system of interest. This definition includes the quantity and characteristics of the resource as well as a record of allocations or ASSIGNMENTS made for the given resource. (Because the ASSIGNMENTS are not defined by the user but result from execution of the program, this portion of \$RESOURCE will be discussed in the next section). As illustrated in Fig. 3.5.1-1, \$RESOURCE provides two levels of resource classification; these are arbitrarily referred to as TYPE and NAME. These two levels allow relatively quick access to a specific resource or a given class of resources. The lower level of classification (resource name) may define either an item-specific resource or resource pool.

As discussed previously, it is necessary to distinguish between item-specific and pooled resources. Therefore, the next sublevel contains a node labeled CLASS with expected values being either SPECIFIC or POOL. In addition, this level provides the initial time and description of the given resource. This description has a node for each descriptive parameter required. The node label names the parameter and the node value gives the corresponding parametric value. Thus, descriptive characteristics such as length, weight, color, location, etc that apply to the resource at the initial time may be specified.

Fig. 3.5.1-1

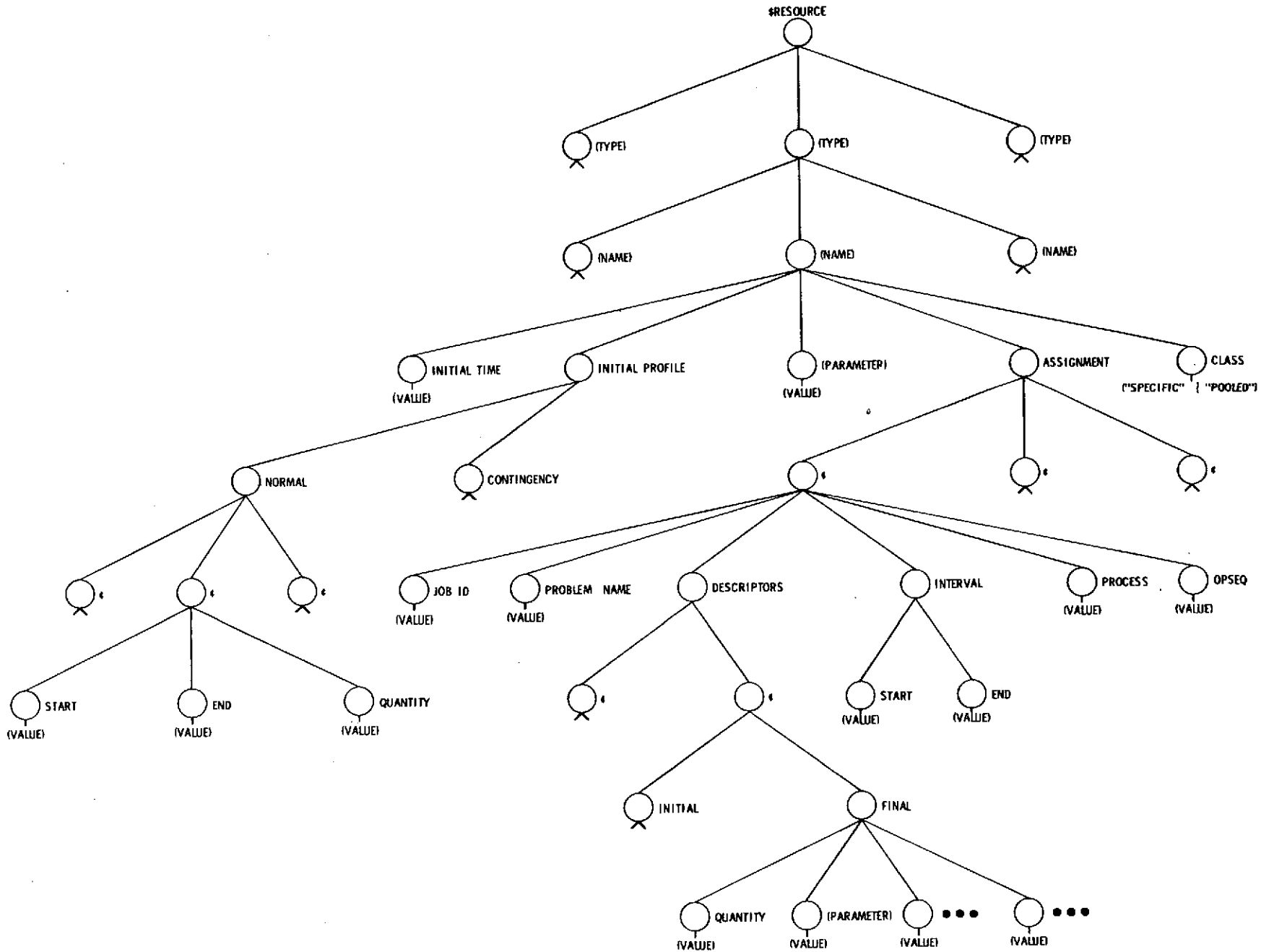


Fig. 3.5.1-1 \$RESOURCE Standard Data Structure

Initial quantities for pooled resources may be time-varying. To accommodate this possibility, a node labeled INITIAL_PROFILE appears in \$RESOURCE. The substructure under this node provides the mechanism for defining variable piecewise constant profiles for both normal and contingency quantity levels. The profile data are typically used with project scheduling techniques. It should be stressed that all the values under the INITIAL_PROFILE node and the other nodes at that level represent initial values. Any subsequent changes to the descriptive characteristics resulting from assignments to specific processes would be recorded in the assignment portion of \$RESOURCE. See Section 3.5.6.

An illustration of the use of \$RESOURCE for a Shuttle application is shown in Fig. 3.5.1-2.

```

$RESOURCE
SRB
  SRB
    QUANTITY - 18
    CLASS - POOL
  VAB HIGH BAY
    HIGH BAY NO. 1
      QUANTITY - 1
      LOCATION - BAY 1
      CLASS - SPECIFIC
    HIGH BAY NO. 2
      QUANTITY - 1
      LOCATION - BAY 2
      CLASS - SPECIFIC
  PERSONNEL
    SRB/EXT. TANK CREW
      QUANTITY - 55
      QUALIFICATIONS - ASSEMBLE SRBS
                    - PREPARE EXT. TANKS
                    - MATE TANK AND SRB
                    - REFURBISH LUT
      CLASS - POOL
    LAUNCH CREW
      QUANTITY - 75
      QUALIFICATIONS - SERVICE SHUTTLE
                    - LAUNCH OPS
                    - REFURBISH PAD
      CLASS - POOL
    ORBITER/PAYLOAD CREW
      QUANTITY - 45
      QUALIFICATIONS - PERFORM PAYLOAD OPS
                    - PREPARE ORBITER
                    - RECYCLE ORBITER
                    - MATE ORBITER AND TANK
      CLASS - POOL
    ASSIGNMENT -
    MISSION CONTROL
      QUANTITY - 65
      QUALIFICATIONS - ON-ORBIT OPS
                    - DEORBIT AND LAND
      CLASS - POOL
    CREW OPS
      QUANTITY - 75
      QUALIFICATIONS - CREW TRAINING
                    - MISSION BRIEFING
                    - FLIGHT CREW PREP
                    - DEBRIEFING
      CLASS - POOL
  LAUNCH UMBILICAL TOWER
  LUT
    QUANTITY - 3
    CLASS - SPECIFIC
  EXT. TANK
  EXT. TANK
    QUANTITY - 7
    CLASS - POOL
  ORBITER
  ORBITER
    QUANTITY - 3
    CLASS - SPECIFIC
  LAUNCH PAD
  LAUNCH PAD NO. 1
    QUANTITY - 1
    LOCATION - PAD 1
    CLASS - SPECIFIC
  LAUNCH PAD NO. 2
    QUANTITY - 1
    LOCATION - PAD 2
    CLASS - SPECIFIC
  CREW
  CREW
    QUANTITY - 15
    CLASS - POOL
  PAYLOAD
  PAYLOAD
    QUANTITY - 163

```

Fig. 3.5.1-2
\$RESOURCE Data Tree for
a Shuttle Application

3.5.2 \$PROCESS

130-a

OPERATIONS MODEL DATA STRUCTURES

The problem descriptions of Section 3.4 can be represented as hierarchical data trees compatible with the PLANS language.

THE USER-DEFINED DATA TREES ARE:

\$RESOURCE	Describes the resources of the system
\$PROCESS	Describes the activities of the system
\$OPSEQ	Describes the operational sequences of the system
\$OBJECTIVES	Describes the objectives and constraints of a problem

The problem descriptions of Section 3.4 can be utilized in PLANS programs to generate schedules and/or resource allocations. The information associated with problem solutions can also be represented in hierarchical data trees compatible with the PLANS language.

THE PROGRAM-CREATED DATA TREES ARE:

\$JOBSET	Describes each single occurrence of a process of a problem
\$SCHEDULE	Describes each job and the associated resources that are assigned to a specific time interval
ASSIGNMENT subnode of \$RESOURCE	Describes the assignments made for each resource of a problem

180-b

3.5.2 \$PROCESS

This data structure defines the activities or processes of the system to be modeled. Obviously, the level of detail to be included determines the content of the \$PROCESS tree. For example, a top-level analysis of a system may group many activities into one process with a corresponding total duration. In such instances some or all of the resources involved in the activities may not be specified. In contrast, the problem objective may require such detail as the skill profile of each person who is considered a resource, and/or variable use of resources during the process time interval. The structure of \$PROCESS, as shown in Fig. 3.5.2-1, allows both extremes to be modeled. Again it should be emphasized that only applicable portions of the structure need to be specified. Thus, conceivably, a process definition could consist of as little as the process name and a corresponding duration.

The first level subnodes to the process name define the process duration and type. In this usage, *type* refers to whether or not a process may be split into segments to facilitate scheduling around constraints. Expected values would be either SPLITTABLE or NOT_SPLITTABLE. A programmer may wish to provide a default value for *type* to permit simplification of program input, but the functional specifications for library modules leave all default nodes to the discretion of the implementer. This level also labels the nodes needed to define the relationship of resources to the process being defined.

Fig. 3.5.2-1

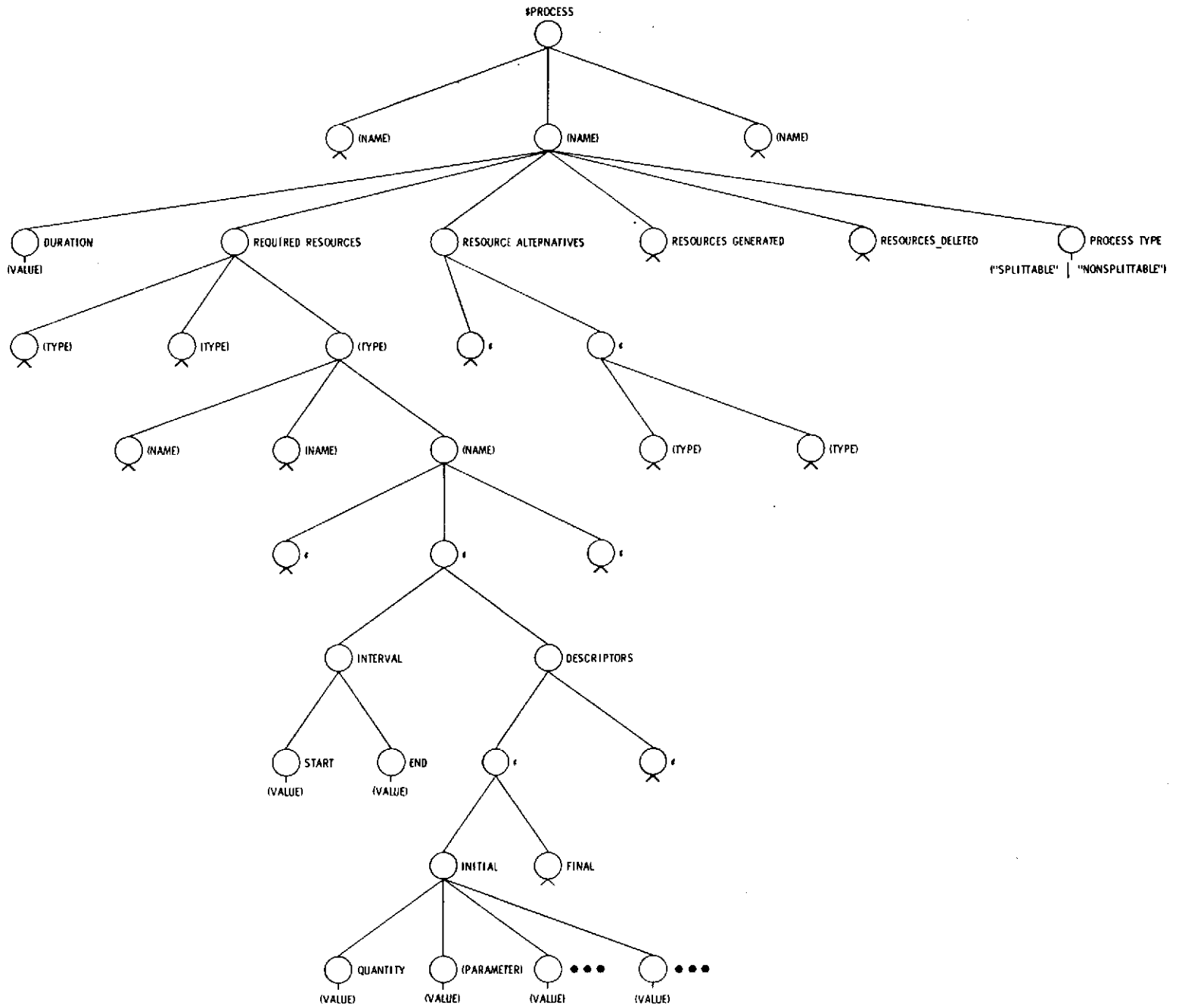


Fig. 3.5.2-1 \$PROCESS Standard Data Structure

The substructure of REQUIRED_RESOURCES contains the most specific information required to ensure a satisfactory resource selection for the process. Thus, if any resource of a given *type* (first sublevel to REQUIRED_RESOURCE) will suffice, only the resource *type* is defined. (The *name* level is left null.) Conversely, a process may require a resource of a given *type* and *name*, and in addition require a particular value of a given *descriptor* at the initiation of the process. In this instance, an INITIAL_DESCRIPTOR would be specified for the given resource. As illustrated in the structural diagram of \$PROCESS, any number of resource *types* and/or *names* may be required by a given process. It is assumed that each resource denoted by a node at the *name* level is required to complete the process. For each resource name (indicated by a node), any number of descriptors may be specified for a given time interval. It is assumed that any initial descriptor defines a requirement on the resource that must be met for a resource to be an acceptable candidate. A FINAL_DESCRIPTOR only indicates a change in a particular descriptor resulting from the accomplishment of a process. The START and END values specified for a given INTERVAL will define an interval relative to the process duration. Thus, if the value of DURATION is 10, a START and END value of 5 and 10, respectively, would indicate that the resource was only required for the last half of the process.

Processes are the mechanism for associating resources with activities. For this reason, multiple requirements for resources defined within a single process will be satisfied by a collection of resources and the assignments of each resource will be identified by the process name but not by a separate requirements identifier. For example, a single process that has a requirement for one crewman and another requirement for a different crewman will ultimately cause two crewmen to have assignments identified by the process name. It is assumed that the association of each crewman with a particular requirement will not be preserved. If the association is to be preserved, two separate processes should be defined. In the Operations Model, a process contains a set of requirements to which resources can be assigned unambiguously.

The format of the data structures subordinate to `ALTERNATE_RESOURCES`, `RESOURCES_GENERATED`, and `RESOURCES_DELETED` is similar to that for `REQUIRED_RESOURCES`, but functionally, each serves a different purpose. `ALTERNATE_RESOURCES` defines any number of sets of alternative resources. Each set is represented with a *null* label (shown as "ø" in Fig. 3.5.2-1). It is assumed that one alternative from each set must be provided for completion of a given process. Thus, an executive program must consider both nodes, `REQUIRED_RESOURCES` and `ALTERNATE_RESOURCES`, when determining the availability of resources required to complete a process.

`RESOURCES_GENERATED` and `RESOURCES_DELETED` specify what happens to resources during a process. In some cases resources will be assembled (or disassembled) to create new resources. Correspondingly,

the initial elements will no longer exist as the described resource. For example, if a process mates an Orbiter to a stack, the resulting assembly may be referred to as a Shuttle. In this case, the resource *Orbiter* and *stack* would be described as RESOURCES_DELETED and the resource *Shuttle* would be a RESOURCE_GENERATED. These nodes, therefore, allow a means of traceability for a particular resource. These two substructures also describe resources that are usually thought of as "expendables" or "consumables". A resource, such as power, dollars, or fuel, that is in fact consumed and will not reappear in the system at a later time, would be a deleted resource. Similarly, a "negative" consumable, such as the refining of a petroleum product, would create resources during a process. An illustration of a portion of \$PROCESS for a Shuttle application is shown in Fig. 3.5.2-2.

```

$PROCESS
  RECYCLE SRB
  DURATION - 11
  REQUIRED RESOURCES
    SRB
    SRB
    INTERVAL
    START - 0
    END - 11
    DESCRIPTORS
    INITIAL
    QUANTITY - 2
    STATUS - TO BE RECOVERED
    FINAL
    QUANTITY - 2
    STATUS - TO BE ASSEMBLED
  ASSEMBLE SRB PAIR
  DURATION - 0.4
  REQUIRED RESOURCES
    SRB
    SRB
    INTERVAL
    START - 0
    END - 47
    DESCRIPTORS
    INITIAL
    QUANTITY - 2
    STATUS - TO BE ASSEMBLED
    FINAL
    STATUS - TO BE MATED
  VAB HIGH BAY
  VAB
  INTERVAL
  START - 0
  END - 47
  DESCRIPTORS
  INITIAL
  STATUS - AVAILABLE
  FINAL
  status - IN USE
  PERSONNEL
  SRB/EXT TANK CREW
  INTERVAL
  START - 0
  END - 47
  DESCRIPTORS
  INITIAL
  QUANTITY - 30
  QUALIFICATIONS - ASSEMBLE SRBS
  LAUNCH UMBILICAL TOWER
  LUT
  INTERVAL
  START - 0
  END - 47
  DESCRIPTORS
  INITIAL
  QUANTITY - 1
  STATUS - AVAILABLE
  FINAL
  STATUS - IN USE
  RESOURCES GENERATED
  SRB PAIR
  SRB PAIR
  DESCRIPTORS
  FINAL
  QUANTITY - 1
  RESOURCES DELETED
  SRB
  SRB
  DESCRIPTORS
  INITIAL
  QUANTITY - 2
  FINAL
  QUANTITY - 0

```

Fig. 3.5.2-2
Use of \$PROCESS for a Shuttle
Application

3.5.3 \$OPSEQ

136-a

OPERATIONS MODEL DATA STRUCTURES

The problem descriptions of Section 3.4 can be represented as hierarchical data trees compatible with the PLANS language.

THE USER-DEFINED DATA TREES ARE:

\$RESOURCE	Describes the resources of the system
\$PROCESS	Describes the activities of the system
\$OPSEQ	Describes the operational sequences of the system
\$OBJECTIVES	Describes the objectives and constraints of a problem

The problem descriptions of Section 3.4 can be utilized in PLANS programs to generate schedules and/or resource allocations. The information associated with problem solutions can also be represented in hierarchical data trees compatible with the PLANS language.

THE PROGRAM-CREATED DATA TREES ARE:

\$JOBSET	Describes each single occurrence of a process of a problem
\$SCHEDULE	Describes each job and the associated resources that are assigned to a specific time interval
ASSIGNMENT subnode of \$RESOURCE	Describes the assignments made for each resource of a problem

136--6

3.5.3 \$OPSEQ

The operations sequence data structure serves to define the relationship between processes that are combined into a network. As illustrated in Fig. 3.5.3-1, the labels of the first level subnodes are the names of different operations sequences. The labels of the second-level subnodes are the names of the elements of the above-named operations sequence. The element is either a process or another operations sequence; if it is an operations sequence, its name will also appear elsewhere at the first level to define its substructure. If the element is a process, its characteristics are described by the \$PROCESS tree. The element TYPE is defined by the next level subnode with an expected value of either PROCESS or OPSEQ. The module CHECK_PROCESS_DEFINITION provides a capability to check whether all processes referred to in the nested operations sequences are defined in \$PROCESS.

This level also defines the general temporal relationship of an element to any other element in the same operations sequence. This data structure (see Fig. 3.5.3-2) allows the user to specify classical predecessors and successors as well as generalized temporal relations. The general temporal relation is specified by six *ordered* subnodes as indicated in Fig. 3.5.3-2. (It is possible that appropriate labels could be designated, and interpreted, to eliminate the requirement to be ordered, but since the order is logical and unambiguous, the labels have not been assumed by any library modules.)

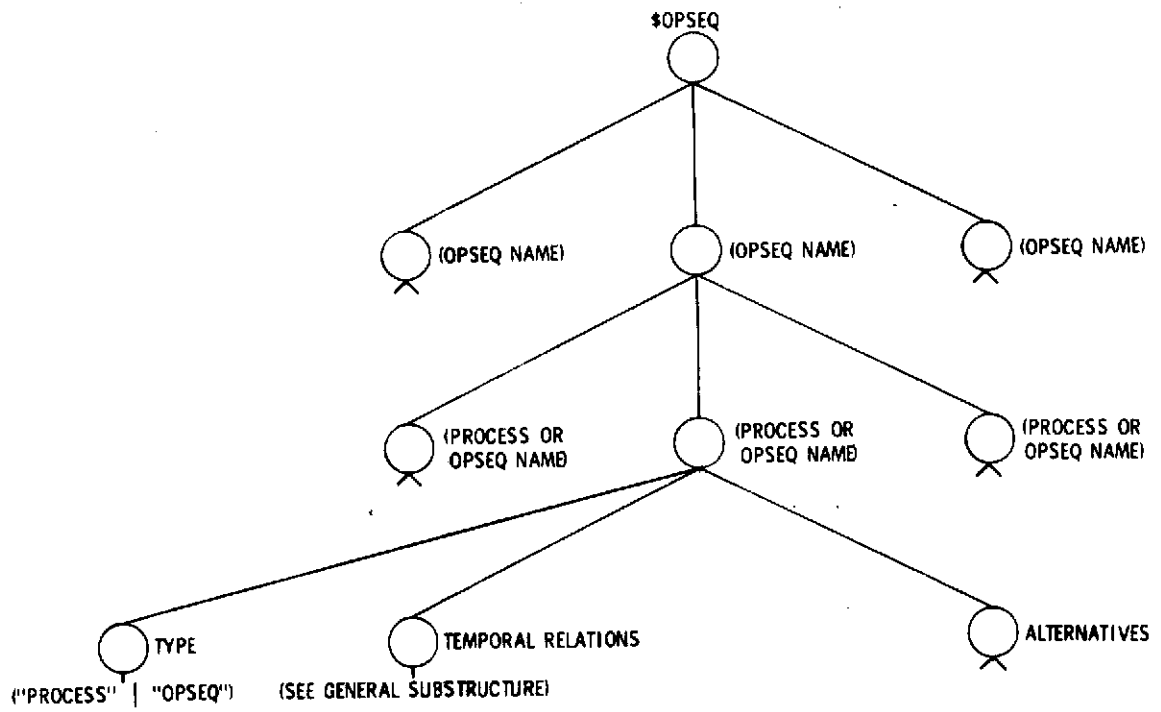


Fig. 3.5.3-1 \$OPSEQ Standard Data Structure

Fig. 3.5.3-2

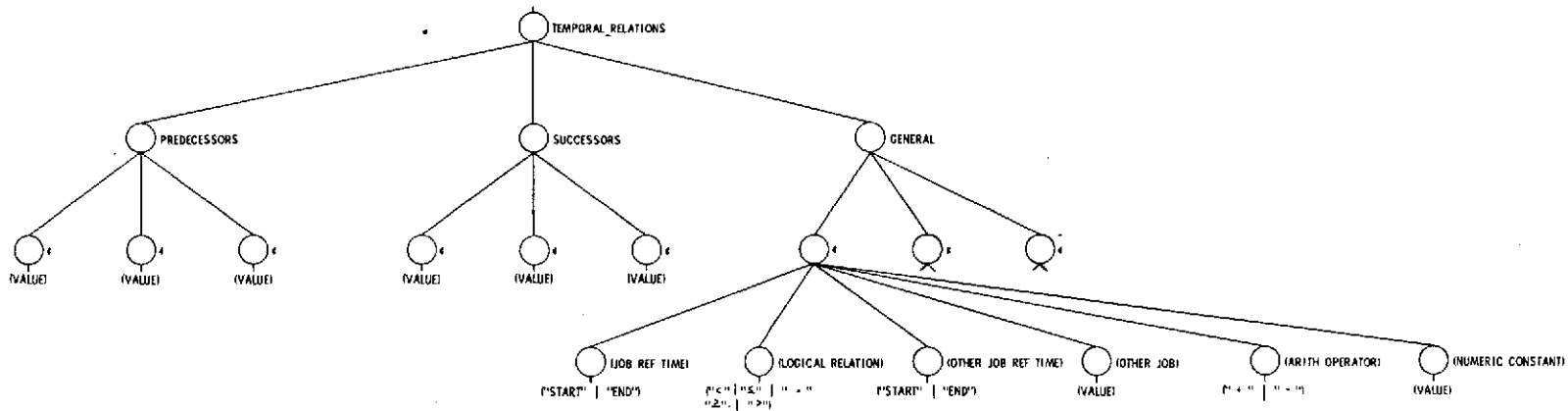


Fig. 3.5.3-2 General Substructure for TEMPORAL_RELATIONS

Finally, any alternatives to an element may be listed as values of subnodes to ALTERNATIVES. Alternatives represent OR gates in an operations sequence. As discussed in the previous section, these alternatives may be either processes or operations sequences.

An illustration of the use of \$OPSEQ for a Shuttle application is shown in Fig. 3.5.3-3.

```

$OPSEQ
SHUTTLE SYSTEM MISSION FLOW
ASSEMBLE SRB PAIR
TYPE - PROCESS
PREPARE EXT. TANK
TYPE - PROCESS
MATE EXT. TANK TO SRB
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
ASSEMBLE SRB PAIR
PREPARE EXT. TANK
MATE ORBITER TO EXT. TANK
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
MATE EXT. TANK TO SRB
PREPARE ORBITER FOR LAUNCH
SERVICE SHUTTLE FOR LAUNCH
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
MATE ORBITER TO EXT. TANK & SRB
LAUNCH PHASE OPERATIONS
TYPE - PROCESS
TEMPORAL RELATIONS
GENERAL
c
START
EQUAL TO
END
SERVICE SHUTTLE FOR LAUNCH
PREDECESSOR
PREPARE CREW FOR FLIGHT
PREPARE CREW FOR FLIGHT
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
PERFORM MISSION BRIEFING
GENERAL
d
END
EQUAL TO
END
SERVICE SHUTTLE FOR LAUNCH
REFURBISH LAUNCH PAD
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
LAUNCH PHASE OPERATIONS
RECYCLE SRB
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
LAUNCH PHASE OPERATIONS
PERFORM ON-ORBIT OPERATIONS
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
LAUNCH PHASE OPERATIONS
DEORBIT, REENTRY AND LAND
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
DEORBIT, REENTRY AND LAND
RECYCLE ORBITER
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
DEORBIT, REENTRY AND LAND
PERFORM CREW TRAINING OPS
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
PERFORM PAYLOAD OPS
PERFORM MISSION BRIEFING
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
PERFORM CREW TRAINING OPS
PERFORM PAYLOAD OPS
TYPE - PROCESS
PREPARE ORBITER FOR LAUNCH
TYPE - PROCESS
TEMPORAL RELATIONS
PREDECESSOR
PERFORM PAYLOAD OPS

```

Fig. 3.5.3-3
Use of \$OPSEQ for a
Shuttle Application

3.5.4 OBJECTIVES

141-a

OPERATIONS MODEL DATA STRUCTURES

The problem descriptions of Section 3.4 can be represented as hierarchical data trees compatible with the PLANS language.

THE USER-DEFINED DATA TREES ARE:

\$RESOURCE	Describes the resources of the system
\$PROCESS	Describes the activities of the system
\$OPSEQ	Describes the operational sequences of the system
\$OBJECTIVES	Describes the objectives and constraints of a problem

The problem descriptions of Section 3.4 can be utilized in PLANS programs to generate schedules and/or resource allocations. The information associated with problem solutions can also be represented in hierarchical data trees compatible with the PLANS language.

THE PROGRAM-CREATED DATA TREES ARE:

\$JOBSET	Describes each single occurrence of a process of a problem
\$SCHEDULE	Describes each job and the associated resources that are assigned to a specific time interval
ASSIGNMENT subnode of \$RESOURCE	Describes the assignments made for each resource of a problem

142

3.5.4 \$OBJECTIVES

The three previous data trees provide a method for describing an operational system (or any number of systems). These trees may be constructed at any time and stored and are independent of the particular problem of interest at any given time. As a matter of fact, they are independent of the scheduling function for which they will probably be used. However, there also exist certain data that are required as input for a particular problem concerning a given system. These data may be structured in a variety of ways, and will contain different information for different problems. \$OBJECTIVES, as defined here, presents some data needed by certain library modules in a structure that may be used and augmented by a program developer. As programs are developed in PLANS, \$OBJECTIVES will certainly evolve beyond those presented here with additional conventions and structure.

As shown in Fig. 3.5.4-1, the first level subnodes include a method of inputting a problem name. Obviously, other problem identifiers may be included by additional nodes. A *Figure of Merit* node is shown to indicate a location for the name input for a routine that calculates the objective function of the problem. Because none of the specified library modules require these particular data, the detailed specification is not provided here. Similarly, a CONSTRAINT node is indicated for probable inclusion by a user of \$OBJECTIVES. These constraints would apply to the overall problem--not to a specific process or operations sequence. Constraints could include the earliest and latest start and end

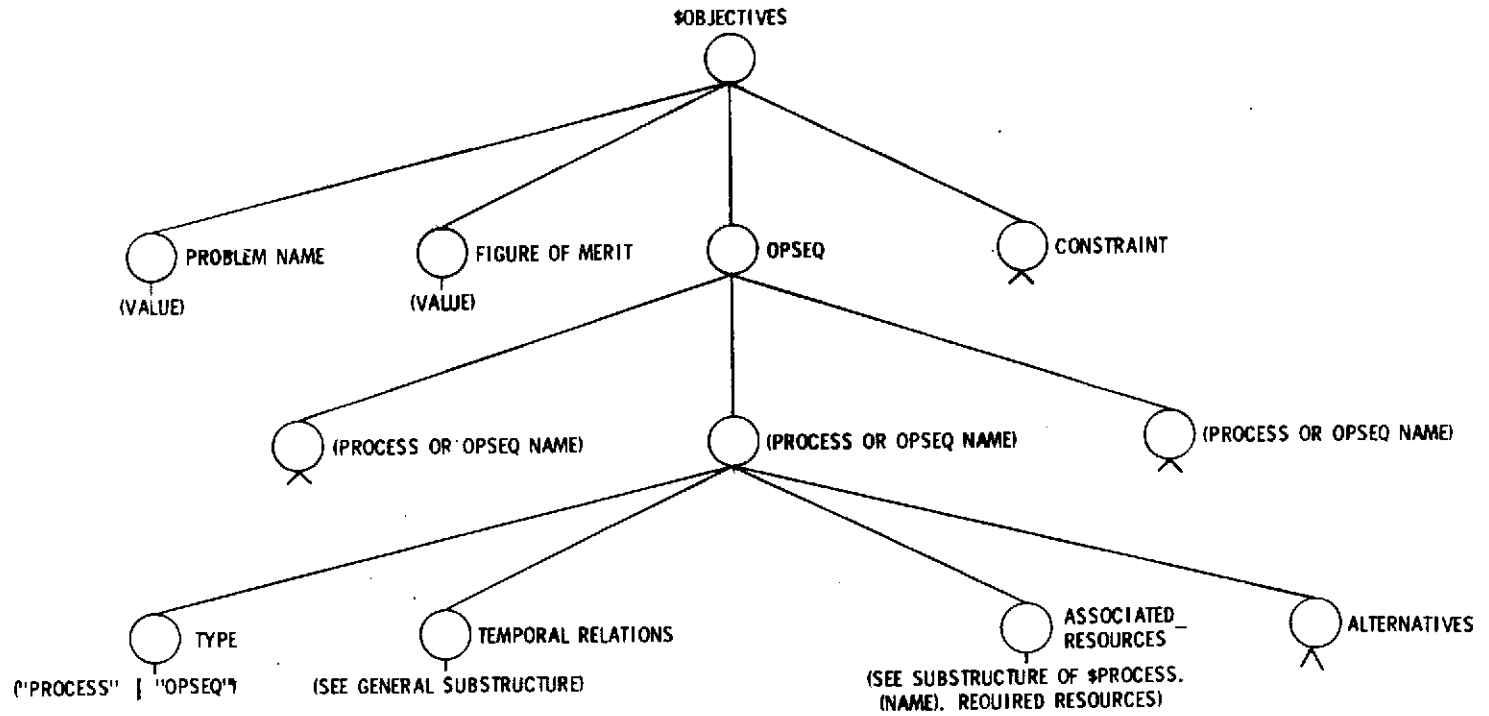


Fig. 3.5.4-1 \$OBJECTIVES Standard Data Structure

times (absolute times), maximum total duration, and/or specific "windows" that may or may not be used for scheduling. For example, a given shop operation to be scheduled may have an earliest start date specified because of expected shipping of supplies. Also, all weekend dates may be unavailable for scheduled activities because the shop works on a five-day work week.

The information in \$OBJECTIVES that is required by the currently-specified library modules is the list of operations sequences to be scheduled and any specific resources to be associated with any process or operations sequence. The first level subnodes to OPSEQ list the processes and/or operations sequences that must be scheduled for successful completion of a problem. The next sublevel lists the TYPE (either PROCESS or OPSEQ) and any temporal relationships that exist between that element and any other element under OPSEQ. The format of the temporal relationship is the same as discussed in the preceding section on \$OPSEQ. Obviously, one way of setting up a problem would be to create an operations sequence in \$OPSEQ containing all the elements desired for a given problem. Then \$OBJECTIVE would only list one operations sequence to be considered and the problem input would be greatly simplified. In fact this would be a recommended approach if a similar problem were to be considered numerous times. However, the basic philosophy has been to consider a more flexible approach in which more elemental subnetworks are created from individual processes. A number of these operations sequences could then be called out in \$OBJECTIVES to synthesize a given problem. The substructure of

the ASSOCIATED_RESOURCES is the same as the \$PROCESS.(NAME).REQUIRED_RESOURCES) substructure and is used to specify any specific resources the user wants associated with the corresponding operations sequence. It is assumed that the characteristics of the particular resource have previously been filed in the \$RESOURCE tree. This substructure would be used if the user wished to execute an operations sequence such as LAUNCH PAYLOAD three separate times with an associated resource for each launch of PAYLOAD 14, PAYLOAD 27, and PAYLOAD 87, respectively.

An example of the use of \$OBJECTIVES for a specific problem appears in Section 4.2.

3.5.5 \$JOBSET

This data tree will be created near the beginning of a problem from the three defined data trees \$OBJECTIVES, \$OPSEQ, and \$PROCESS by the module GENERATE_JOBSET (see Volume III). As illustrated by Fig. 3.5.5-1 this tree combines processes with all required resources (whether they are specified or generic) into *jobs*. Thus, a *job is a single occurrence of a process* and is the element of the Operations Model that is scheduled by the solution algorithm. Once created, \$JOBSET contains most of the input data required to work a given problem and makes further reference to the stored trees \$OPSEQ and \$PROCESS unnecessary.

The first-level subnodes of \$JOBSET represent a collection of single occurrences of the processes in an operations sequence. The descendants of these nodes are unique job identifiers. The first level subnodes to the job identifiers specify a JOB_TYPE, which indicates whether a job may be interrupted and scheduled in more than one segment. The JOB_INTERVAL node contains a relative interval equal to the duration of the process involved. The substructure of RESOURCES will be created for each required resource and the most specific input information. Other information such as identifiers, alternatives, and temporal relationships to other jobs are also included in \$JOBSET.

Fig. 3.5.5-1

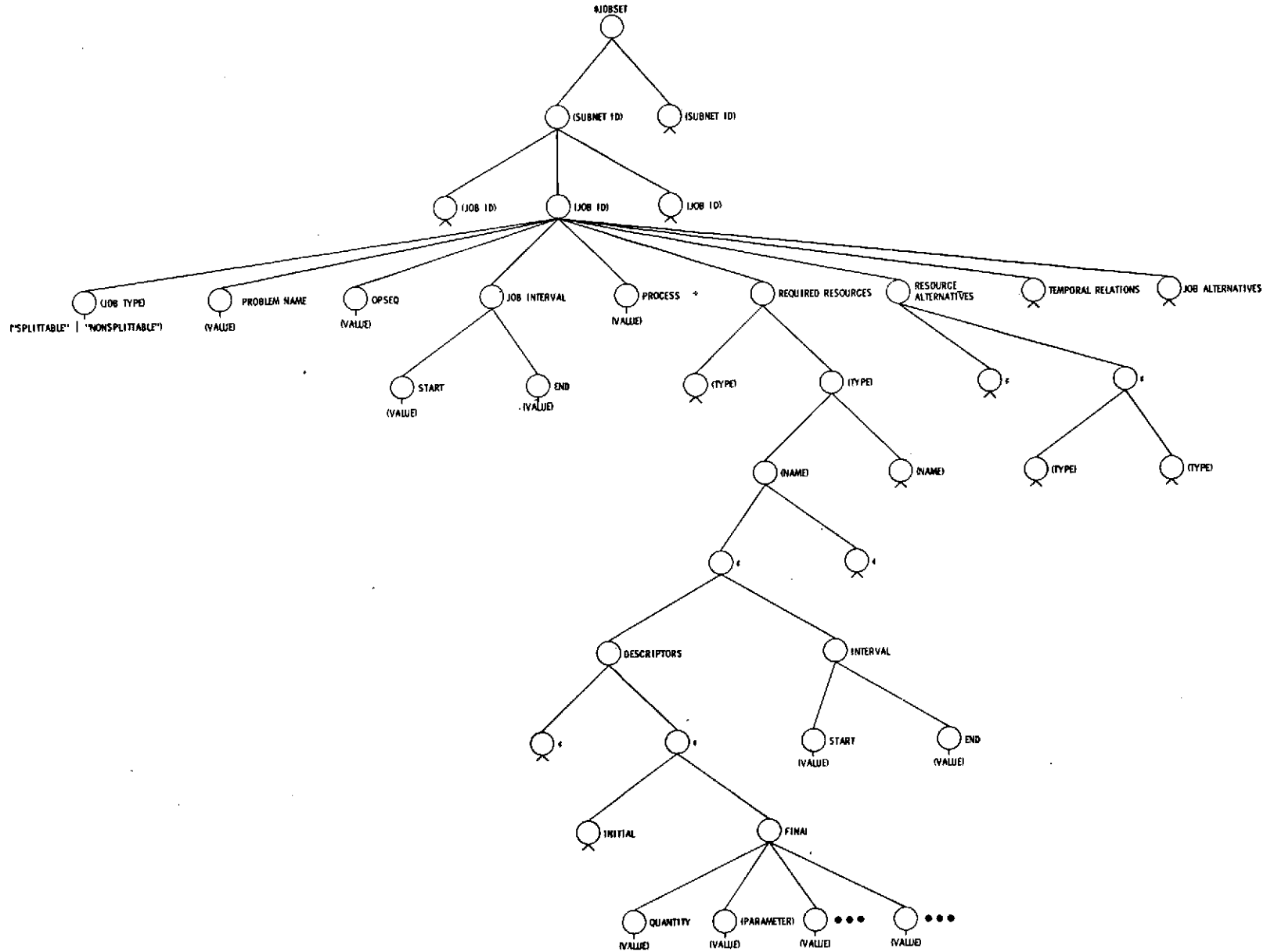


Fig. 3.5.5-1 \$JOBSET Standard Data Structure

3.5.5 \$JOBSET

148a

OPERATIONS MODEL DATA STRUCTURES

The problem descriptions of Section 3.4 can be represented as hierarchical data trees compatible with the PLANS language.

THE USER-DEFINED DATA TREES ARE:

\$RESOURCE	Describes the resources of the system
\$PROCESS	Describes the activities of the system
\$OPSEQ	Describes the operational sequences of the system
\$OBJECTIVES	Describes the objectives and constraints of a problem

The problem descriptions of Section 3.4 can be utilized in PLANS programs to generate schedules and/or resource allocations. The information associated with problem solutions can also be represented in hierarchical data trees compatible with the PLANS language.

THE PROGRAM-CREATED DATA TREES ARE:

\$JOBSET	Describes each single occurrence of a process of a problem
\$SCHEDULE	Describes each job and the associated resources that are assigned to a specific time interval
ASSIGNMENT subnode of \$RESOURCE	Describes the assignments made for each resource of a problem

148-b

3.5.6 \$SCHEDULE

148-C

OPERATIONS MODEL DATA STRUCTURES

The problem descriptions of Section 3.4 can be represented as hierarchical data trees compatible with the PLANS language.

THE USER-DEFINED DATA TREES ARE:

\$RESOURCE	Describes the resources of the system
\$PROCESS	Describes the activities of the system
\$OPSEQ	Describes the operational sequences of the system
\$OBJECTIVES	Describes the objectives and constraints of a problem

The problem descriptions of Section 3.4 can be utilized in PLANS programs to generate schedules and/or resource allocations. The information associated with problem solutions can also be represented in hierarchical data trees compatible with the PLANS language.

THE PROGRAM-CREATED DATA TREES ARE:

\$JOBSET	Describes each single occurrence of a process of a problem
\$SCHEDULE	Describes each job and the associated resources that are assigned to a specific time interval
ASSIGNMENT subnode of \$RESOURCE	Describes the assignments made for each resource of a problem

148-d

3.5.6 \$SCHEDULE

\$SCHEDULE contains the *answer* to the scheduling problem. That is, it contains the same jobs as \$JOBSET, but now each job interval contains specific times (i.e., not relative) and specific resources have been identified for each job. No resource or job alternatives exist in \$SCHEDULE because these selections have been made and no temporal relationships are needed because absolute times have been assigned for each job. Resource intervals relative to the job interval have been replaced with absolute times. In other words, the schedule has been *concretized* (see Fig. 3.5.6-1 for the basic structure). \$SCHEDULE consists of any number of *schedule units*, which contain all of the substructure for any one job identifier. This schedule unit is considered the smallest element that may be scheduled. The individual schedule unit contains all information necessary to maintain a record of resource allocations made and corresponding descriptors that apply to the specified resource and time interval.

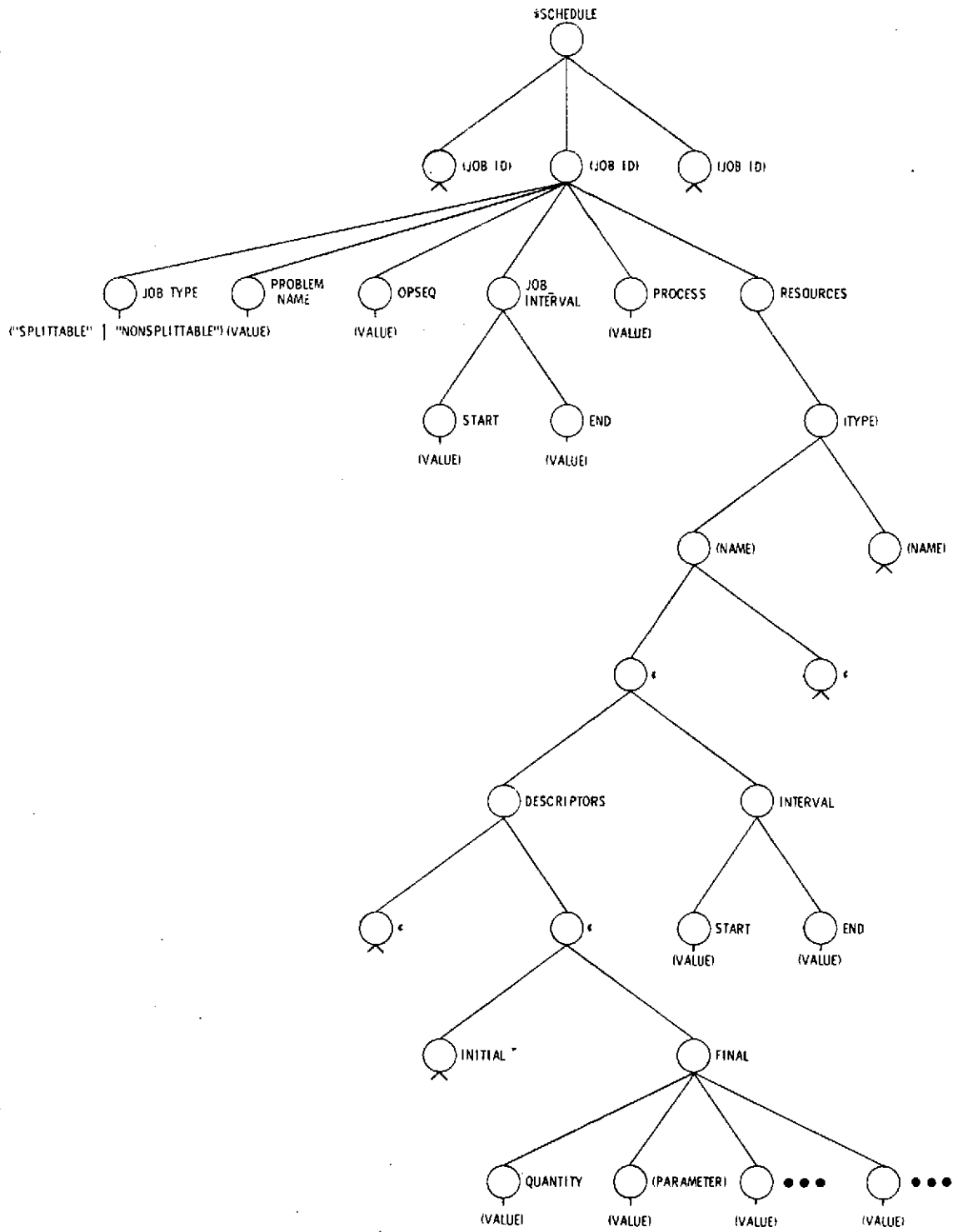


Fig. 3.5.6-1 \$SCHEDULE Standard Data Structure

3.5.7 Assignment Subnode of
\$RESOURCE

3.5.7 Assignment Subnode of \$RESOURCE

150-a

OPERATIONS MODEL DATA STRUCTURES

The problem descriptions of Section 3.4 can be represented as hierarchical data trees compatible with the PLANS language.

THE USER-DEFINED DATA TREES ARE:

\$RESOURCE	Describes the resources of the system
\$PROCESS	Describes the activities of the system
\$OPSEQ	Describes the operational sequences of the system
\$OBJECTIVES	Describes the objectives and constraints of a problem

The problem descriptions of Section 3.4 can be utilized in PLANS programs to generate schedules and/or resource allocations. The information associated with problem solutions can also be represented in hierarchical data trees compatible with the PLANS language.

THE PROGRAM-CREATED DATA TREES ARE:

\$JOBSET	Describes each single occurrence of a process of a problem
\$SCHEDULE	Describes each job and the associated resources that are assigned to a specific time interval
ASSIGNMENT subnode of \$RESOURCE	Describes the assignments made for each resource of a problem

150-6

3.5.7 ASSIGNMENT Subnode of \$RESOURCE

While the basic inputs to \$RESOURCE consist of resources and their characteristics as defined by a user, an equally important function of the \$RESOURCE tree is to maintain a record of all allocations made for each resource. Even though these allocations are handled by the scheduling program, the close relationship between the initial resource descriptions and the subsequent descriptions brought about by assignments to certain activities, justifies the existence of both in the same data tree. Therefore, as shown in the \$RESOURCE tree in Fig. 3.5.1-1, all assignments are recorded as a substructure to the ASSIGNMENT node of \$RESOURCE.

Assignments are arranged by increasing start times with equal start times being ordered by earliest end time. Such an ordering will facilitate the checking for resources by subsequent scheduling attempts. Each assignment is indicated on the \$RESOURCE tree diagram by a *null* labeled (c) node. The assignment will consist of an INTERVAL, DESCRIPTORS, and any other information included as a corresponding part of the *schedule unit* from which the update of the assignment file is made. Library modules WRITE_ASSIGNMENT and UPDATE_RESOURCE have been designed to accomplish this update. Additional identification information will have to be included in the basic schedule unit to allow subsequent *un-scheduling*. These data, to be included as first-level subnodes to the null assignments nodes, could include the problem name, the operations sequence involved, etc. The library module UNSCHEDULE has been designed to remove assignments from the \$RESOURCE tree.

The DESCRIPTORS portion of the assignment structure allows subsequent investigation to determine the *history* of any given resource. That is, the state of an item-specific resource or the partitioning and corresponding quantities of pooled resources may be determined as a function of time. The library module RESOURCE_PROFILE is designed to determine the quantity in a resource pool as a function of time for the pooled resources with implicit descriptors. Two variables determine the available quantity of a pooled resource. The assignments affect the available quantity because any resources assigned are assumed to be unavailable during the assignment interval. Secondly, resources may be either generated or deleted as the result of a process. For purposes of creating a profile, it is assumed that any quantity deleted or generated is reflected at the end time of the process interval. This would be indicated by an appropriately labeled FINAL_DESCRIPTOR with a value of either GENERATED or DELETED. The corresponding quantity would be indicated with the subnode QUANTITY.

DESCRIPTOR_PROFILE is a module that uses the assignment information in \$RESOURCE to determine the history of changes to the descriptors for an item-specific resource. These resources have an initial description that is updated as assignments are made. Obviously, if a resource is being considered for a specific process, it will be necessary to know the current set of descriptors that apply for a time interval of interest. Also, recognizing that an item-specific resource may be assembled with other resources to generate a new resource, a FINAL_DESCRIPTOR of DELETED

could apply. Therefore, when assessing the availability of an item-specific resource, it is not sufficient to check only for conflicting assignments. A further check must be made of the final descriptors for the most recent assignment to ensure that the resource was not deleted. Similarly, an item specific resource may be generated (or regenerated) by the disassembly of a resource.

3.6 Heuristic and Mathematical Programming Solution Techniques

154

3.6 HEURISTIC AND MATHEMATICAL PROGRAMMING SOLUTION TECHNIQUES

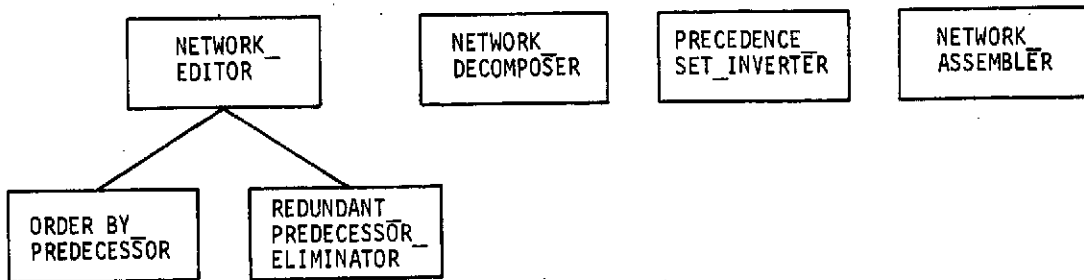
An analysis of the state-of-the-art in computerizable scheduling techniques leads to the conclusion that two major bodies of standardized methodology are available (see Appendix). The first is referred to here as project scheduling and is based on heuristic scheduling rules. The second is called mathematical programming and is based on procedures that produce mathematically optimal solutions. The specified PLANS module library contains routines in both classes of solution methods. However, each class of methods is capable of dealing with problem models that have only limited generality or dimensionality. Problems that are described with more general models and/or higher dimensionality, however, will be solved by building problem-dependent logic. While PLANS is designed to aid in programming such logic, the analyst will make better use of the PLANS programming system by describing his problem in a format that makes one of the two standardized methodologies applicable. If this is done, the capabilities represented in the module library can then be applied, and will permit much more rapid program development and checkout.

The purpose of this section is to describe the characteristics of the problem models that are compatible with existing standardized solution methodologies so that the user of both PLANS and the PLANS module library will have the maximum capabilities available to him. It should be noted at this point that nothing in PLANS precludes its use with nonstandard methodologies. To be sure, PLANS makes customized scheduling logic much easier to program.

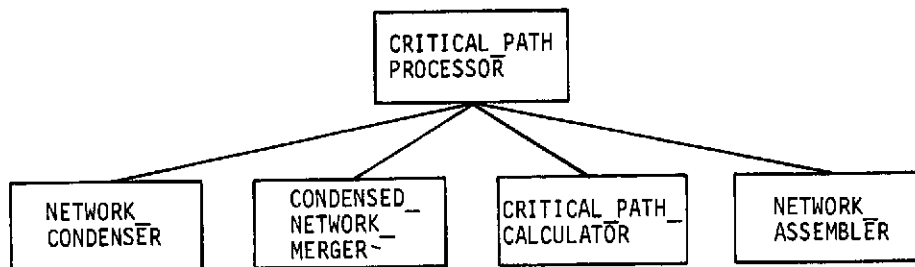
Similarly, the module library is not limited to capabilities associated with project scheduling, mathematical programming, or solution methods; in fact, many of the modules perform functions that must be performed in typical problem-dependent scheduling logic.

Combination of the generalized network modules and the resource allocation and smoothing modules specified for the module library constitutes a *project scheduling system* that is applicable to very large problems that have the problem models described above. The logical relationships of these modules is shown in Fig. 3.6-1. The collection of project scheduling modules provides state-of-the-art solution capability for that class of problems. It should be noted that each individual module performs a separable and useful function in its own right and thus may be used in any custom-made logic that the user designs. Executive modules (i.e., NETWORK_EDITOR, CRITICAL_PATH_PROCESSOR, and HEURISTIC_SCHEDULING_PROCESSOR) are also provided, however, that call other modules in order to execute a particular solution strategy. This strategy produces near-minimal time schedules, which satisfy both resource level constraints and network constraints (precedences) and also produce *smoothed* resource demand profiles.

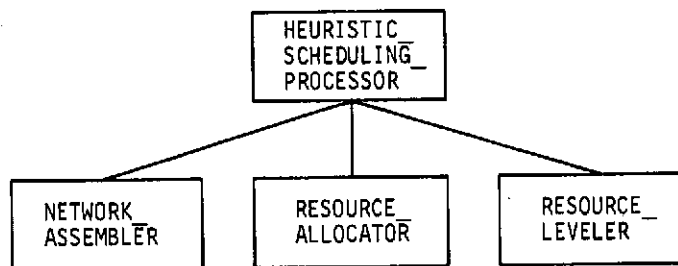
The problem characteristics that are accommodated by the PLANS project scheduling modules include those described in Subsections 3.4.1 through 3.4.5 and illustrated in Fig. 3.6-2. In addition, project scheduling techniques can be applied to more general problem descriptions by applying special reformatting



(a) Network-Processing Modules Calling Hierarchy



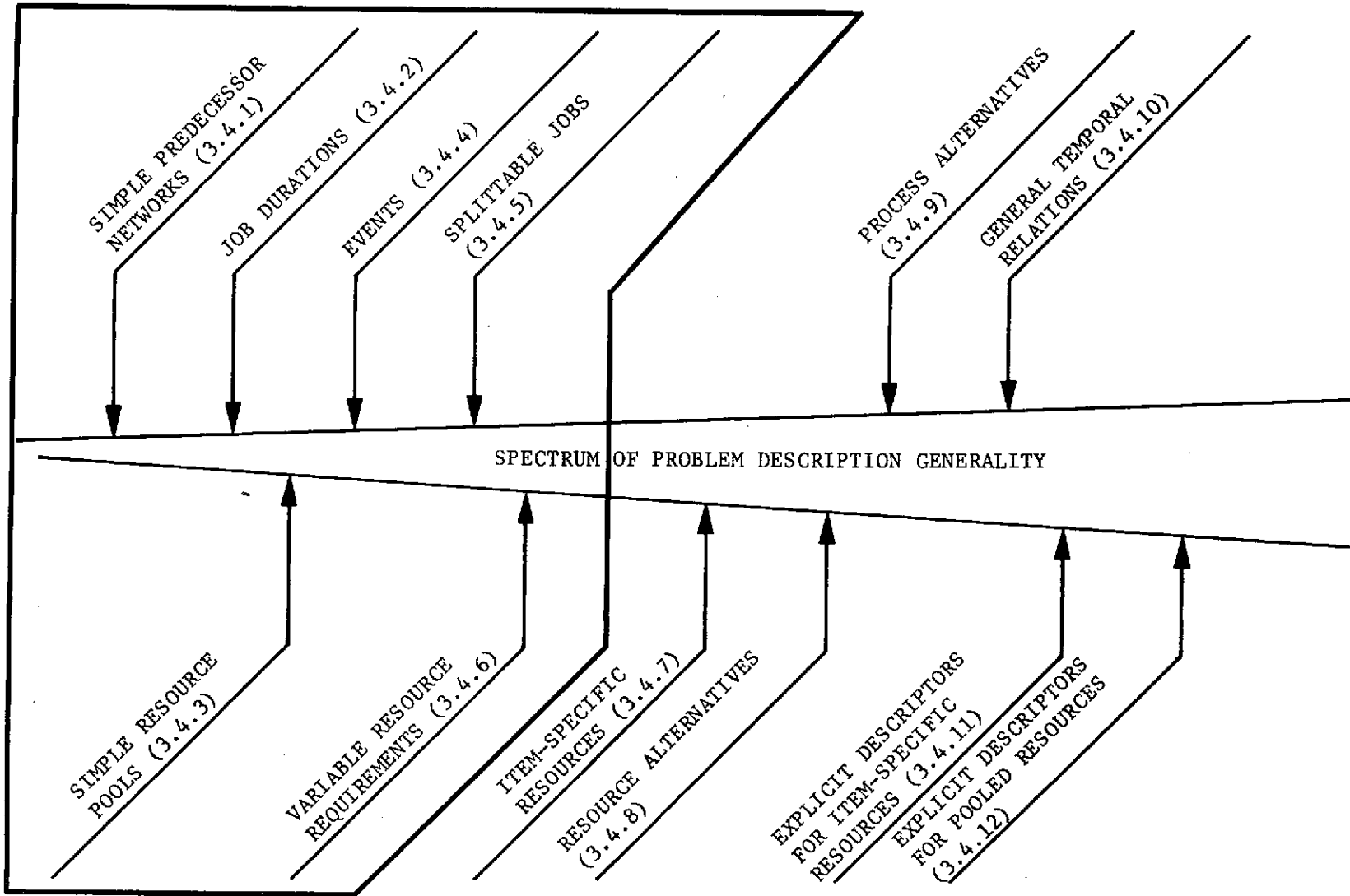
(b) Critical-Path Modules Calling Hierarchy



(c) Heuristic-Scheduling Modules Calling Hierarchy

Fig. 3.6-1
Project Schedule System within the PLANS Module Library

JOBS AND THEIR RELATIONSHIPS TO OTHER JOBS



RESOURCES AND THEIR RELATIONSHIPS TO JOBS

Fig. 3.6-2
Problem Models for Plans Project Scheduling Modules

techniques. Examples include creating dummy processes and regarding item-specific resources as pools of the quantity one. Studies are underway to identify and automate such reformatting techniques that will extend the capabilities provided by project scheduling. Results will be reported in subsequent documentation.

In a strict sense, the problem characteristics that can be handled by mathematical programming methods are not limited. It is the dimensionality of the problem that limits its compatibility with mathematical programming techniques. All scheduling problems that can be solved using mathematical programming techniques can be characterized as *small in dimensionality*. Such problems may occur in preliminary scheduling or may represent intentional simplifications made for the purpose of establishing performance goals or limits. If a problem involves only a small number of processes, each with a small number of resources, and if few alternatives exist for choosing between resources or processes, then mathematical programming solution techniques could be applicable. An increase in generality of any problem model leads to a very rapid increase in the dimensionality of the corresponding mathematical programming formulation. For example, jobs that require nonconstant quantities of resources, as illustrated in Fig. 3.6-3, lead to additional choices (i.e., decision variables) and, therefore, to an increase in dimensionality.

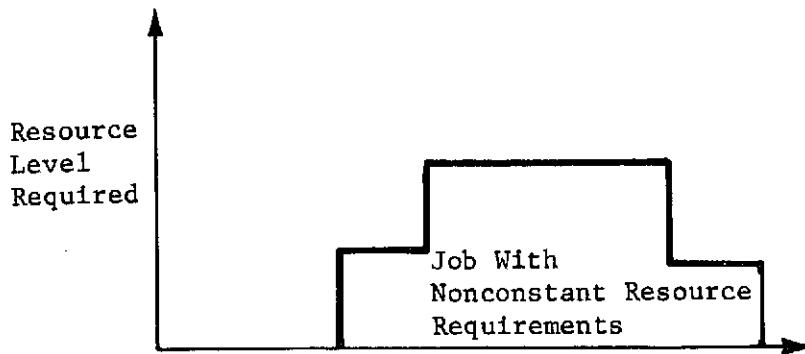


Fig. 3.6-3 Example of Nonconstant Resource Demand Profile

It follows that model simplicity is an indirect necessary condition for using mathematical programming techniques for scheduling.

Some problems associated with resource allocation may have small enough dimensionality to be amenable to mathematical programming techniques. An example is the development of a set of compatible combinations of resources such as grouping payloads that have composite length, weight, and power requirements that fall below a set of limits. The PLANS library contains modules, called `COMPATIBILITY_SET_GENERATOR` and `FEASIBLE_PARTITION_GENERATOR`, which apply to this problem. They are based on quasi-enumerative techniques and produce mathematically optimal solutions.

Other problems with small dimensionality can be solved with other PLANS mathematical programming modules. Figure 3.6-4 represents a decision structure that leads either to an appropriate algorithm choice, or to an indication that mathematical programming techniques are not applicable. This simple diagram contains order-of-magnitude decision thresholds concerning dimensionality

and should assist the problem analyst in determining whether he should consider using mathematical programming. Secondly, the figure is useful in suggesting characteristics of his problem that prevent the applicability of existing mathematical programming methods. Such knowledge could lead to minor restructuring of the problem model to take advantage of logic from the module library.

The use of Fig. 3.6-4 presumes the ability to scope the dimensionality of a problem. For estimating purposes, the dimensionality of a problem can be approximated by summing (over all jobs) the number of time intervals during which each job may start. The upper bound on the number of intervals during which a job may start is, of course, the number of intervals in the scheduling horizon being considered. If a critical path analysis is performed, the user can determine the exact number of intervals for each job as the number of slack intervals for that job.

The tests for Generalized Upper Bounding structure (abbreviated GUB in Fig. 3.6-4) refer to a special structure within the tableau in linear programming that permits the use of this technique. The GUB structure arises in problems that require the selection of one, and only one, candidate from each of several groups of candidates. For example, in a scheduling problem one, and only one, start time must be chosen for each job (from the groups of candidate start times for that job). No jobs may be left out and no jobs may be assigned more than one start time. Another example where the GUB structure would occur is in assigning personnel with special skills to a job requiring one electrician, one plumber, one mason, etc given that sets of personnel with these skills exist.

4.0 Illustrative Examples

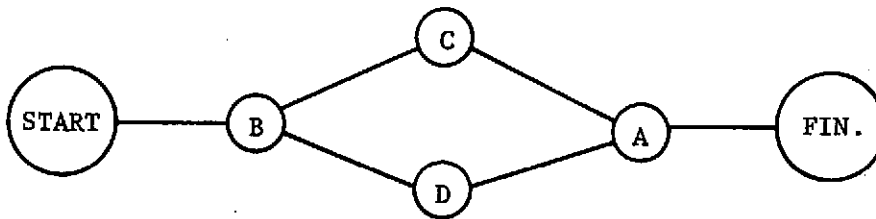
162a

4.0 ILLUSTRATIVE EXAMPLES

4.1 ORDERING OF A PRECEDENCE NETWORK

4.1.1 Problem Statement

In Section 2.0 of this volume, the logic of the module ORDER_BY_PREDECESSORS is used to illustrate the PLANS language features. The same example is repeated here with emphasis placed on how the data structures are modified as the logic is executed. A simple four-job network is used for illustration; the network can be depicted as shown in the sketch.



The problem is simply to generate, from a randomly ordered list of jobs, an ordered list that has the property that any job will appear in the ordered list only after all its predecessors have appeared.

4.1.2 Problem Model

For this simple problem, the input network information is provided in a tree called \$JOBLIST that, initially, has the structure shown in Stage 1 of Fig. 4.1-1.

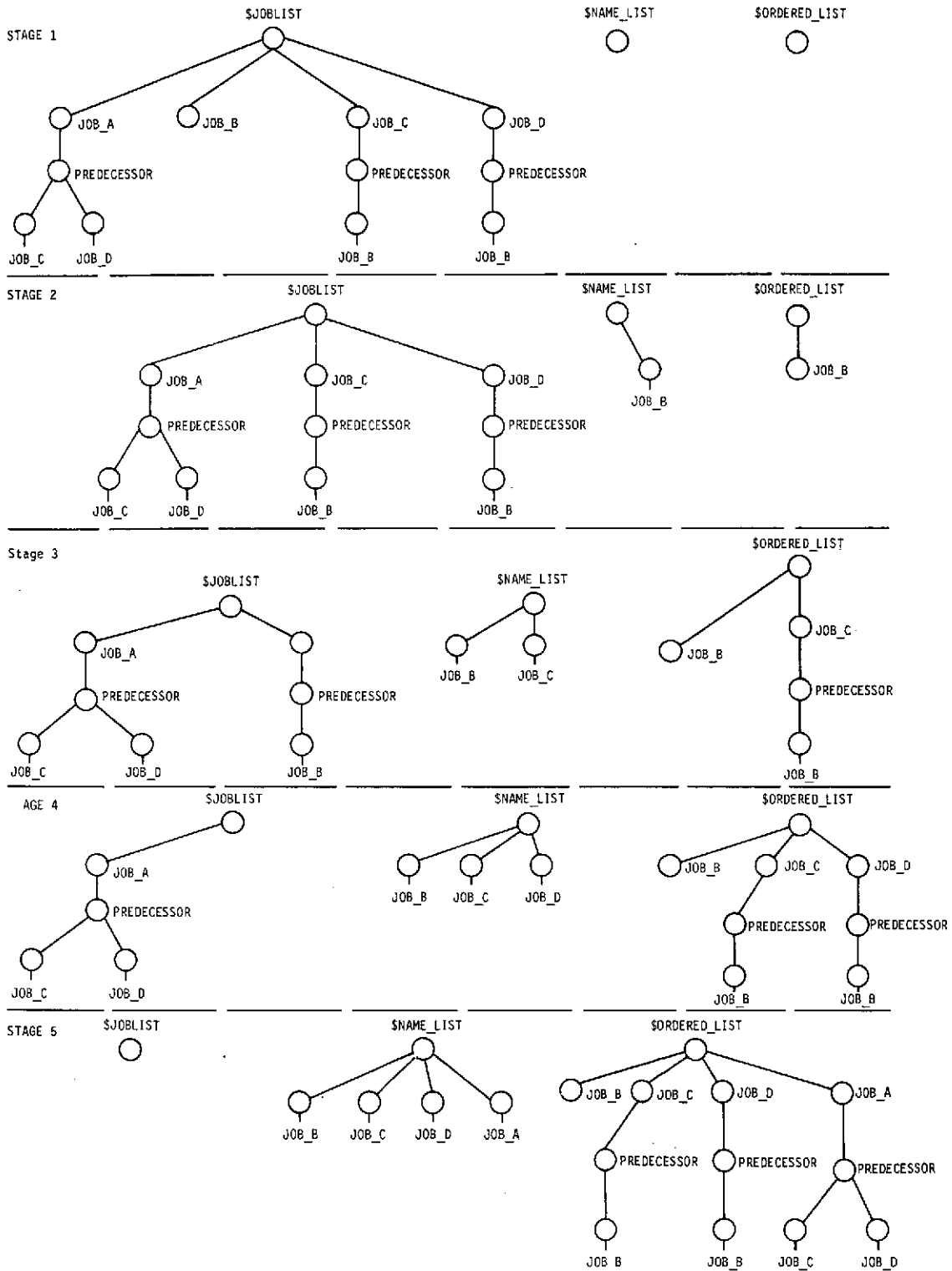


Fig. 4.1-1 Data Structures Illustrating ORDER_BY_PREDECESSORS

4.1.3 Program Logic

The \$JOBLIST tree shown at Stage 1 is passed as a parameter to the module, ORDER_BY_PREDECESSORS (see Fig. 4.1-2). Because \$NAME_LIST is declared to be local to the module, it is empty each time the module is entered. \$ORDERED_LIST is usually empty, but may contain jobs that are to precede all jobs in \$JOBLIST. This allows ORDER_BY_PREDECESSORS to be restartable, in case a required job was missing from \$JOBLIST on a previous call to ORDER_BY_PREDECESSORS. \$ORDERED_LIST is passed to the module as a parameter. Its initialization is the responsibility of the calling program.

```
1 ORDER BY PREDECESSORS: PROCEDURE ($JOBLIST, $ORDERED_LIST) ;
2   DECLARE $NAME_LIST, $TEMP LOCAL ;
3 LOOP:
4   GRAFT $JOBLIST.FIRST: (ELEMENT.PREDECESSOR SUBSET OF $NAME_LIST)
5     AT $TEMP;
6   IF $TEMP IDENTICAL TO $NULL THEN RETURN ;
7   $NAME_LIST (NEXT) = LABEL ($TEMP) ;
8   GRAFT $TEMP AT $ORDERED_LIST (NEXT) ;
9   GO TO LOOP ;
10  END ORDER_BY_PREDECESSORS ;
```

Fig. 4.1-2 PLANS Subroutine for Ordering Jobs by Predecessors

Since \$NAME_LIST is initially empty, the GRAFT statement at line 4 searches for the first job in \$JOBLIST that has no predecessors (JOB_B), removes that job from \$JOBLIST, and places it at \$TEMP. Because a job was found (i.e., because \$TEMP is not null), line 6 fails to cause a return. The name of the job found is, therefore, added to \$ORDERED_LIST. The GO TO statement at line 9 then starts the process over again at LOOP. The tree status at this point is shown in Fig. 4.1-1, Stage 2.

The GRAFT statement at line 4 again searches for the first job in \$JOBLIST whose predecessors are a subset of \$NAME_LIST. In this case, \$NAME_LIST contains only the name, JOB_B. Thus, the first job (JOB_C) in \$JOBLIST that either has no predecessors or has only JOB_B as a predecessor will satisfy the condition and be removed from \$JOBLIST and placed at \$TEMP. The exit test fails, the name JOB_C is added to \$NAME_LIST, and the job itself is removed from \$TEMP and added to \$ORDERED_LIST. The tree status at this point is shown in Fig. 4.1-1, Stage 3. Note that the entire substructure representing information about JOB_C is now in \$ORDERED_LIST.

The third pass starting at LOOP transfers JOB_D to \$ORDERED_LIST because its only predecessor, JOB_B, is named in \$NAME_LIST. The tree status at this point is shown in Stage 4.

Now that both JOB_C and JOB_D are named in \$NAME_LIST, the conditional GRAFT statement is satisfied by JOB_A, which is therefore moved to \$ORDERED_LIST, yielding the status shown in Stage 5.

Finally, the GRAFT statement at line 4 fails to find in \$JOBLIST (which is empty) a new job whose predecessors are in \$NAME_LIST. A null node is, therefore, placed at \$TEMP, causing the exit test (line 6) to succeed. Note that the module would be exited if the conditional GRAFT statement failed even if \$JOBLIST were not empty. This condition would indicate a cycle or a missing job in \$JOBLIST. It is the responsibility of the calling program to test for this condition if such a test is considered necessary.

4.2 PAYLOAD GROUPING

4.2.1 Problem Statement

The problem is to find a composite payload whose characteristics satisfy Shuttle Orbiter limits. For simplicity, we restrict the characteristics to be those that can be evaluated by summing the properties of the individual payload (e.g., length, weight, etc). Any combination of payloads that satisfies all the Shuttle limits is considered a candidate for a mission. The best composite payload is to be selected by an existing routine. Therefore, the logic to be written must find all feasible combinations and pass those combinations to the selection routine. A composite payload could consist of a single payload, a pair of payloads, a triplet of payloads, etc, i.e. combinations of order 1, 2, ... up to a prescribed maximum order. For example, payload 12 could, itself, be a feasible composite payload as could the triplet consisting of payload 12, 5, and 17.

4.2.2 Problem Model

This problem is not, in the strictest sense of the word, a scheduling problem because no assignment of times to the activities is involved. However, neither PLANS nor the library modules have been designed with such a limited view of scheduling in mind. This problem deals with the characteristics of resources that can be described within the \$RESOURCE tree structure, as:

```
$RESOURCE
  SHUTTLE_WITH_KICK_STAGE
    ¢
      LIMIT
        BAY_LENGTH - 32
        WEIGHT - 4400
```



```

PAYLOAD
  AS001
    CHARACTERISTICS
      LENGTH - 13
      WEIGHT - 1807
  PH007
  .
  .
  .

```

The information needed to generate the candidate-feasible combinations and to call the payload selection routine can be arranged within the \$OBJECTIVES structure as shown below:

```

$OBJECTIVE
  FIG OF MERIT
    ¢ - MINIMIZE
    ¢ - NUMBER
  CONSTRAINT
    MAX_IN_COMBO - 3

```

4.2.3 Program Logic


The logic consists of nested loops of PLANS code. The innermost loop sums one characteristic over all payloads in a particular combination. This summation is repeated for each characteristic as the next higher loop is executed. The combination loop, a unique capability of PLANS, generates each payload combination of order K, where K ranges from 1 to the value of MAX_IN_COMBO in the outermost loop. The PLANS code is shown below.

```

1  FIND_BEST_COMPOSITE_PAYLOAD: PROCEDURE ($OBJECTIVES, $RESOURCE) ;
2  /* GENERATE ALL INTERNALLY FEASIBLE PAYLOAD COMBINATIONS.          */
3  DO K = 1 TO $OBJECTIVES.CONSTRAINT.MAX_IN_COMBO ;
4    DO FOR ALL COMBINATIONS OF $RESOURCE.PAYLOAD TAKEN K AT A TIME ;
5      DO J = 1 TO NUMBER($RESOURCE.SHUTTLE_WITH_KICK_STAGE(1).LIMIT) ;
6        $SUM(J) = 0 ;
7        DO L = 1 TO K ;
8          $SUM(J) = $SUM(J) + $COMBINATION(L).CHARACTERISTIC(J) ;
9        END ;
10       IF $SUM(J) > $RESOURCE.SHUTTLE_WITH_KICK_STAGE(1).LIMIT(J)
11         THEN GO TO END_COMBO_LOOP ;
12       END ;
13       $FEASIBLE_SET(NEXT) = $COMBINATION ;
14       GRAFT $SUM AT $FEASIBLE_SET(LAST).SUMS ;
15     END_COMBO_LOOP: END ;
16   END ;
17  CALL BESTSET($OBJECTIVES, $FEASIBLE_SET, $REST) ;
18  WRITE $REST ;
19  END FIND_BEST_COMPOSITE_PAYLOAD ;

```

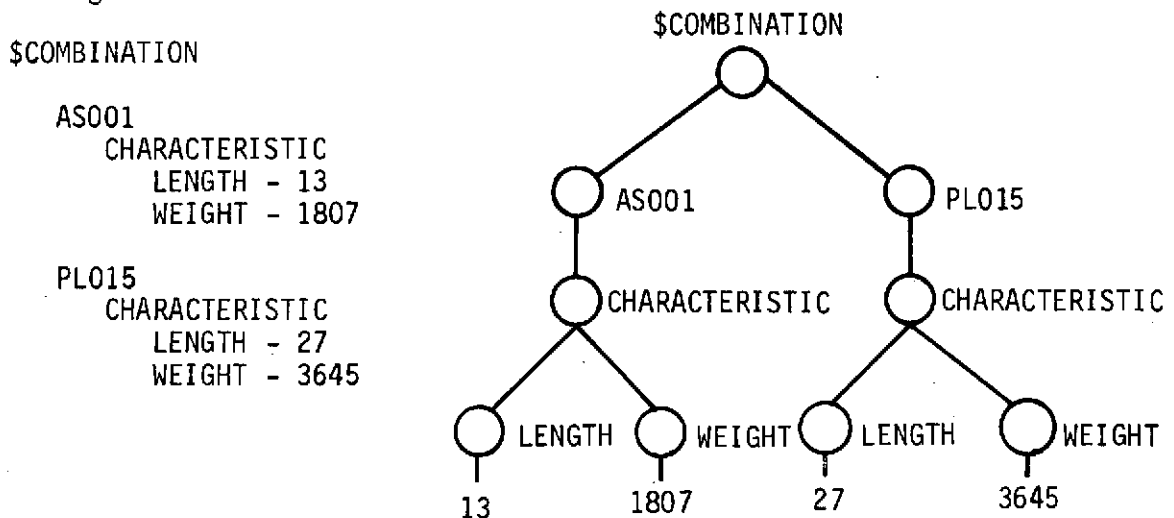
Reproduced from
best available copy.



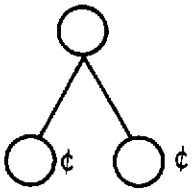
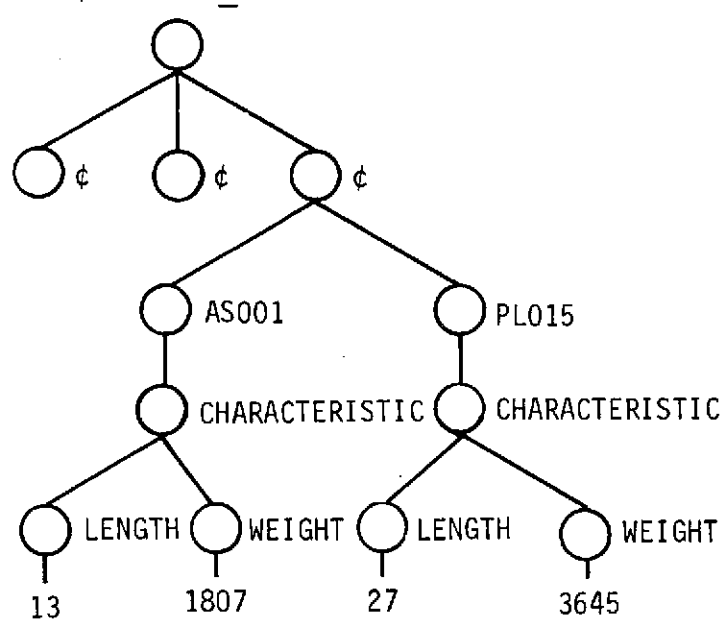
The code presented assumes that the subroutine BESTSET has the capability to interpret the information and structure of \$OBJECTIVES.

Notice should be taken of the use of the special indices, NEXT and LAST, in statements 13 and 14. These statements add a new combination of payloads to \$FEASIBLE_SET. Because PLANS is a tree manipulation language, the structure of the data trees actually changes during program execution.

As the combination loop is executed, the structure \$COMBINATION is maintained. \$COMBINATION is a special tree that does not actually have subnodes of its own. Instead, \$COMBINATION is maintained, with a set of pointers, in the already existing structure from which the combinations are to be formed (\$RESOURCE: PAYLOAD). This economizes on time and storage by avoiding unnecessary duplication of data. For a second-order combination, it might look like:

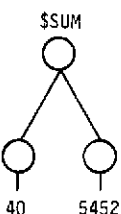

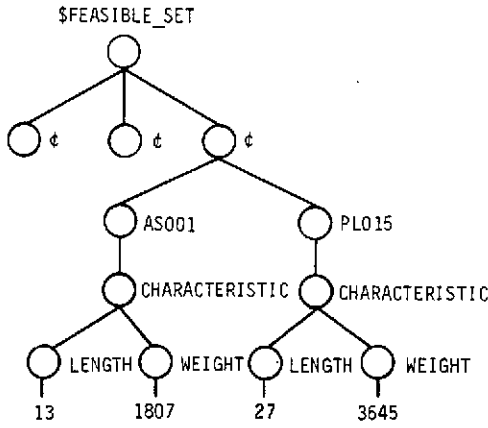
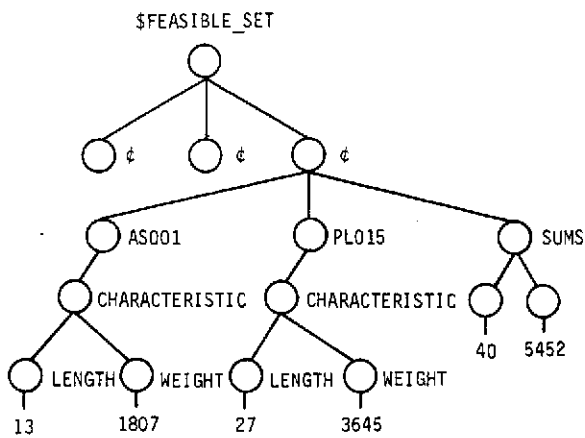


where both payload designators are first order subnodes of \$COMBINATION. If this combination were feasible, the statement \$FEASIBLE_SET (NEXT) = \$COMBINATION would cause the following modification of \$FEASIBLE_SET.

Structure prior to Statement execution	Structure after Statement execution
<p>\$FEASIBLE_SET</p> <p>⌀</p> <p>⌀</p> <p>or</p> <p>\$FEASIBLE_SET</p> 	<p>\$FEASIBLE_SET</p> <p>⌀</p> <p>⌀</p> <p>⌀</p> <p>AS001</p> <p>CHARACTERISTIC</p> <p>LENGTH - 13</p> <p>WEIGHT - 1807</p> <p>PL015</p> <p>CHARACTERISTIC</p> <p>LENGTH - 27</p> <p>WEIGHT - 3645</p> <p>or</p> <p>\$FEASIBLE_SET</p> 

The next statement, GRAFT \$SUM AT \$FEASIBLE_SET(LAST).SUMS results

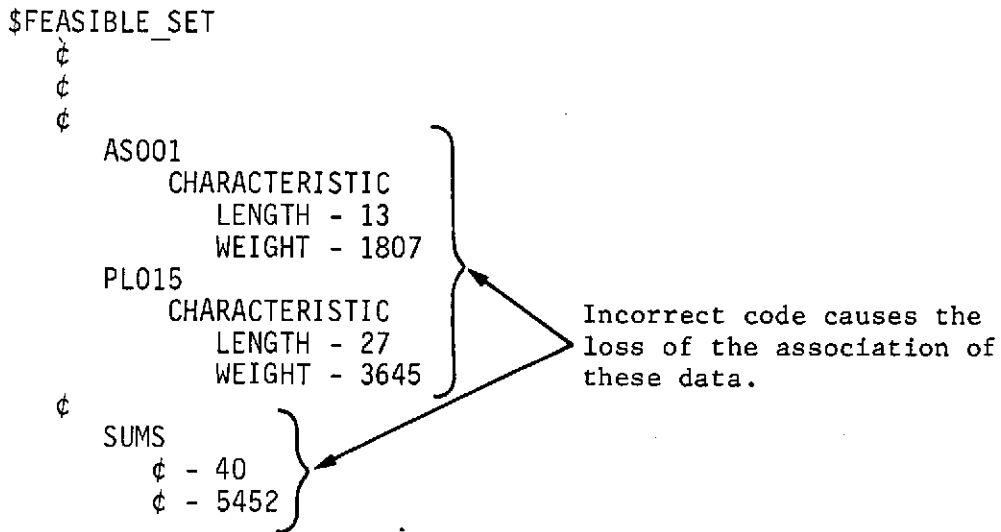
in:

Structure prior to Statement execution	Structure after Statement execution
<p>\$SUM € - 40 € - 5452</p> <p>or</p> 	<p>\$SUM</p> <p>or</p> 
<p>\$FEASIBLE_SET € € €</p> <p>AS001 CHARACTERISTIC LENGTH - 13 WEIGHT - 1807</p> <p>PL015 CHARACTERISTIC LENGTH - 27 WEIGHT - 3645</p> <p>or</p> 	<p>\$FEASIBLE_SET € € €</p> <p>AS001 CHARACTERISTIC LENGTH - 13 WEIGHT - 1807</p> <p>PL015 CHARACTERISTIC LENGTH - 27 WEIGHT - 3645</p> <p>SUMS € - 40 € - 5452</p> <p>or</p> 

Note that because the assignment statement 13 builds a new node in \$FEASIBLE_SET, the GRAFT statement (14) must refer to the LAST (not NEXT) node if the SUMS are to apply to the correct combination. If the code had been incorrectly written as:

```
$FEASIBLE_SET(NEXT) = $COMBINATION; GRAFT $SUM AT $FEASIBLE_SET
(NEXT).SUMS;
```

the structure of \$FEASIBLE_SET would have been:



4.2.4 Changes to Problem Scope

It can be recognized that the code is independent of the number of payloads in the problem, the number of characteristics being considered, and the maximum number of individual payloads allowable in any combination. Furthermore, it is possible to write the BESTSET logic in PLANS to interpret tree data in \$OBJECTIVES so that changes in the problem objectives are easily accommodated.

PLANS permits coding of this routine to be even less sensitive to problem changes than the code illustrated. Suppose, for example, the power requirements and weights of instruments were

to be summed to check for feasibility as a part of a sortie module design. The appearance of the label PAYLOAD in the code is therefore restrictive. The problem would have been avoided by placing the resource name to be considered in a special place in the data structure such as in \$OBJECTIVES as follows:

```
$OBJECTIVES
  FIG_OF_MERIT
    c - MINIMIZE
    c - NUMBER
  CONSTRAINT
    MAX_IN_COMBO - 3
  PROBLEM_RESOURCE - PAYLOAD
```

The statements in the illustrated coding that contain the label PAYLOAD could have been coded with an indirect reference. For example:

```
DO FOR ALL COMBINATIONS OF
  $RESOURCE.#($OBJECTIVES.PROBLEM_RESOURCE)
  TAKEN K AT A TIME ;
```

Thus to change from payloads to instruments would be accomplished by adding instruments to \$RESOURCE and changing the value of the PROBLEM_RESOURCE node of \$OBJECTIVES.

4.3 PROJECT SCHEDULING

4.3.1 Problem Statement

A project consists of 11 activities, each requiring resources from three different resource pools. To be specific, consider each pool to be a manpower pool, containing six men with a common skill. The activities in the project are related to each other by simple precedence relations; i.e., certain activities cannot begin before others are finished. The objective of the schedule is to find the earliest time that all jobs can be completed without using more from any pool than are available.

4.3.2 Problem Model

This problem is precisely the basic project scheduling model.

The characteristics that make it so are:

- 1) Relationships between jobs (temporal relations) are simple predecessors;
- 2) Resources required are pooled resources with no special characteristics that are altered after an assignment. (Pooled resources with implicit descriptors only, i.e., with no explicit descriptors).

The temporal relations of the problem can be illustrated by a network diagram. The diagram of Fig. 4.3-1 shows the job duration below each job and the quantities of required resources from each of the three pools above each job.

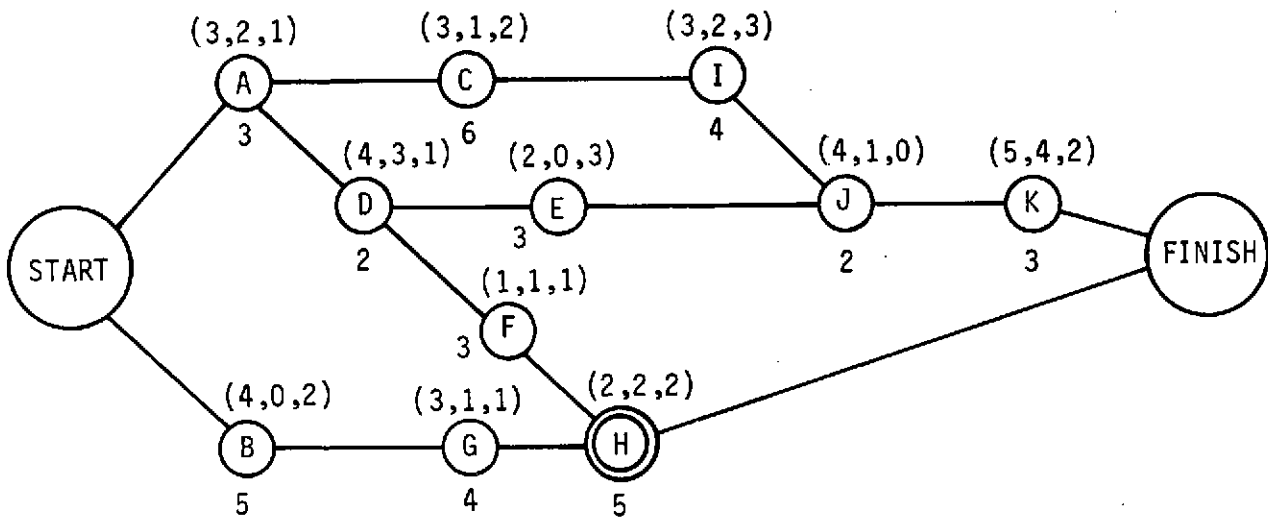


Fig. 4.3-1 Network Diagram for Project Scheduling

The standard data structures presented previously accommodate project scheduling information. Relations between jobs are included in the structure \$OPSEQ which, for this problem, would look like:

```
$OPSEQ
PROJECT - A
A
  TYPE - PROCESS
B
  TYPE - PROCESS
C
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
  φ - A
D
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
  φ - A
E
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
  φ - D
F
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
  φ - D
G
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
  φ - B
H
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
  φ - F
  φ - G
I
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
  φ - C
```



```

J
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
    ¢ - I
    ¢ - E
K
  TYPE - PROCESS
  TEMPORAL RELATIONS
  PREDECESSOR
    ¢ - J

```

The relationships between jobs and their resources are described in the \$PROCESS tree. For this project scheduling problem, \$PROCESS looks like the following:

```

$PROCESS
  A
    REQUIRED RESOURCES
    MANPOWER
    LABOR_1
      ¢
      DESCRIPTORS
        ¢
        INITIAL
        QUANTITY - 3
    LABOR_2
      ¢
      DESCRIPTORS
        ¢
        INITIAL
        QUANTITY - 2
    LABOR_3
      ¢
      DESCRIPTORS
        INITIAL
        QUANTITY - 1
    DURATION - 3
  B
    REQUIRED RESOURCES
    MANPOWER
    LABOR_1
      ¢
      DESCRIPTORS
        ¢
        INITIAL
        QUANTITY - 4

```

```

LABOR_3
  DESCRIPTORS
    ¢
      INITIAL
        QUANTITY - 2
DURATION - 5

```

The descriptions of the resources modeled in the system are defined in the \$RESOURCE tree. For this problem, \$RESOURCE would include:

```

$RESOURCE
  MANPOWER
    LABOR_1
      CLASS - POOL
      INITIAL PROFILE
      NORMAL
      ¢
        QUANTITY-6
    LABOR_2
      CLASS - POOL
      INITIAL PROFILE
      NORMAL
      ¢
        QUANTITY-6
    LABOR_3
      CLASS - POOL
      INITIAL PROFILE
      NORMAL
      ¢
        QUANTITY-6

```

The fact that there is a single occurrence of the operation sequence PROJECT_A, is modeled simply by constructing \$OBJECTIVE as shown below. The PROBLEM_RESOURCE node identifies MANPOWER as the critical resource to be considered in this problem. This allows the programmer to use indirect references, thus making his code more generally applicable.

```

$OBJECTIVE
  OPSEQ
    PROJECT_A
      TYPE - OPSEQ
        PROBLEM_RESOURCE - MANPOWER

```

4.3.3 Program Logic

Because this problem is purely a project scheduling problem, the project scheduling routines from the module library apply directly. The recommended method of solution for this example would be to use the time-progressive heuristic of the RESOURCE_ALLOCATOR module. Volume III of this report contains the detailed functional specifications for this module, and the solution for this example problem that results from the application of the heuristic procedures is specified in RESOURCE_ALLOCATOR.

The PLANS program code to solve this example merely calls the specified library module called RESOURCE_ALLOCATOR. Therefore, the code is simply:

```

PROJECT_SCHEDULING: PROCEDURE ;
  READ $OPSEQ,$PROCESS,$RESOURCE,$OBJECTIVES,$INTEGER,$START,$END ;
  $TYPE = $OBJECTIVES,$PROBLEM_RESOURCE ;
  CALL GENERATE_JOBSET($OBJECTIVES,$OPSEQ,$PROCESS,$INTEGER,$JOBSET) ;
  DO I=1 TO NUMBER($RESOURCE,$TYPE) ;
    CALL RESOURCE_PROFILE($RESOURCE,$TYPE,
      LABEL($RESOURCE,$TYPE(I)),$START,$END,$PROFILE) ;
    GRAFT $PROFILE AT $PROFILES(NEXT) ;
  END ;
  CALL RESOURCE_ALLOCATOR($JOBSET,$PROFILES,$SCHEDULE) ;
  WRITE $SCHEDULE ;
END PROJECT_SCHEDULING ;

```

4.3.4 Alternative Approach

An alternative to the time-progressive heuristic scheduling strategy, used within the RESOURCE_ALLOCATOR module, is a time-transcendent strategy. A single time-transcendent strategy with sufficiently general applicability cannot be specified and therefore is not provided in the module library. However, the PLANS

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

code for constructing such a solution strategy is quite simple. To illustrate this, consider the logic of Fig. 4.3-2, which is a time-transcendent logic. For the example problem presented, the program shown in Fig. 4.3-3 could be written to implement the logic on the flow chart.

4.3.5 Changes to Problem Scope

It should be noted that this program is independent of the total number of jobs, the definition of their predecessor relationships, the number of resource pools, the problem resource type, and the time increment. That is, in order to vary these problem characteristics, the programmer needs only to change the input data. However, throughout this example it has been assumed that \$JOBSET contains only one SUBNET_ID subnode. This will be the case for this particular example problem because there is only one subnode of \$OBJECTIVES.OPSEQ; but, generally, there will be several such subnodes. This problem can be eliminated by inserting the following block of code after the CALL GENERATE_JOBSET statement near the beginning of the program.

```
DO I=1 TO NUMBER ($JOBSET);
  DO J=1 TO NUMBER ($JOBSET(I));
    GRAFT $JOBSET(I)(J) AT $JOBLIST(NEXT);
  END;
END;
PRUNE $JOBSET;
```

This code eliminates all of the SUBNET_ID nodes found in \$JOBSET. It creates a new data tree called \$JOBLIST that contains all of the "job" nodes one level below the root node. Therefore, in the remainder of the program all occurrences of \$JOBSET(1) should be replaced with \$JOBLIST.

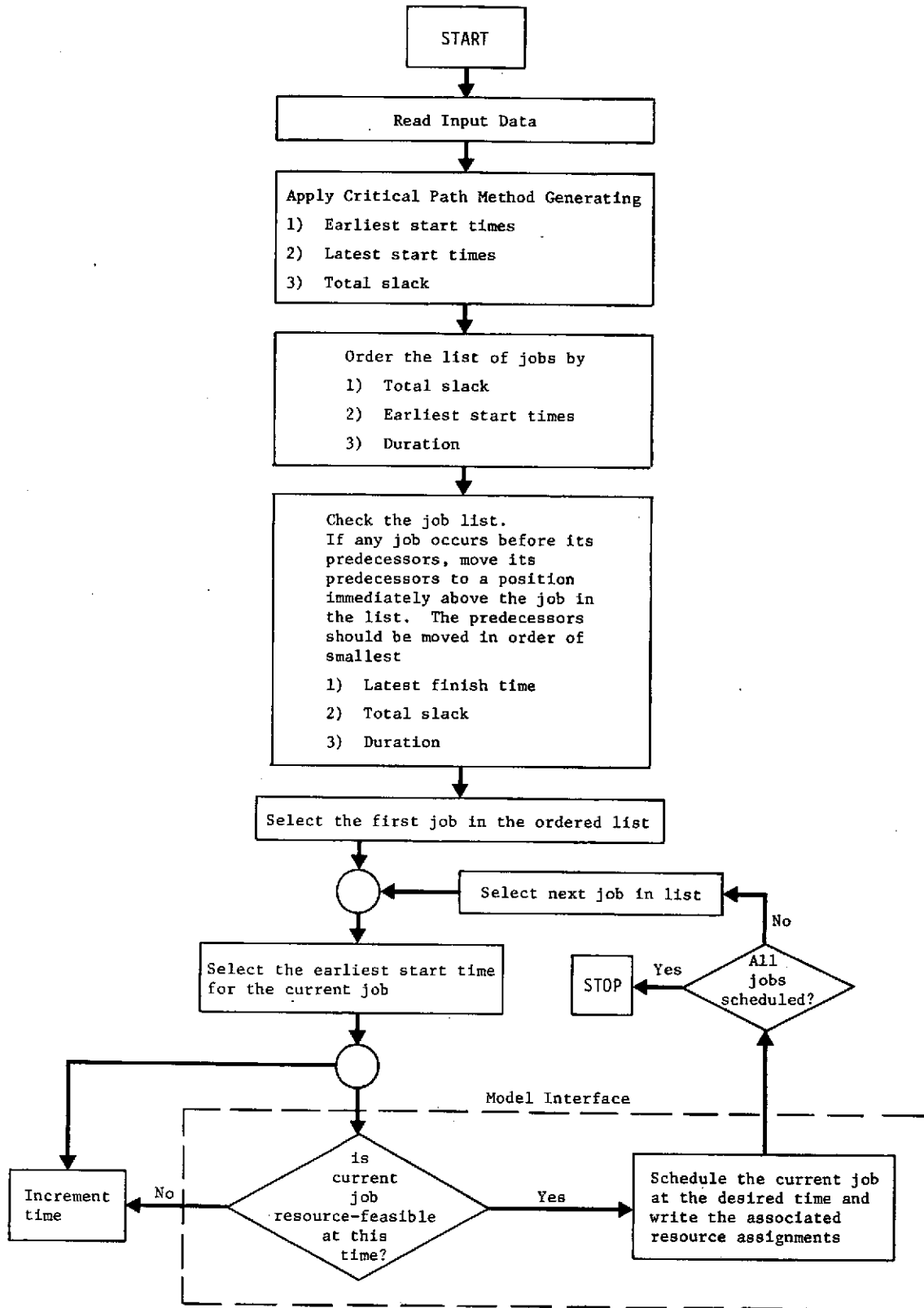


Fig. 4.3-2

Flow Logic for Time Transcendent Project Scheduling Algorithm

```

/* THIS PROGRAM USES A TIME-TRANSCENDENT STRATEGY TO SOLVE THE      */
/* BASIC PROJECT SCHEDULING PROBLEM.                                */
PROJECT_SCHEDULING: PROCEDURE ;
  READ $OBJECTIVES, $OPSEQ, $PROCESS, $RESOURCE, DELTA_TIME, INTEGER ;
  CALL GENERATE_JOBSET($OBJECTIVES, $OPSEQ, $PROCESS, INTEGER, $JOBSET) ;
  CALL CRITICAL_PATH_CALCULATION($JOBSET) ;
  ORDER $JOBSET(1) BY -FLOAT.TOTAL, -START.EARLY, -JOB.INTERVAL.END ;
  NUMBER_OF_JOBS = NUMBER($JOBSET(1)) ;
/*
  THIS LOOP ORDERS THE JOBS BY PREDECESSORS. WITHIN EACH GROUP OF
  PREDECESSORS THE JOBS ARE PUT IN ASCENDING ORDER ACCORDING TO LATEST
  FINISH TIMES, TOTAL SLACK, AND DURATION. */
  DO I=1 TO NUMBER_OF_JOBS ;
  SCAN_JOBSET:
    IF $JOBSET(1)(I).TEMPORAL_RELATIONS.PREDECESSORS SURSET OF
      $NAMELIST
    THEN $NAMELIST(NEXT) = LABEL($JOBSET(1)(I)) ;
    ELSE DO ;
      DO K=I+1 TO NUMBER_OF_JOBS ;
      IF LABEL($JOBSET(1)(K)) SURSET OF
        $JOBSET(1)(I).TEMPORAL_RELATIONS.PREDECESSORS
      THEN GRAFT $JOBSET(1)(K) AT $TEMP(NEXT) ;
      END ;
      ORDER $TEMP BY FINISH.LATE, FLOAT.TOTAL, DURATION ;
      DO L=1 TO NUMBER($TEMP) ;
      GRAFT INSERT $TEMP(1) AS $JOBSET(1)(I) ;
      END ;
      GO TO SCAN_JOBSET ;
    END ;
  PRUNE $NAMELIST ; $TYPE = $OBJECTIVES.PROBLEM_RESOURCE ;
/*
  THIS LOOP GOES THROUGH THE JOB LIST AND SCHEDULES EACH JOB AS SOON
  AS A TIME IS DETERMINED IN WHICH ALL OF ITS REQUIRED RESOURCES ARE
  AVAILABLE. */
  DO J=1 TO NUMBER_OF_JOBS ; TIME = $JOBSET(1)(J).START.EARLY ;
  CHECK_RESOURCE_AVAILABILITY:
    DO K=1 TO NUMBER($RESOURCE.#($TYPE)) ;
    CALL RESOURCE_PROFILE($RESOURCE, $TYPE,
      LABEL($RESOURCE.#($TYPE)(K)), TIME, TIME, $PROFILE) ;
    IF $PROFILE.IN_USE(1).QUANTITY +
      $JOBSET(1)(J).REQUIRED_RESOURCES.#($TYPE)(K)(1).DESCRIPTORS
      (1).INITIAL.QUANTITY
    > $RESOURCE.#($TYPE)(K).INITIAL_PROFILE.NORMAL(1).QUANTITY
    THEN DO ; TIME = TIME + DELTA_TIME ;
      GO TO CHECK_RESOURCE_AVAILABILITY ;
    END ;
  END ;
/*
  NOW THE JOB CAN BE SCHEDULED BY UPDATING THE ASSIGNMENT INFORMATION
  FOUND IN THE TREE. $RESOURCE. */
  $ASSIGNMENT_UNIT.INTERVAL.START = TIME ;
  $ASSIGNMENT_UNIT.INTERVAL.END = TIME + $JOBSET(1)(J).DURATION ;
  DO L=1 TO NUMBER($RESOURCE.#($TYPE)) ;
  NEEDED_AMT = $JOBSET(1)(J).REQUIRED_RESOURCES.#($TYPE)(L)(1).
    DESCRIPTORS(1).INITIAL.QUANTITY ;
  IF NEEDED_AMT = 0
  THEN DO ;
    $ASSIGNMENT_UNIT.DEScriptor(1).INITIAL.QUANTITY =
      NEEDED_AMT ;
    CALL WRITE_ASSIGNMENT($ASSIGNMENT_UNIT,
      $RESOURCE.#($TYPE)(L).ASSIGNMENT) ;
  END ;
  END ;
  END ;
/*
  THE ENTIRE SCHEDULE IS PRINTED OUT BY THE LOOP BELOW.
  ALL RESOURCES AND THEIR SPECIFIC JOB ASSIGNMENTS ARE DISPLAYED. */
  DO I=1 TO NUMBER($RESOURCE.#($TYPE)) ;
  $NAME = LABEL($RESOURCE.#($TYPE)(I)) ;
  WRITE $NAME, $RESOURCE.#($TYPE)(I).ASSIGNMENT ;
  END ;
END PROJECT_SCHEDULING ;

```

REPRODUCIBILITY OF THE
 ORIGINAL PAGE IS POOR

Fig. 4.3-3 PLANS Code for Time Transcendent Project Scheduling Algorithms

4.4 FLIGHT ASSIGNMENT

4.4.1 Problem Statement

The problem to be solved is to find launch dates for a set of payloads, each of which has a launch window; i.e., an interval during which it must be launched if it is launched at all. Resources that are required to launch the payloads are specified by quantity and type. For example, each launch requires one orbiter, two solid rocket motors, three crewmen, etc. Since the resources may be reused after a flight, the scheduling must assign the cycling resources in a way that permits as many payloads to be launched as possible.

4.4.2 Problem Model

All resources for the problem can be modeled as item-specific resources. That is, each specific orbiter, crewman, launch pad, etc should be given a separate identity so that each can be tracked through the launch and turnaround processes separately.

It is sufficient in this example to define a single process, FLIGHT, with the appropriate required resources as illustrated below:

```
$PROCESS
  FLIGHT
    REQUIRED_RESOURCES
      PAYLOAD
        ¢
          ¢
            INTERVAL
              START - 0
              END   - 21
```

ORBITER

¢

¢

INTERVAL
START - 0
END - 21

SRM

¢

¢

INTERVAL
START - 0
END - 14
DESCRIPTORS

¢

INITIAL
QUANTITY - 2

Each of the resource types appearing in \$PROCESS should appear in the \$RESOURCE tree with the specific resources under each type. In addition, the payloads and their windows should appear in \$RESOURCE. At input, \$RESOURCE would have the structure:

```
$RESOURCE
  ORBITER
    ORBITER_02
    ORBITER_05
    ORBITER_06
    .
    .
  SRM
    SRM_S190
    SRM_A06
    .
    .
  PAYLOAD
    PAYLOAD_07
    WINDOW
      START - 217
      END - 238
    PAYLOAD_09
    WINDOW
      START - 240
      END - 271
    .
    .
```


4.4.3 Program Logic

In the program, illustrated in Figure 4.4-1, the module NEXTSET performs the function of finding the next time after a given input time that a complete set of required resources of the correct types and quantities will be available. It examines the assignments in \$RESOURCE to find this time. In addition, the module returns the identifiers of the specific resources that correspond to this time. Thus, NEXTSET provides the fundamental logic needed to build a time-progressive scheduling heuristic. (See Volume III, Section 2.4.12 for a complete functional description of the NEXTSET module).

```

FLIGHT_ASSIGNMENT: PROCEDURE ($RESOURCE, $PROCESS) !
  ORDER $RESOURCE, PAYLOAD BY -ELEMENT, WINDOW, START,
    -ELEMENT, WINDOW, END !
  $FLIGHT = $PROCESS, FLIGHT !
  $FLIGHT, JOB_INTERVAL, START = 0 !
  GRAFT $FLIGHT, DURATION AT $FLIGHT, JOB_INTERVAL, END !
BUILD_SCHEDULE_UNIT:
/*
  THE MODULE 'NEXTSET' MAKES ALL OF THE RESOURCE ASSIGNMENTS FOR THE
  NEXT FLIGHT AND CREATES A TREE ($NEXTSET) WHICH IS READY TO BE
  PLACED IN $$SCHEDULE. HERE IT IS ASSUMED THAT THE PAYLOAD WITH THE
  NEAREST WINDOW OPENING TIME WILL BE USED. */
  CALL NEXTSET(<$FLIGHT, $RESOURCE, PAYLOAD(1), WINDOW, START,
    $RESOURCE, PAYLOAD(1), WINDOW, END, $RESOURCE, $NEXTSET, $WINDOWS) !
  IF $NEXTSET IDENTICAL TO $NULL THEN GO TO OUTPUT !
  START_TIME = $NEXTSET, JOB_INTERVAL, START !
/*
  THE CODE BELOW DETERMINES IF A DIFFERENT PAYLOAD SHOULD BE SUBSTI-
  TUTED ON THIS FLIGHT DUE TO ITS NEARER WINDOW CLOSING TIME. */
  $CANDIDATES = $RESOURCE, PAYLOAD, ALL: (ELEMENT, WINDOW, START <=
    START_TIME & ELEMENT, WINDOW, END >= START_TIME) !
  IF $CANDIDATES IDENTICAL TO $NULL
    THEN $KEEP = LABEL($RESOURCE, PAYLOAD(1)) !
    ELSE DO ! N = INFINITY !
  FIND_MINIMUM_END_TIME:
    GRAFT $CANDIDATES, FIRST: (ELEMENT, WINDOW, END < N) AT $TEMP !
    IF $TEMP NOT IDENTICAL TO $NULL
      THEN DO ! N = $TEMP, WINDOW, END ! $KEEP = LABEL($TEMP) !
      GO TO FIND_MINIMUM_END_TIME !
    END !
  END !
/*
  SINCE THE PAYLOAD HAS BEEN CHOSEN, THE NEXT FLIGHT CAN BE SCHEDULED
  AFTER UPDATING THE 'ASSIGNMENT' INFORMATION IN $RESOURCE. */
  LABEL($NEXTSET, RESOURCES, PAYLOAD(1)) = LABEL($KEEP) !
  GRAFT $RESOURCE, PAYLOAD, #LABEL($KEEP) AT $$SCHEDULED_PAYLOADS(NEXT) !
  GRAFT $NEXTSET AT $JOB(1) !
  CALL UPDATE_RESOURCE($JOB, $RESOURCE) !
  GRAFT $JOB(1) AT $$SCHEDULE(NEXT) !
  GO TO BUILD_SCHEDULE_UNIT !
OUTPUT: WRITE $$SCHEDULE !
/*
  SINCE THIS PROGRAM MODIFIES THE STRUCTURE OF $RESOURCE, THE LOOP
  BELOW IS NEEDED TO RESTORE IT TO ITS ORIGINAL FORM. */
  DO I=1 TO NUMER($$SCHEDULED_PAYLOADS) !
    GRAFT INSERT $$SCHEDULED_PAYLOADS(1) AT $RESOURCE, PAYLOAD(1) !
  END !
  STOP !
END FLIGHT_ASSIGNMENT !

```

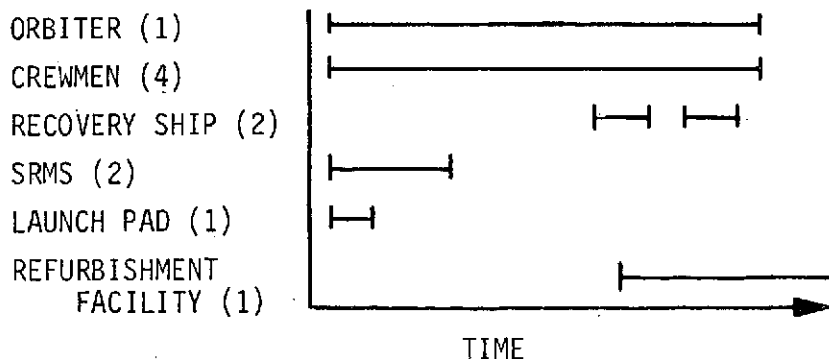
Fig. 4.4-1 Example of PLANS Code for FLIGHT ASSIGNMENT Algorithm

After the next flight time and set of resources are determined by NEXTSET, a choice of payloads may exist for that flight. The logic illustrated finds all the payloads whose windows contain the next flight time and then chooses the one whose window closes the soonest after the flight time.

Finally, the logic updates \$RESOURCE by calling the module UPDATE_RESOURCE and adds a new flight to the schedule being stored in \$SCHEDULE. The detailed specifications for UPDATE_RESOURCE are found in Volume III, Section 2.4.16.

4.4.4 Changes to Problem Scope

The code illustrated applies to any combination of resources in any quantities properly defined in \$PROCESS and \$RESOURCE. For example, new cycling resources could be added; that is, vertical assembly building, flight control centers, etc could be added to the flight resources without changing the code. Furthermore, the process called FLIGHT need not require all of its resources for the same time intervals; the diagram below illustrates a resource set that is accommodated merely by changing the \$PROCESS data without changing the illustrated code.



Another illustration of a change in the problem scope that could be accommodated without a coding change concerns the modeling of resources that undergo descriptor changes as a result of being assigned. Suppose that the process FLIGHT were defined to require crewmen with no previous flight experience. \$PROCESS might look like:

```

$PROCESS
  FLIGHT
    REQUIRED_RESOURCES
      CREWMAN
        ¢
          ¢
            DESCRIPTORS
              ¢
                INITIAL
                  EXPERIENCE - NONE
                FINAL
                  EXPERIENCE - VET

```

The appearance of the initial descriptor, EXPERIENCE, with a value NONE would cause the module NEXTSET to look only for crewmen with no experience. After being assigned to a flight, a crewman would have a value VET for EXPERIENCE as a result of the action of the module UPDATE_RESOURCE and thus, would not be chosen again. Thus, without changing the code, we have introduced non-recycling resources into the system. In the terminology of the operations model, this has been accomplished by generalizing item-specific resources, with implicit descriptors only, to item-specific resources with explicit descriptors. Note that explicit descriptors have the distinguishing property of being changed by a process and retaining their new value after the process has terminated.