DOCUMENT IS MISSING MANY PAGES!

File with N74-33706

MARTIN MARIETTA AEROSPACE

DENVER DIVISION
POST OFFICE BOX 179
DENVER, COLORADO 80201
TELEPHONE (303) 794-5211

50475

CR-140283

To: Recipients of MCR-74-314 NAS9-13616

Subject: Revision of MCR-74-314 NAS9-13616, Phase 1 Final Report, Scheduling Language and Algorithm Development Study, Volume III. Detailed Functional Specifications for the Language and Module Library

Attached are revision pages for the subject report. Please add the attached new page vii to your book and replace all pages currently in your book with the remaining attached replacement pages.

John F. Flater

John F. Flater
Program Manager
NAS9-13616

Page Intentionally Left Blank

"Page missing from available version"

III

CONTENTS

Page Intentionally Left Blank

Table

REVISION STATUS SUMMARY

The following list identifies the revisions made to this specification by symbol, a brief summary of the purpose of each revision, and the pages revised.

REVISION SYMBOL	REVISION PURPOSE / PAGE NUMBER(S)	REVISION DATE / ORIGINATOR/APPROVAL
A	To incorporate revision status summary page. vii To correct typographical and editorial errors in original issue. iv, vi, 1-14, 2-11, 2-12, 2-13, 2-14, 2-15, 2-16, 2.4.1-5, 2.4.3-1, 2.4.4-1, 2.4.5-2, 2.4.6-2, 2.4.7-1, 2.4.7-2, 2.4.7-3, 2.4.8-4, 2.4.8-5, 2.4.9-3, 2.4.9-4, 2.4.10-2, 2.4.10-3, 2.4.10-4, 2.4.11, 2.4.11-1, 2.4.11-3, 2.4.11-5, 2.4.12, 2.4.12-1, 2.4.12-2, 2.4.12-3, 2.4.12-4, 2.4.12-5, 2.4.12-6, 2.4.13-2, 2.4.13-3, 2.4.14-3, 2.4.14-5, 2.4.14-8, 2.4.14-9, 2.4.15-2, 2.4.15-3, 2.4.16-2, 2.4.16-3, 2.4.17-2, 2.4.18-3, 2.4.21-1, 2.4.21-2, 2.4.22-2, 2.4.22-3, 2.4.23-1, 2.4.23-2, 2.4.23-3, 2.4.23-4, 2.4-23-5, 2.4.23-6, 2.4.23-7, 2.4.23-8, 2.4.24-2, 2.4.24-3, 2.4.24-5, 2.4.25-4, 2.4.25-5, 2.4.25-6, 2.4.25-7, 2.4.25-8, 2.4.26-2, 2.4.26-5, 2.4.27-2, 2.4.27-3, 2.4.27-4, 2.4.28-2, 2.4.28-5, 2.4.29-2, 2.4.29-3, 2.4.29-4, 2.4.30-4, 2.4.30-5, 2.4.30-6, 2.4.31-2, 2.4.32-4, 2.4.32-5, 2.4.32-6, 2.4.32-17, 2.4.32-24, 2.4.33-1, 2.4.33-2, 2.4.33-8, 2.4.34-3, 2.4.34-4, 2.4.34-6, 2.4.34-7, 2.4.34-8, 2.4.34-9, 2.4.34-10, 2.4.34-11, 2.4.34-17, 2.4.35, 2.4.36-3, 2.4.36-4, 2.4.38-5	18 October 1974 J. Willoughby/J. T. Slater

Reading the definition from the top, it is said to be an augmented grammar definition of the language METASYNTAX. The rule defining METASYNTAX indicates that a language definition starts with the string .AUG_GRAM, followed by an identifier (the name of the language), followed by a RULE. The next character of the rule METASYNTAX is an iteration operator. The dollar sign has the meaning "zero or more occurrences of the element..." Thus, after the mandatory RULE. zero or more additional RULEs may appear. Finally, the string .END terminates the language definition.

The next rule defines a RULE as an identifier (the metavariable name), followed by the string ":=", followed by an EXPRESSION. Then zero or more occurrences of an element may occur, where the element consists of the single character "|" (vertical bar meaning "or") *followed by an* EXPRESSION. *Note the use of parentheses* to form a group that can be treated as a single unit. The

EXPRESSION $("|" EXPRESSION)

portion of the rule allows a RULE to contain alternatives. An example is the NOUN rule of Fig. 1.1.1-1, which is read "a NOUN consists of the word 'BOY', *or* the word 'GIRL', *or* the word 'DOG', *or* the word 'CAT'". Finally, the RULE is said to be terminated by a semicolon (the one in quotation marks) and the RULE rule, having been completed, is itself terminated (by a semicolon without quotation marks). Keep in mind that symbols in quotation marks represent terminal symbols in the language *being defined* (in this case METASYNTAX), while symbols not in quotation marks have meaning in the language in which the definition is written.

An EXPRESSION consists of an ELEMENT followed by zero or more additional ELEMENTs. An ELEMENT is an identifier (used for meta-variable names), a string (used for specific terminal symbols), or any of the specific symbols ".ID", ".STRING", ".LABEL," ".TREE", ".NUM", or ".EMPTY". ".EMPTY" is a reference to the null character string, which represents a condition that, during parsing, is always satisfied. It is used when optional elements are involved. If, for example, one wished to define a number with or without preceding signs (unary operators), the rule might take the form

```
NUMBER  :=  ( "+" | "-" | .EMPTY )  .NUM ;
```

which specifies that either "+", "-", or nothing at all may precede the number itself.

An additional alternative for ELEMENT (after ".EMPTY") is

```
".PEEK"  "("  .STRING  ")"
```

which represents a look-ahead capability. The effect is to peek ahead at the next input symbol to determine whether it is a specified string. The significance will be clearer when parsing is discussed in more detail. The final alternatives for ELEMENT represent a parenthesized expression and an iteration respectively. Finally, the definition of METASYNTAX is terminated by the string ".END".

It is suggested that the serious reader consider this definition of METASYNTAX as a grammar for the simple language of Fig. 1.1.1-1 and as a grammar for the language in which the definition of METASYNTAX is written. This familiarization will help considerably when the metalanguage is used later in the definition of PLANS.

"Page missing from available version"

(2-1
to
2-10)

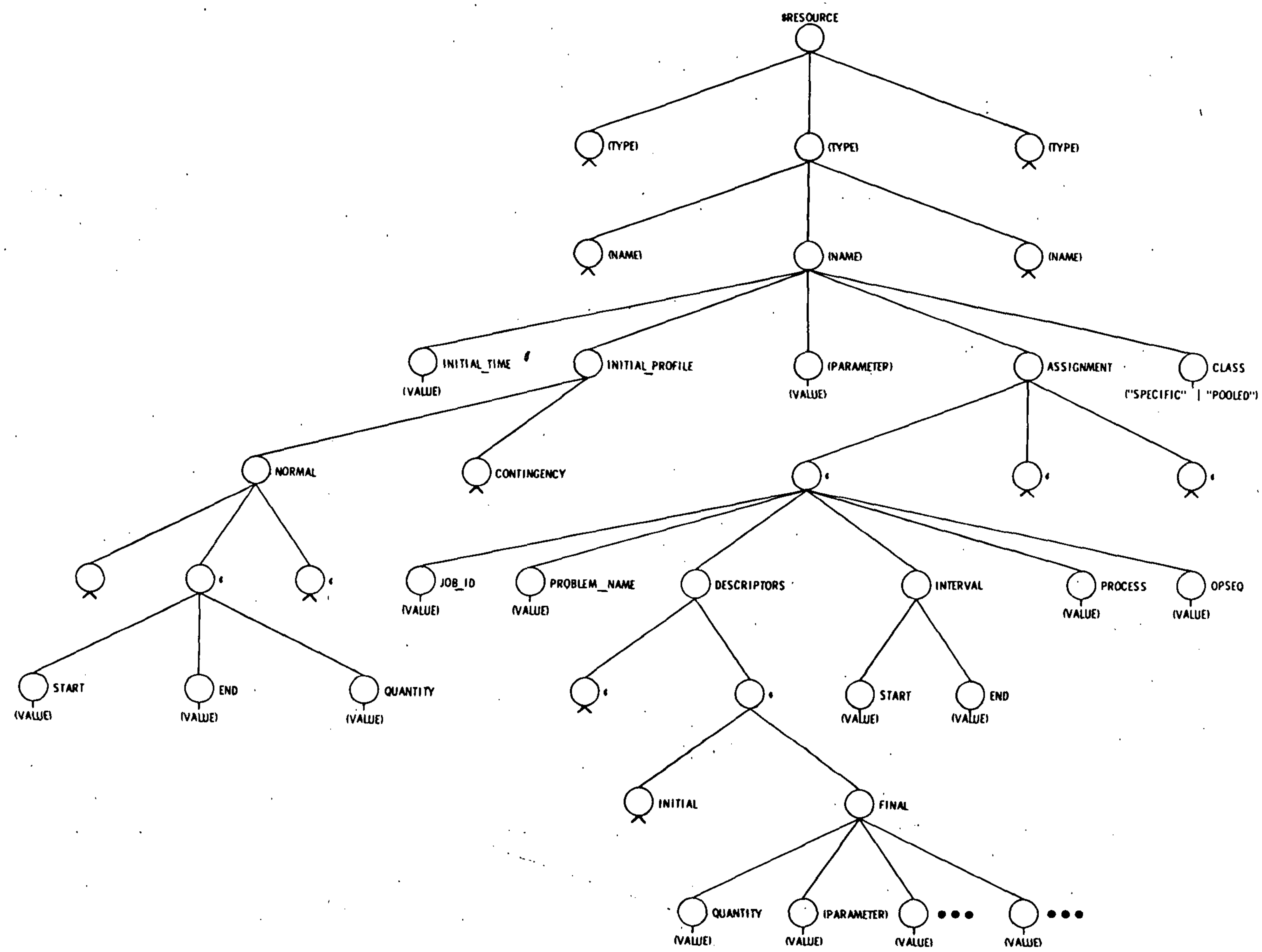


Fig. 2.2-1 $RESOURCE *Standard Data Structure*

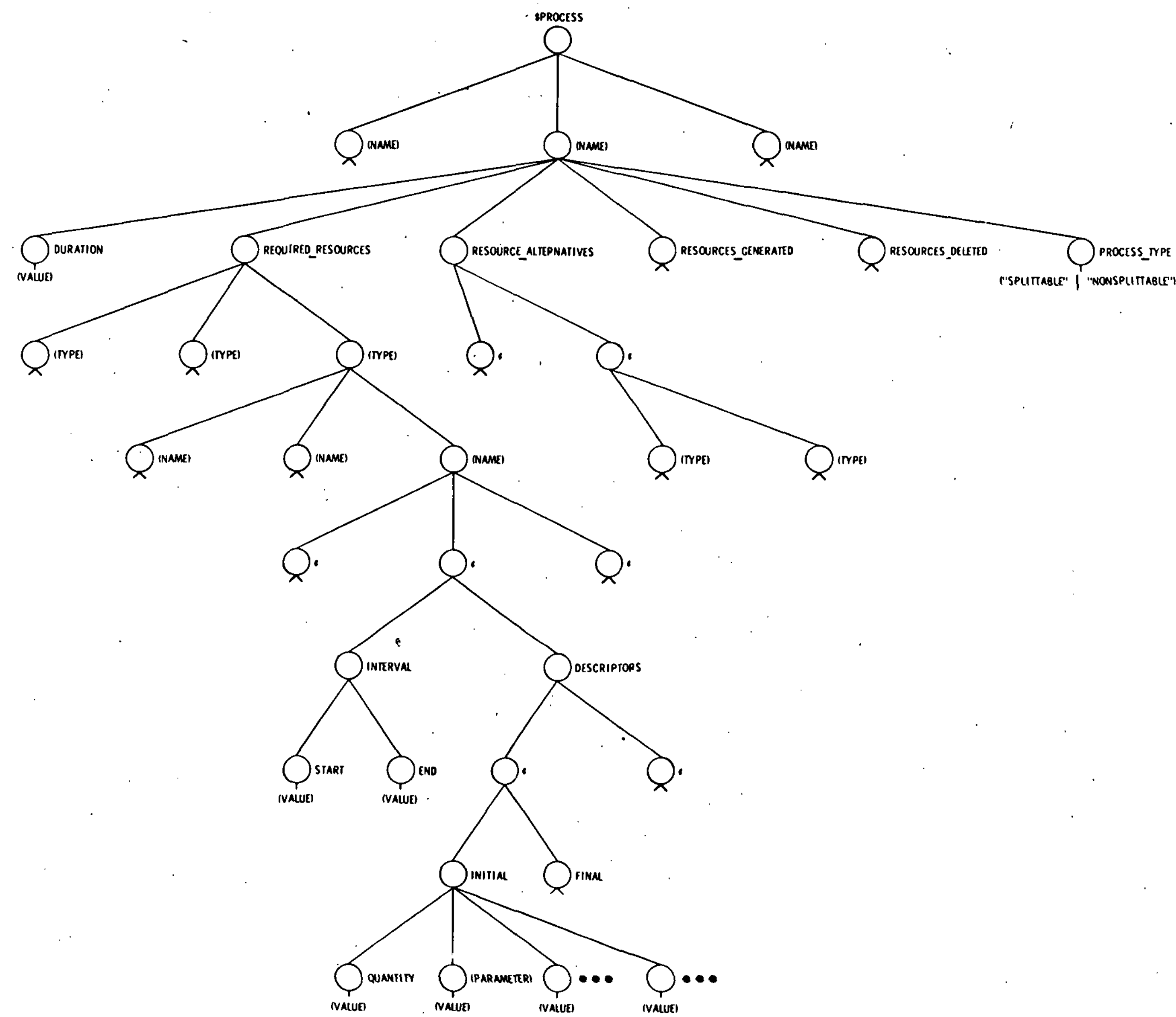


Fig. 2.2-2 $PROCESS *Standard Data Structure*

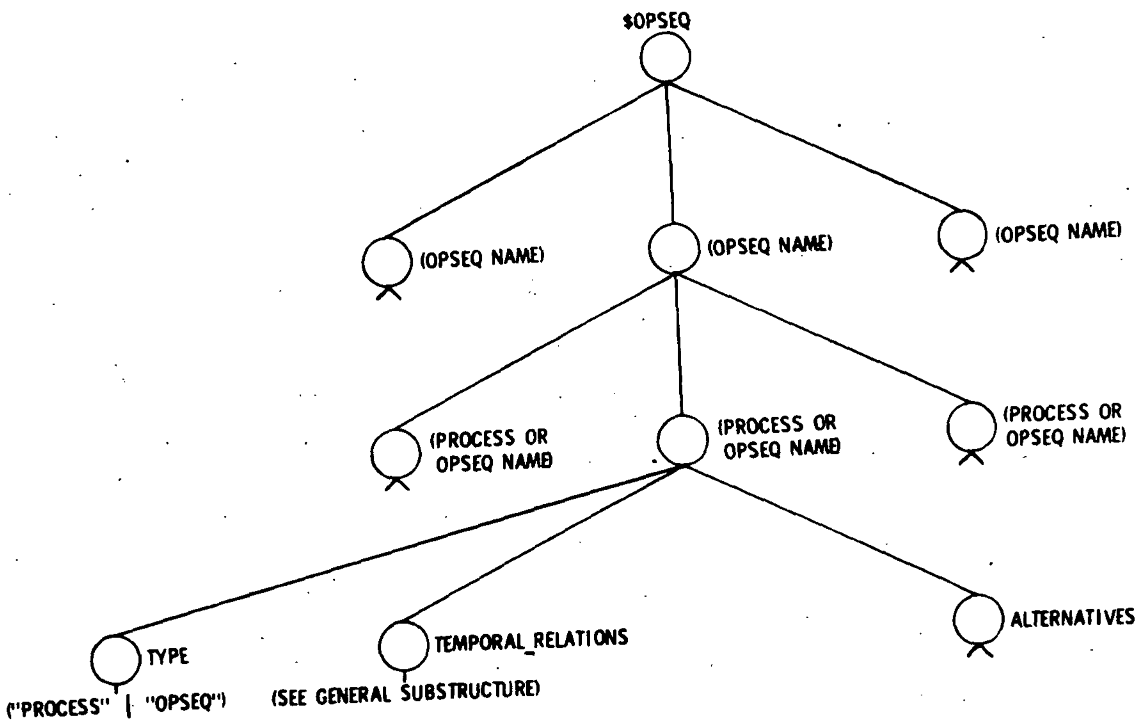


Fig. 2.2-3 $OPSEQ *Standard Data Structure*

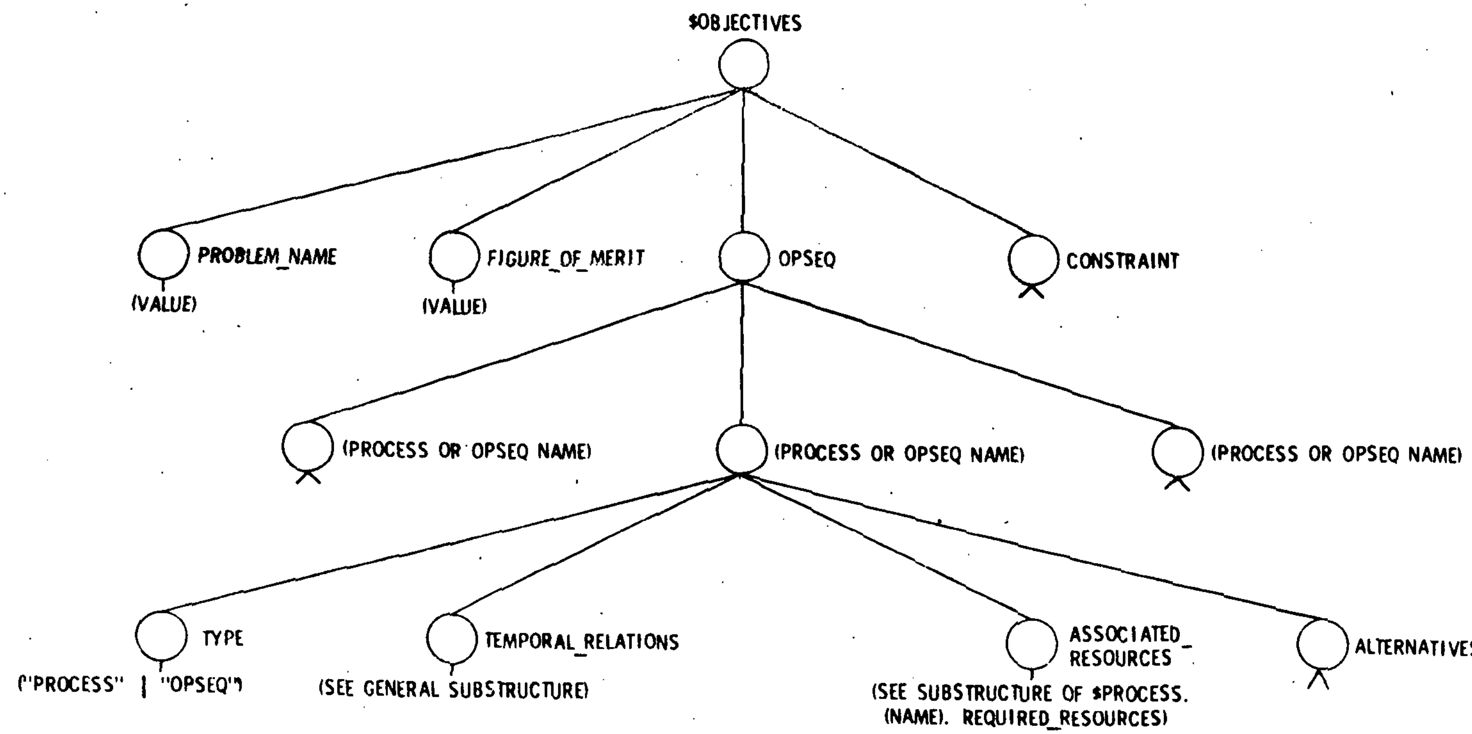


Fig. 2.2-4 $OBJECTIVES *Standard Data Structure*

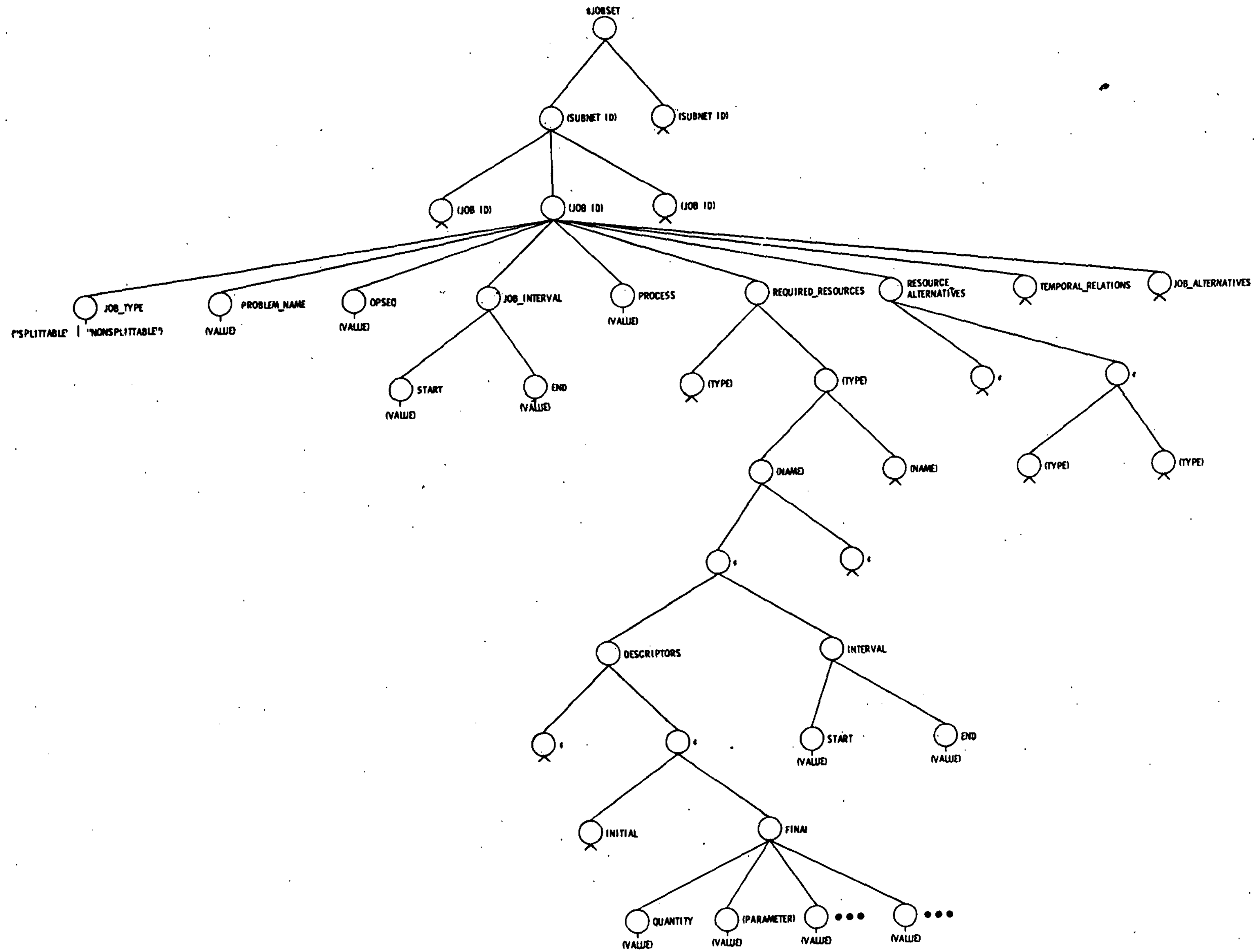


Fig. 2.2-5 $JOBSET *Standard Data Structure*

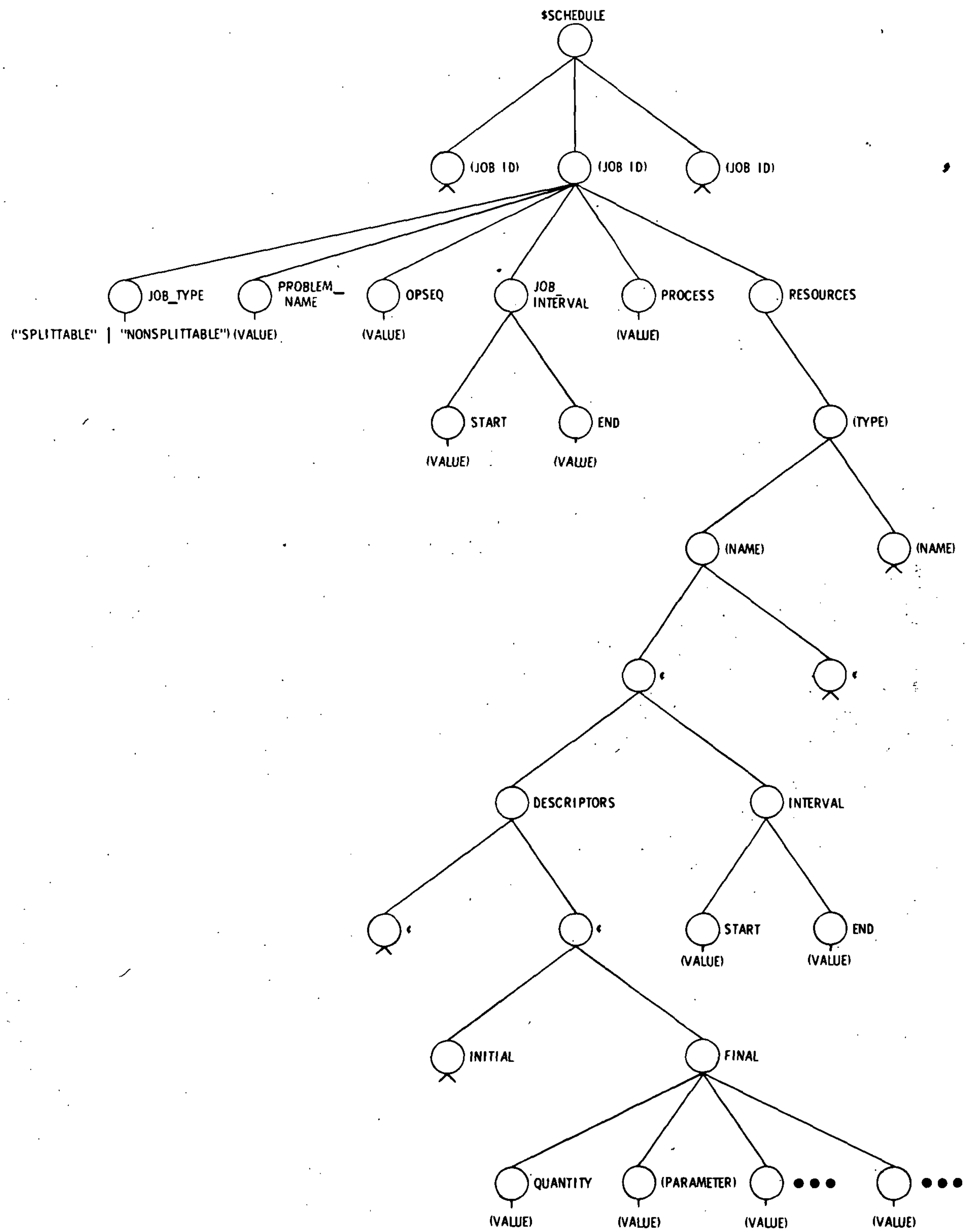


Fig. 2.2-6 $SCHEDULE *Standard Data Structure*

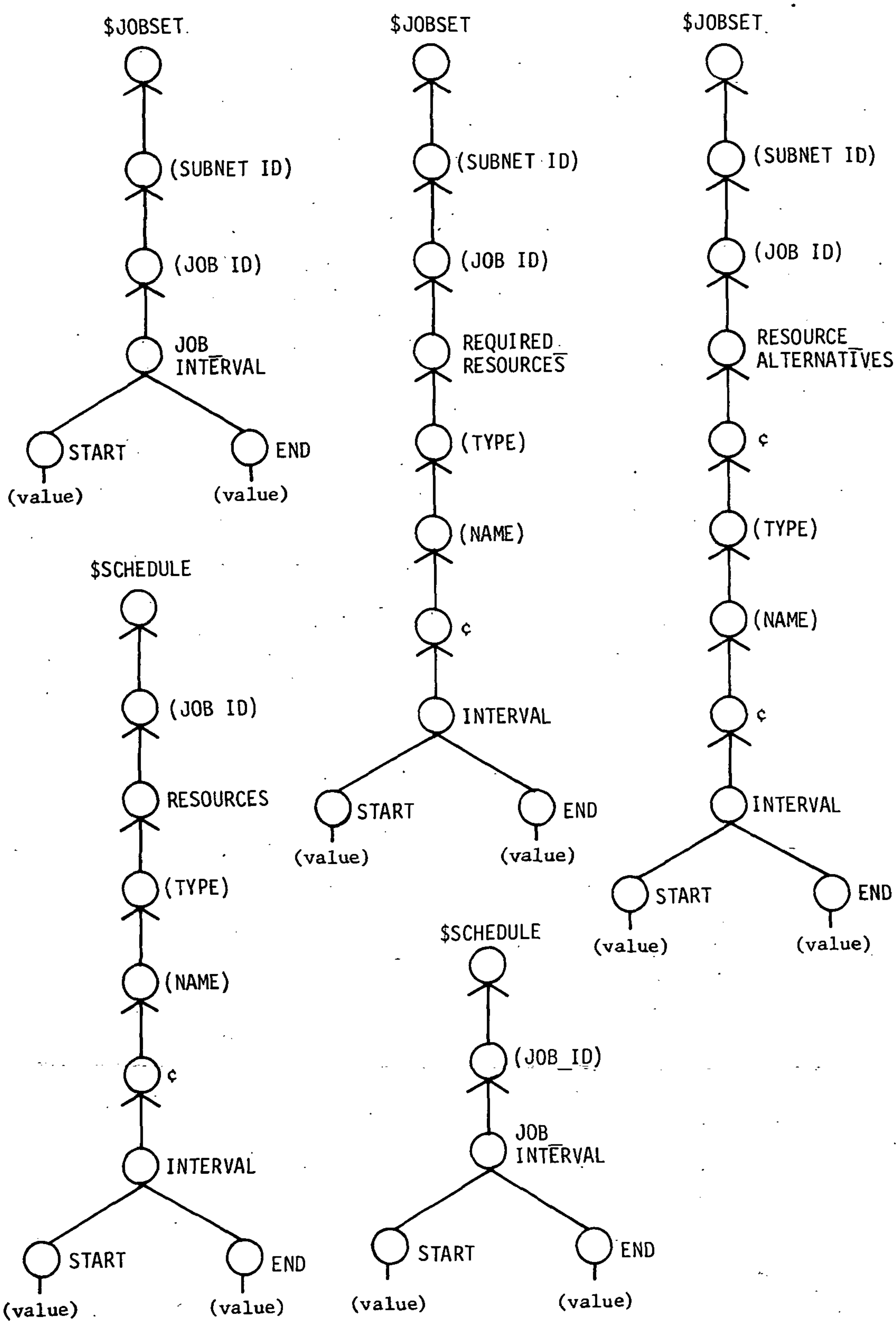
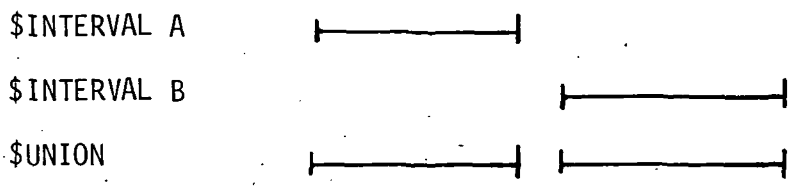


Fig. 2.4.1-2 (concl)

2.4.3 INTERVAL_UNION

2.4.3.1 Purpose and Scope

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



2.4.3.2 Modules Called

None

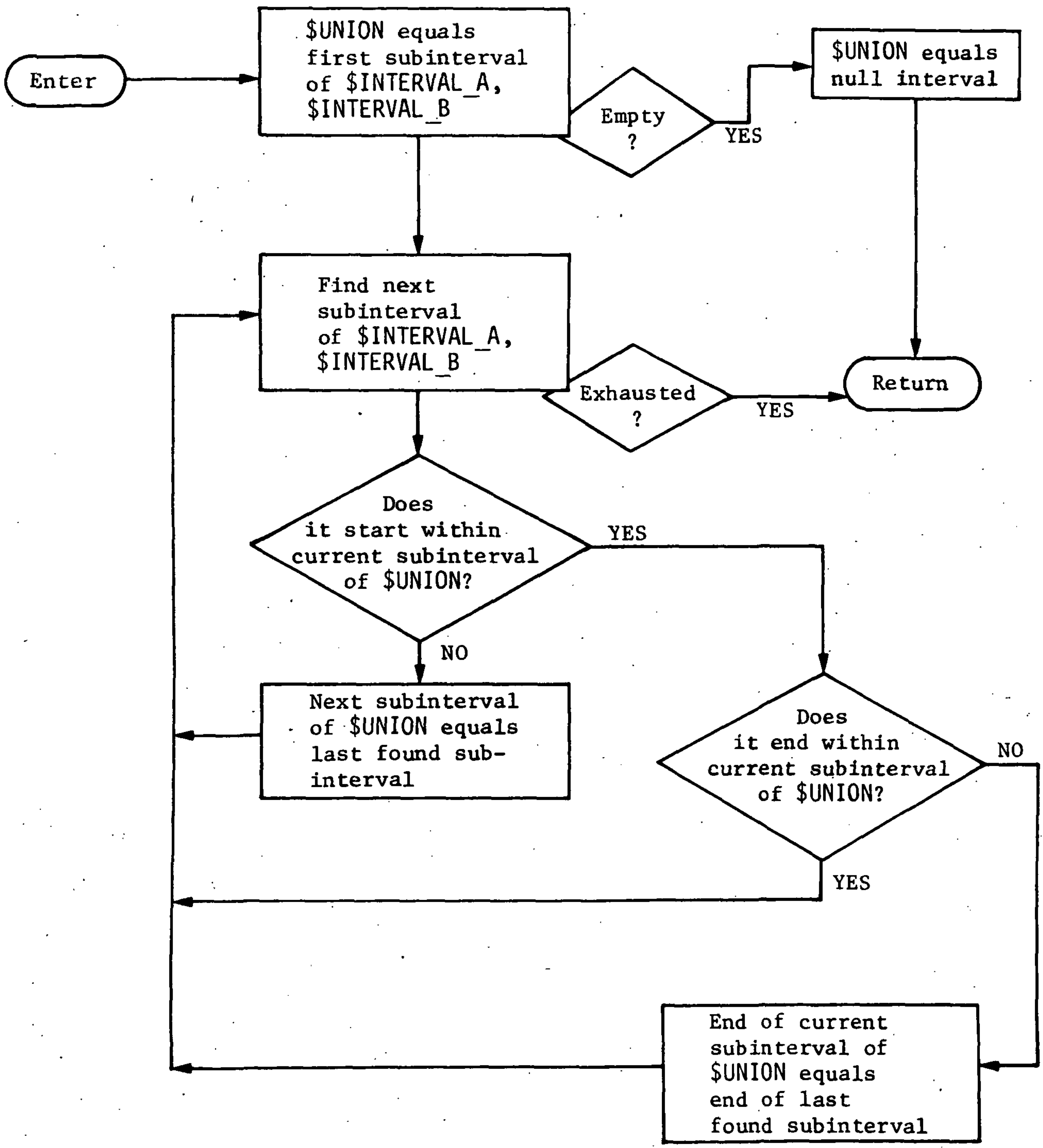
2.4.3.3 Module Input

$INTERVAL_A and $INTERVAL_B are standard intervals.

2.4.3.4 Module Output

$UNION is a standard interval.

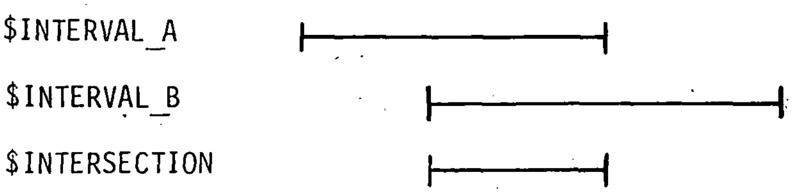
2.4.3.5 Functional Block Diagram



2.4.4 INTERVAL_INTERSECTION

2.4.4.1 Purpose and Scope

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



2.4.4.2 Modules Called

None

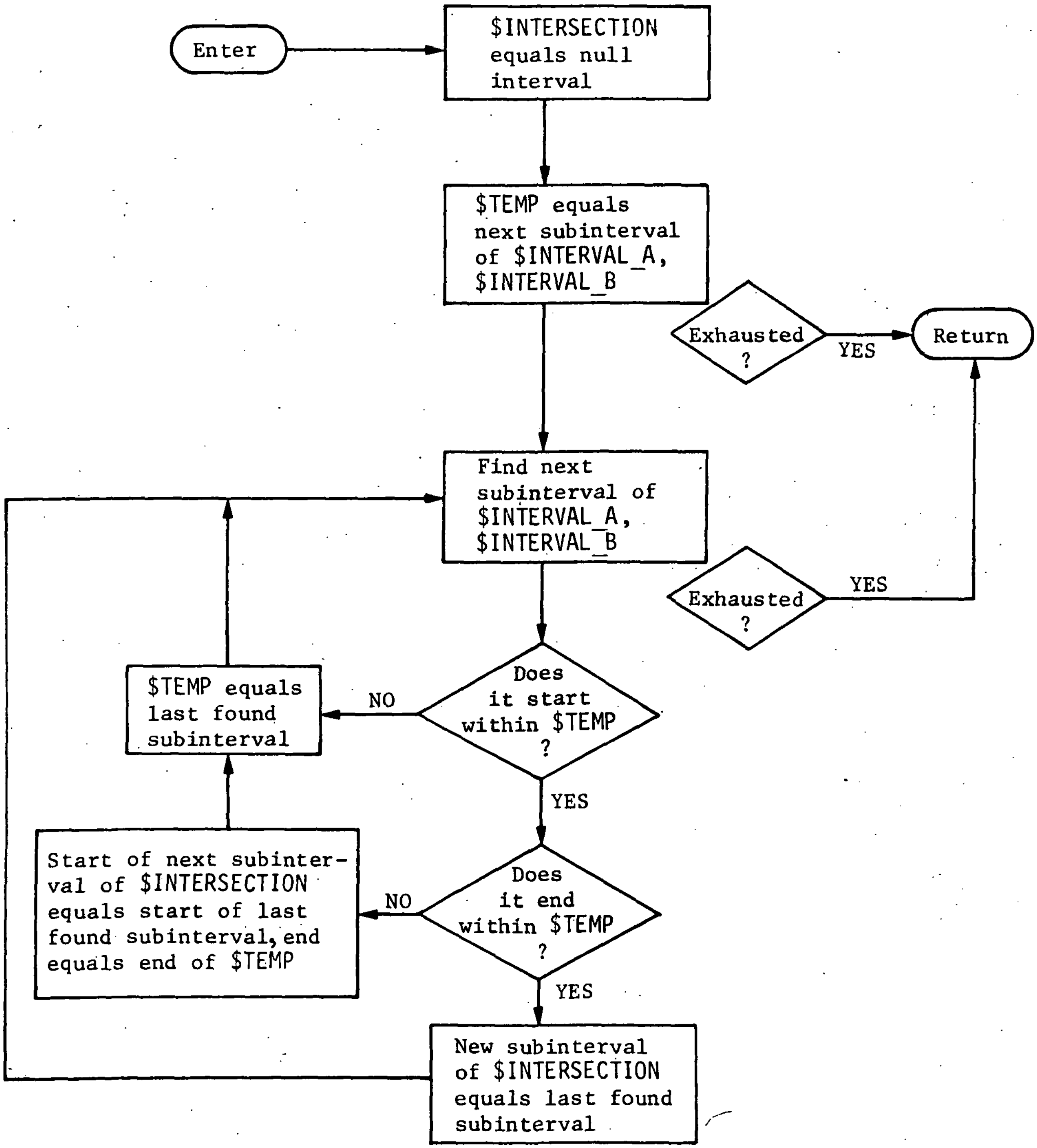
2.4.4.3 Module Input

$INTERVAL_A and $INTERVAL_B are standard intervals.

2.4.4.4 Module Output

$INTERSECTION is a standard interval.

2.4.4.5 Functional Block Diagram



2.4.5 FIND_MAXIMUM

2.4.5.1 Purpose and Scope

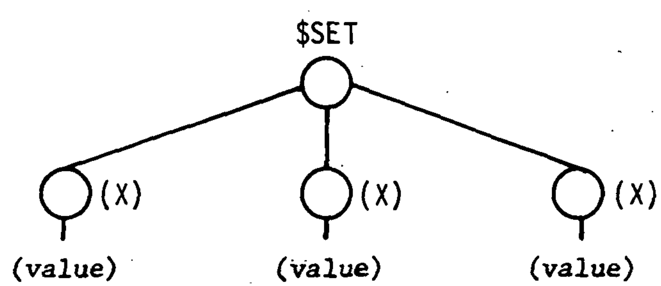
Given a set of numerical values (i.e., a node of a tree for which each of the next lower level subnodes is terminal and has a numerical value), find the maximum (minimum) of the values and find the indices (i.e., the ordinal positions in the original set) of each of the subnodes for which the value equals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

2.4.5.2 Modules Called

(None)

2.4.5.3 Module Input

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

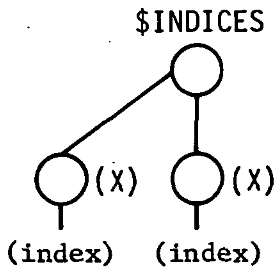


where each value is numeric.

2.4.5.4 Module Output

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

$INDICES is a tree of the form



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

2.4.6 FIND_MINIMUM

2.4.6.1 <u>Purpose and Scope</u>

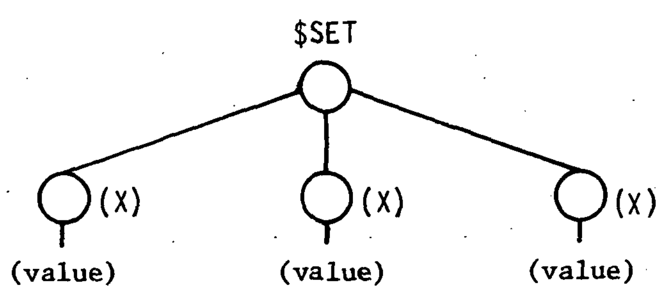
Given a set of numerical values (i.e., a node of a tree for which each of the next lower level subnodes is terminal and has a numerical value), find the minimum of the values and find the indices (i.e., the ordinal positions in the original set) of each of the subnodes for which the value equals the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

2.4.6.2 <u>Modules Called</u>

(None)

2.4.6.3 <u>Module Input</u>

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

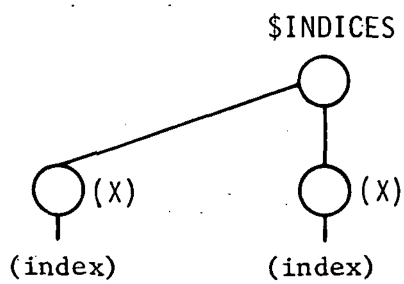


where each value is numeric.

2.4.6.4 <u>Module Output</u>

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

$INDICES is a tree of the form



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

2.4.7 CHECK_FOR_PROCESS_DEFINITION

2.4.7.1 Purpose and Scope

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

2.4.7.2 Modules Called

None

2.4.7.3 Module Input

Input to this module consists of $OBJECTIVES, $OPSEQ and $PROCESS. The minimum required data structure from these Standard Data Structures is illustrated in Fig. 2.4.7.3-1.

2.4.7.4 Module Output

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

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

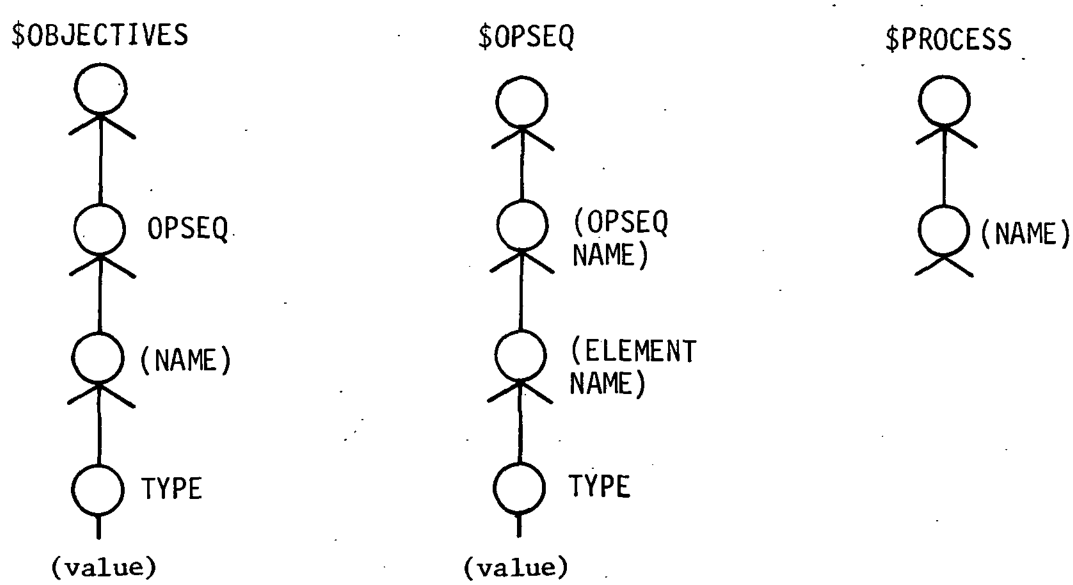
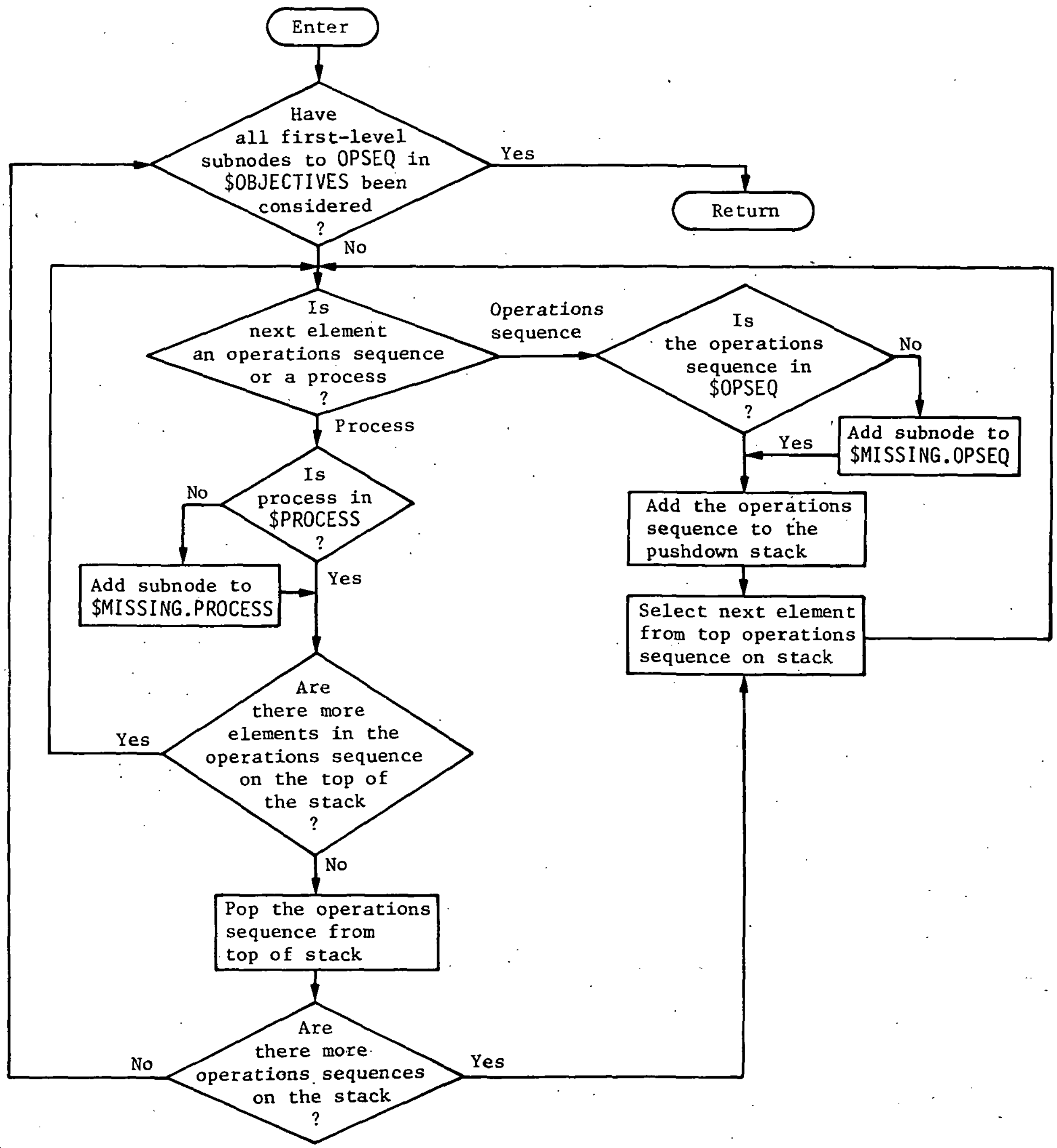


Fig. 2.4.7.3-1
Minimum Required Input Structures from Standard Data Structures for Module: CHECK_FOR_PROCESS_DEFINITION

2.4.7.5 Functional Block Diagram



2.4.7.6 Typical Applications

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

2.4.8.3 Module Input

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

2.8.4.4 Module Output

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

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

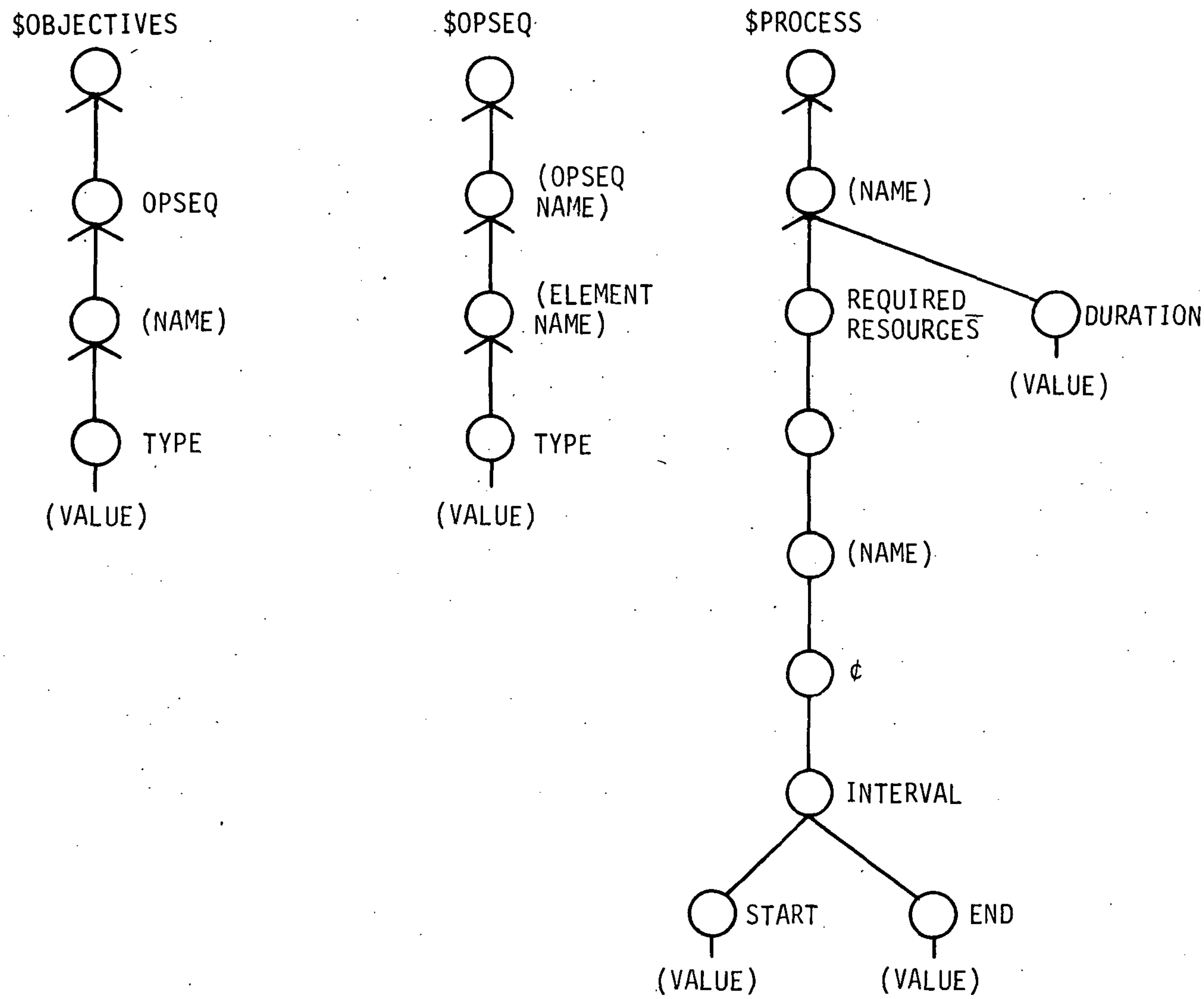


Fig. 2.4.8-1
Minimum Required Input Structures from Standard Data Structures for Module Generation

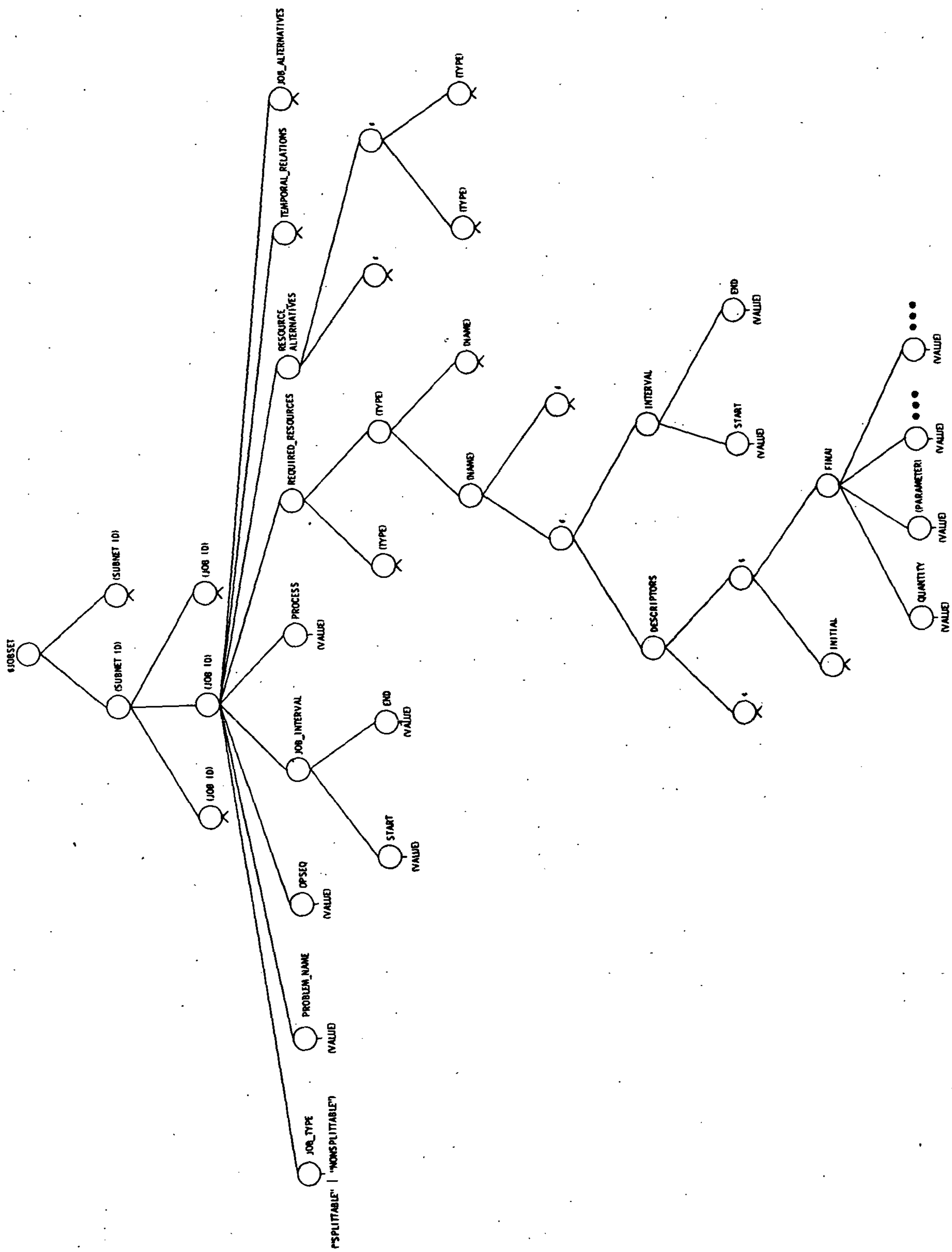
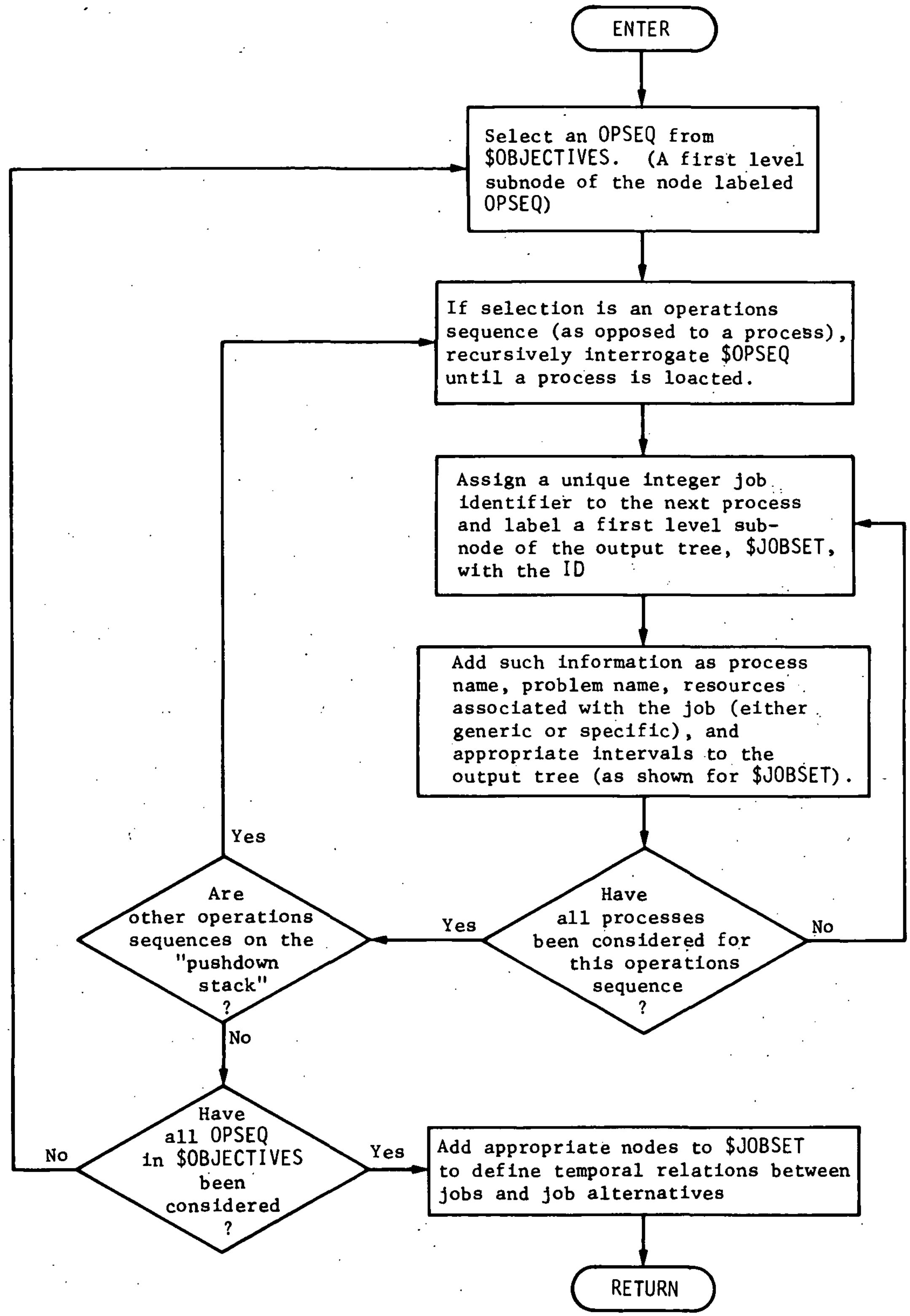


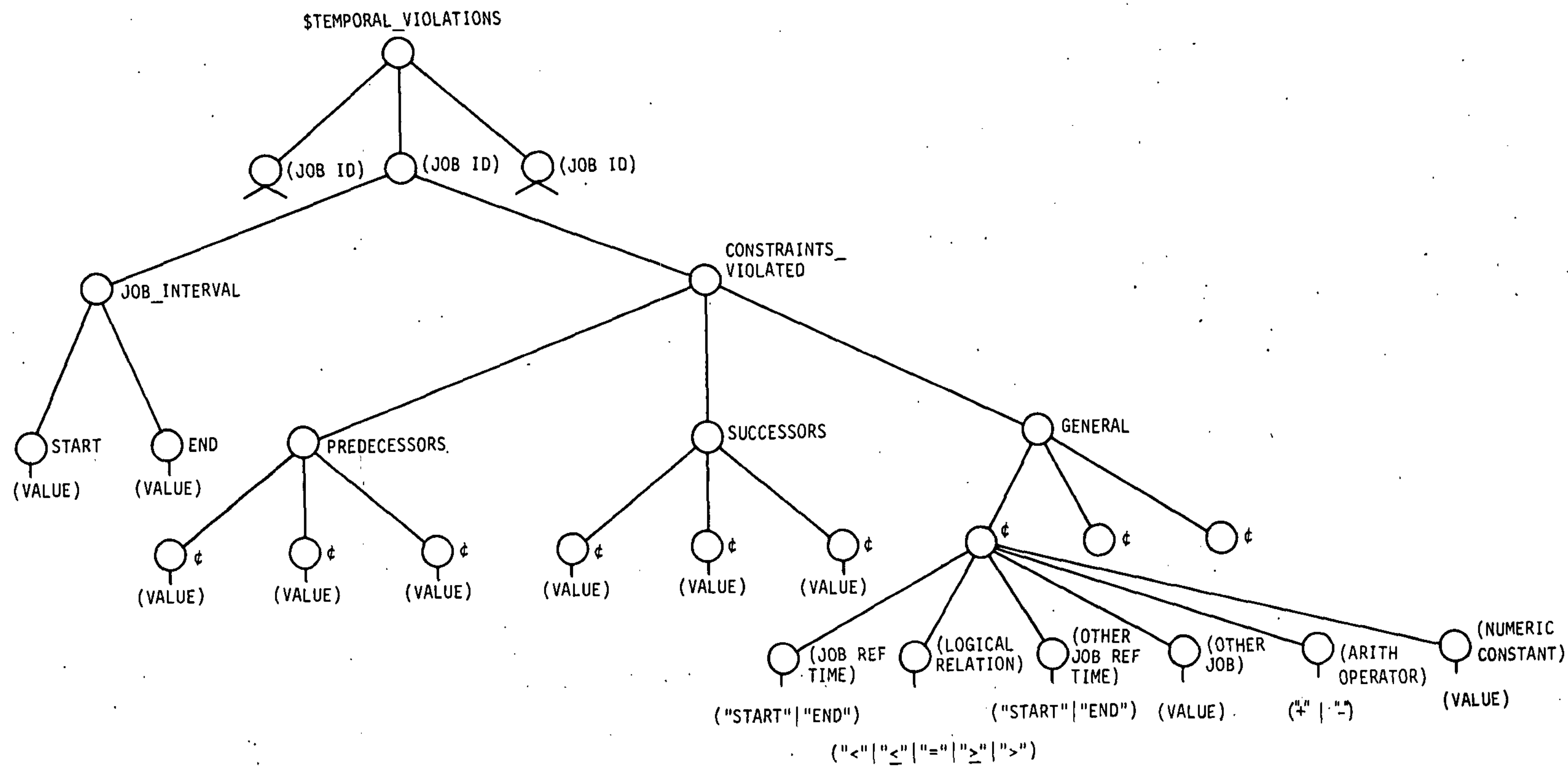
Fig. 2.4.8-2 GENERATE_JOBSET Standard Data Structure

Fig. 2.4.8-2

2.8.4.5 Functional Block Diagram



OUTPUT DATA STRUCTURE



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

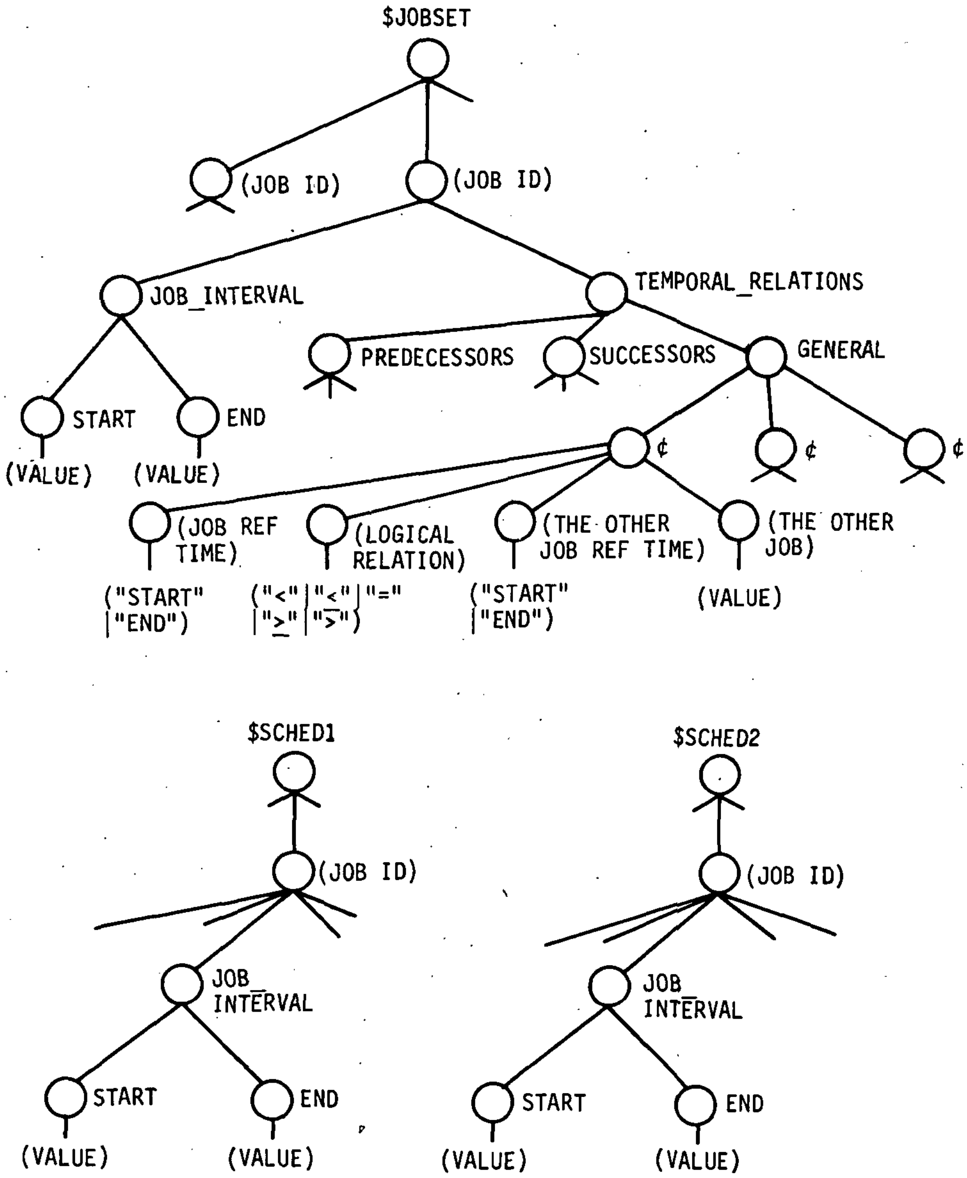


Fig. 2.4.9-1
Minimum Required Input Structures from Standard Data Structures for Module:
CHECK_EXTERNAL_TEMP_RELATIONS

2.4.10 CHECK_INTERNAL_TEMP_RELATIONS

2.4.10.1 Purpose and Scope

This module will determine the temporal relations specified for jobs in $JOBSET that are violated within a single partial schedule that has two or more jobs.

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

2.4.10.2 Modules Called

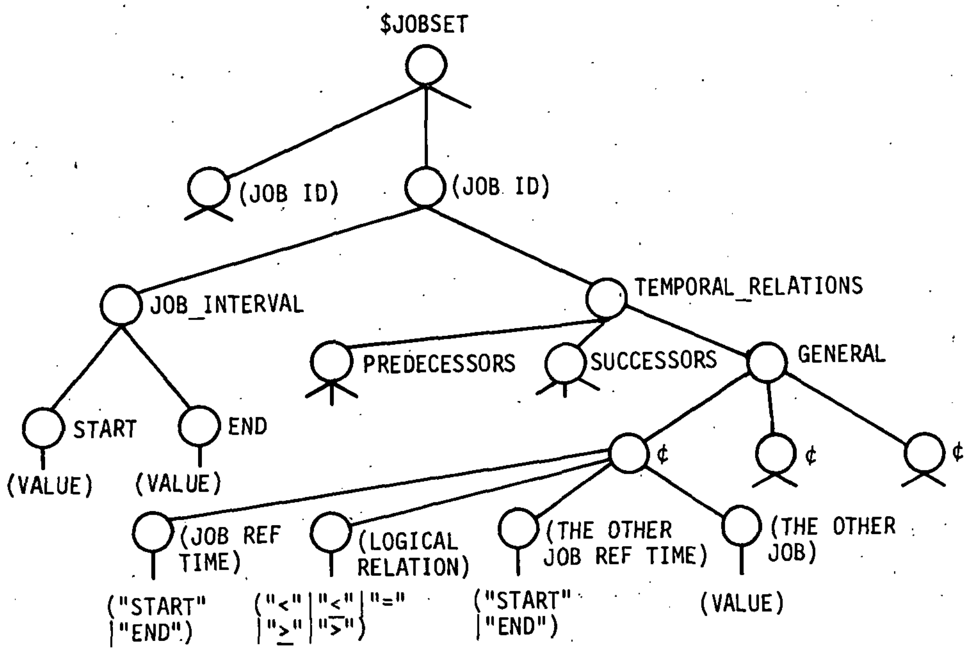
CHECK_ELEMENTARY_TEMP_RELATION

2.4.10.3 Module Input

This module will be called with three arguments. There are two input arguments: $JOBSET and $SCHED. The structure of $JOBSET is identical to the structure output from the module GENERATE_JOBSET. The structure of $SCHED is that of the standard schedule unit.

The minimum data structures required from the standard structures $JOBSET and $SCHEDULE are shown on the following page. Note that in the minimum structure the fifth and sixth subnodes of a relation in the TEMPORAL_RELATIONS substructure are not mandatory in every case.

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



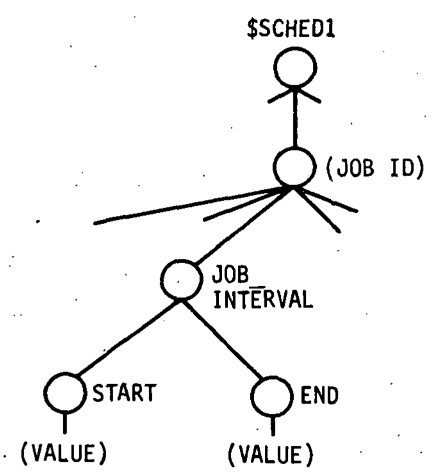
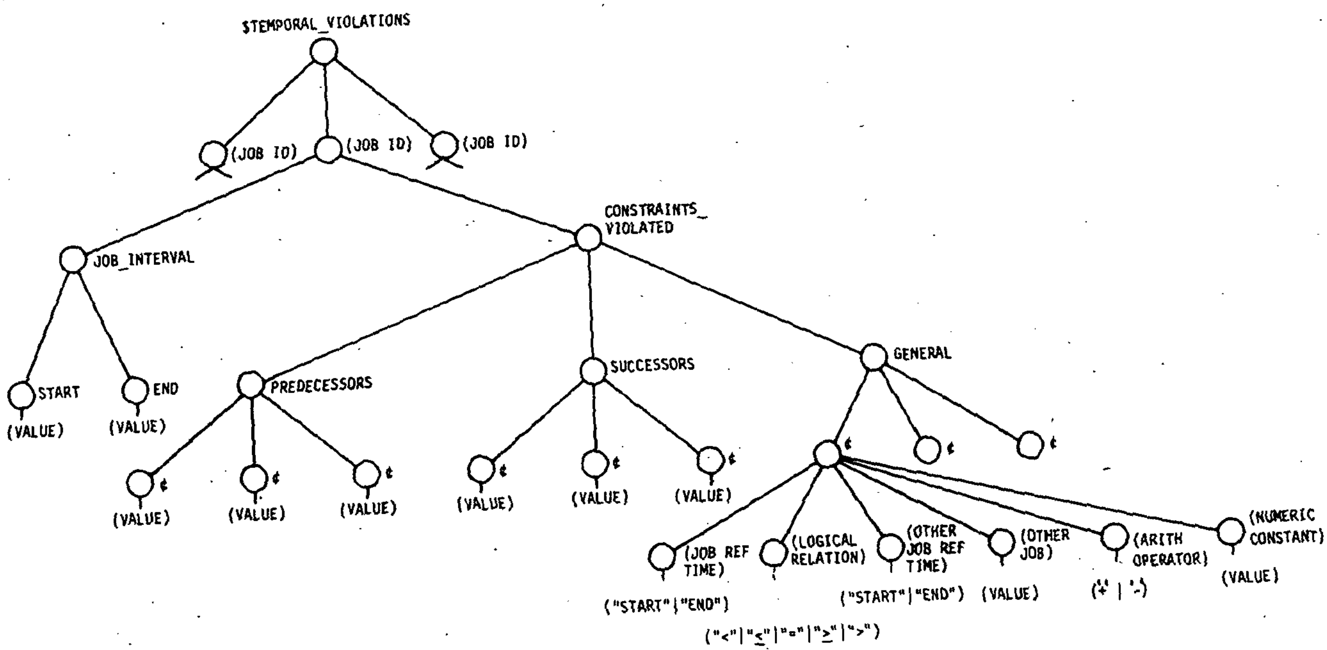


Fig. 2.4.10-1
Minimum Required Input Structures from Standard Data Structures for Module: CHECK_INTERNAL_TEMP_RELATIONS

2.4.10.4 Module Output

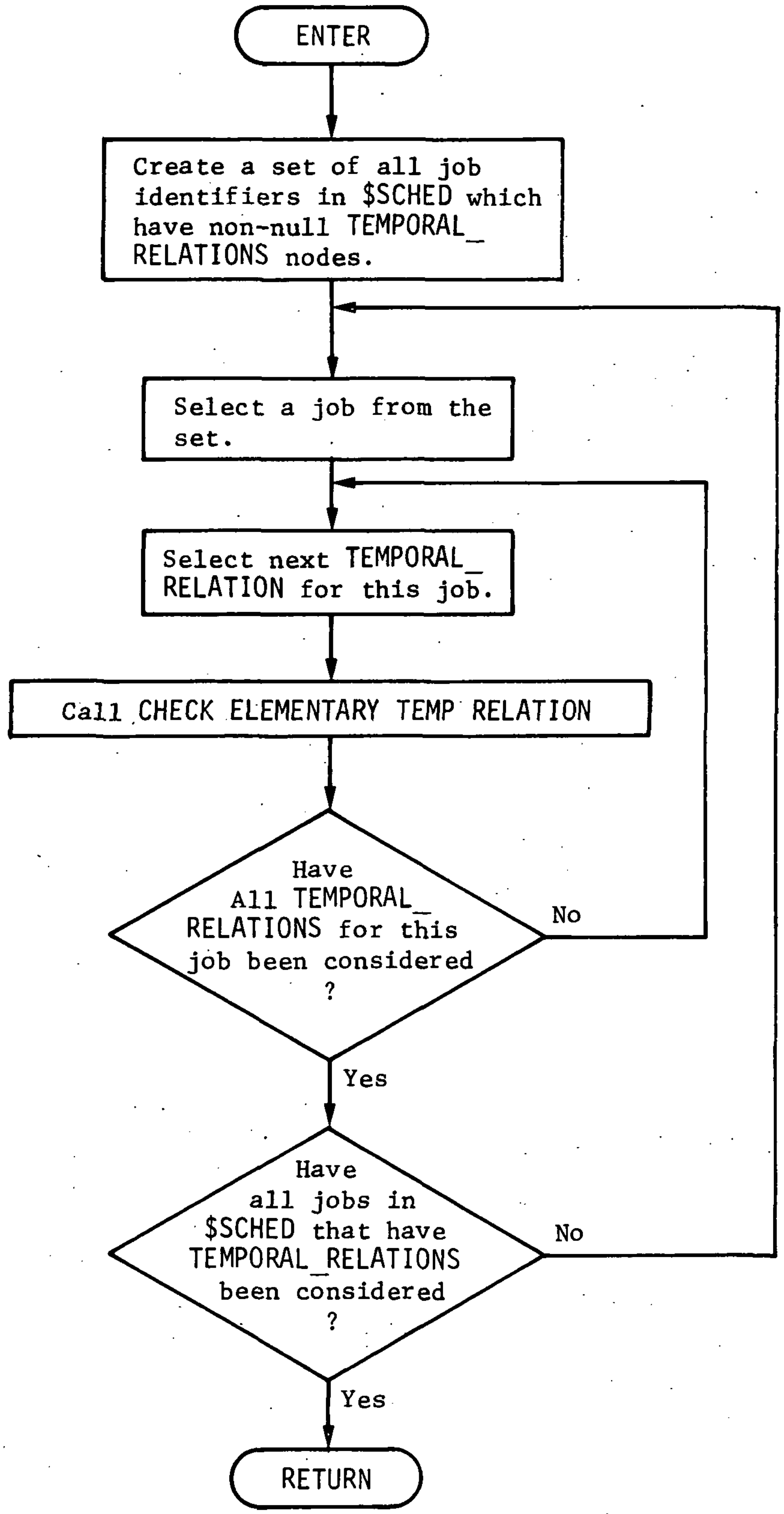
This module will build and return an output tree with the structure shown below:

OUTPUT DATA STRUCTURE



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

2.4.10.5 Functional Block Diagram



2.4.11 CHECK_ELEMENTARY_TEMP_RELATION

CHECK_ELEMENTARY_TEMP_RELATION

2.4.11 CHECK_ELEMENTARY_TEMP_RELATION

2.4.11.1 Purpose and Scope

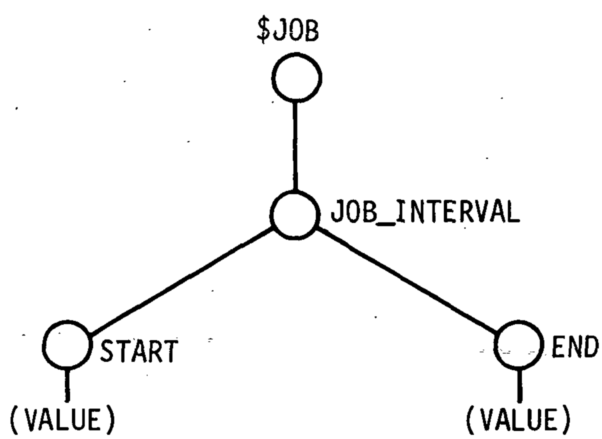
This module is elementary in the sense that it determines satisfaction or nonsatisfaction of a single input relationship involving the start or end times of two jobs for which specific start and end times have been assigned. The principal use of this module is to service higher level logic that is checking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

2.4.11.2 Modules Called

None

2.4.11.3 Module Input

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



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

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

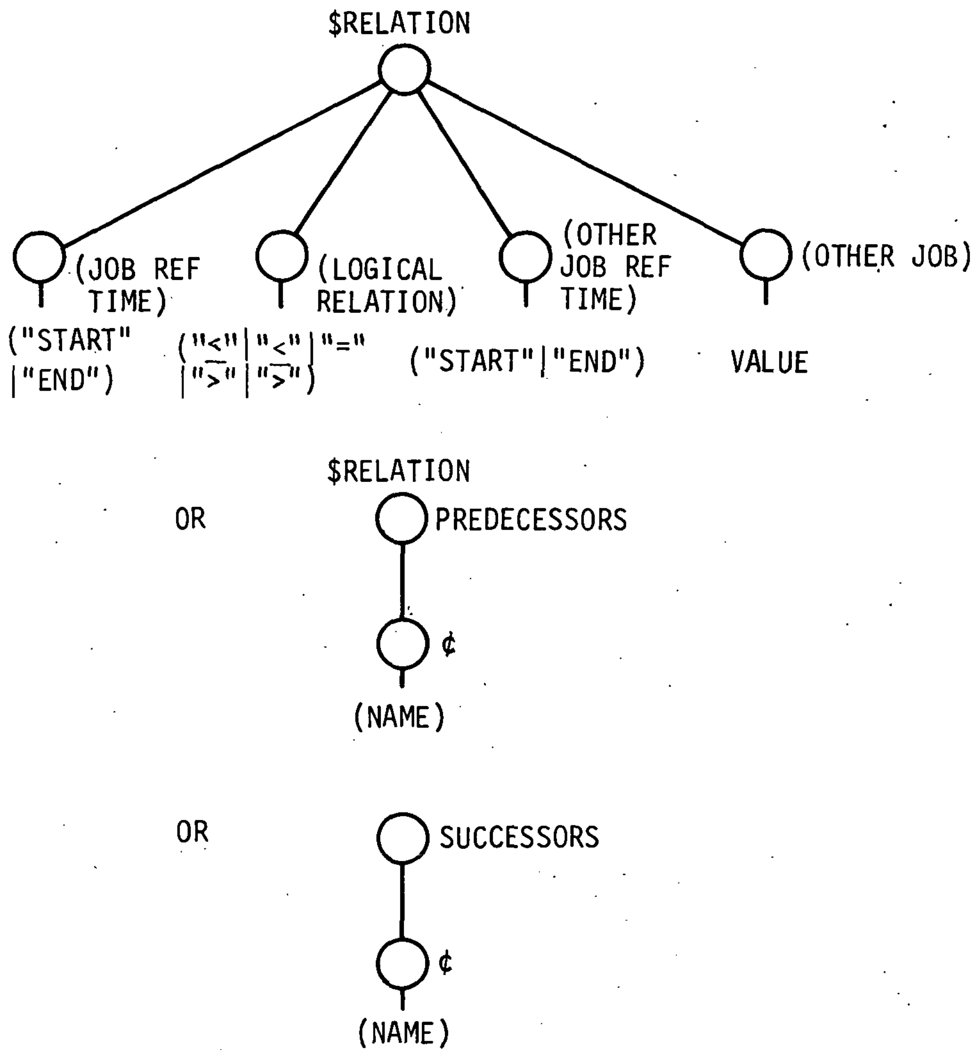
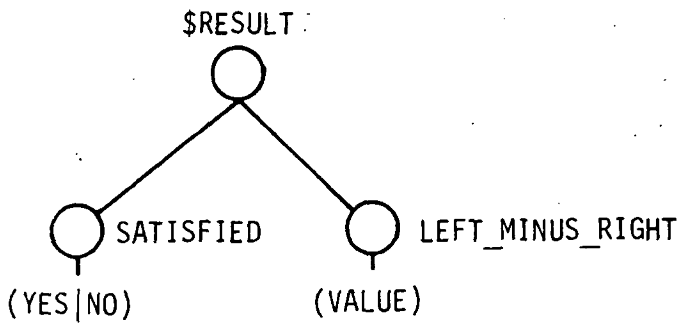


Fig. 2.4.11-1
Minimum Required Input Structures from Standard Data Structures for Module: CHECK_ELEMENTARY_TEMP_RELATION

2.4.11.4 Module Output

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



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

GENERAL RELATION

{START | END} {< | ≤ | = | ≥ | >} {START | END} OF {(JOB_ID)} {+ | -} {(CONSTANT)}

LEFT_SIDE: {START | END}
RIGHT_SIDE: {START | END} OF {(JOB_ID)} {+ | -} {(CONSTANT)}

PREDECESSOR

END OF JOB2 ≤ START OF JOB1

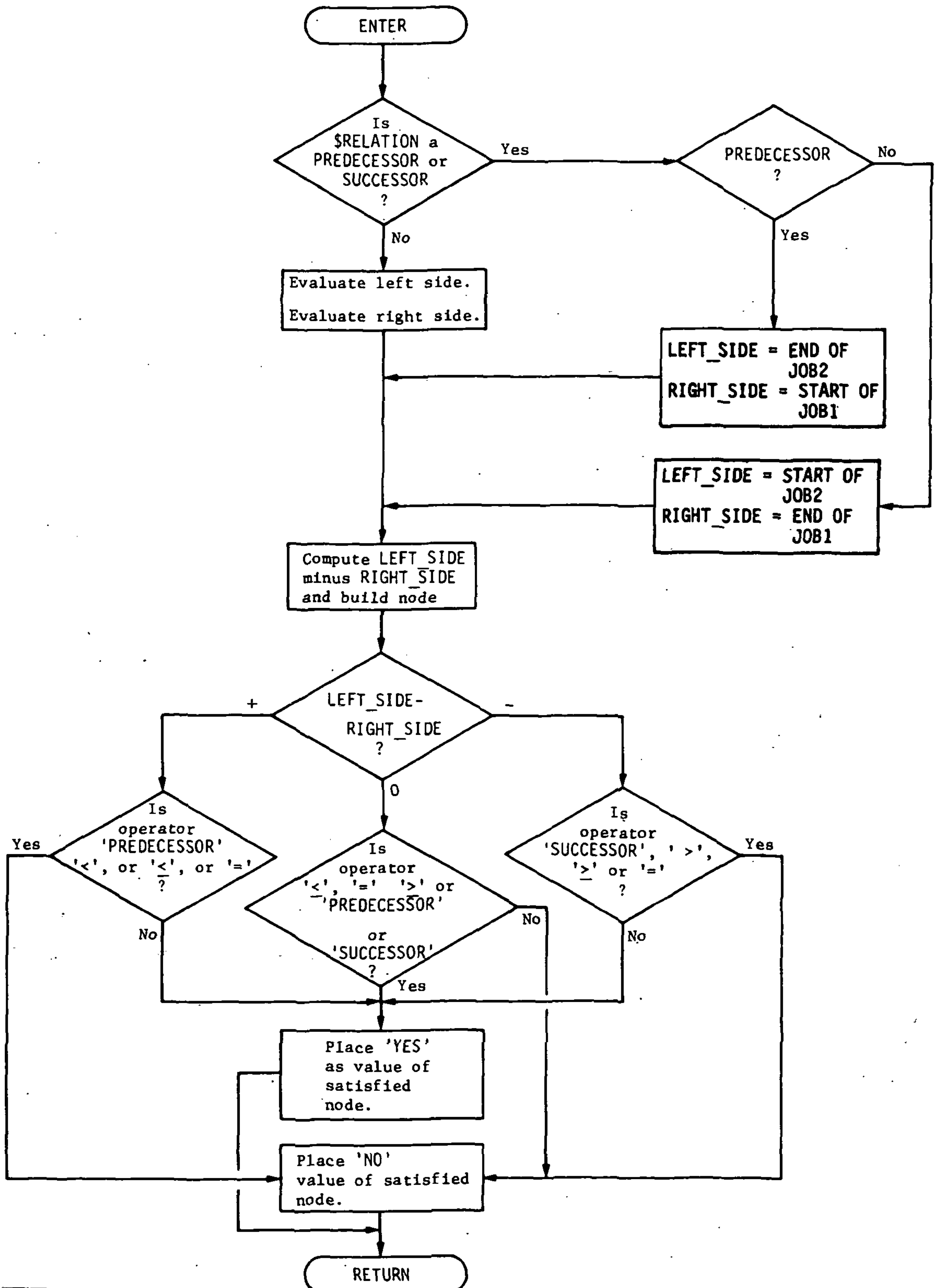
LEFT_SIDE: END OF JOB2
RIGHT_SIDE: START OF JOB1

SUCCESSOR

START OF JOB2 ≥ END OF JOB 1

LEFT_SIDE: START OF JOB2
RIGHT_SIDE: END OF JOB 1

2.4.11.5 Functional Block Diagram



2.4.12 NEXTSET

2.4.12 NEXTSET

2.4.12.1 Purpose and Scope

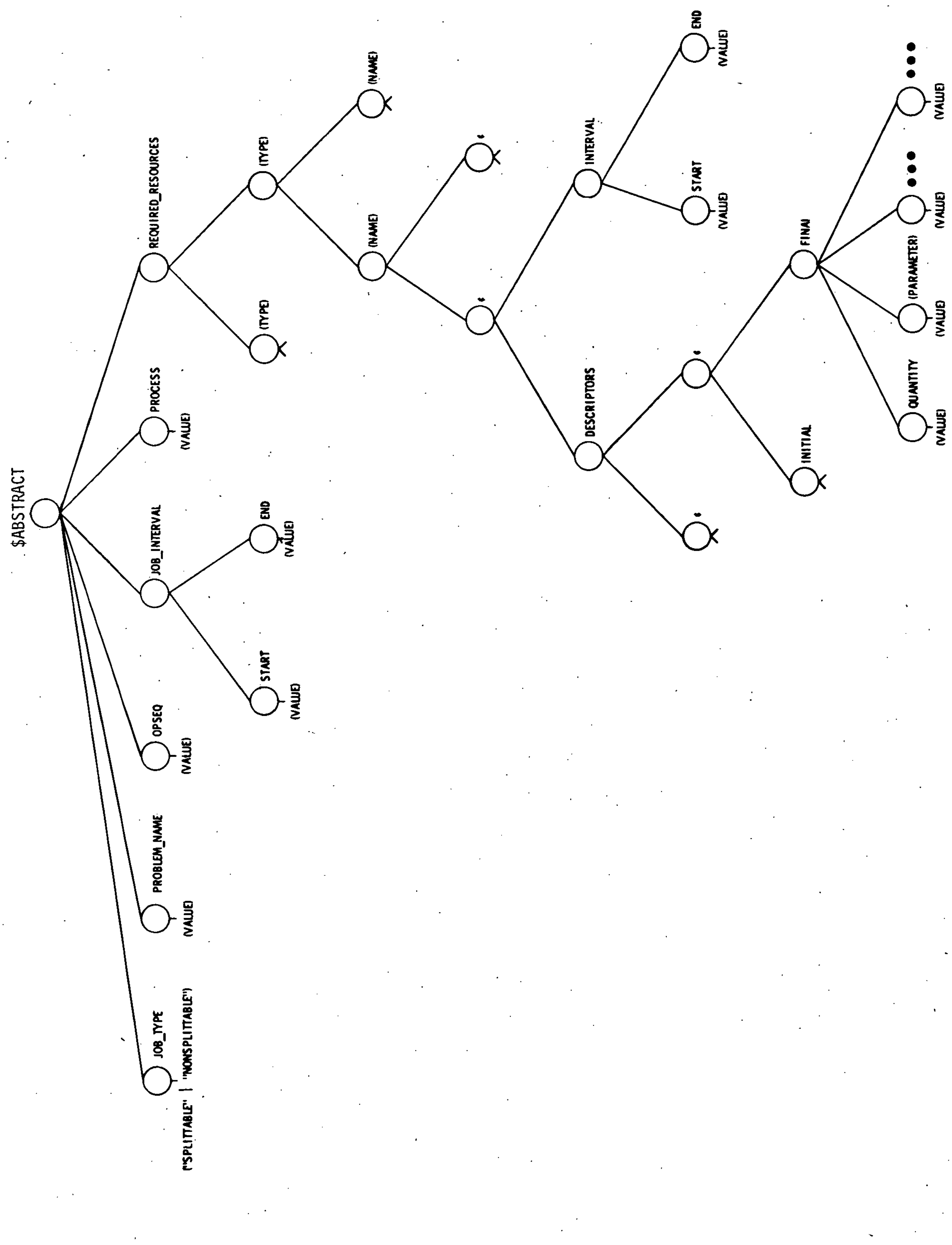
This module accepts an abstract description of item specific resource requirements associated with a specific job and, by referring to information about the assignments already scheduled for the resources, determines the earliest possible time (within a designated interval) at which the resource requirements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

2.4.12.2 Modules Called

DURATION
INTERVAL_UNION
INTERVAL_INTERSECTION

2.4.12.3 Module Input

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


$ABSTRACT
REQUIRED_RESOURCES
(TYPE)
(NAME)
(TYPE)
(NAME)
INTERVAL
START
(VALUE)
END
(VALUE)
DESCRIPTORS
FINAL
(VALUE)
(PARAMETER)
(VALUE)
QUANTITY
(VALUE)
(VALUE)
INITIAL
PROCESS
(VALUE)
JOB_INTERVAL
END
(VALUE)
START
(VALUE)
OPSEQ
(VALUE)
PROBLEM_NAME
(VALUE)
JOB_TYPE
("SPLITABLE" | "NONSPLITABLE")

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

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

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

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

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

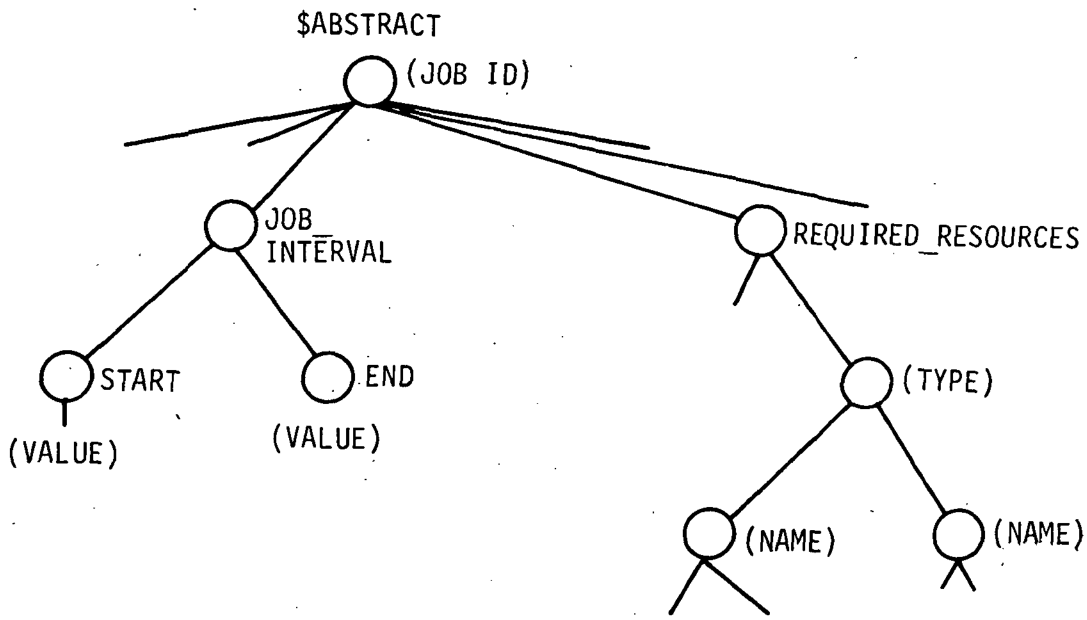
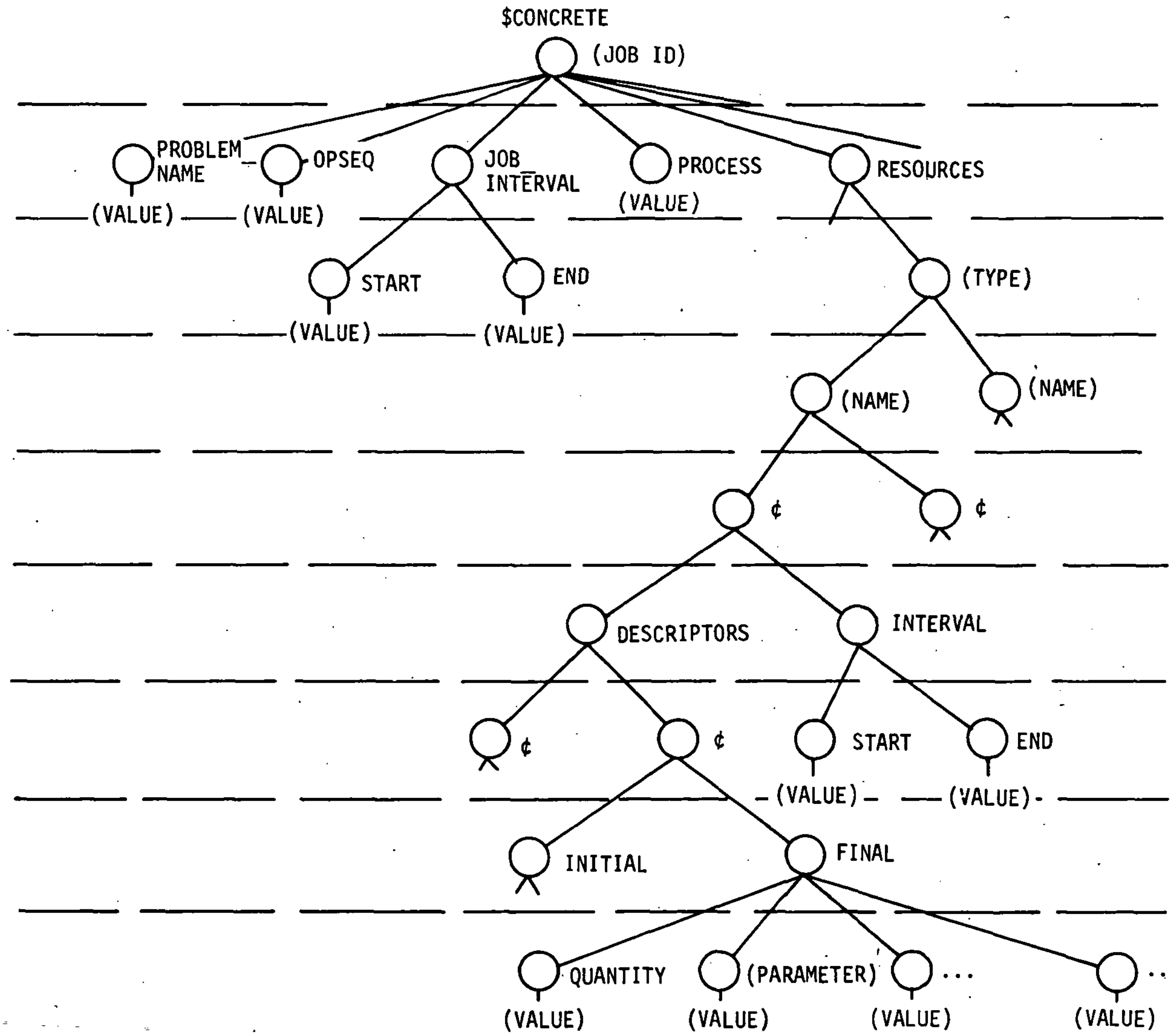


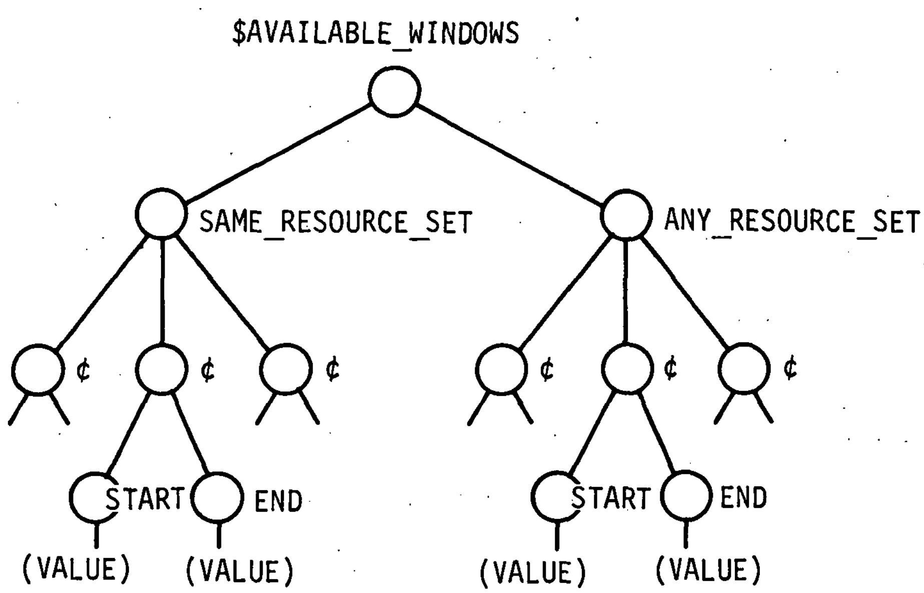
Fig. 2.4.12-1
Minimum Required Input Structures from Standard Data Structures for Module: NEXTSET

OUTPUT DATA STRUCTURE



2.4.12.4 Module Output

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



2.4.13 RESOURCE_PROFILE

2.4.13.1 Purpose and Scope

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

2.4.13.2 Modules Called

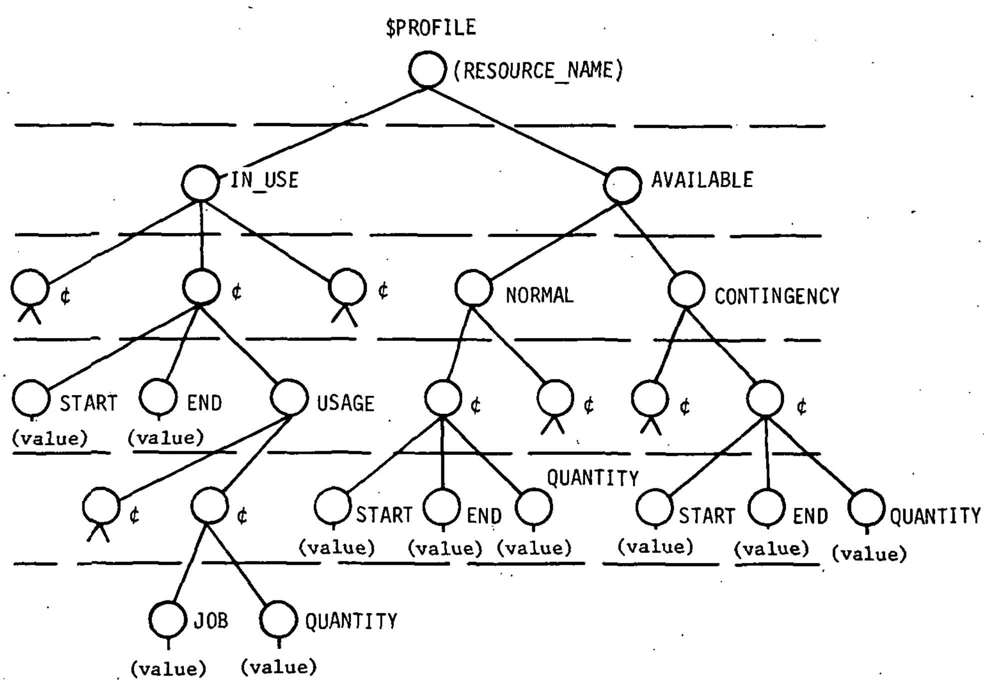
None.

2.4.13.3 Module Input

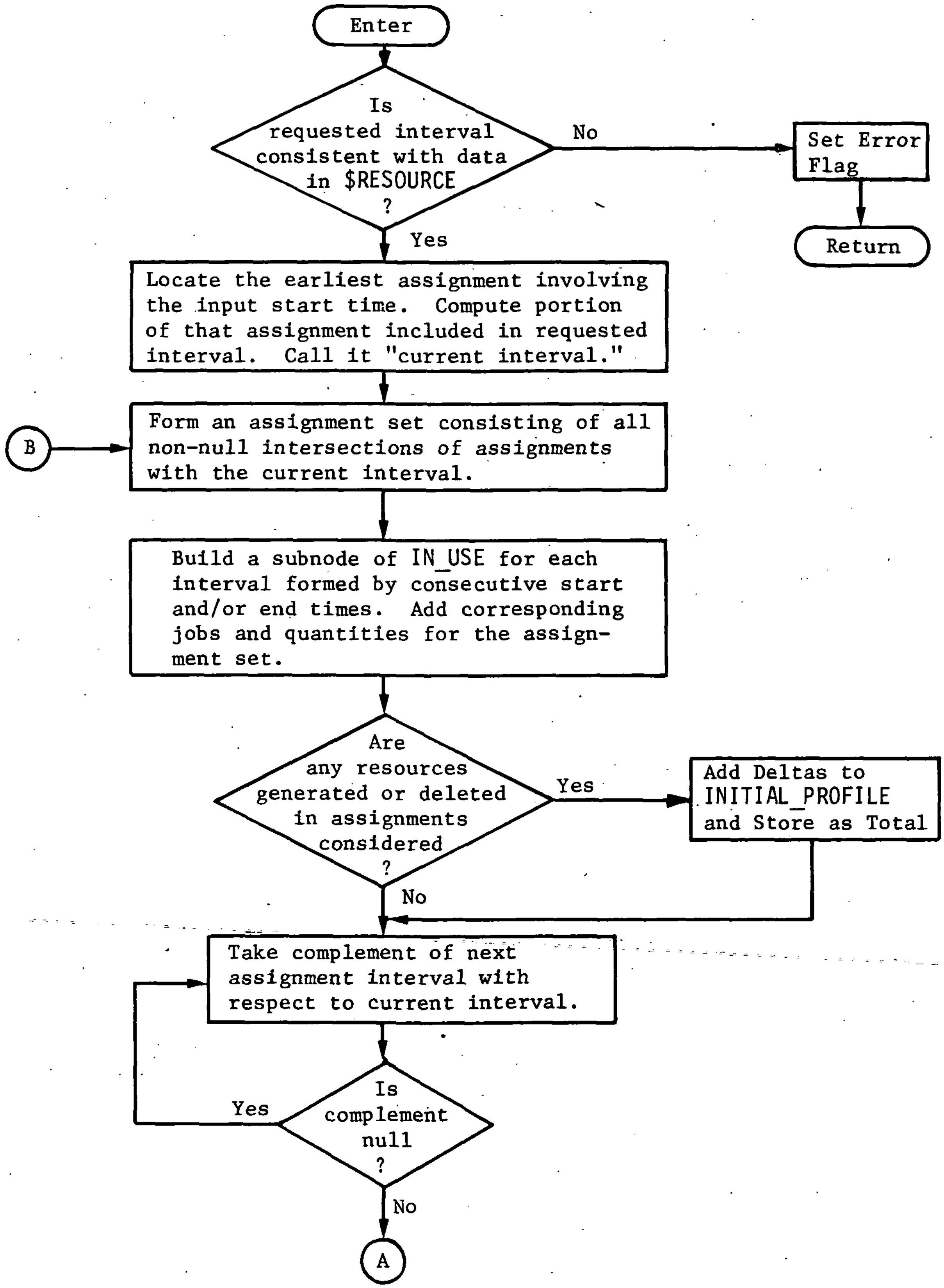
The input to this module will consist of the pooled resource type and name whose profile is to be generated, the time interval for which the profile is to be generated, and the $RESOURCE tree.

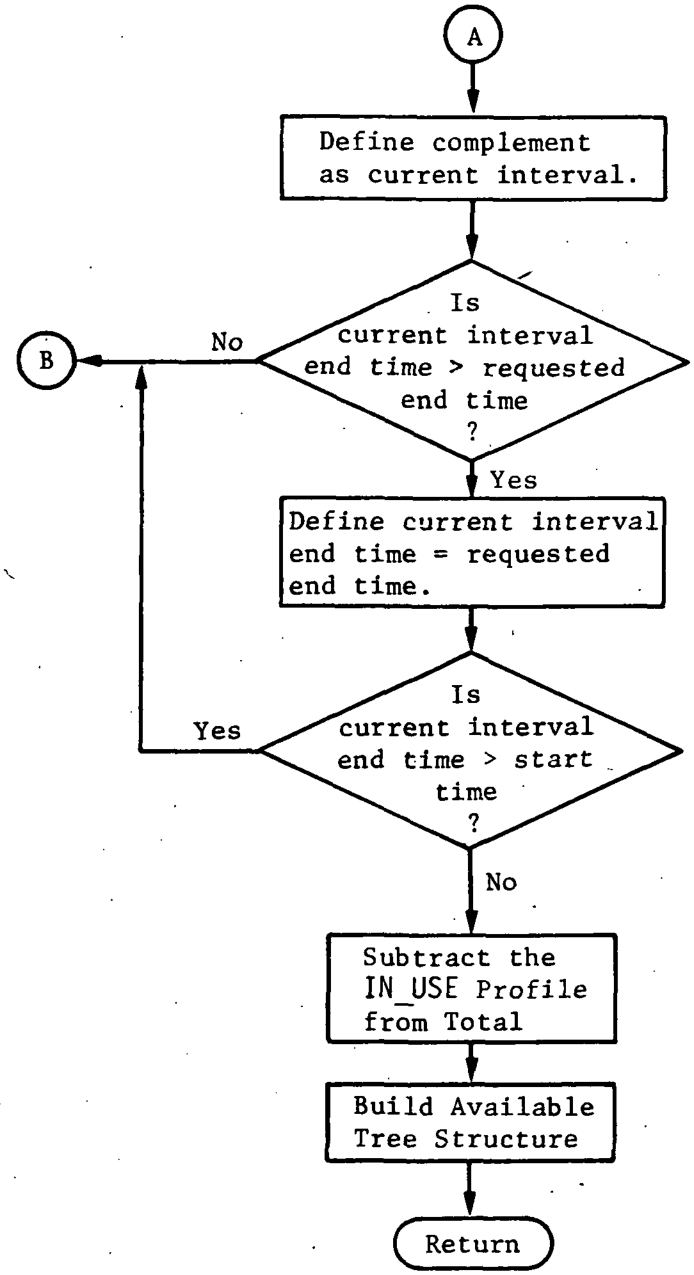
2.4.13.4 Module Output

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

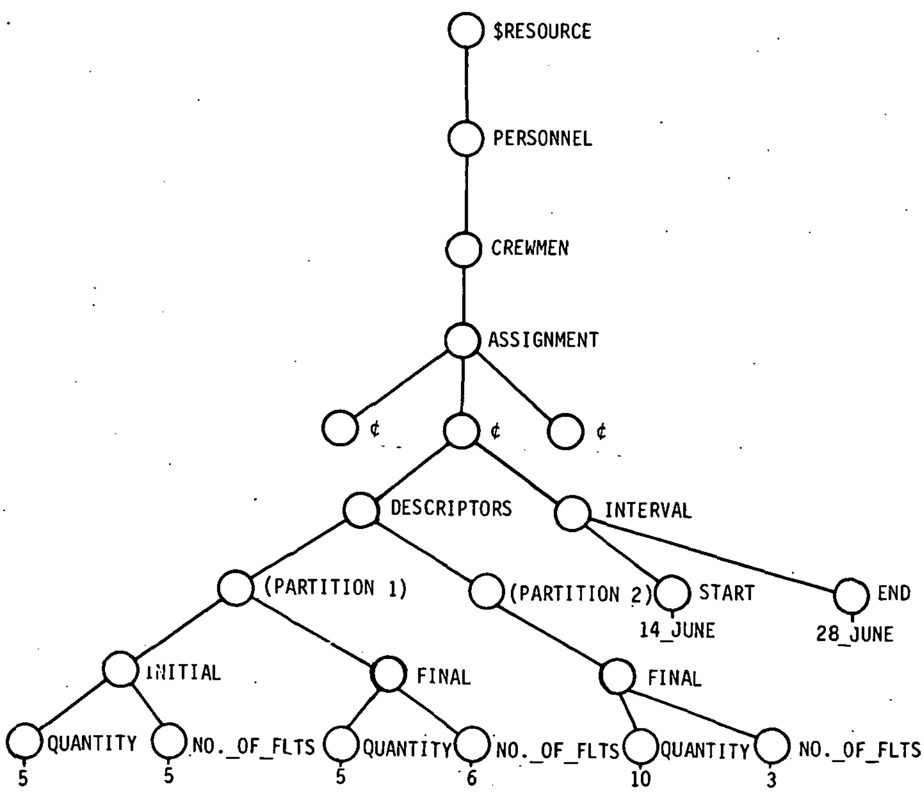


2.4.13.5 Functional Block Diagram



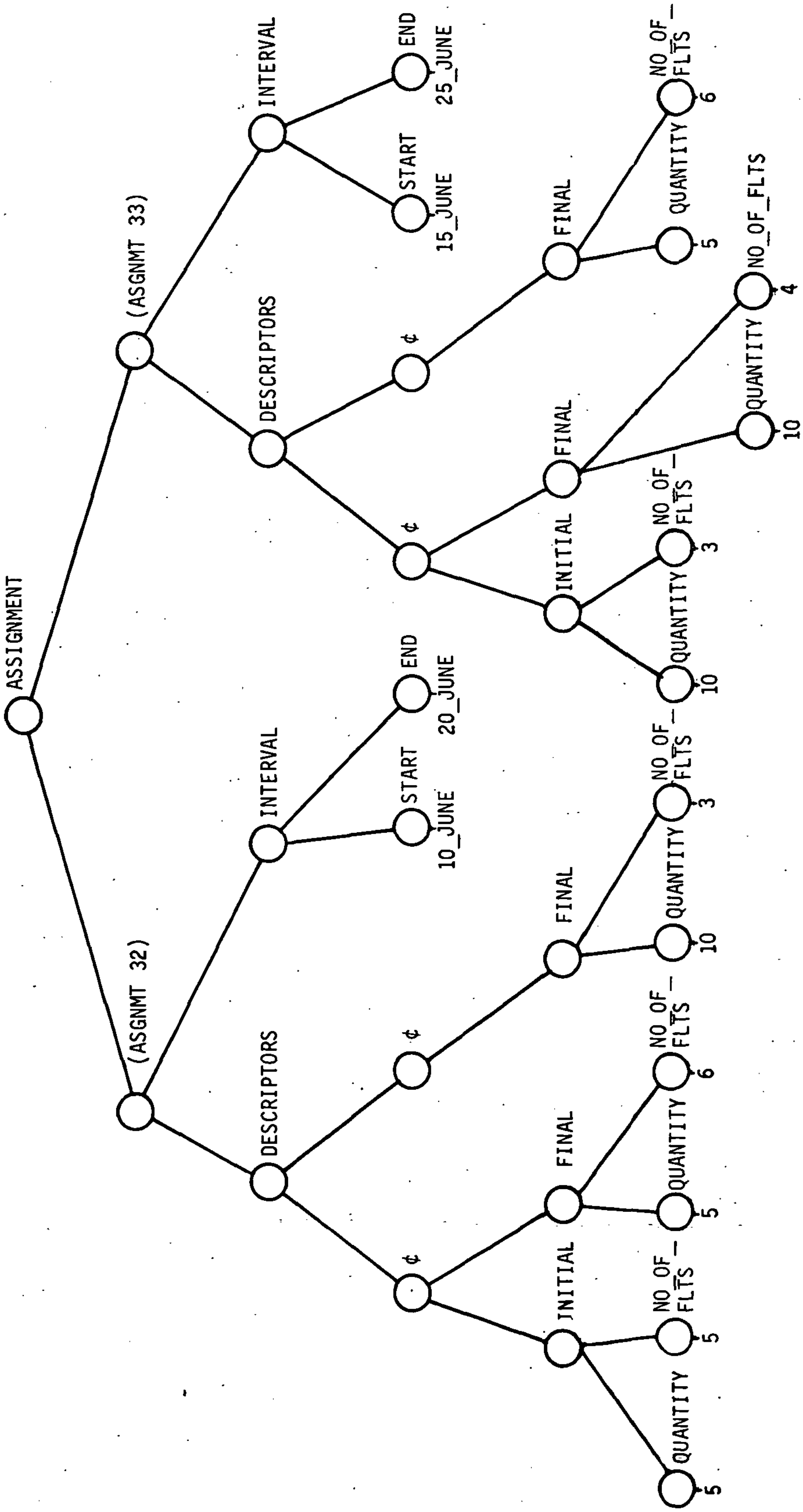

A
Define complement as current interval.
Is current interval end time > requested end time ?
No
B
Yes
Define current interval end time = requested end time.
Is current interval end time > start time ?
Yes
No
Subtract the IN_USE Profile from Total
Build Available Tree Structure
Return

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



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

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



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

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

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

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

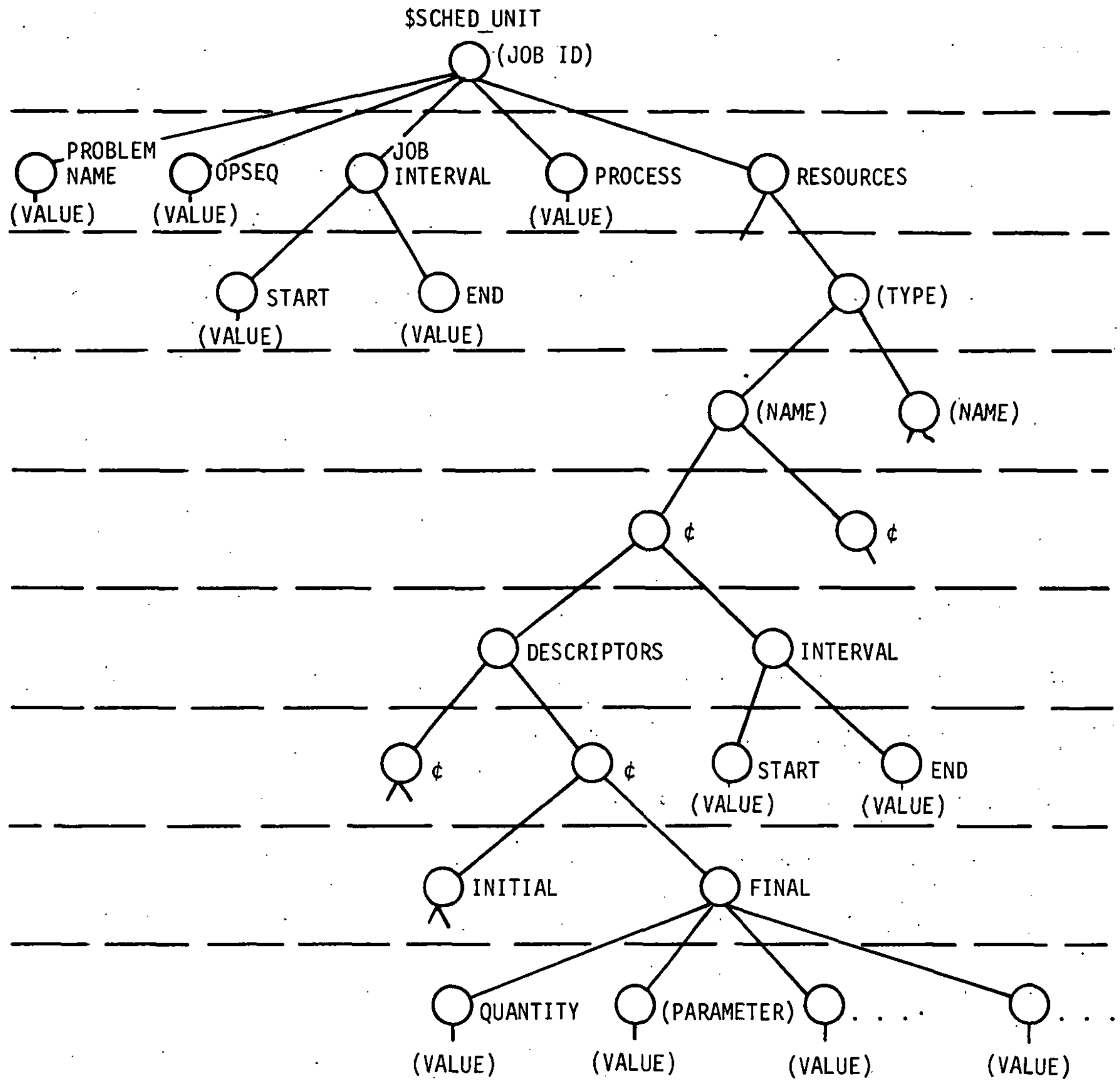
2.4.14.2 Modules Called

None

2.4.14.3 Module Input

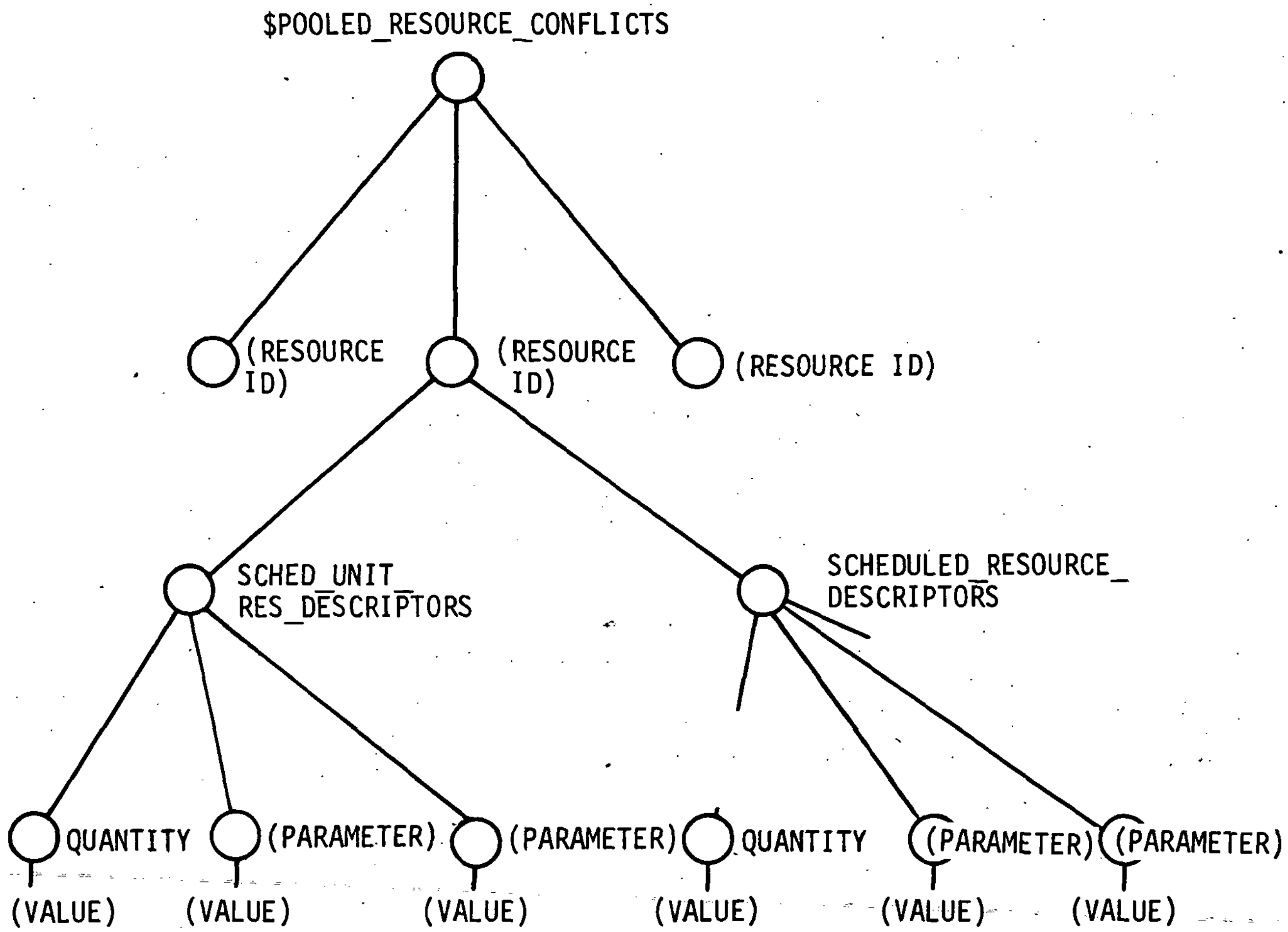
This module is called with two arguments: $RESOURCE and $SCHED_UNIT. $RESOURCE has the general structure given in paragraph 2.4.14.1; $SCHED_UNIT has the general structure of a schedule unit shown in the following illustration.

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

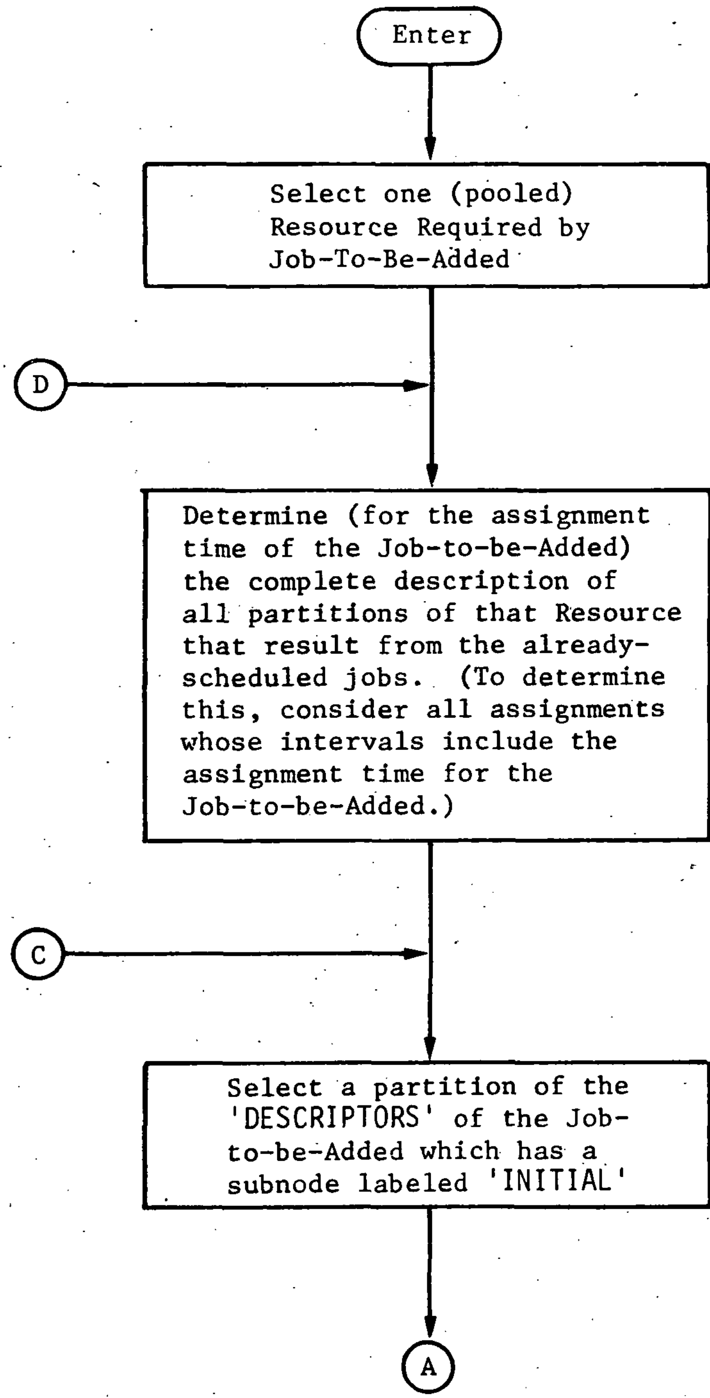

$SCHED_UNIT
(JOB ID)
PROBLEM NAME
(VALUE)
OPSEQ
(VALUE)
JOB INTERVAL
PROCESS
(VALUE)
RESOURCES
START
(VALUE)
END
(VALUE)
(TYPE)
(NAME)
(NAME)
¢
¢
DESCRIPTORS
INTERVAL
¢
¢
START
(VALUE)
END
(VALUE)
INITIAL
FINAL
QUANTITY
(VALUE)
(PARAMETER)
(VALUE)
. . . .
(VALUE)
. . .
(VALUE)

2.4.14.4 Module Output

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



2.4.14.5 Functional Block Diagram



2.4.15 DESCRIPTOR_PROFILE

2.4.15.1 Purpose and Scope

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

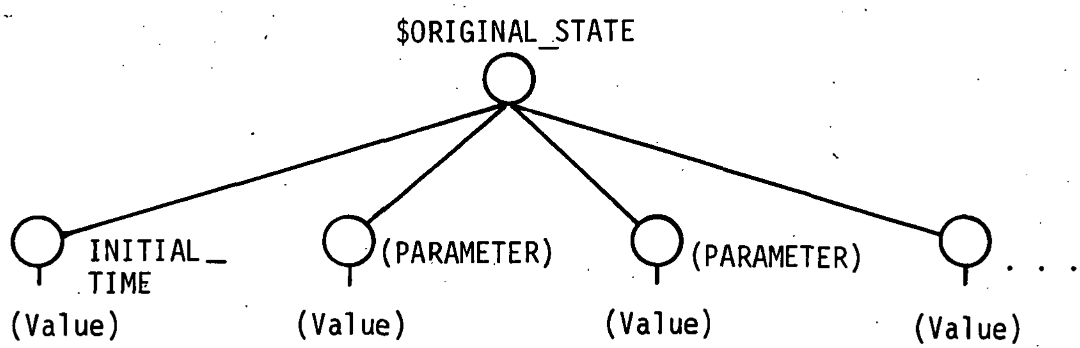
Any number of descriptive parameters may have been used in the resource assignments, but any one parameter will be assumed to contain only mutually exclusive values. For example, if the descriptive parameter, LOCATION is specified, values of DENVER, DALLAS, or DETROIT are obviously mutually exclusive.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

2.4.15.2 Modules Called

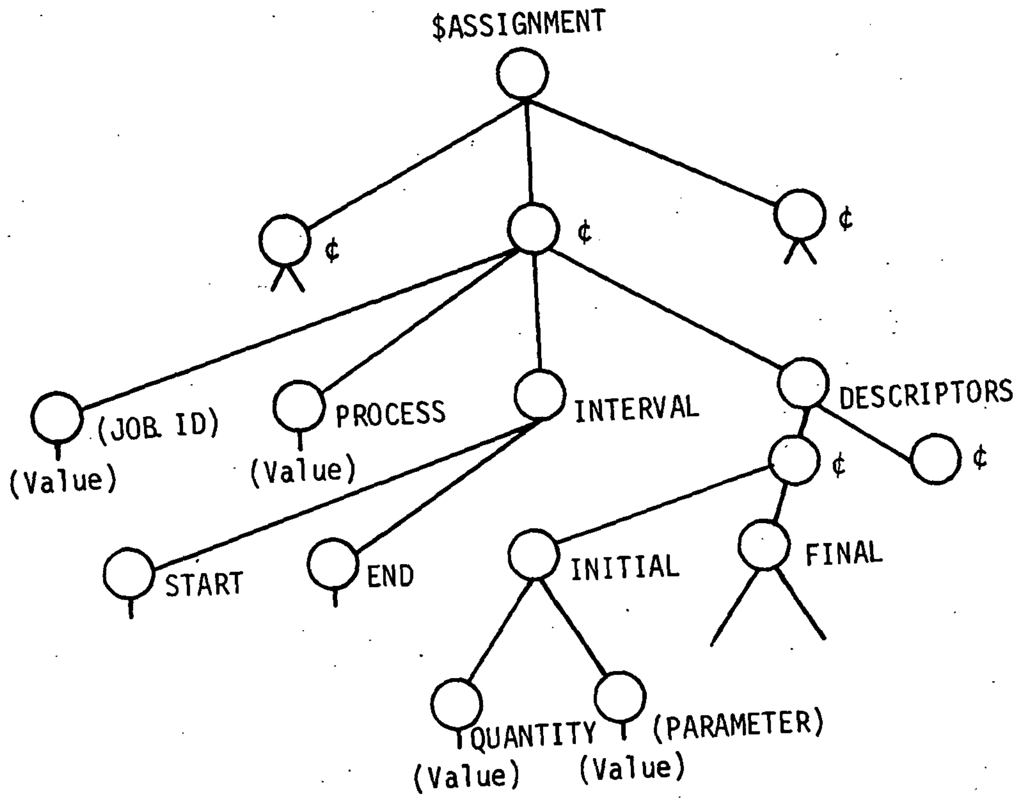
None.

2.4.15.3 Module Input

Input consists of the item-specific resource to be considered, the original values of descriptors to be updated and the corresponding time, the assignments to be considered, and the interval of time that assignments are to be considered. The original descriptors and their values are defined in a tree structure as shown in the sketch. This format corresponds to the first level subnodes of the resource names in the $RESOURCE tree.

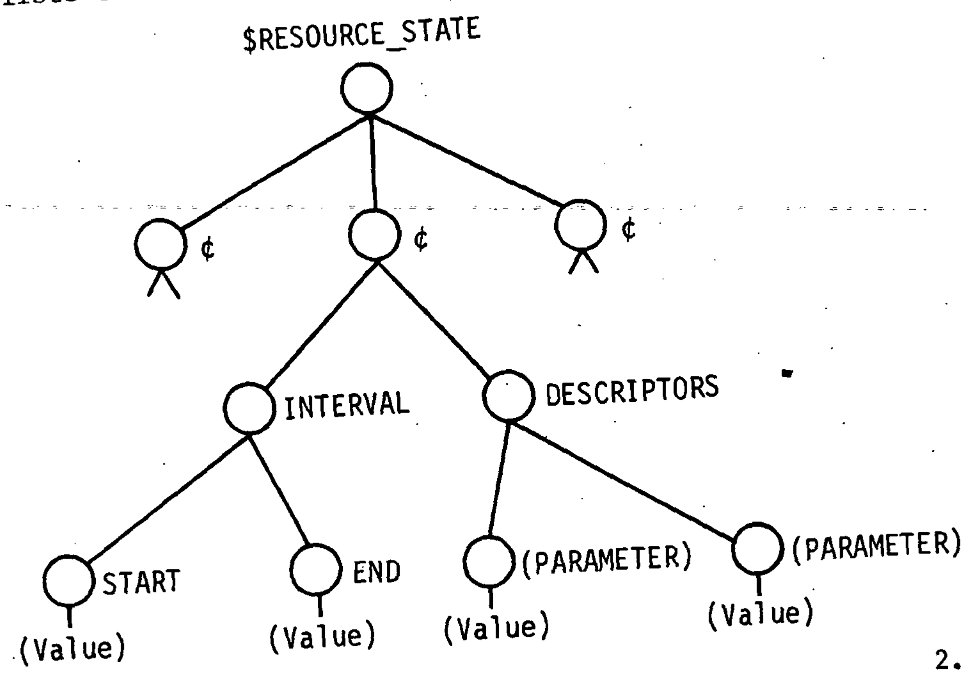


The assignments to be considered would have a format corresponding to the subnode levels of the ASSIGNMENT node in the $RESOURCE tree as illustrated in the sketch. Any nodes, other than the time interval and descriptors (which are required), will be retained for aiding traceability.

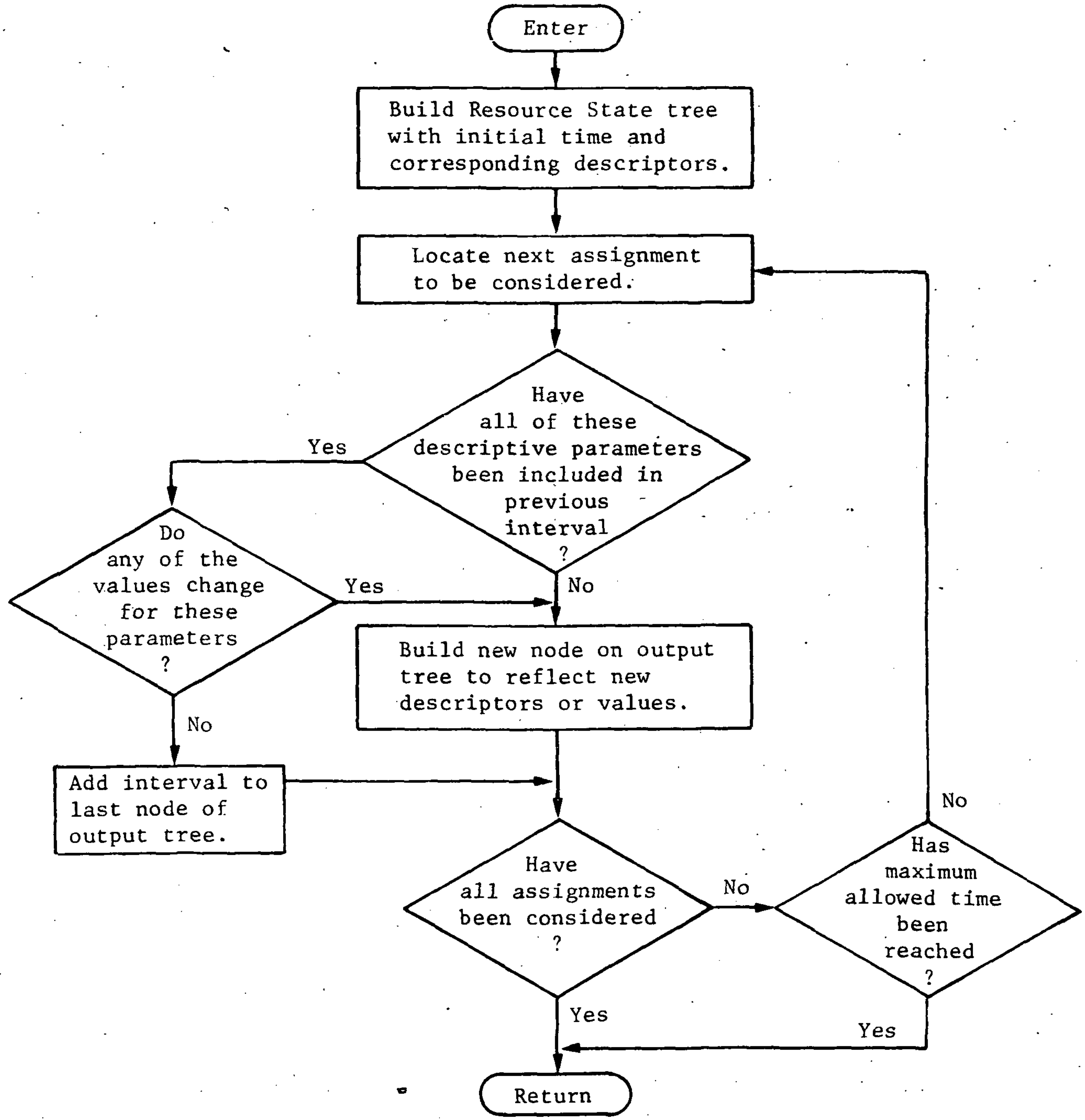


2.4.15.4 Module Output

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



2.4.15.5 Functional Block Diagram



2.4.16 UPDATE_RESOURCE

2.4.16.1 Purpose and Scope

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

2.4.16.2 Modules Called

WRITE_ASSIGNMENT

2.4.16.3 Module Input

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

2.4.16.4 Module Output

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

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

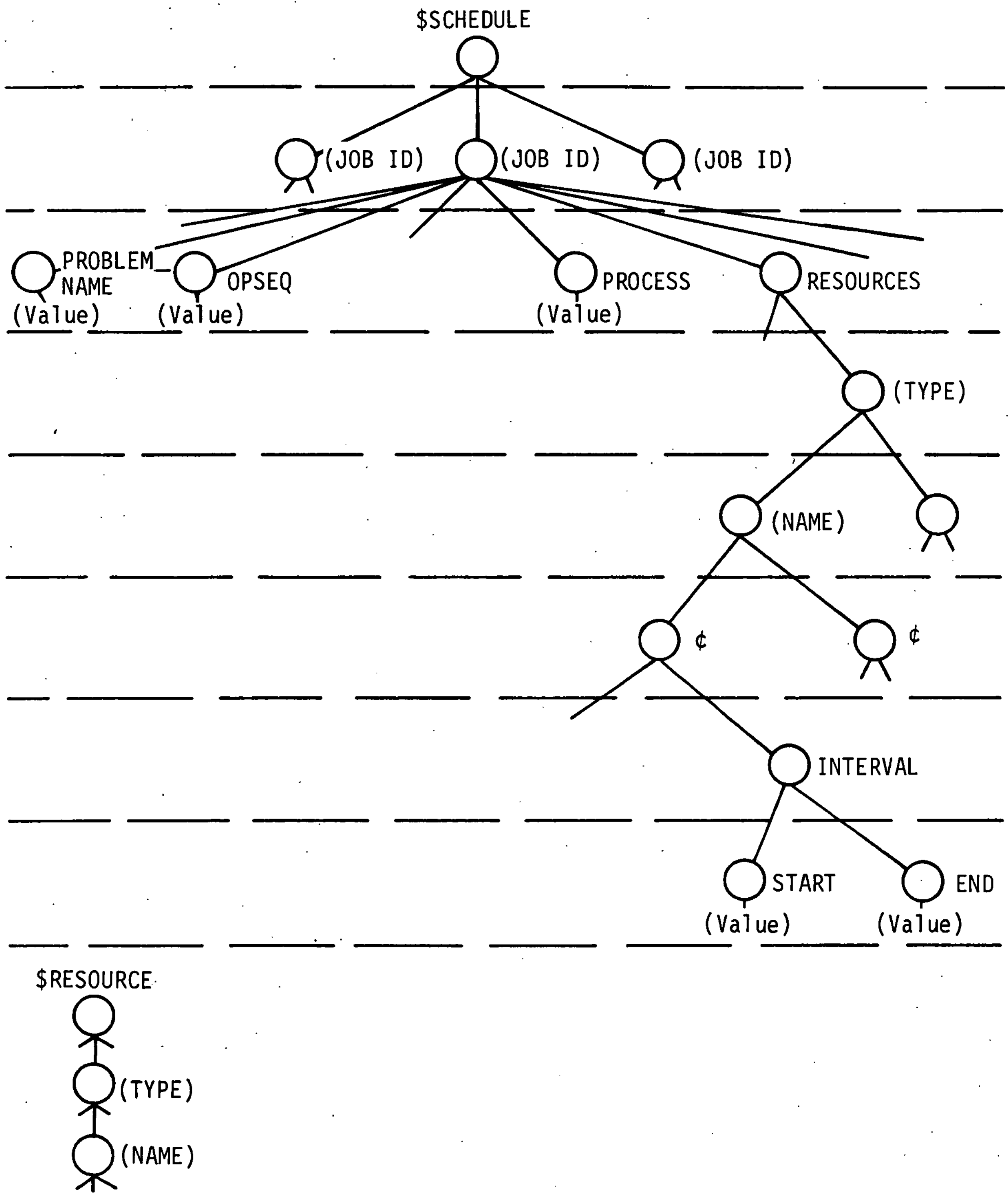
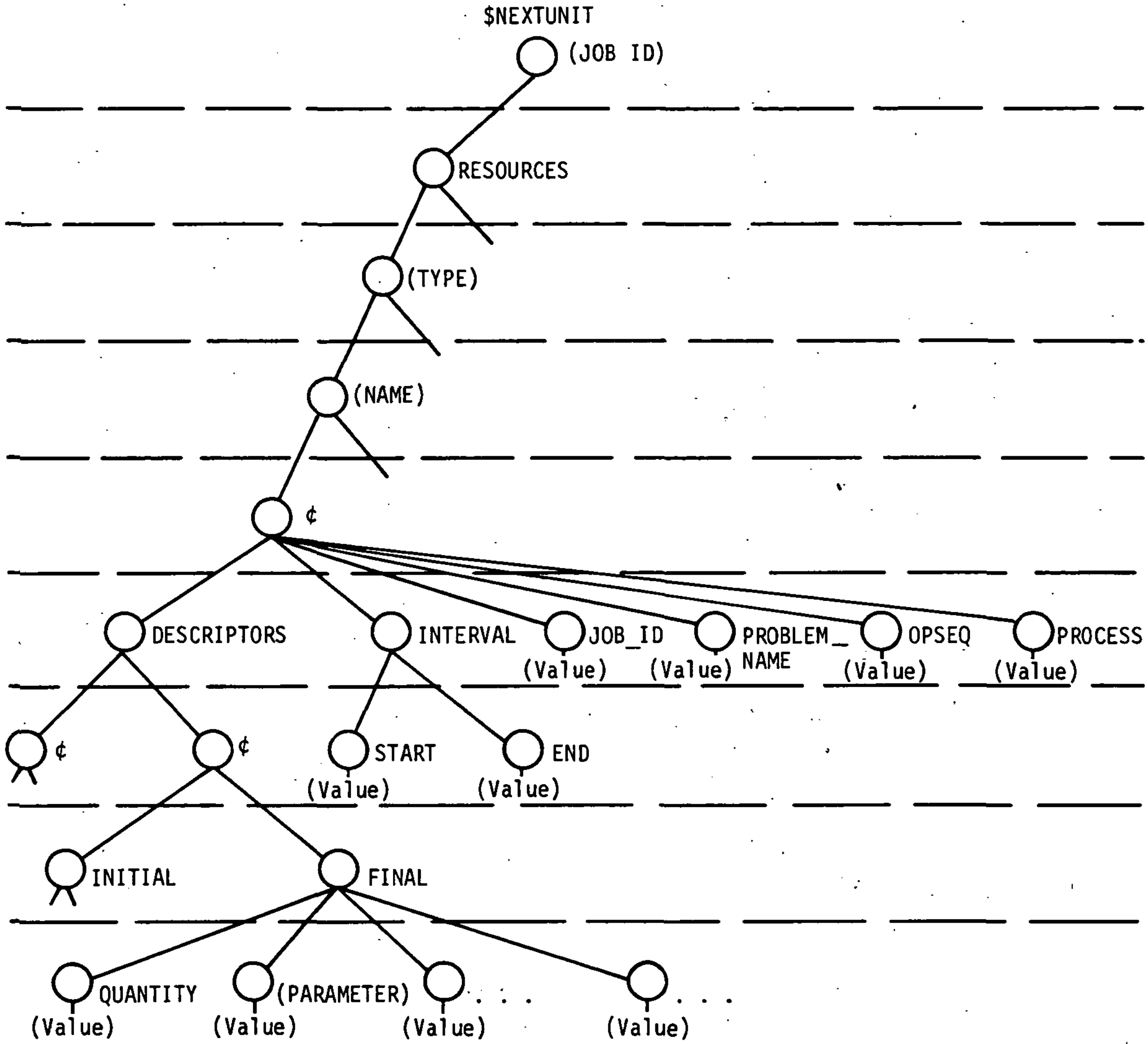
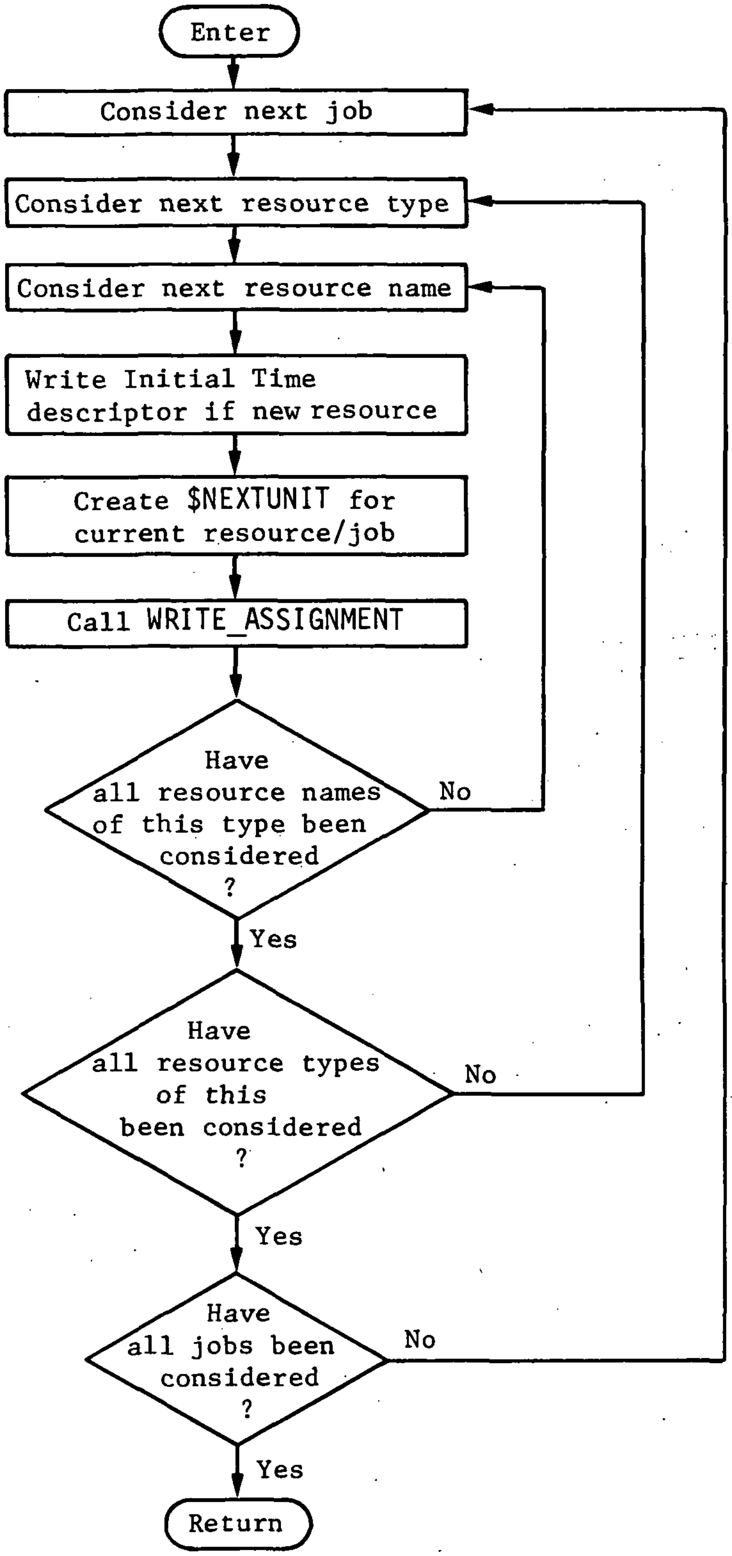


Fig. 2.4.16.4-2
Minimum Required Input Structures from Standard Data Structures for Module: UPDATE_RESOURCE

OUTPUT DATA STRUCTURE



2.4.16.5 Functional Block Diagram



2.4.17 WRITE_ASSIGNMENT

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

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

2.4.17.2 Modules Called

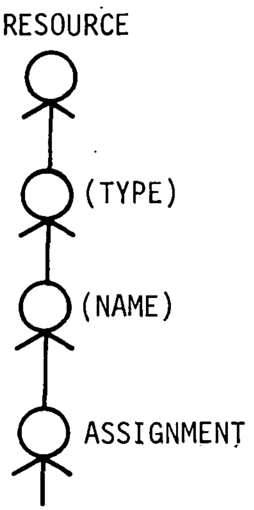
None

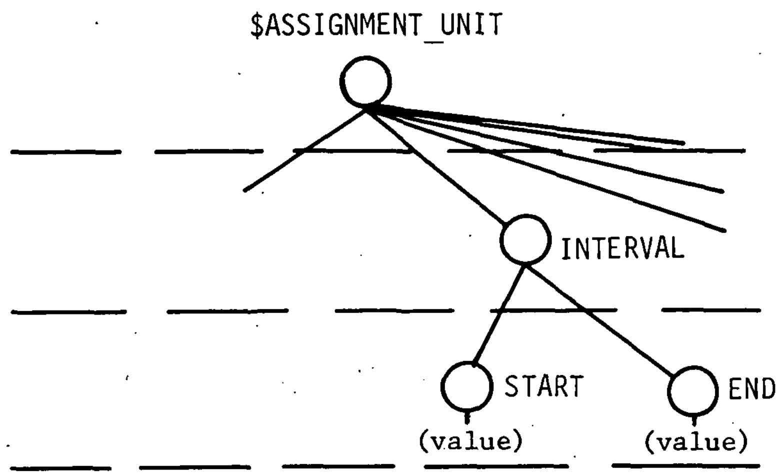
2.4.17.3 Module Input

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

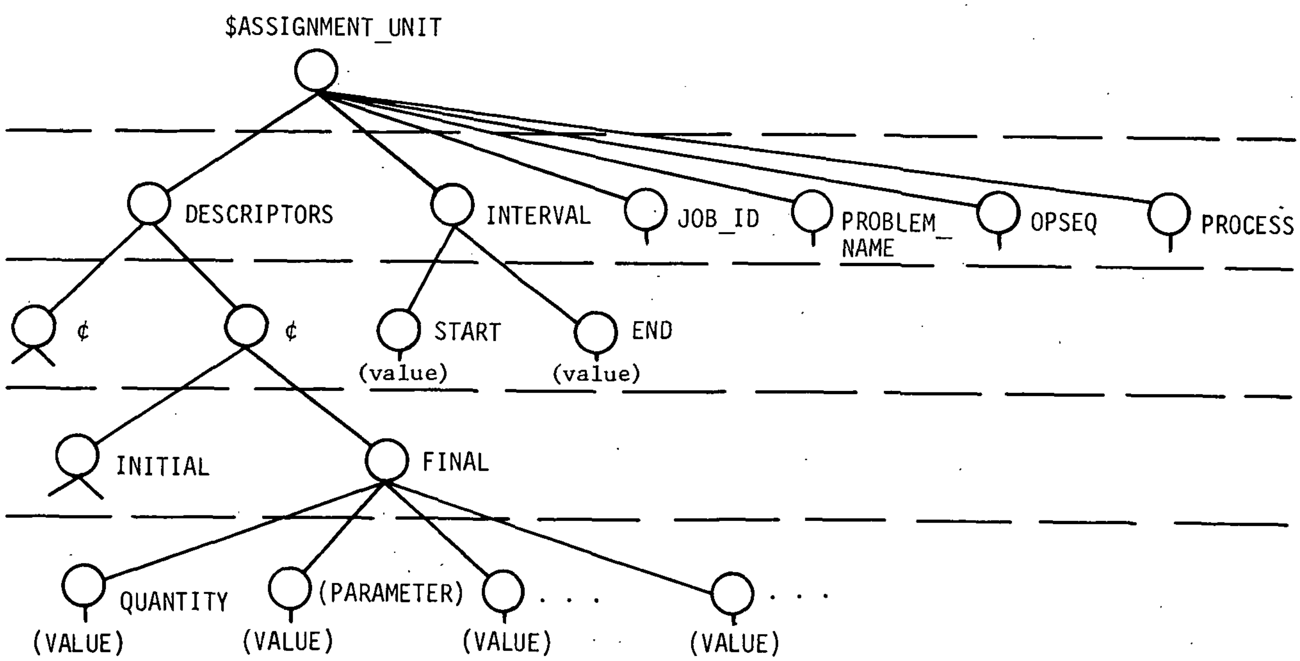
MINIMUM REQUIRED INPUT STRUCTURES FROM STANDARD DATA STRUCTURES
FOR MODULE: WRITE_ASSIGNMENT

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

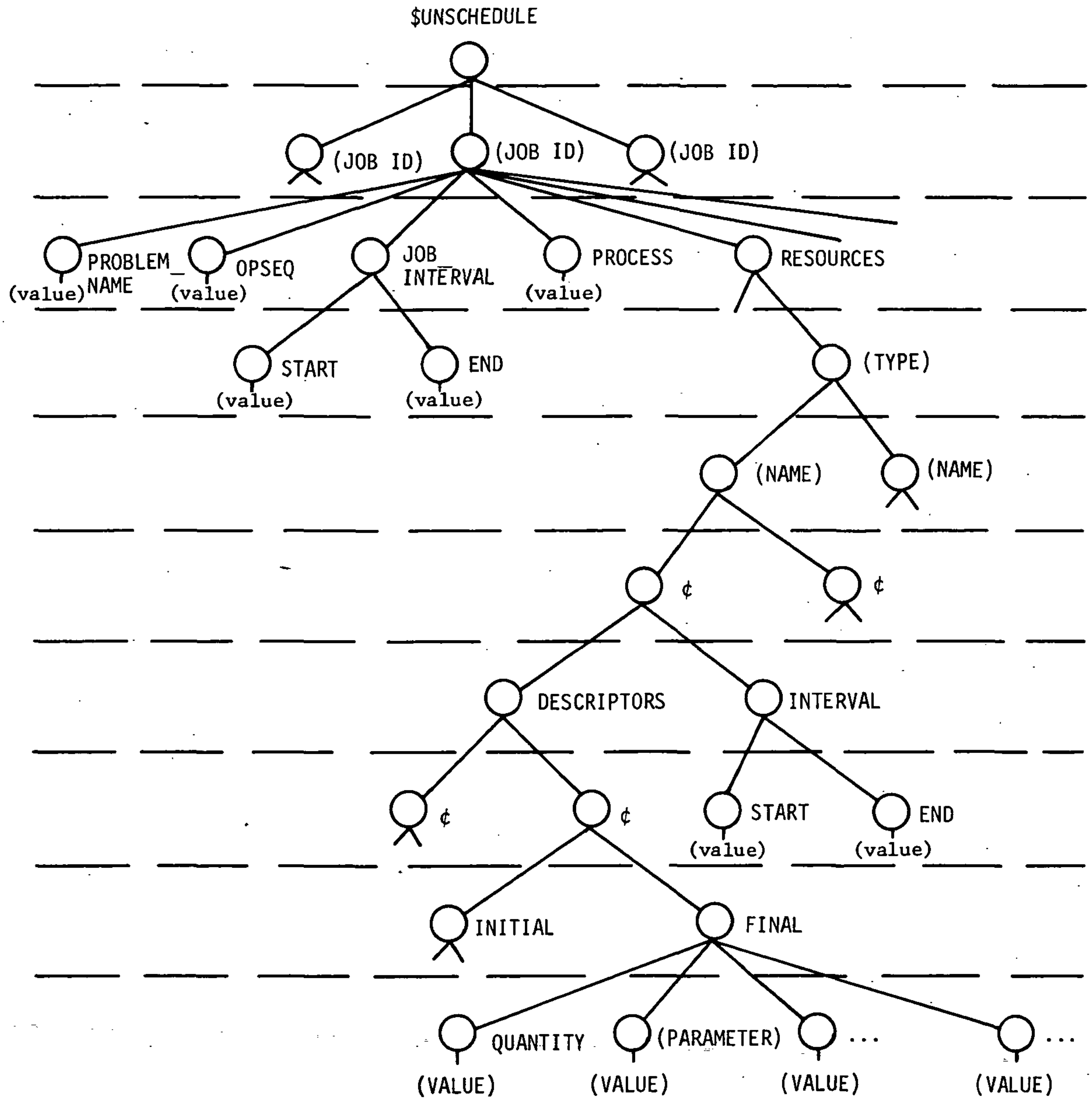




INPUT DATA STRUCTURE



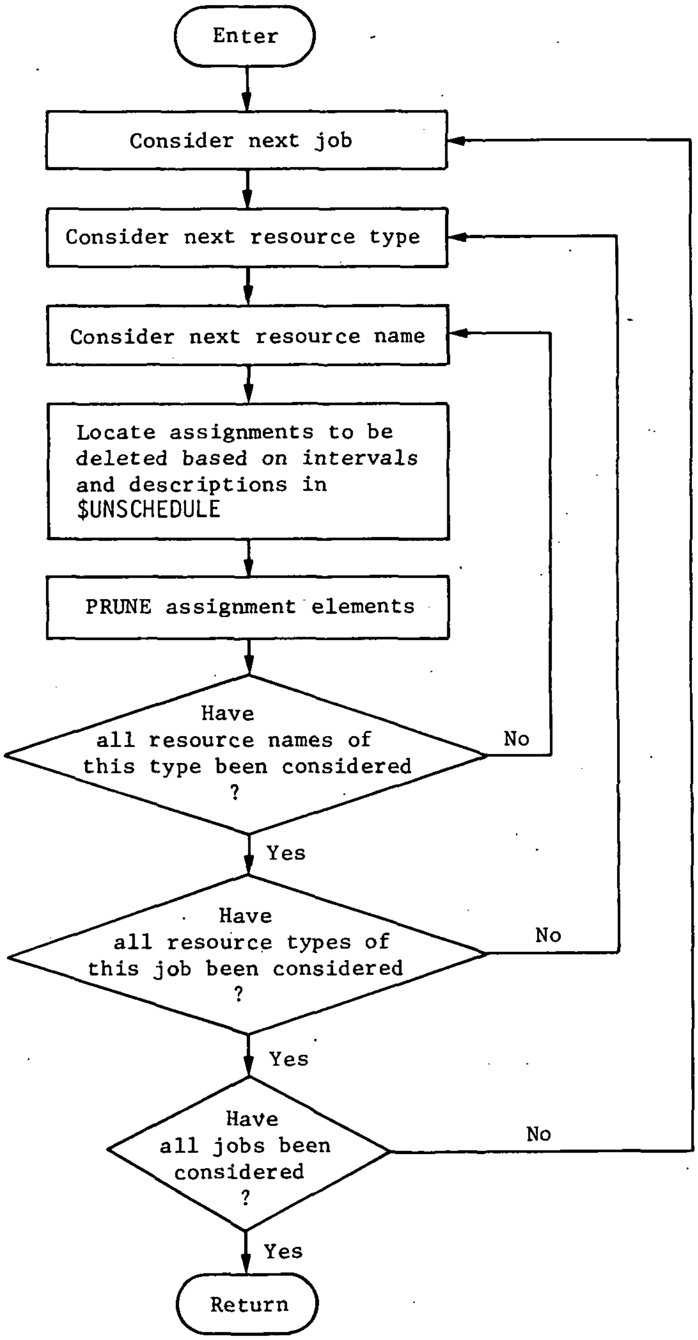
INPUT DATA STRUCTURE



2.4.18.4 Module Output

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

2.4.18.5 Functional Block Diagram



2.4.21 PROJECT_DECOMPOSER

2.4.21.1 Purpose and Scope

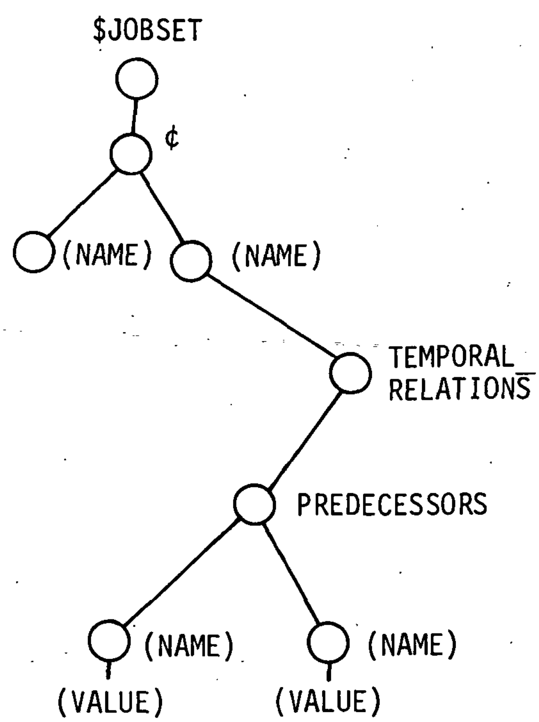
This module will identify all subprojects contained within a specified project. Frequently these subprojects, which are sometimes apparent to the scheduler, are difficult to recognize in the complete network. Identification of the subprojects can significantly reduce the computational effort required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

2.4.21.2 Modules Called

None

2.4.21.3 Module Input

Critical path input data $JOBSET



2.4.21.4 Module Output

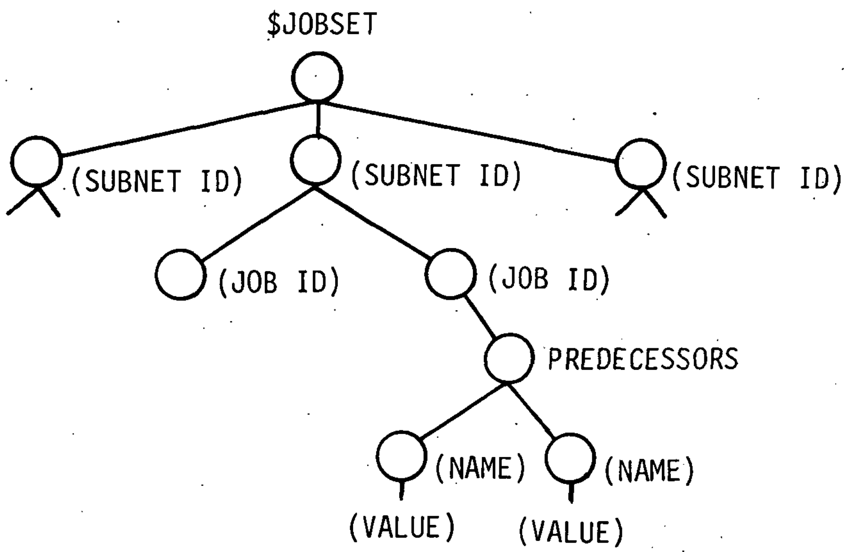
Tree defining the unique subproject decomposition $JOBSET

Subproject identifier (user supplied label)

Member activity or event identifer

Predecessor of activity or event identifer

.
.
.



2.4.21.5 Functional Description

In order to construct an algorithm for identifying "subprojects" this term must be precisely defined. A subproject is a subnetwork containing all the predecessors and successors of its member activities. (These, of course, do not include the events START and FINISH.) Recall that a network for scheduling purposes is a set of activities and events denoted by nodes together with all their

2.4.22 REDUNDANT_PREDECESSOR_CHECKER

2.4.22.1 Purpose and Scope

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

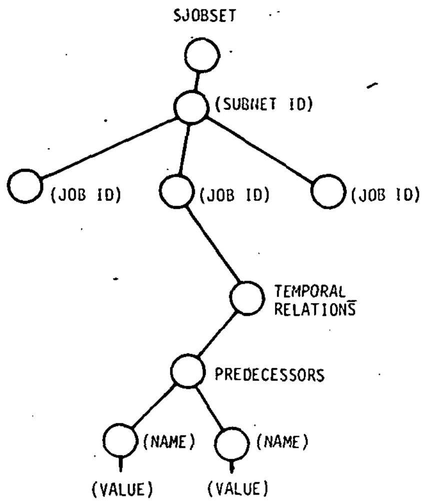
Expressing a project in terms of a collection of nonredundant predecessors serves two useful purpose: (1) it expedites considerably critical path calculations; (2) its facilities comprehension of the precedence relations by representing the project in terms of the most logically concise precedence network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

2.4.22.2 Modules Called

None

2.4.22.3 Module Input

Network definition $JOBSET - including redundant precedessors.



2.4.22.4 Module Output

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

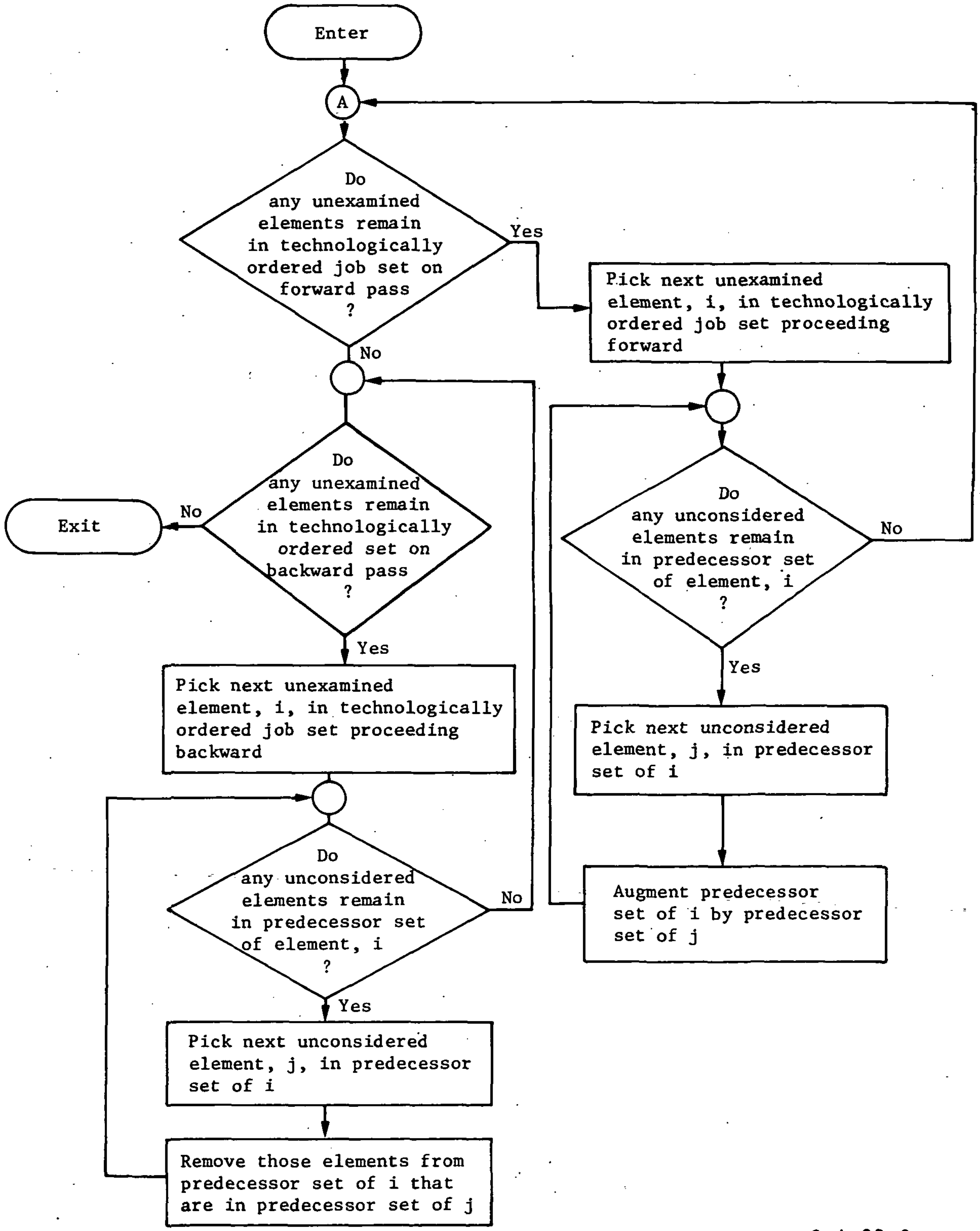
2.4.22.5 Functional Description

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

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

2.4.22.6 Functional Block Diagram

REDUNDANT_PREDECESSOR_CHECKER



2.4.22.7 Typical Application

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

2.4.22.8 References

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

2.4.23 CRITICAL_PATH_CALCULATOR

2.4.23.1 Purpose and Scope

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

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

1) It must consist of a *finite* collection of well-defined activities and events (with no unspecified alternatives) which, when completed, mark the end of the project.
2) The activities may be started and stopped independently of each other within a given sequence. This requirement precludes the analysis of continuous flow processes.
3) The predecessor relationships among the activities and events must not contain cycles; that is there can be no predecessor chains implying that a job precedes itself. Thus a project is nonrepetitive. It is essentially a one-time effort such as a R&D task or a construction project.

2.4.23.2 Modules Called

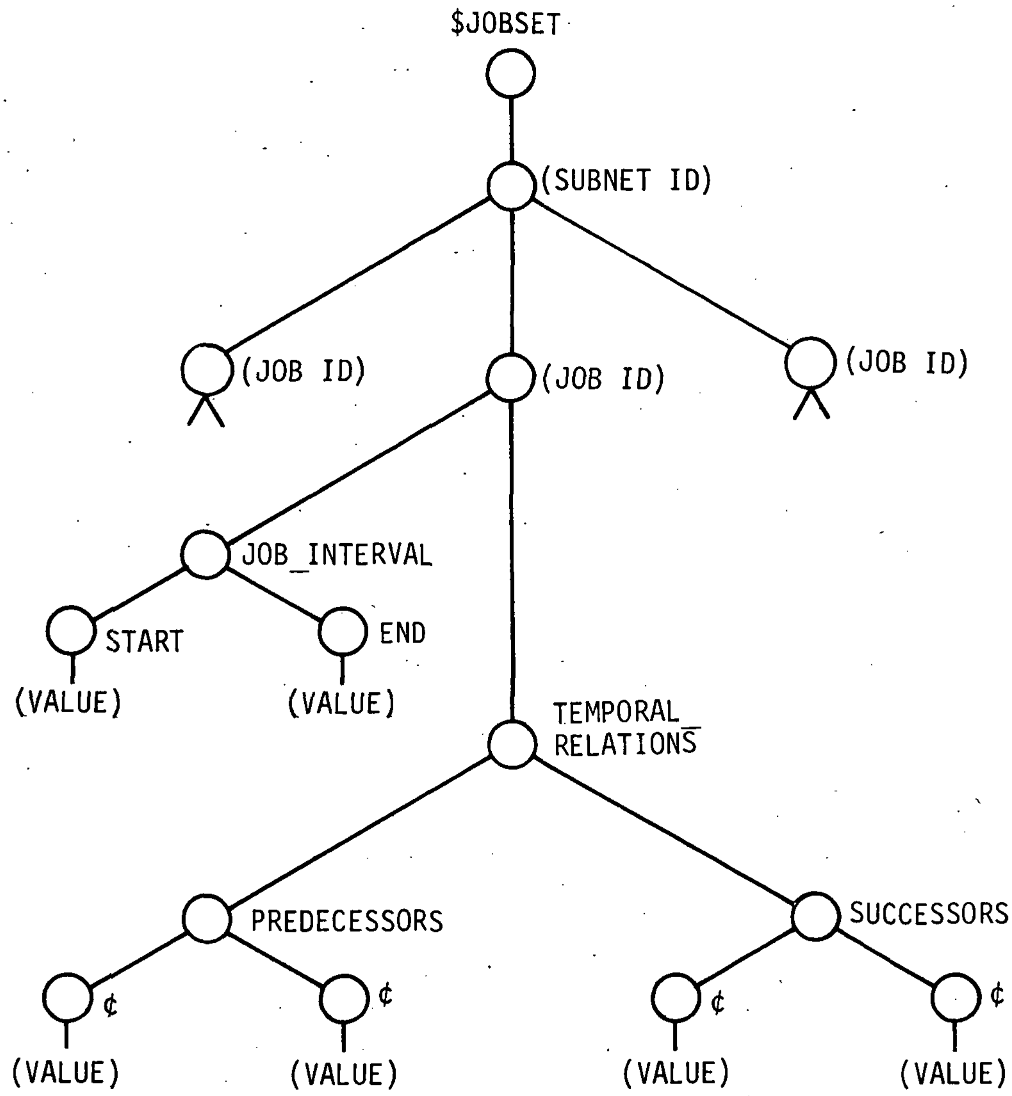
ORDER_BY_PREDECESSORS

FIND_MAXIMUM

FIND_MINIMUM

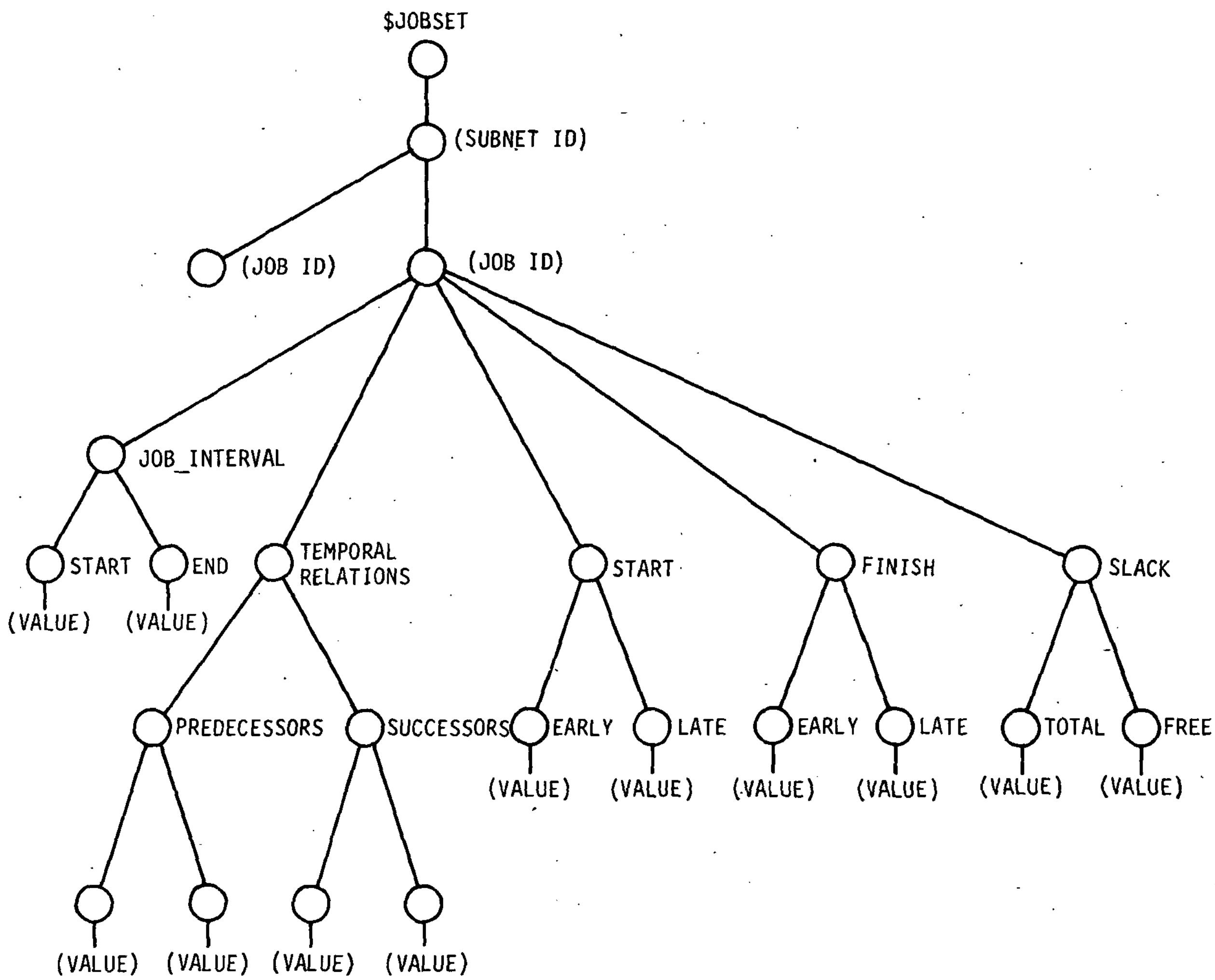
2.4.23.3 Module Input

Critical Path Input Data ($JOBSET)



2.4.23.4 Module Output

Critical Path Output Data ($JOBSET)



2.4.23.5 Functional Description

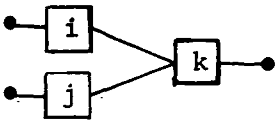
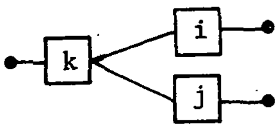
Critical path analysis is a powerful but simple technique for analyzing, planning, scheduling, and controlling complex projects. In essence, the method provides a means of determining (1) which activities are "critical" in their effect upon total project duration, and (2) how to schedule all activities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

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

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

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

Table 2.4.23-1 Basic Precedence Relationship

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

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

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

and the early finish of activity i is clearly

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

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

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

and the late start is

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

For any activity, the quantity

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

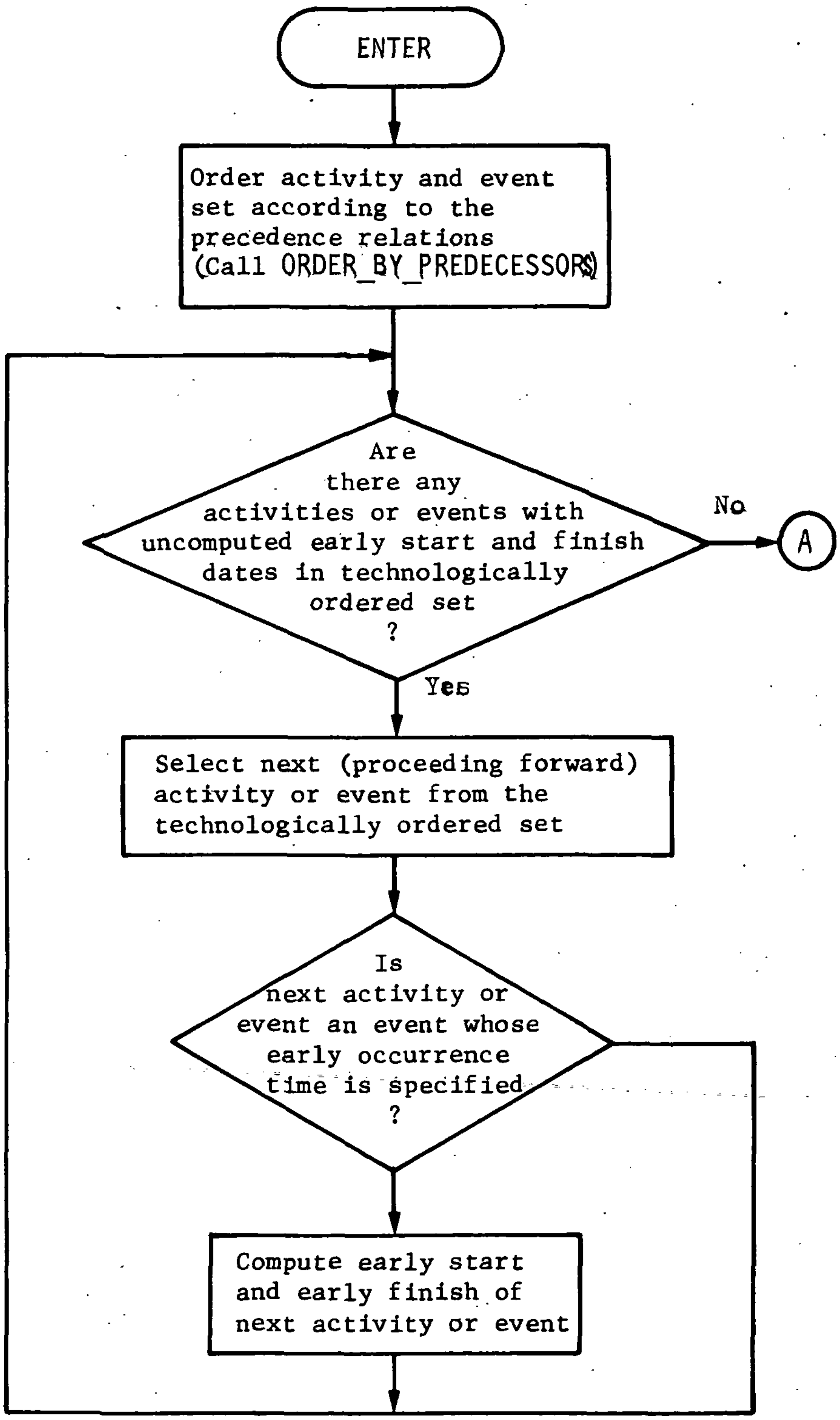
is defined to be the total slack. The set of critical activities is then the subset of activities having minimum total slack.

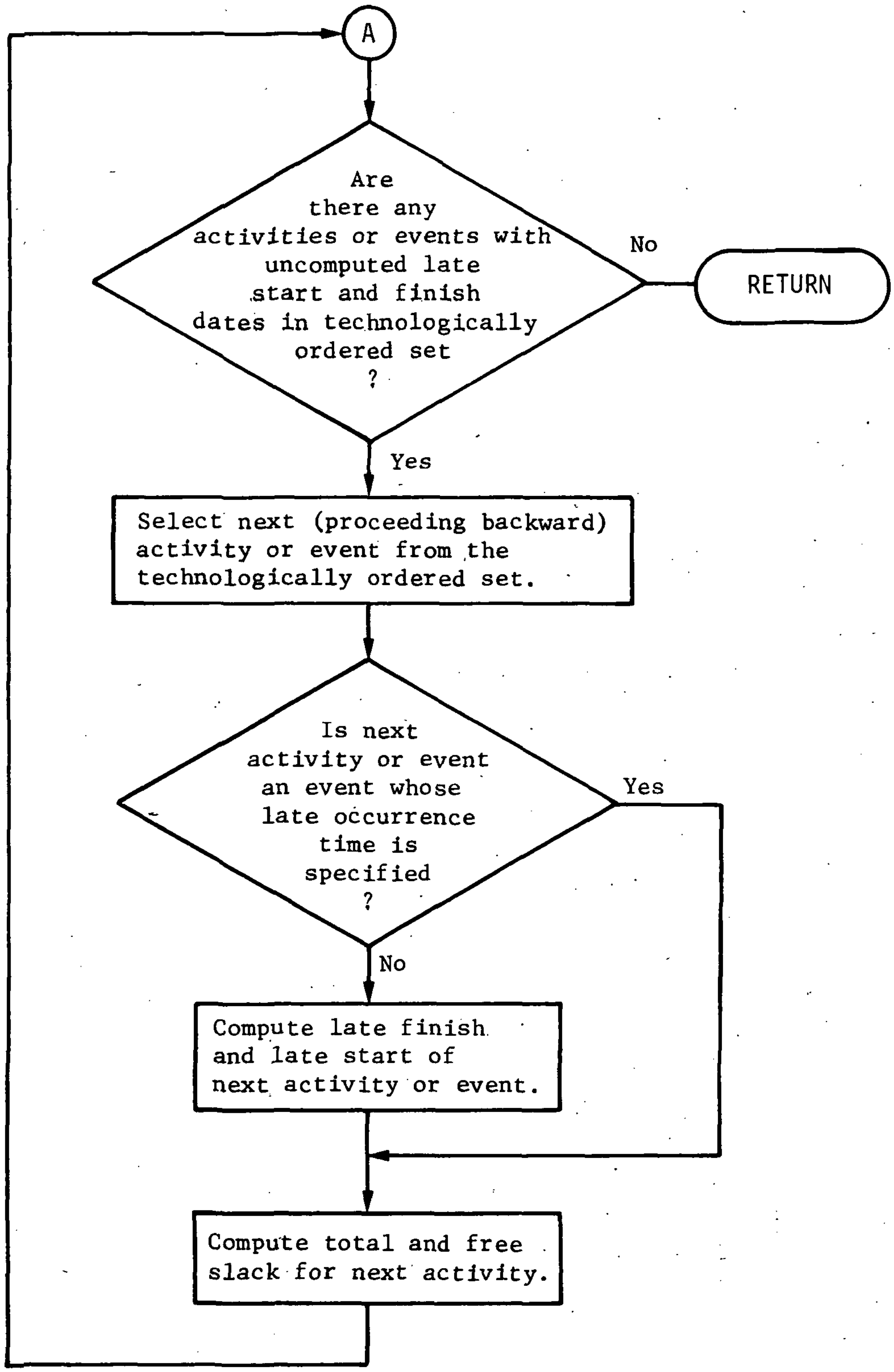
Another useful variable is free slack, S^f. Free slack is defined as the amount by which an activity may be delayed without affecting any other activity. It is computed as

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

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

2.4.23.6 Functional Block Diagram




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

2.4.24 PREDECESSOR_SET_INVERTER

2.4.24.1 Purpose and Scope

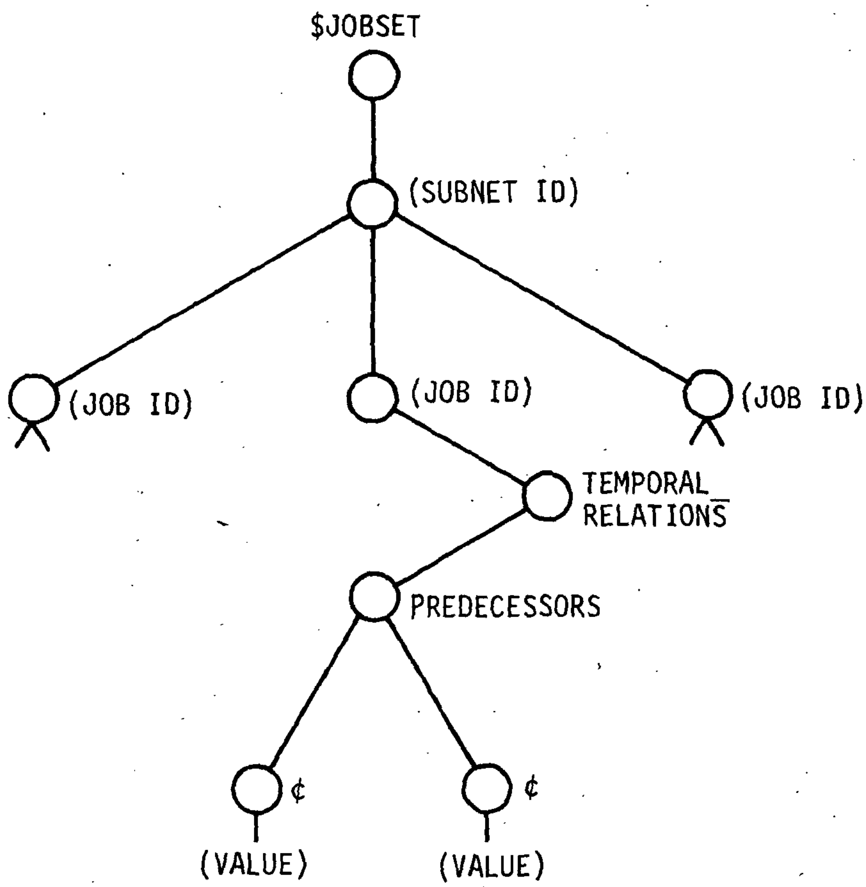
Given a set of activities and their respective predecessor sets, this module will form the respective successor sets. This inversion process is necessary for critical path computation. The project scheduling system assumes throughout that stating precedence relations in terms of predecessor sets is more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

2.4.24.2 Modules Called

None

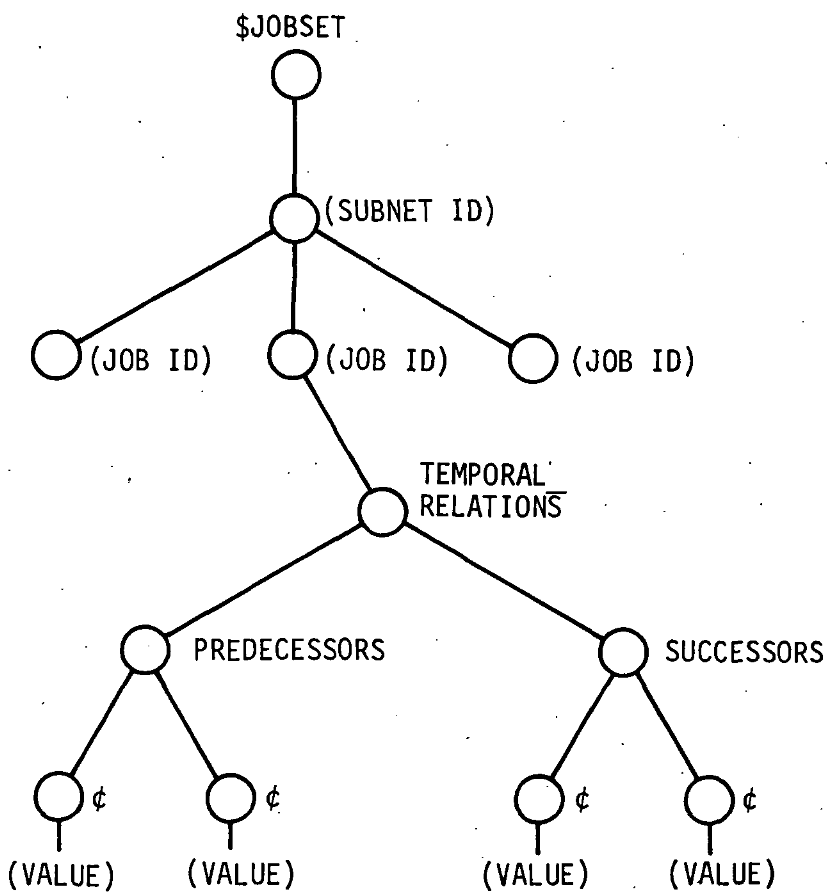
2.4.24.3 Module Input

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



2.4.24.4 Module Output

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

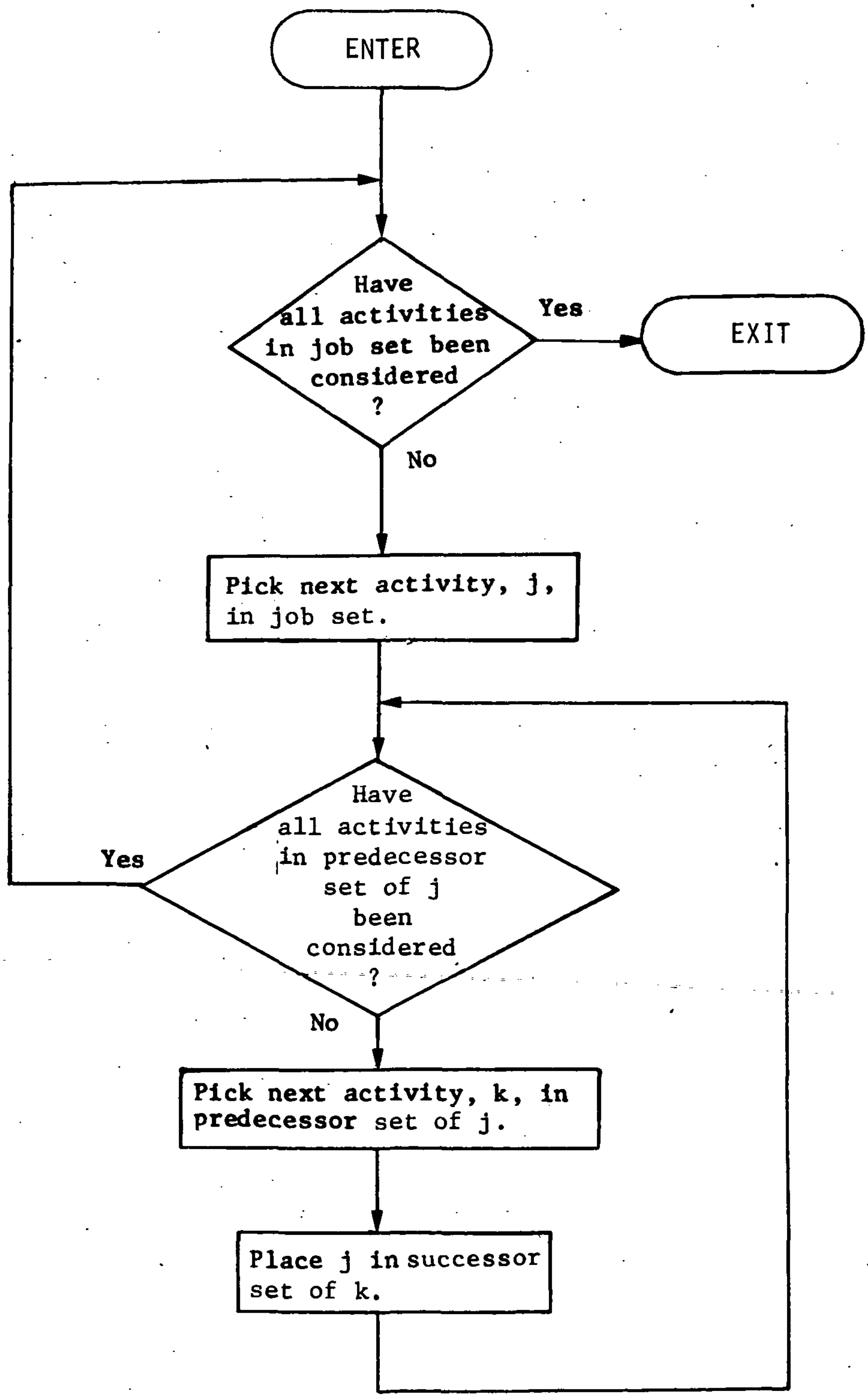


2.4.24.5 Functional Description

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

2.4.24.6 Functional Block Diagram

PREDECESSOR_SET_INVERTER



2.4.24.7 Typical Application

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

matter what its size, can be viewed as one comprehensible summarized network. Without this capability network analysis would be of little value to project scheduling.

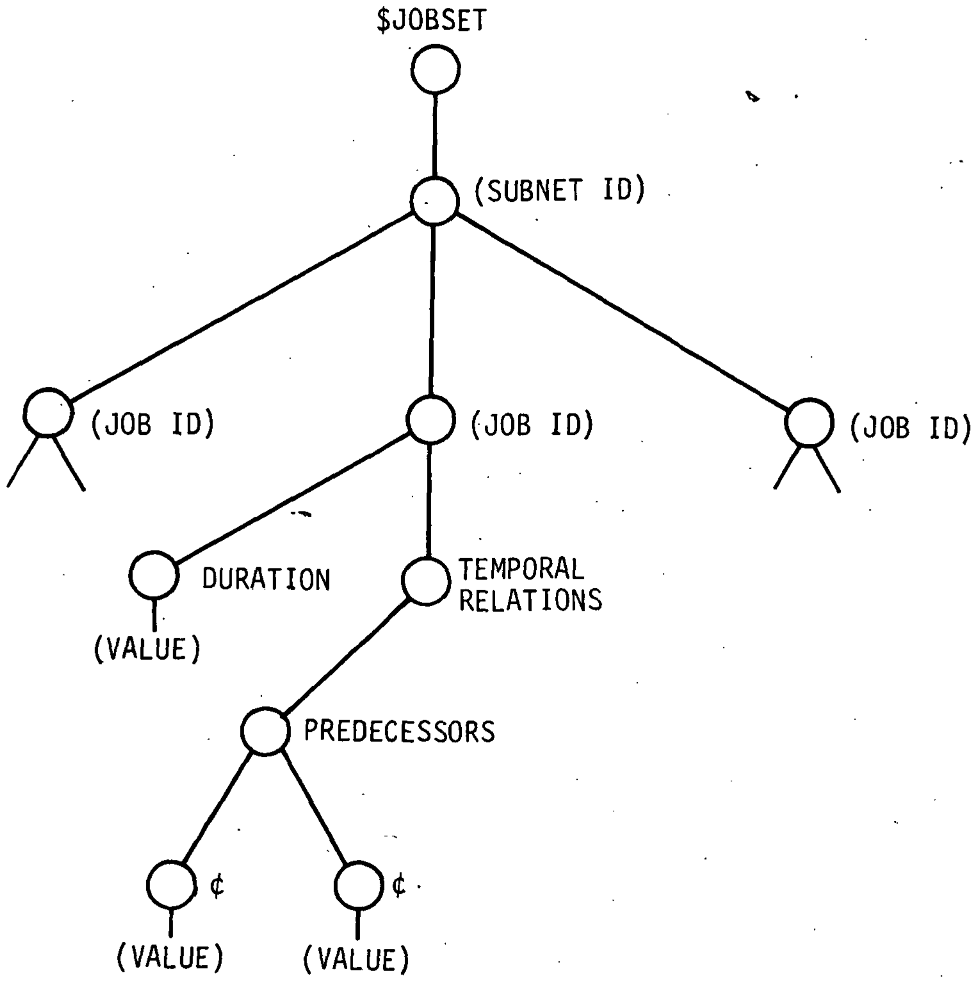
The purpose of this module is then to convert a network, specified in terms of a jobset with its corresponding family of predecessor sets and durations, into a condensed network defined by its event and pseudo-activity set with its corresponding collection of predecessor sets and durations.

2.4.25.2 Modules Called

None

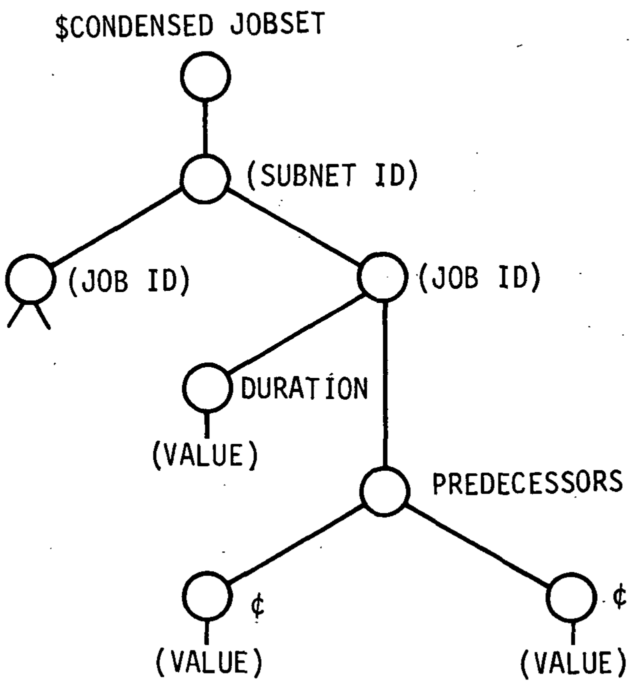
2.4.25.3 Module Input

Critical Path Input Data ($JOBSET)



2.4.25.4 Module Output

Tree Defining the Condensed network

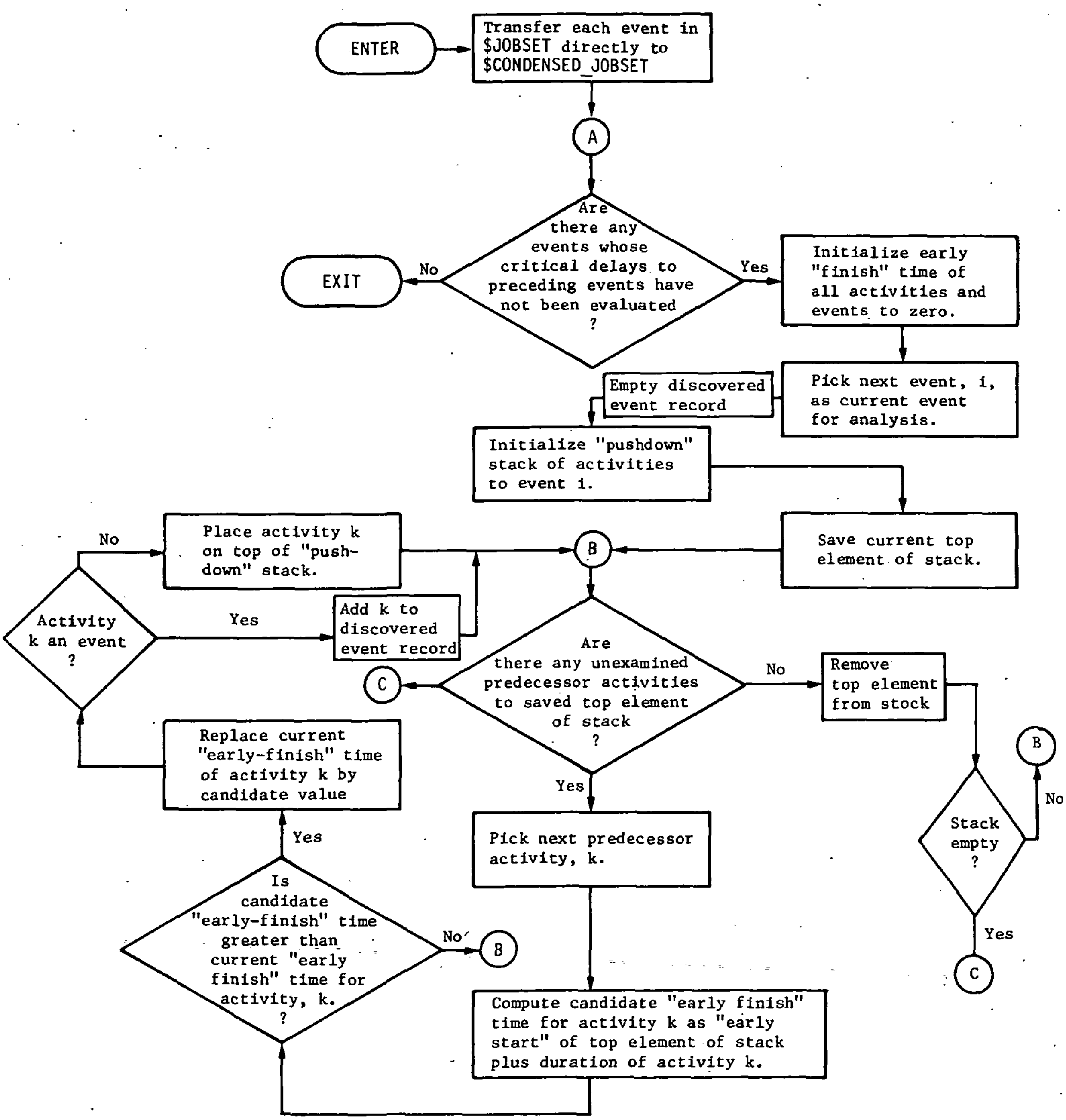


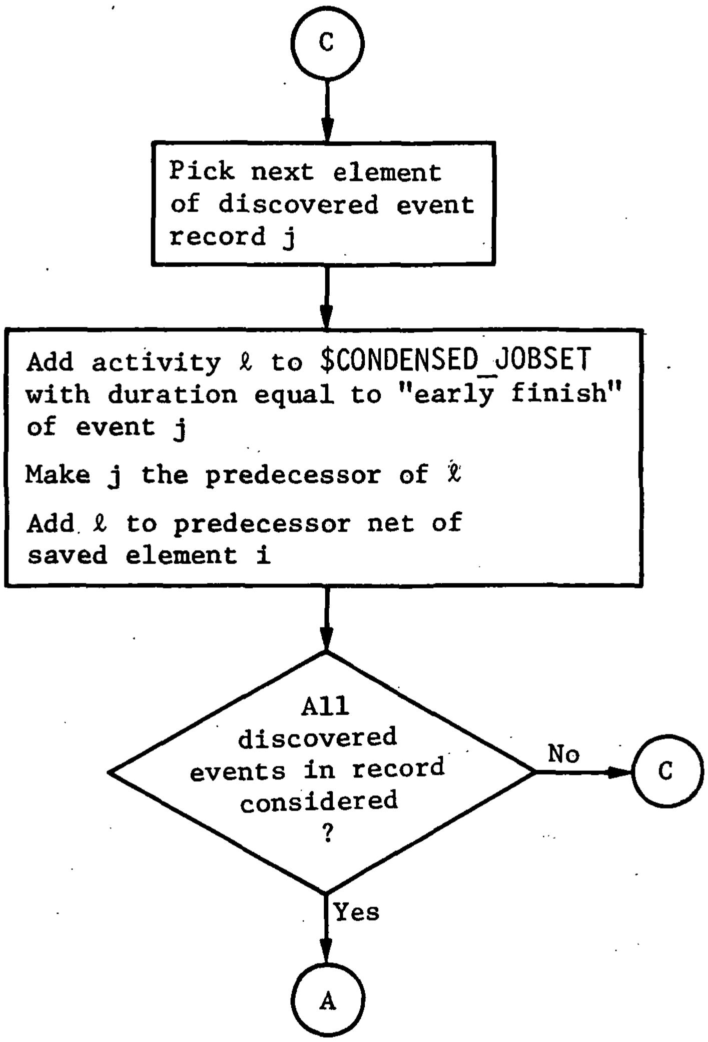
2.4.25.5 Functional Description

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

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

2.4.25.6 Functional Block Diagram




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

2.4.26 CONDENSED_NETWORK_MERGER

2.4.26.1 Purpose and Scope

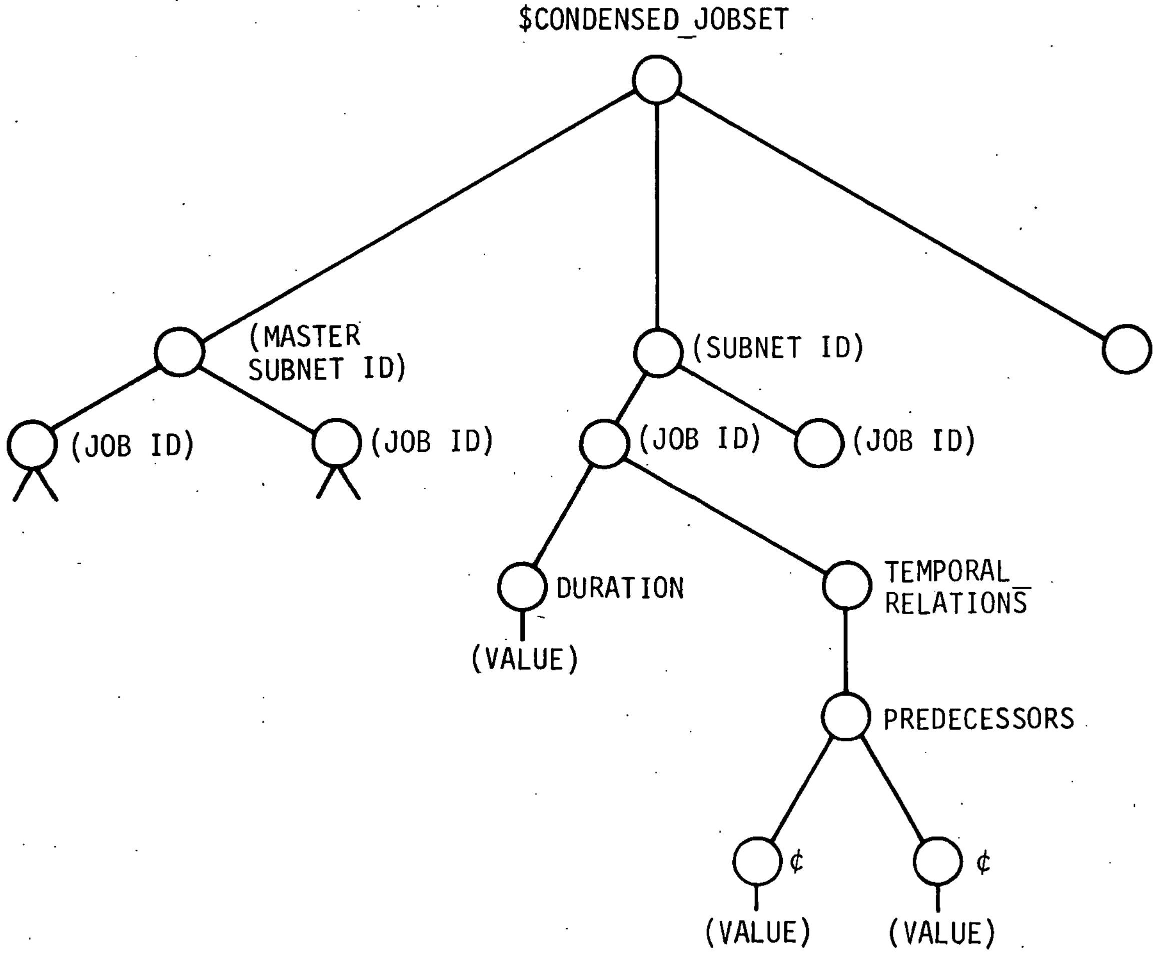
This module will merge two condensed subnetworks into a composite condensed network. This process is essential in merging subnetworks into a self-contained master network.

2.4.26.2 Modules Called

None

2.4.26.3 Module Input

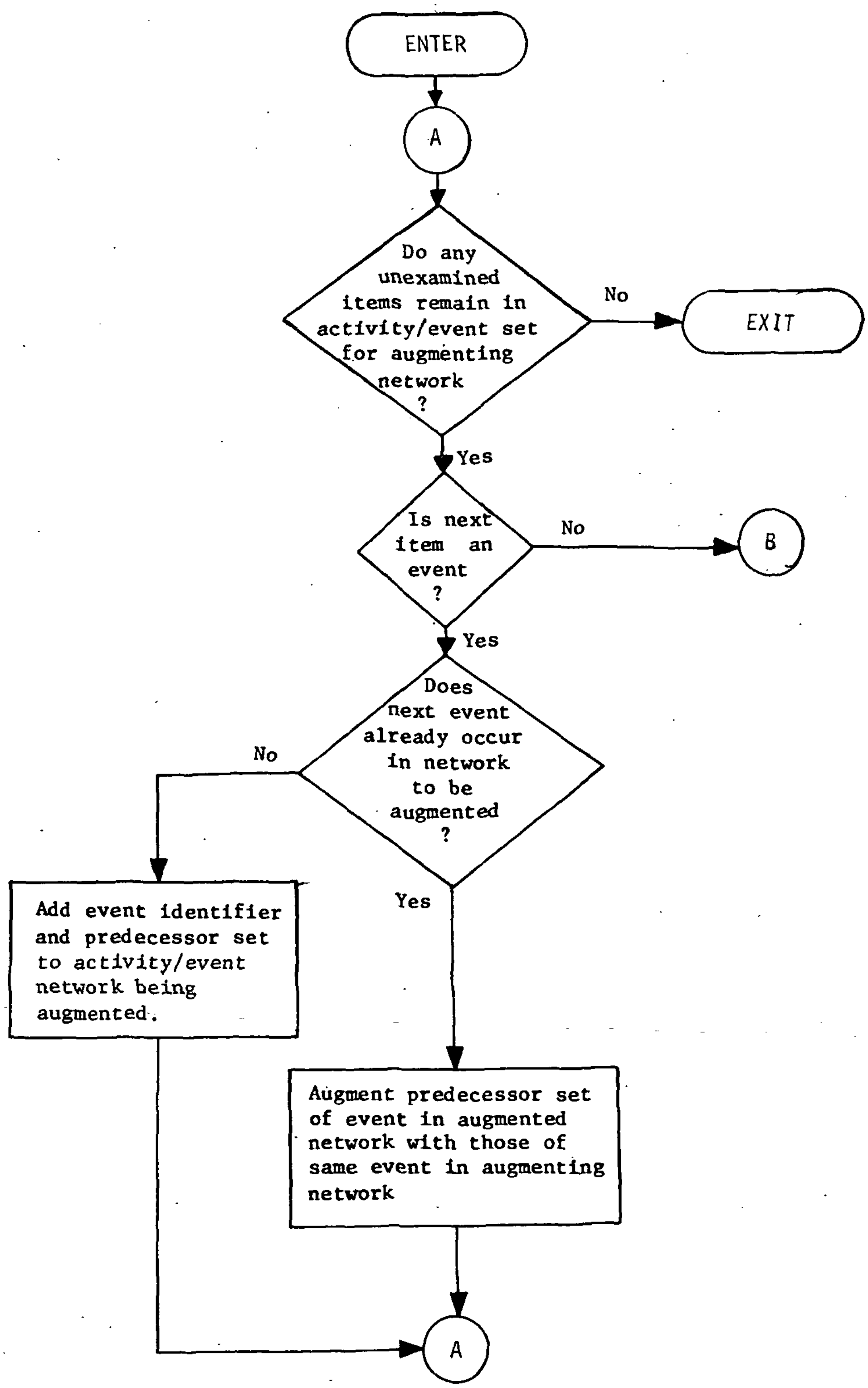
Critical path data for condensed subnetwork and condensed master subnetworks $CONDENSED_JOBSET

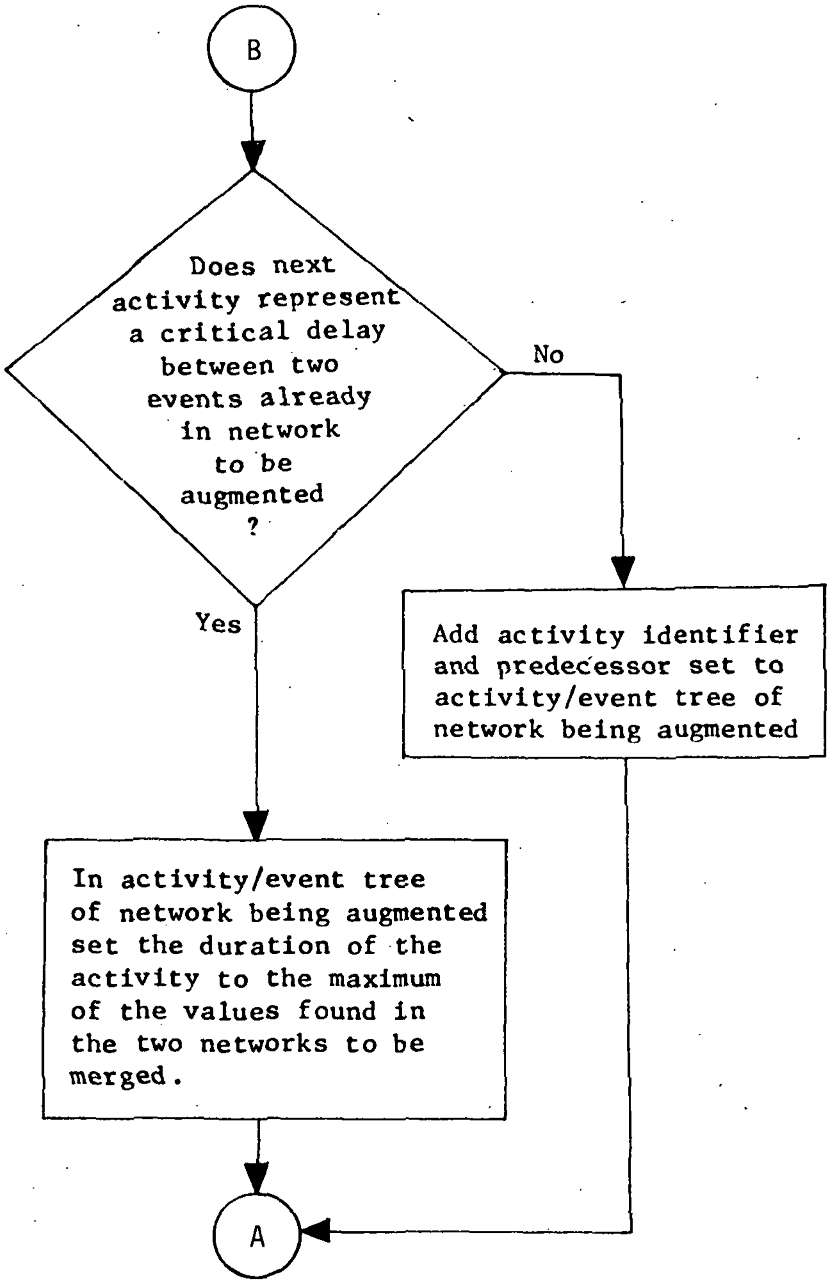


2.4.26.4 Module Output

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

2.4.26.6 Functional Block Diagram




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

2.4.27 NETWORK_ASSEMBLER

2.4.27.1 Purpose and Scope

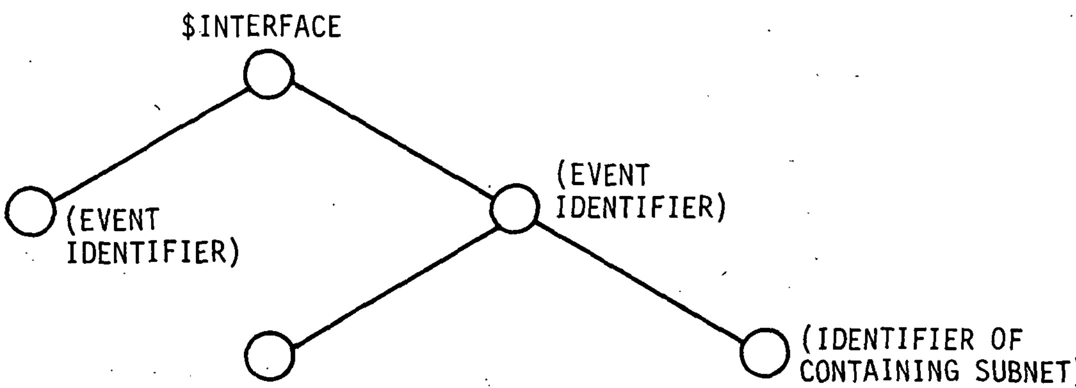
Given a master subnetwork and its prescribed interfacing events, this module will assemble this subnetwork and all of its interfacing subnetworks into a master network. This assembly capability facilitates the heuristic scheduling of any combination of subnetworks that may share common resources. The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

2.4.27.2 Modules Called

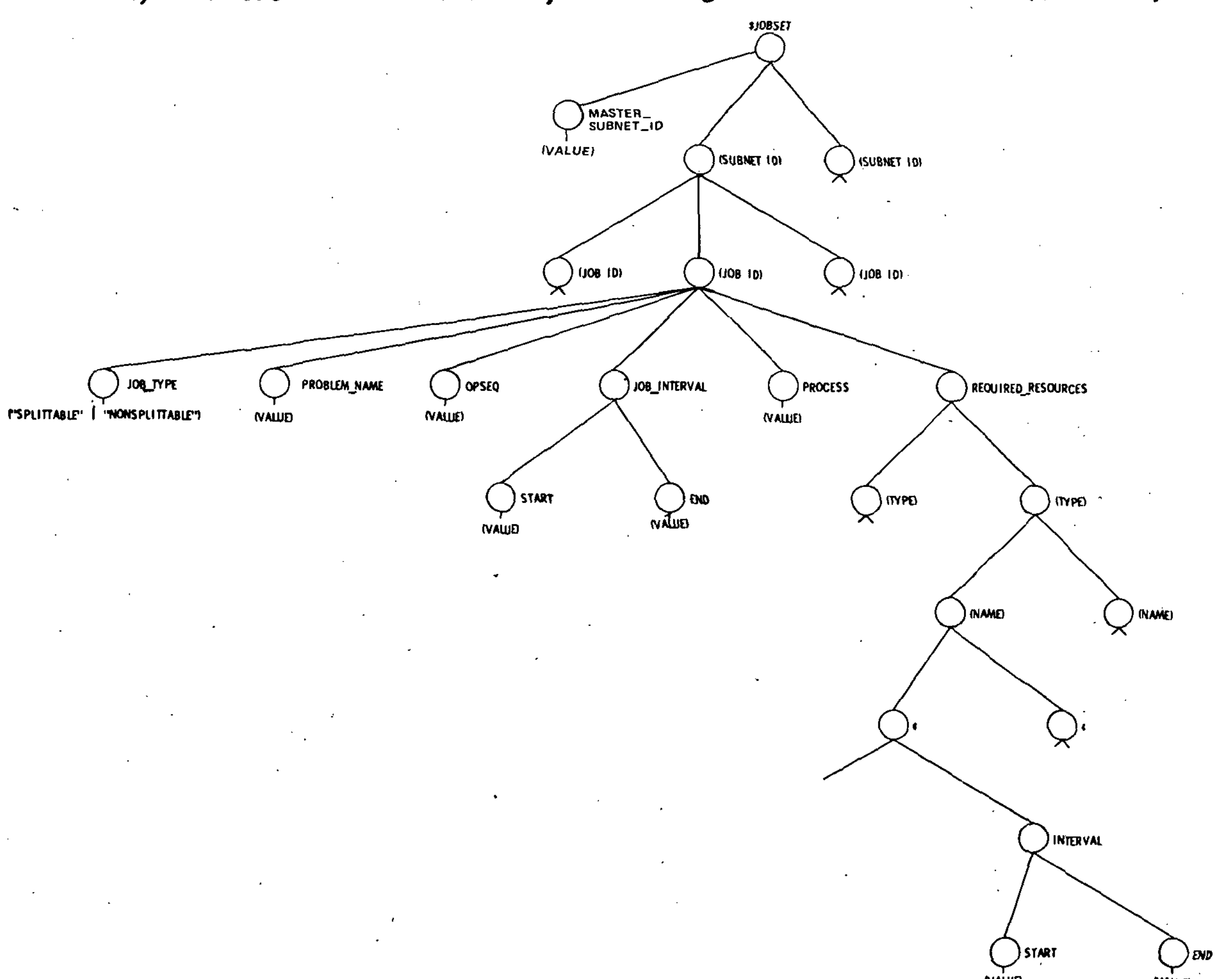
None

2.4.27.3 Module Input

1) Interface event definition ($INTERFACE)



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

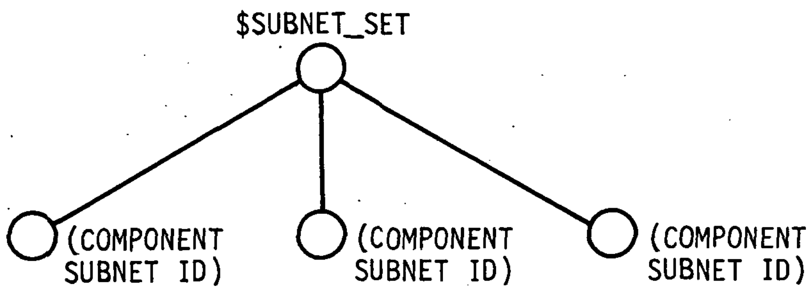


2.4.27.4 Module Output

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

2) Component Subnetworks of Master Network ($SUBNET_SET)

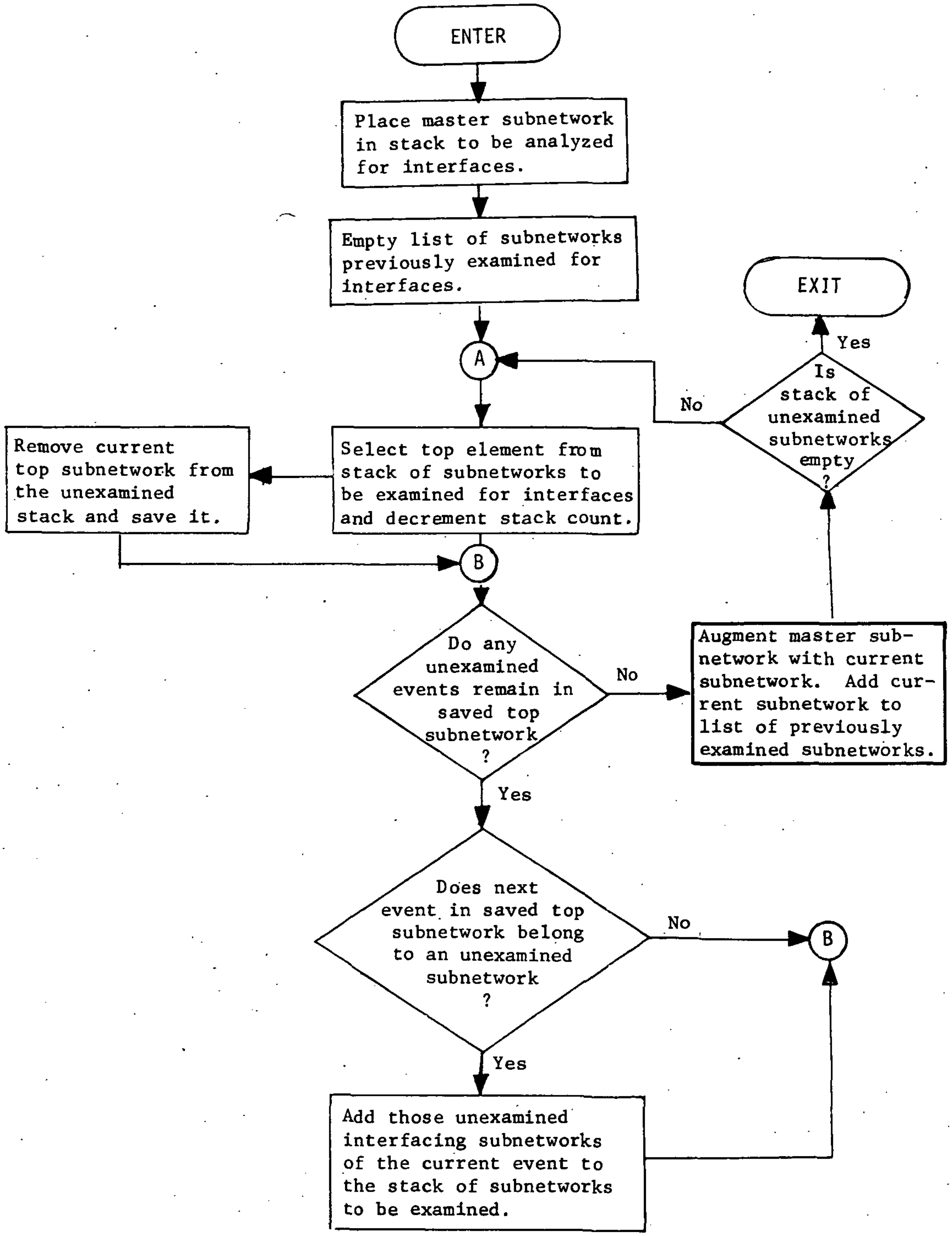
   A. Component subnet identifier



2.4.27.5 Functional Description

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

2.4.27.6 Functional Block Diagram



2.4.28 CRITICAL_PATH_PROCESSOR

2.4.28.1 Purpose and Scope

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

1) Integrate the master subnetwork and all of its interfacing subnetworks into a condensed master network.

2) Compute the early- and late-occurrence dates of all the interface events.

3) Compute all critical-path data for the activities in the master subnetwork and all of its interfacing subnetworks.

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

2.4.28.2 Modules Called

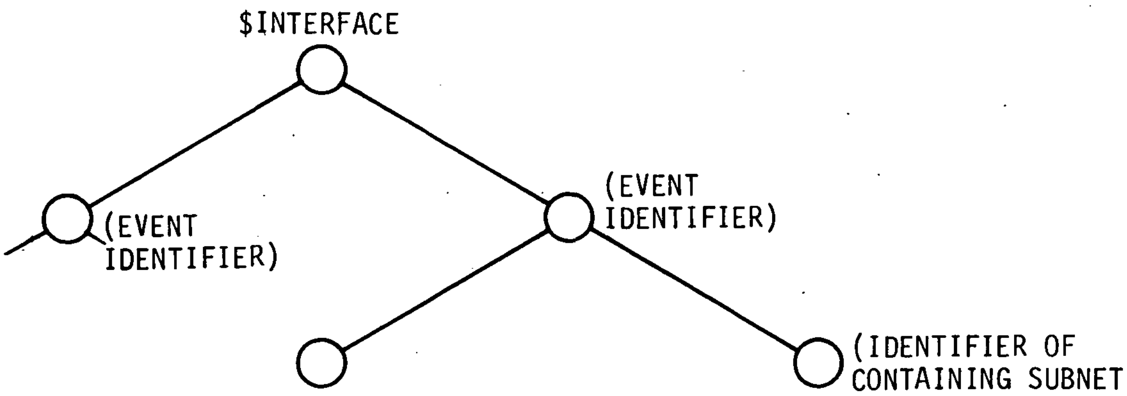
NETWORK_CONDENSER

CONDENSED_NETWORK_MERGER

CRITICAL_PATH_CALCULATOR

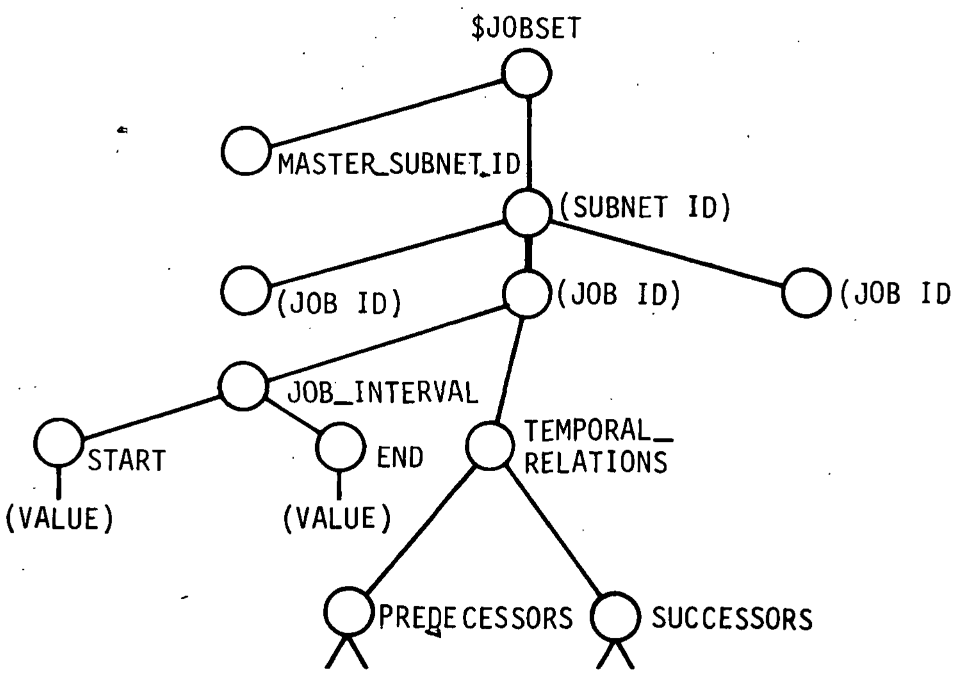
2.4.28.3 Module Input

1) Interface Event Definitions ($INTERFACE)



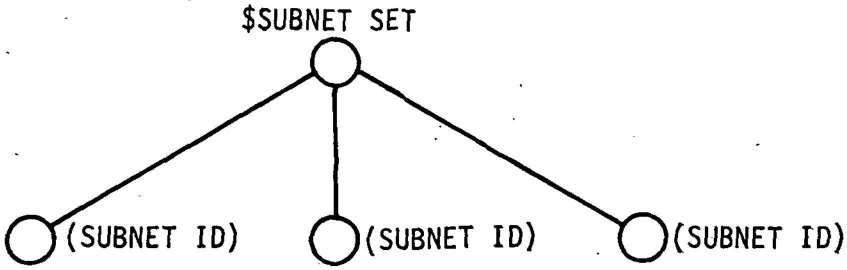
This data structure is illustrated in Fig. 2.4.28-1 for the subnetwork complex of Fig. 2.4.28-2

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

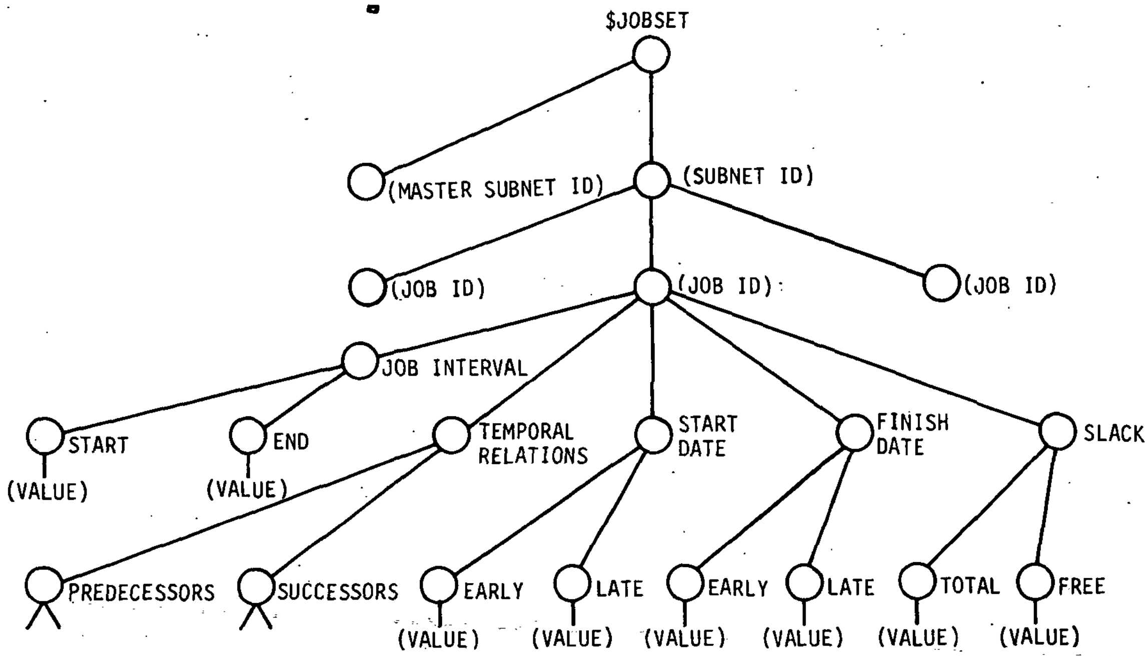


2.4.28.4 Module Output

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



2) Critical Path Output Data ($JOBSET)



2.4.28.5 Functional Description

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

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

The second objective of the module, calculation of the early and late occurrence dates of all the interfacing events, is accomplished by applying the module CRITICAL_PATH_CALCULATOR to the condensed master network. To do so one need only construct the single tree $JOBSET, including the successor set substructure,

2.4.29 NETWORK_EDITOR

2.4.29.1 Purpose and Scope

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

Four types of errors may occur in precedence data:

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

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

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

2.4.29.2 Modules Called

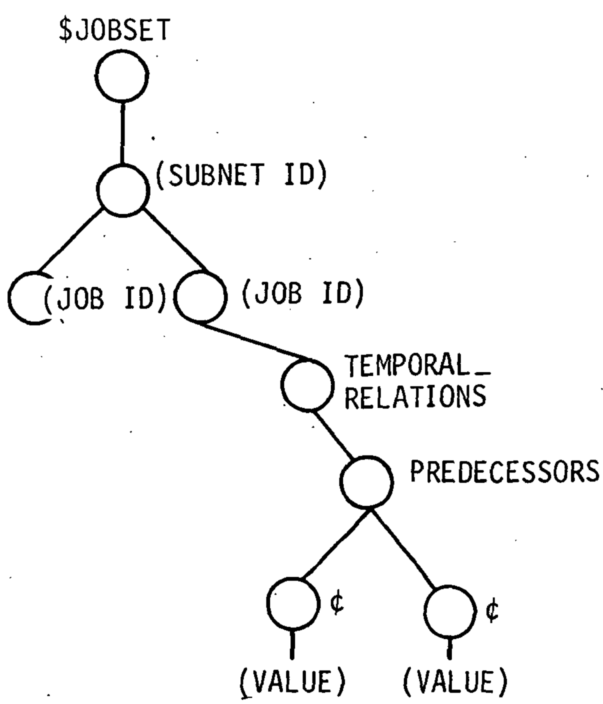
ORDER_BY_PREDECESSORS

REDUNDANT_PREDECESSOR_CHECKER

2.4.29.3 Module Input

1) Network definition $JOBSET - unedited version

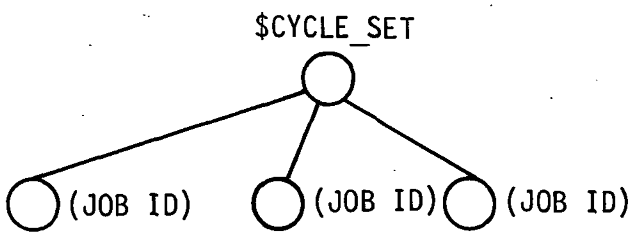
2) Redundant-predecessor-elimination option indicator (SIMPLIFY)



2.4.29.4 Module Output

1) Network definition $JOBSET - edited version

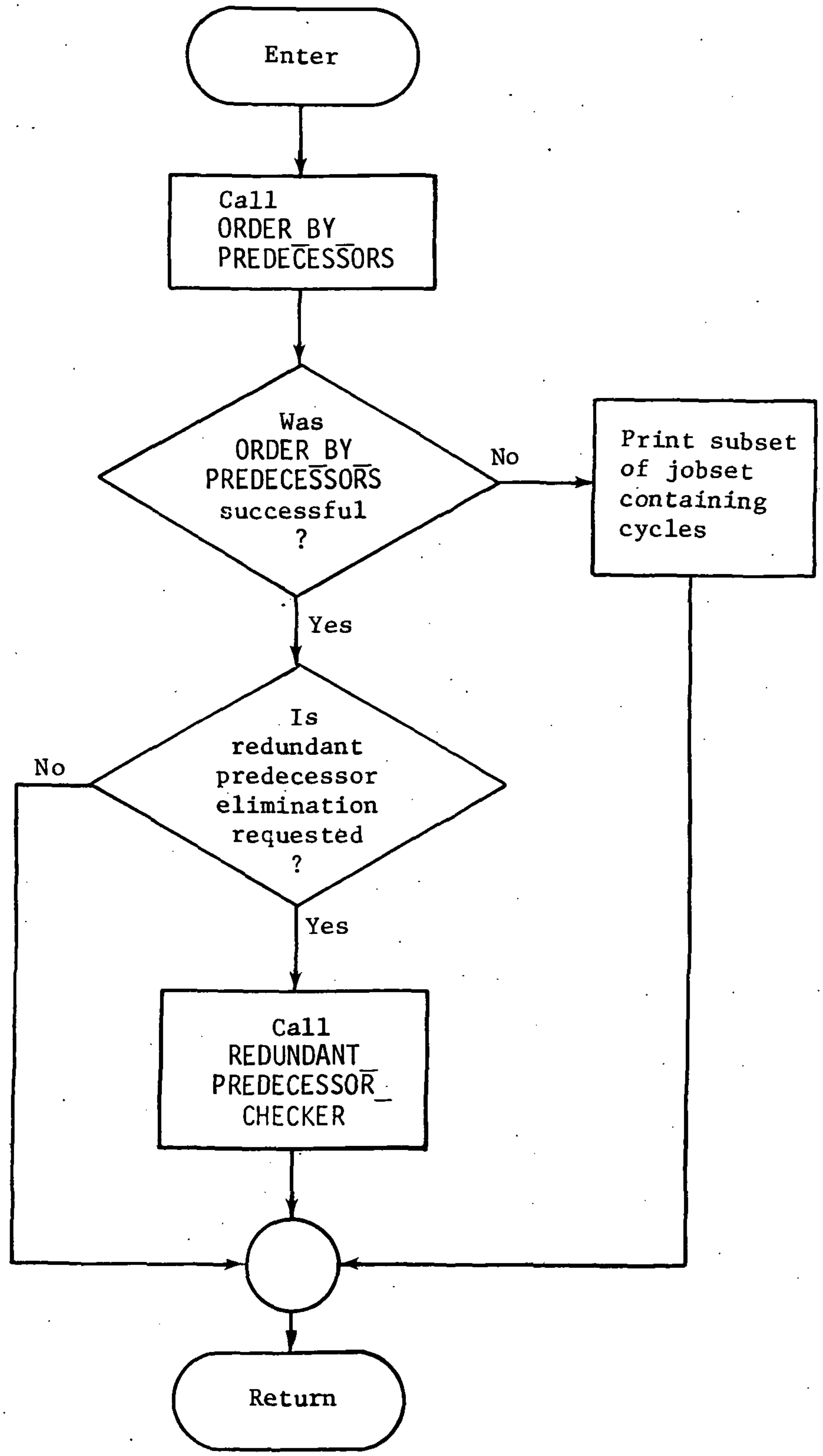
2) Cycle-containing subset of activities or events $CYCLE_SET



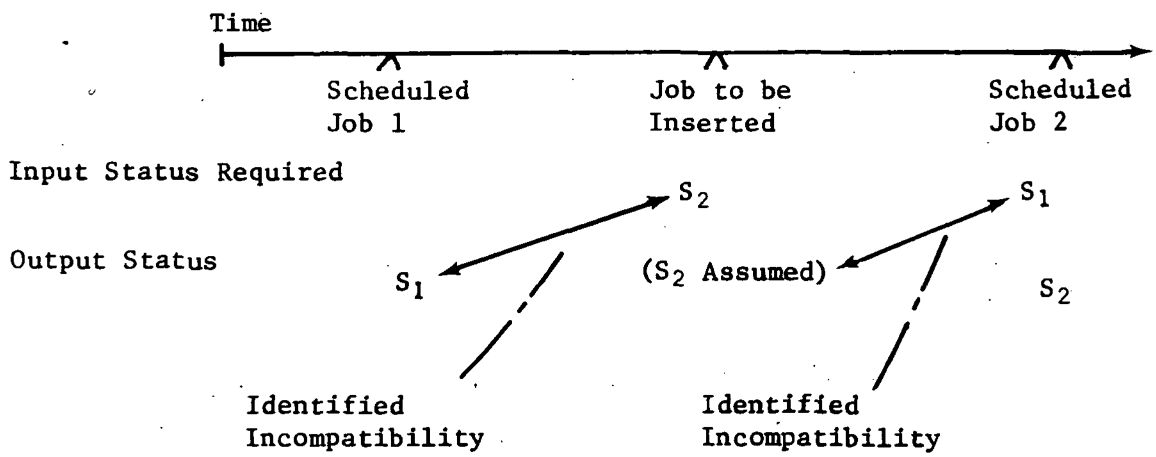
2.4.29.5 Functional Description

The module NETWORK_EDITOR serves primarily as a coordinator of the two editing modules ORDER_BY_PREDECESSORS and REDUNDANT_PREDECESSOR_CHECKER. This module is intended to prevent the user from attempting to use REDUNDANT_PREDECESSOR_CHECKER without first having called ORDER_BY_PREDECESSORS to place the second level subnodes of $JOBSET in a technological ordering. The user may opt not to eliminate redundant predecessors by setting the flag SIMPLIFY.

2.4.29.6 Functional Block Diagram



descriptors at the assignment time, the incompatibilities that are identified for times after the assignment time are those that result assuming compatibility between the scheduled resource descriptors and the required descriptors for the job to be inserted. This is illustrated below.



2.4.30.2 Modules Called

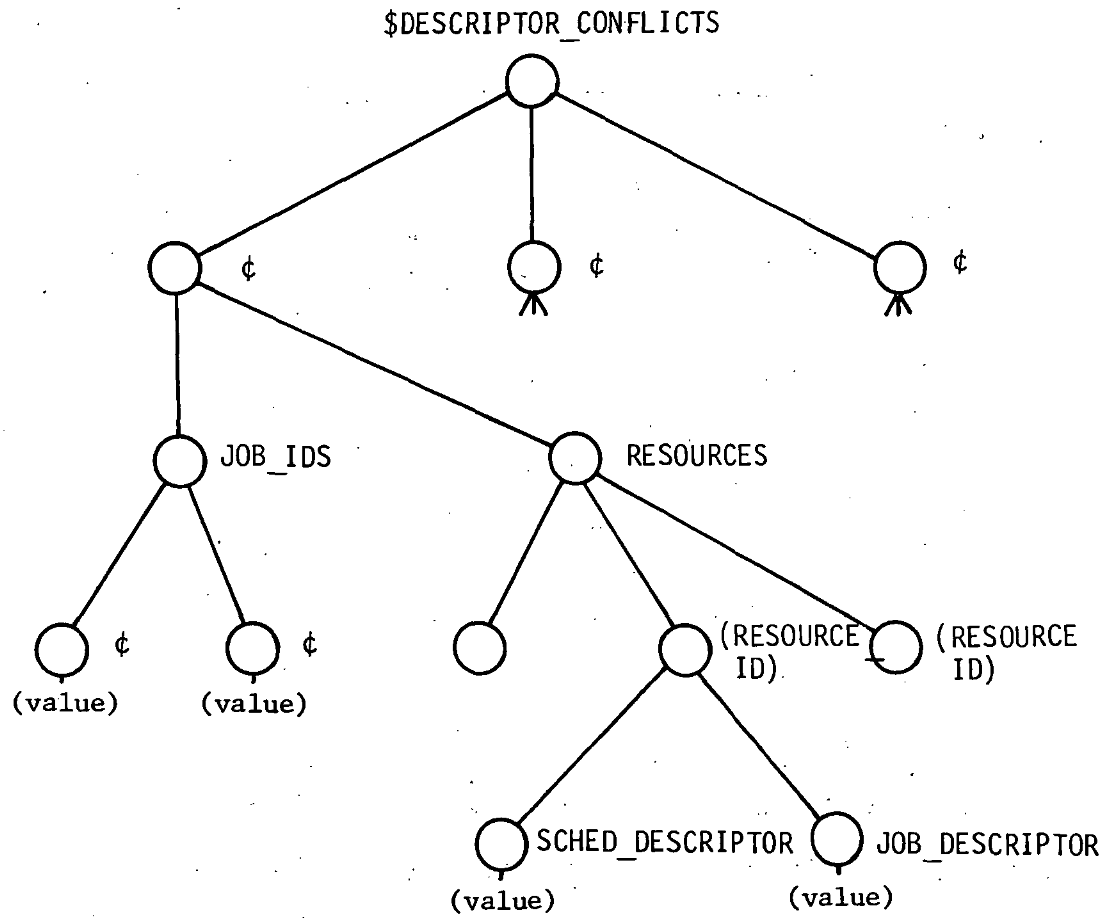
DESCRIPTOR_PROFILE

2.4.30.3 Module Input

This module is called with two input arguments. They are $RESOURCE and $SCHEDULE_UNIT. $RESOURCE has the general structure given in Section 2.2 and must contain initial descriptors at a reference time and all assignment and descriptor changes that are to be considered after that time. This information is required by this module so that it can call DESCRIPTOR_PROFILE.

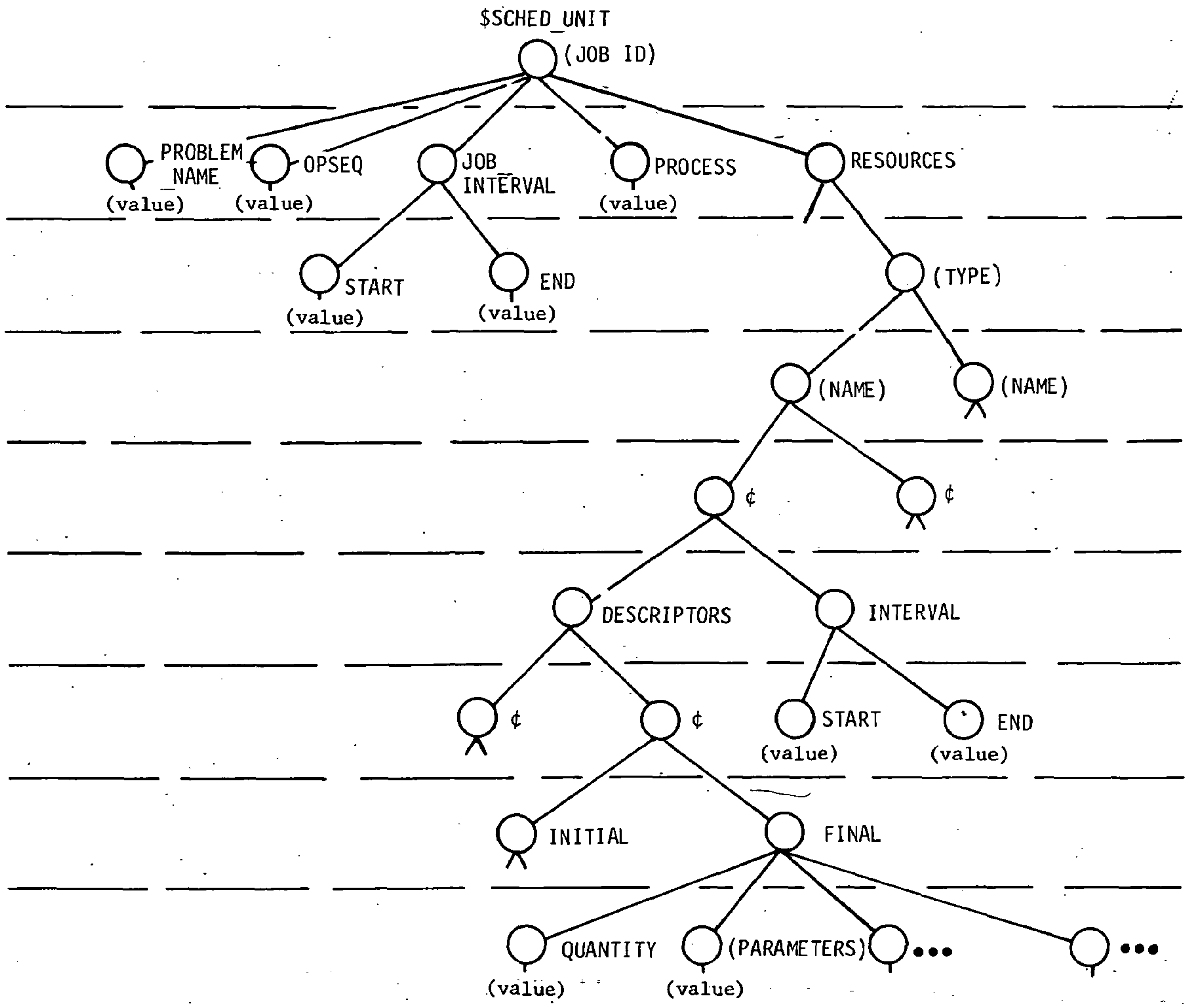
2.4.30.4 Module Output

This module returns a structure called $DESCRIPTOR_CONFLICTS, which contains information about the conflicts that would result if $SCHED_UNIT were assigned at its specified time. The general structure of $DESCRIPTOR_CONFLICTS is shown below:



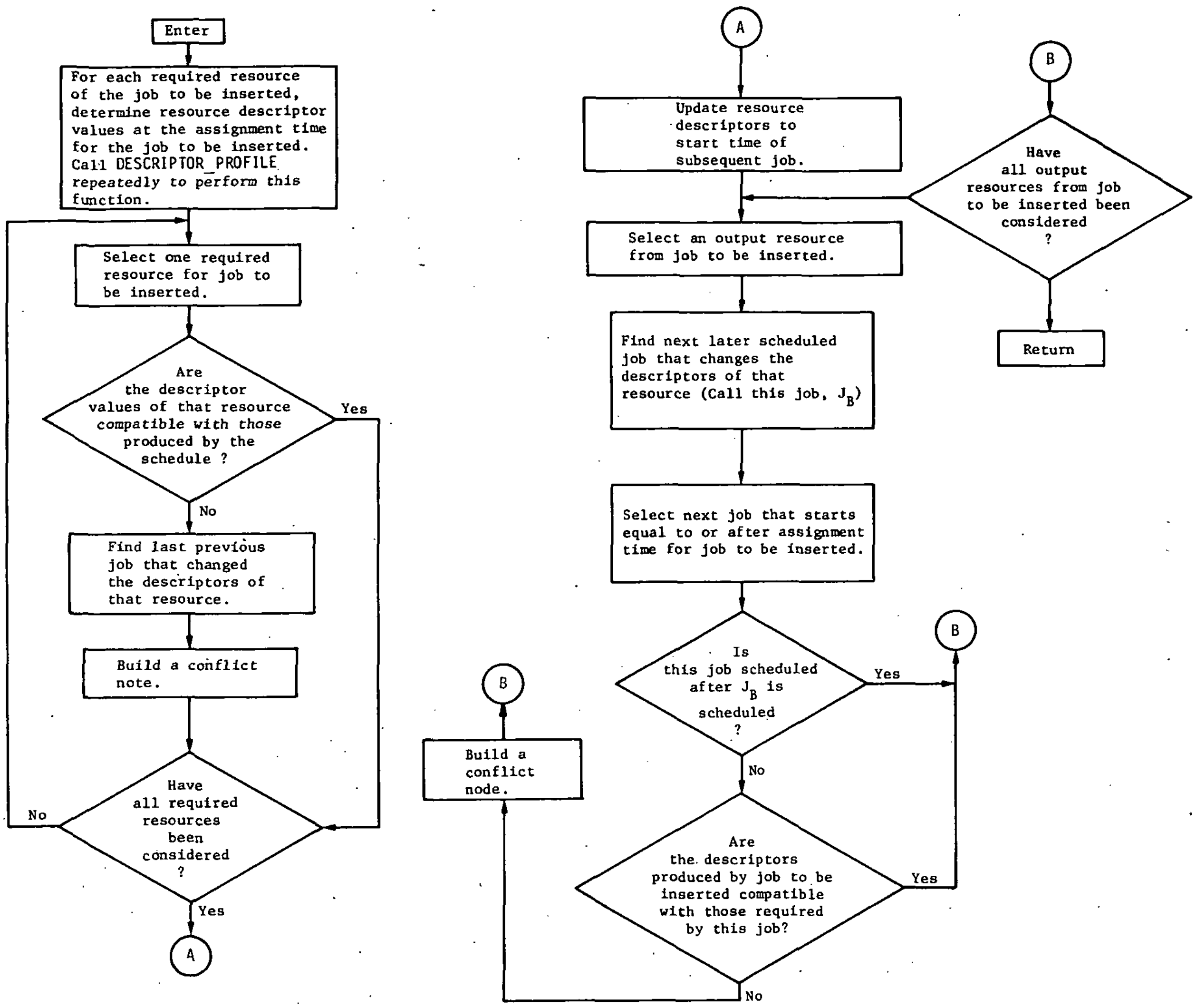
Each first-level subnode represents a resource status conflict that would result from the assignment of $SCHED_UNIT at the specified time.

$SCHED_UNIT has the general structure of a schedule unit shown below:



Note that in $SCHED_UNIT, the JOB_INTERVAL.START must contain the assignment time for the job to be inserted.

2.4.30.5 Functional Block Diagram



2.4.31 ORDER_BY_PREDECESSORS

2.4.31.1 Purpose and Scope

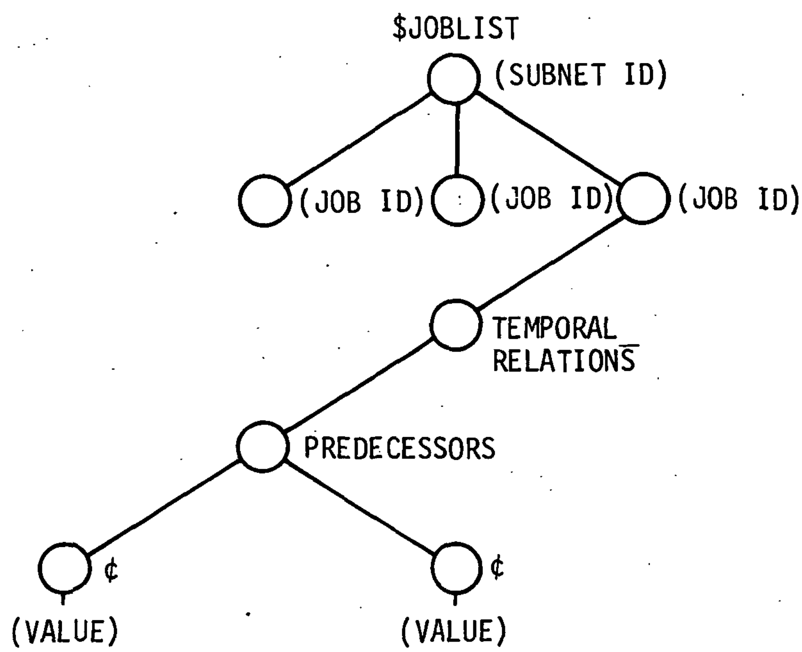
Given a set of activities and events and their respective predecessor sets, this module either places them in a technological order if one exists or identifies a subset of the activities containing a cycle. A technological ordering of the events and activities means an ordering such that any activity or event is preceded by all of its predecessors or equivalently followed by all of its successors. A cycle, on the other hand, is a chain of predecessor-successor related activities or events implying that some event or activity is a predecessor of itself. Such an activity or event could never be scheduled because one of its predecessors, namely itself, could never be completed beforehand. Hence, the presence of cycles in a precedence network precludes any scheduling or critical path analyses.

2.4.31.2 Modules Called

None

2.4.31.3 Module Input

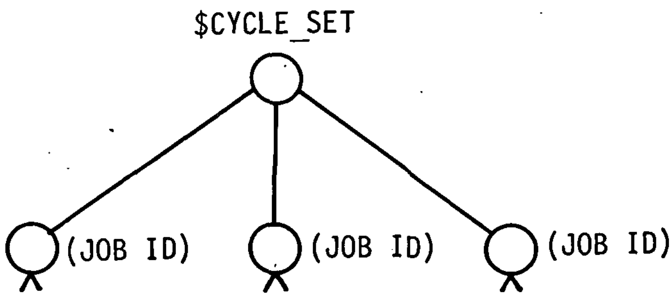
Network definition ($JOBSET) - activities or events (first level subnodes) are not technologically ordered.



2.4.31.4 Module Output

1) Network definition ($JOBSET) - activities or events (second-level subnodes) are technologically ordered.

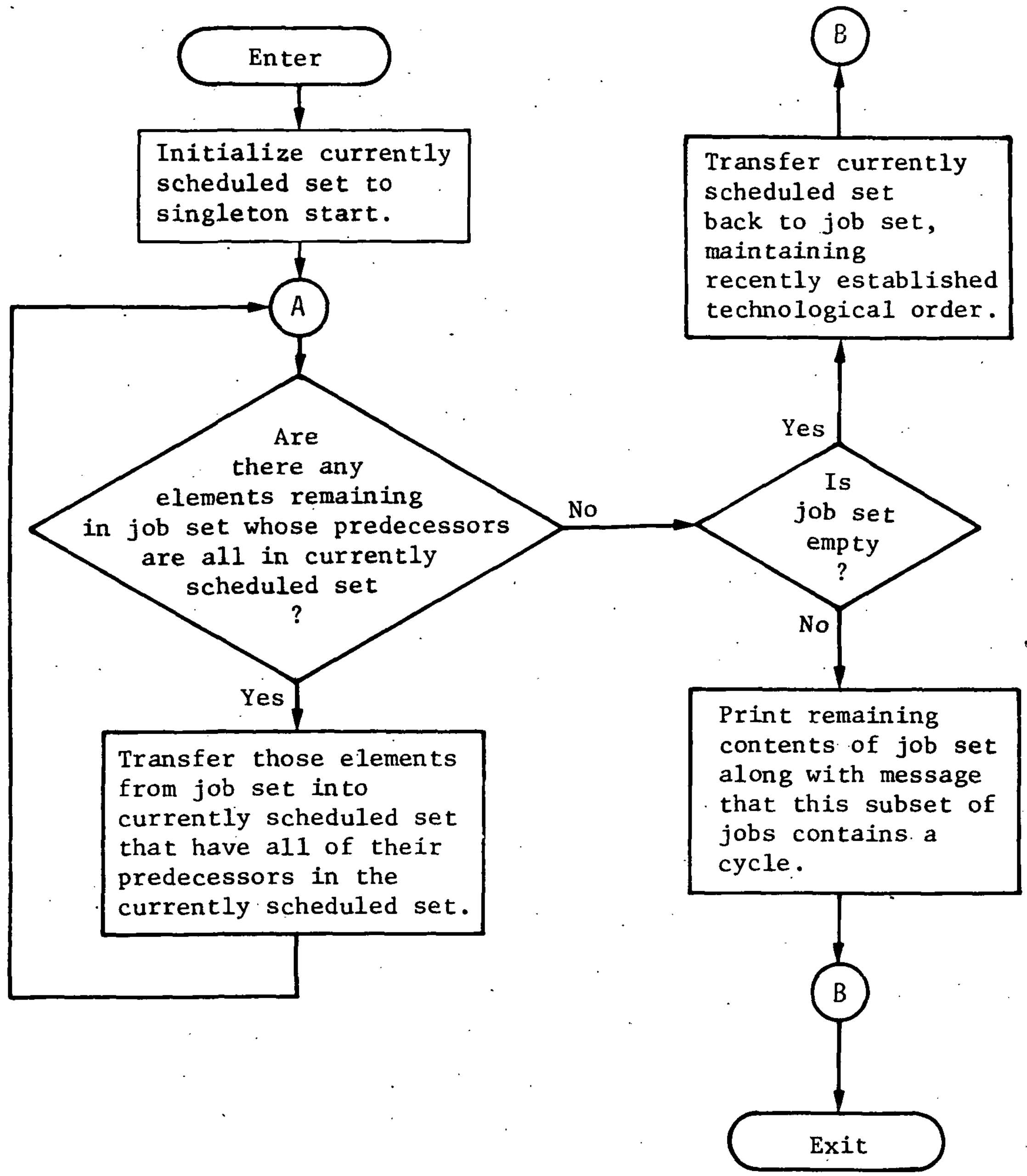
2) Subset of jobs containing cycles (if any exist) ($CYCLE_SET)



2.4.31.5 Functional Description

It can be shown that the activities and events of a project can be technologically ordered if, and only if, the precedence relations contain no cycles. It must be noted, however, that if cycles are absent, the technological ordering is by no means unique. The particular ordering produced by this module results from inductively "scheduling" in cycles all those activities or events whose predecessors are "scheduled." Eventually a cycle arises where there are no activities or events with all of their predecessors "scheduled." If some activities or events remain unscheduled, they contain a cycle. A more precise description of the logic of the module is provided in the functional block diagram.

2.4.31.6 Functional Block Diagram



limits of their residual slack to produce heuristically the most level resource-loaded schedule.

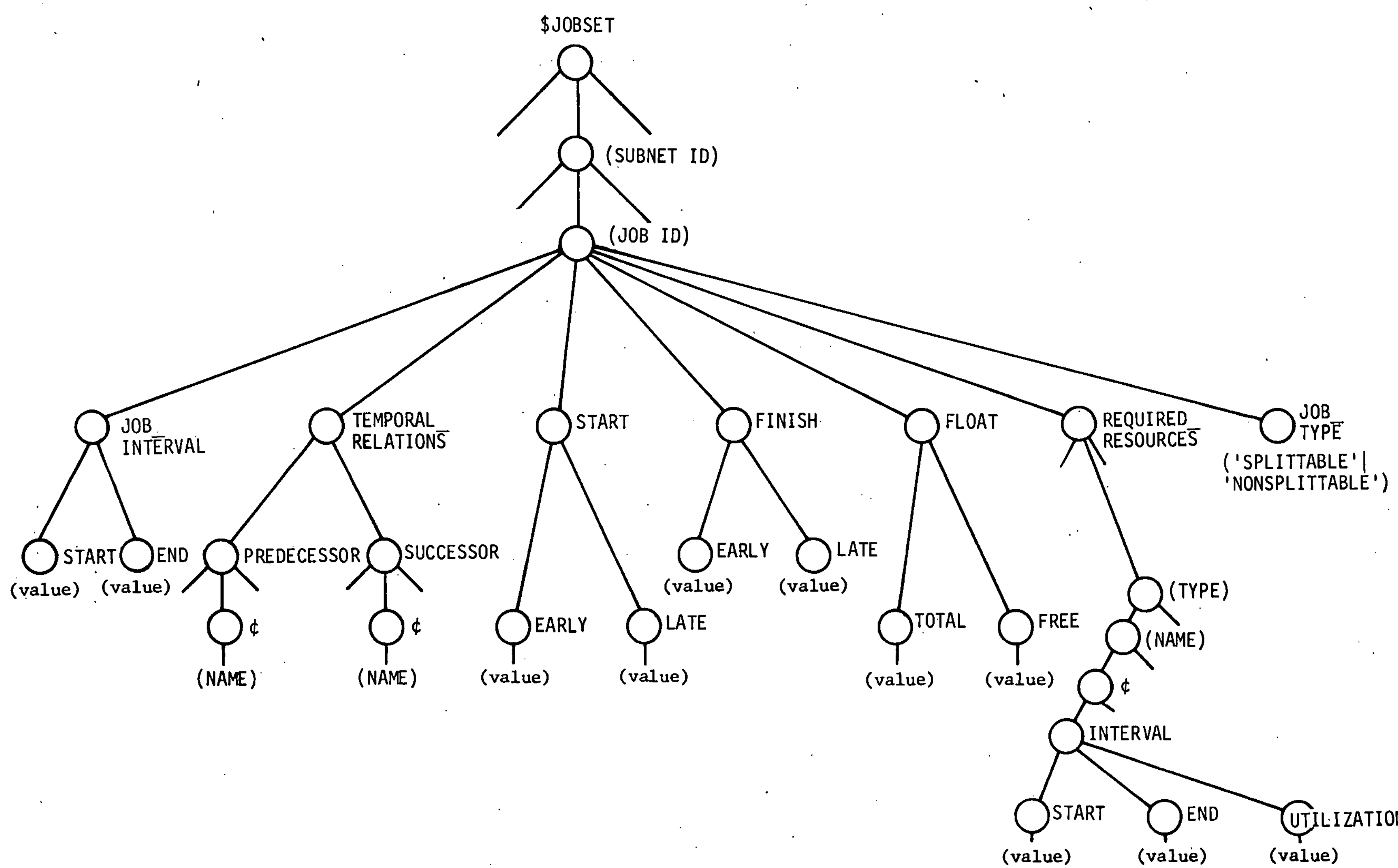
2.4.32.2 Modules Called

None

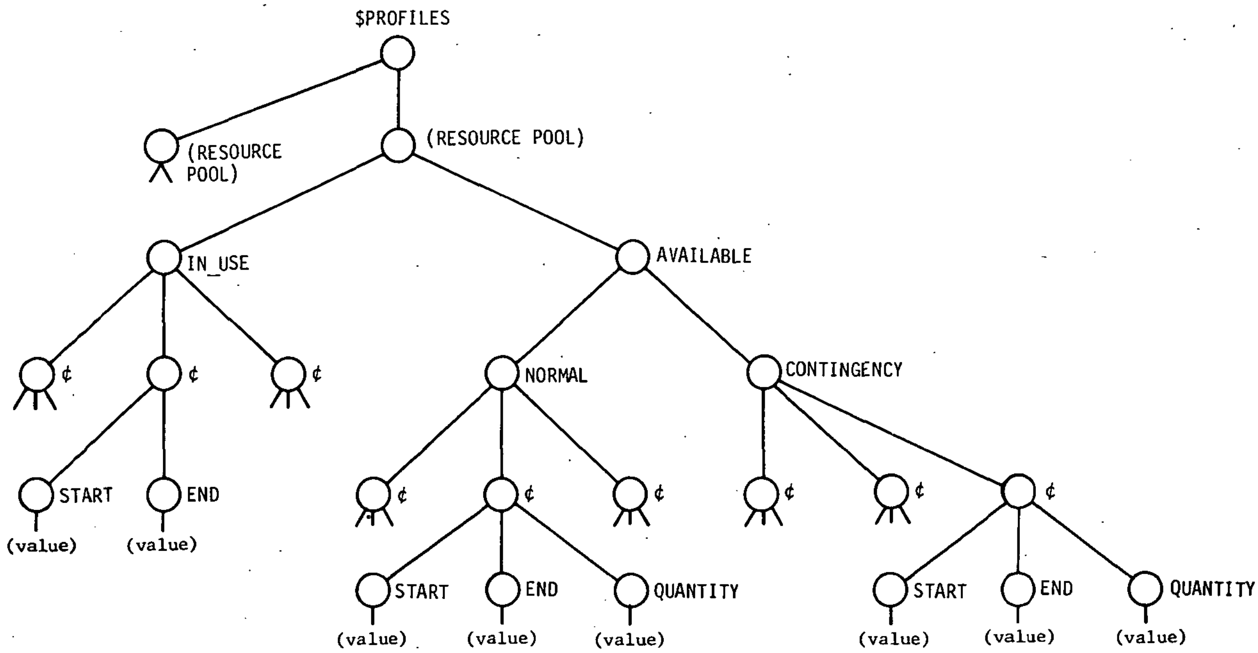
2.4.32.3 Module Input

1) Network, Critical Path Data and Activity or Event Definitions

$JOBSET

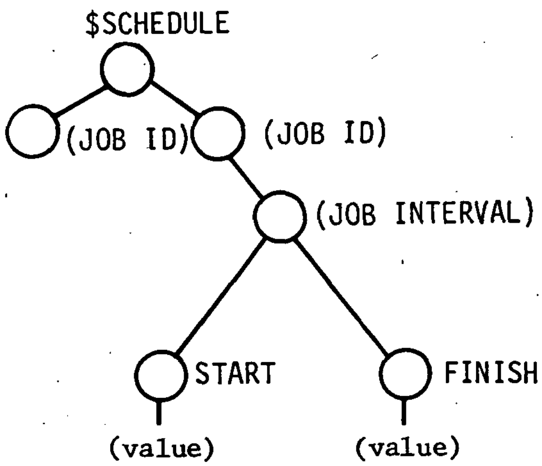

$JOBSET
(SUBNET ID)
(JOB ID)
JOB INTERVAL
TEMPORAL RELATIONS
START
FINISH
FLOAT
REQUIRED RESOURCES
JOB TYPE
('SPLITTABLE'|
'NONSPLITTABLE')
START
END
(value)
(value)
PREDECESSOR
SUCCESSOR
¢
¢
(NAME)
(NAME)
EARLY
LATE
(value)
(value)
EARLY
LATE
(value)
(value)
TOTAL
FREE
(value)
(value)
(TYPE)
(NAME)
¢
INTERVAL
START
END
UTILIZATION
(value)
(value)
(value)

2) Resource Definitions ($PROFILES)

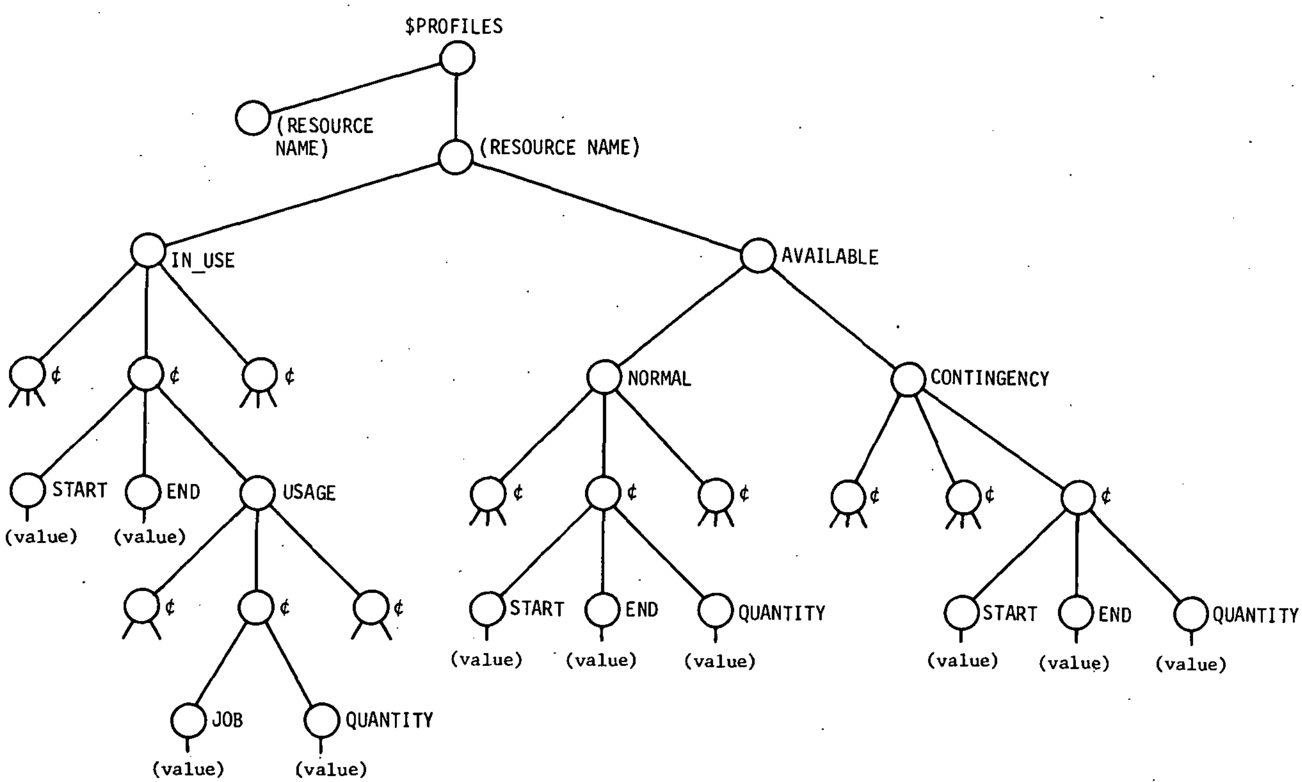


2.4.32.4 Module Output

1) Resulting Heuristic Schedule ($SCHEDULE)



2) Revised Resource Profile Including Usage ($PROFILES)



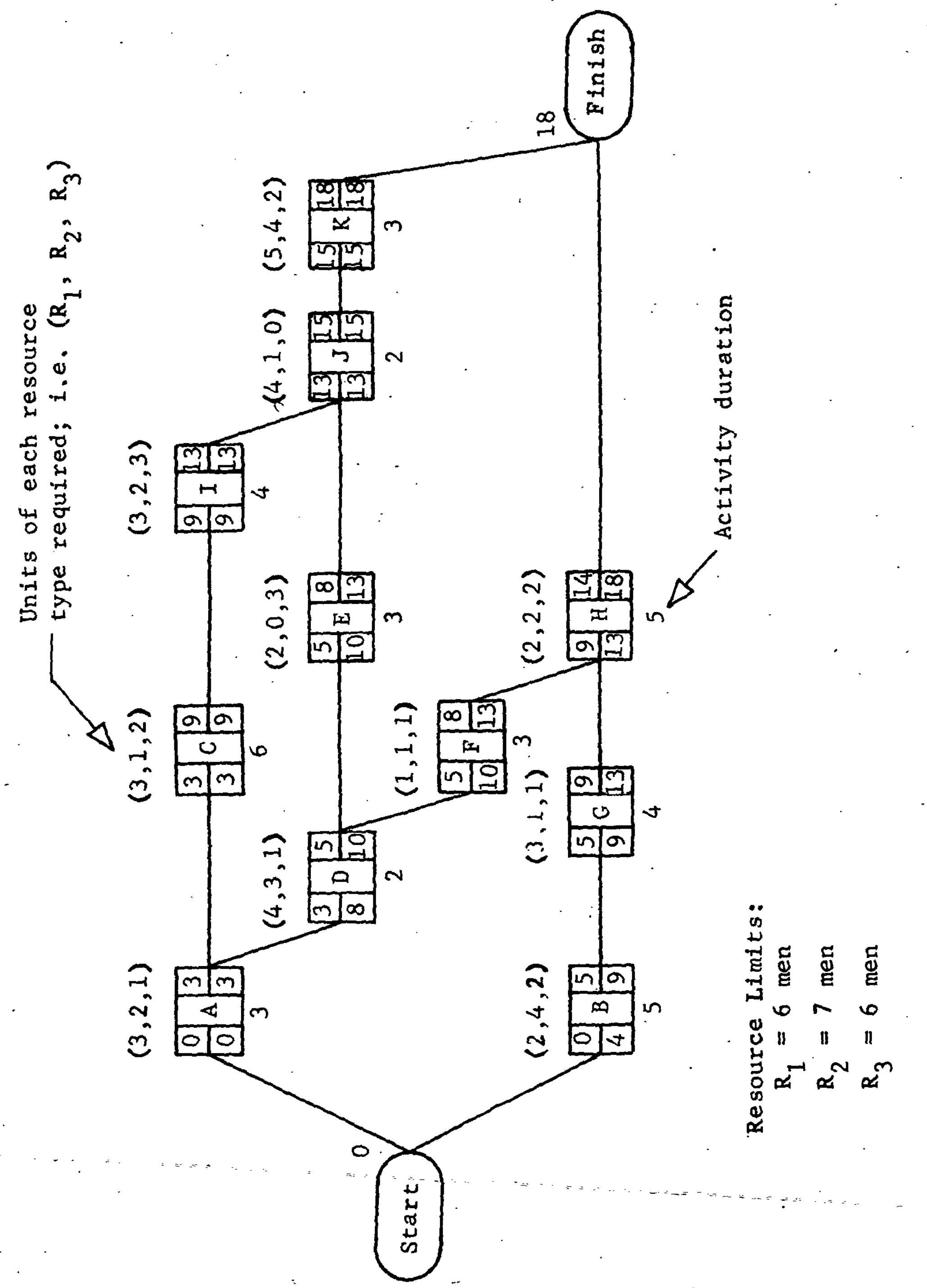


Fig. 2.4.32-1
Constrained-Resource Problem with Three Resource Types

Fig. 2.4.32-1

ACTIVITIES									
ACTIVITY	DURATION	LATEST START	RESOURCE REQUIREMENTS			CYCLE OF ENTRY	SCHEDULED START	SCHEDULED FINISH
A	3	0	3	2	1	1	0	3
B	5	4	2	4	2	1	0	5
C	6	3	3	1	2	2	3	9
D	2	8	4	3	1	2	5	7
E	3	10	2	0	3	4	7	10
F	3	10	1	1	1	4	10	13
G	4	9	3	1	1	3	7	11
H	5	13	2	2	2	6	13	18
I	4	9	3	2	3	5	9	13
J	2	13	4	1	0	6	13	15
K	3	15	5	4	2	7	15	18

CYCLES			
CYCLE	PROCESS TIME	SET	
1	0	A	B
	1		
	2		
2	3	C	D
	4	D	
3	5	D	G
	6	D	
	7	D	
	8	D	

RESOURCES					
DAY	RESOURCES AVAILABLE			NORMAL ALLOCATION	CONTINGENCY ALLOCATION
1	~~6~~	~~3~~	1	6	0
	7	~~5~~	1	7	0
	~~6~~	~~5~~	3	6	0
2	~~6~~	~~3~~	1	6	0
	7	~~5~~	1	7	0
	~~6~~	~~5~~	3	6	0
3	~~6~~	~~3~~	1	6	0
	7	~~5~~	1	7	0
	~~6~~	~~5~~	3	6	0
4	~~6~~	~~4~~	1	6	0
	7	~~3~~	2	7	0
	~~6~~	~~4~~	2	6	0
5	~~6~~	~~4~~	1	6	0
	7	~~3~~	2	7	0
	~~6~~	~~4~~	2	6	0
6	~~6~~	~~3~~	0	6	0
	7	~~6~~	5	7	0
	~~6~~	4	3	6	0
7	~~6~~	~~3~~	0	6	0
	7	~~6~~	5	7	0
	~~6~~	~~4~~	3	6	0
8	~~6~~	~~3~~	0	6	0
	7	~~6~~	5	7	0
	~~6~~	~~4~~	3	6	0
9	~~6~~	~~3~~	0	6	0
	7	~~6~~	5	7	0
	~~6~~	~~4~~	3	6	0

Reschedule to Expedite D (days 4–9)

Fig. 2.4.32-2
Trace of the Execution of the RESOURCE_ALLOCATOR *Algorithm on the Constrained-Resource Problem Shown in Fig. 2.4.32-1, Using Contingency Resource Thresholds on the First and Third Resources, Respectively*

Fig. 2.4.32-2

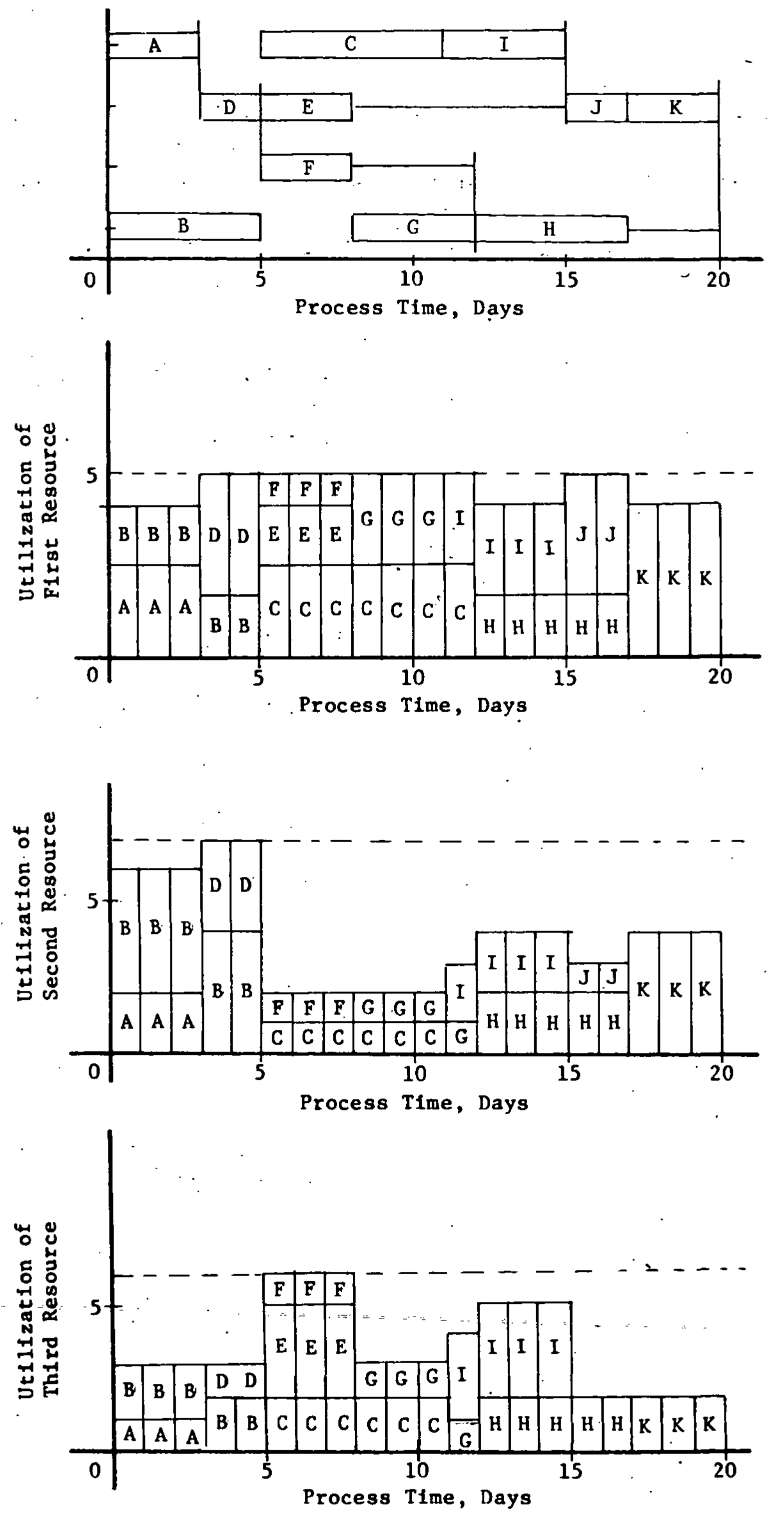


Fig. 2.4.32-4
Minimum Duration Solution to Constrained-Resource Problem Using No Resource Contingency Levels

The optimal schedule requires two more days than the contingency-resource schedule. Which schedule is suprior depends on the availability of supplemental resource units; that is, on the "hardness" of the resource constraints. It is obvious that the optimal schedule is superior to the 25-day RESOURCE_ALLOCATOR schedule generated assuming no resource contingency levels, as shown in Fig. 2.4.32-5. Thus, it is apparent that the simple priority rule scheduling of the RESOURCE_ALLOCATOR, which is in force when no resource thresholds are present, is greatly enhanced by the modifying heuristic that invokes contingency resources when an activity's late-start date is slipped. Finally, it should be noted that by executing a series of parametric funs with varying resource contingency thresholds, a thorough analysis of the tradeoff between project duration and resource availability can be made.

2.4.32.8 References

Davis, Edward W. and Heidorn, George E., "An Algorithm for Optimal Project Scheduling under Multiple Resource Constraints", *Management Science*, August 1971.

Davis, Edward W., "Networks: Resource Allocation", *Journal of Industrial Engineering*, April 1974.

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

2.4.33 RESOURCE_LEVELER

2.4.33.1 Purpose and Scope

In many project scheduling situations, the pattern of resource utilization is often more important than the quantity of resources used. For example, a resource feasible schedule that results in rapidly changing resource requirements is clearly undesirable from the project control standpoint. In these situations it is useful to perform resource leveling in order to reduce resource profile fluctuations.

Conceptually, a resource utilization profile is level when the actual quantity of resource used in each time period is constant. Unfortunately, it is not generally possible to maintain perfectly level profiles and simultaneously satisfy all of the scheduling constraints. As a consequence, some fluctuations will inevitably remain in the resource profiles. The purpose of this module is then to minimize these remaining resource variations. This is accomplished by heuristically minimizing the sum of the squares of the resources over time, subject to the network, resource availability, and activity completion constraints.

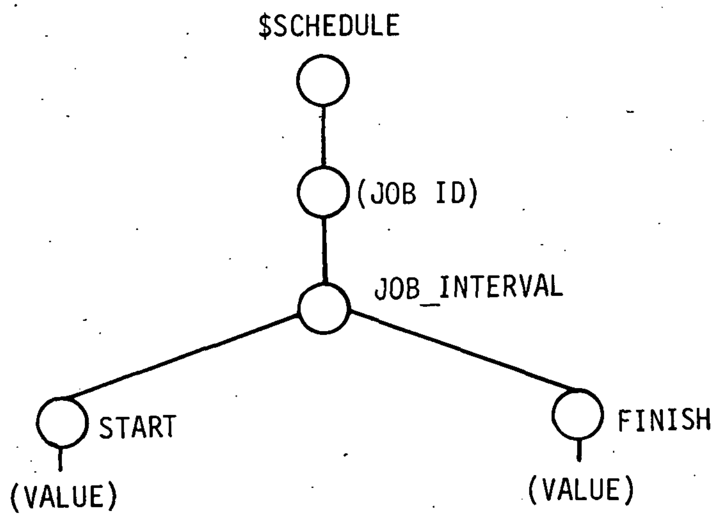
This module is applicable to the general class of project scheduling problems that includes multiple resources with time varying pool levels.

2.4.33.2 Modules Called

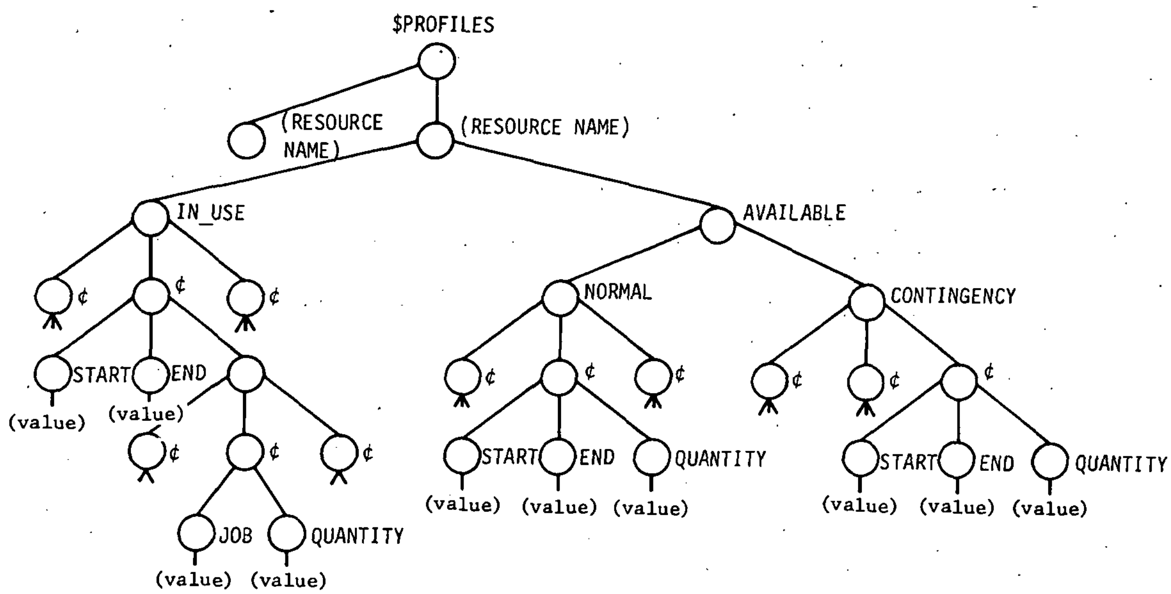
None.

2.4.33.3 Module Input

1) Nominal Schedule ($SCHEDULE)



2) Nominal Resource Profile ($PROFILES)



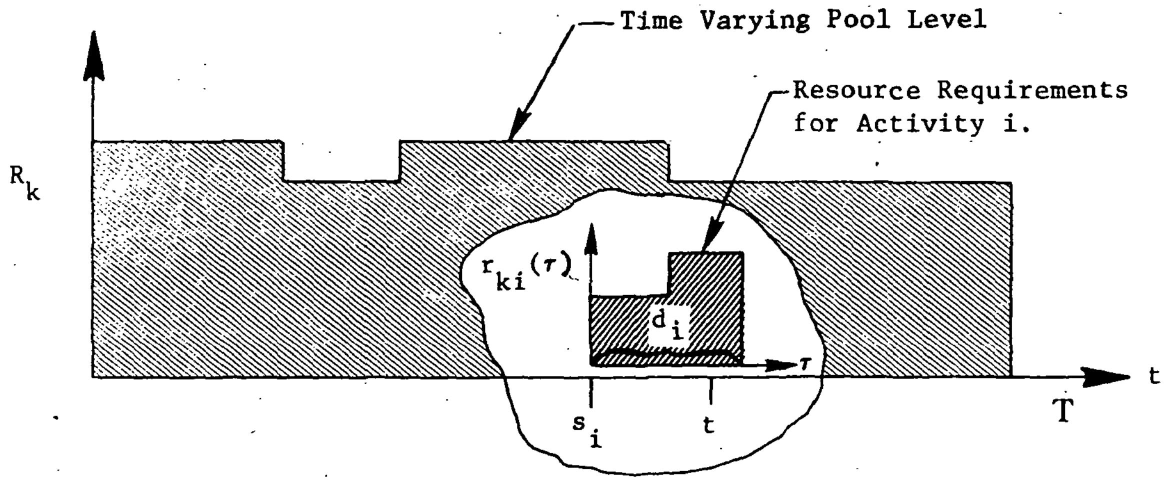
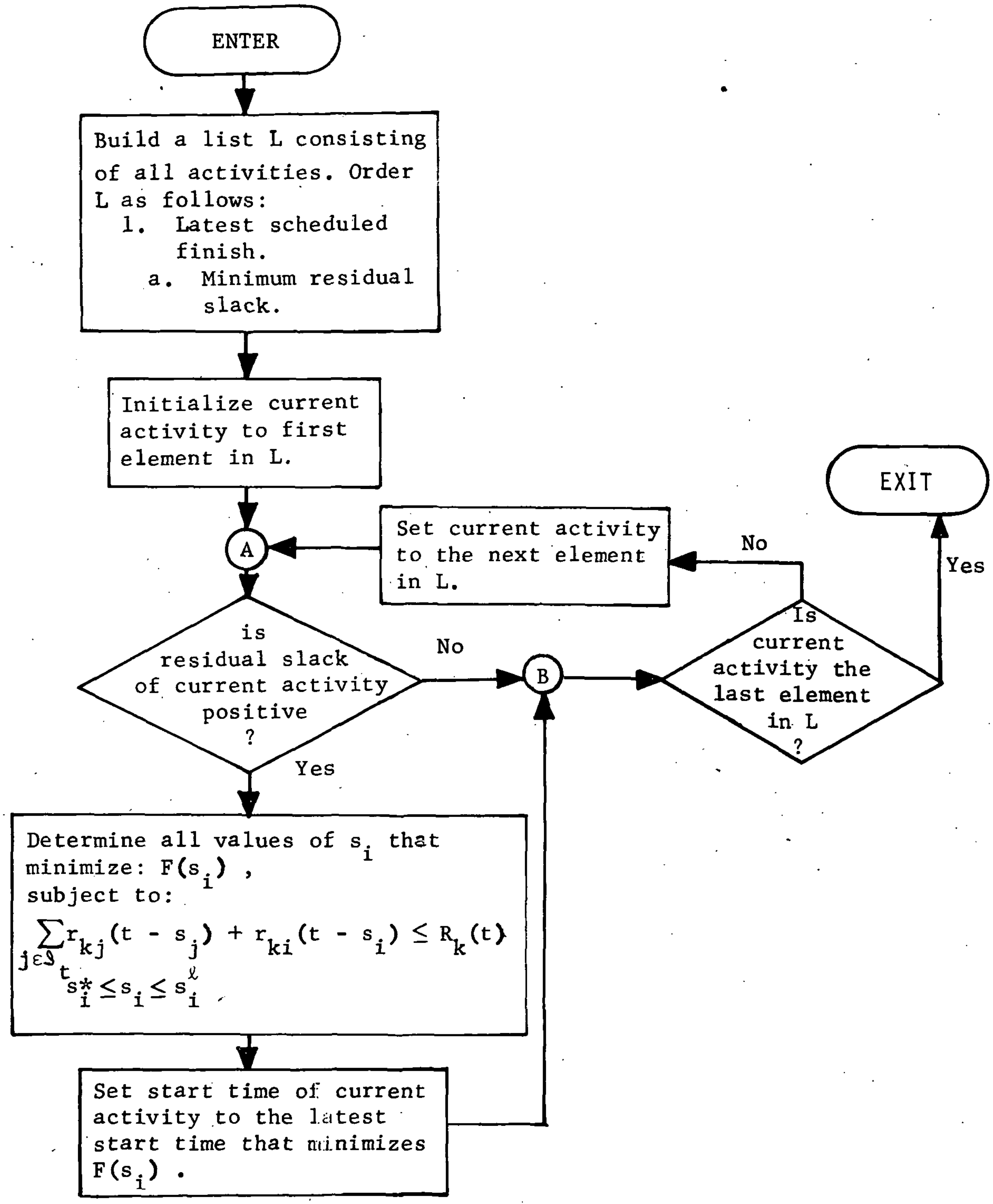


Fig. 2.4.33-2 Time-Varying Resource Variables

This module can also be easily modified to solve the resource profile shaping problem. This can be accomplished by minimizing the square of the differences between actual and desired resource profiles.

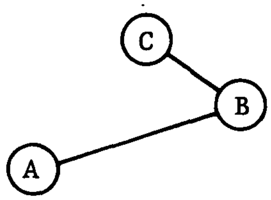
2.4.33.6 Functional Block Diagram



ingenuity. Furthermore, questions that arise in modeling the project as a precedence network, frequently shed light on the entire scheduling problem.

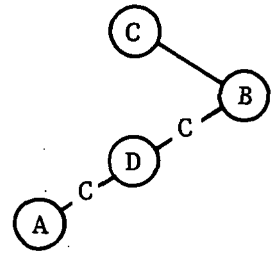
Burman (Burman, 72) has suggested a sophistication of the ordinary precedence network that would permit the simple representation of all temporal relations among activities and events. Indeed, a somewhat more involved critical path algorithm can be developed to generate critical path data for his sophisticated networks. Unfortunately, however, the new networks hopelessly complicate any heuristic scheduling process. As is so often the case in problem solving, it is far easier to generalize a problem than to solve it.

Basically, what Burman has done is to identify a new type of successor--the closely-continuous successor. Such a successor must begin at the instant of completion of its predecessor. To see how this new concept facilitates the simulation of general temporal relations, consider the following examples. Consider the most difficult case of two activities whose respective start and finish are constrained to differ by a fixed time interval with the successor activity having an ordinary second predecessor as shown in Fig. 2.4.34-1.



$s_B = f_A + a$

$s_B \geq f_c$



$d_D = a$

Fig. 2.4.34-1
Sample Representation of a General Temporal Relation Using Closely-Continuous Successors

To represent this temporal relation in terms of closely-continuous successors one has only to introduce a single dummy activity D requiring no resources of duration equal to the fixed intervale length "a." Activity D is then made a closely-continuous successor of activity A and B, in turn, is made a closely continuous successor of D. Activity B is made an ordinary successor of activity C. Consider next the case illustrated in Fig. 2.4.34-2, wherein one activity cannot start until a second activity has started.

$s_B \geq s_A$

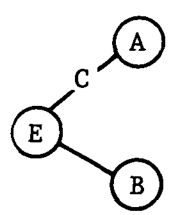


Fig. 2.4.34-2
Sample Representation of a General Temporal Relation Using Closely Continuous Successors

To represent this temporal relation, one need only introduce a single dummy event E. Then activity A is made a closely-continuous successor of event E while activity B is made an ordinary successor.

Although the closely-continuous successor concept provides a generalized network presentation of all of the general temporal relations, no simple heuristic procedure can be devised to schedule such a network. Long multibranch trees of closely-continuous successors of a given activity have to be scheduled before that activity itself can be scheduled. This considerably complicates the resource allocation logic perhaps to the point of diminishing returns. Any complications in a heuristic procedure must be justified by their results. Without establishing the utility of the relatively simple resource allocator for ordinary precedence networks, it seems pointless to build a vastly more complicated allocator for generalized precedence networks. Nonetheless, in Subsection 2.4.34.7, a proof is given that any general temporal relation can be moldeled using only ordinary and closely continuous successors.

This module has the capability of scheduling interfacing subnetworks. It assembles a user supplied master subnetwork and all of its interfacing subnetworks into a master network. All the activities of this master network are to be scheduled subject to common resource availability levels.

A time-progressive heuristic program is used to obtain short, but not necessarily minimal, project durations. The heuristic employs a critical-path-based priority rule tempered by a modifying heuristic using contingency resource thresholds. By utilizing late-start time as the priority value of each activity or event, a dynamic priority function is obtained that does not require updating each time a new acticity is scheduled. This results from the fact that the late-start date of an activity is independent of the actual scheduled start dates of any of its predecessor as long as none of them are delayed beyond its late-start date. Nonetheless, the late-start date does represent a good priority rule in terms of scheduling the least flexible activities first. That unscheduled activity with the earliest late-start date, other factors being equal, is the activity most likely to lengthen project duration beyond the critical-path value. The modifying heuristic is activated whenever an activity cannot be scheduled before its late-start date. The resource that prevents the scheduling of the activity is augmented by a user-input contingency threshold from the time the activity's predecessors were all completed until the activity is successfully scheduled.

Finally, an option is provided for leveling the resource utilization profiles via a least squares heuristic after a tentative initial schedule has been obtained from the late-start-date heuristic. The leveling procedure involves sequentially considering the activities in order of latest scheduled finish. A weighted sum of squares of the resource profiles over time is then computed for each activity for each start date in its residual float. That start date in the float interval is selected that will minimize the weighted resource sum of squares. Two underlying principles motivate this heuristic procedure. First, by sequentially delaying activities considered, in order of their latest scheduled finish, the float of activities with earlier scheduled finishes can only be increased, thereby improving their subsequent scheduling flexibility. Second, the weighted sum of squares of the resource profiles over time is decreased by reducing any jump in the utilization level of any resource from one time interval to the next. In fact, the unconstrained minimum sum of the squares is achieved when all the resource profiles are such that the utilization levels of any given resource in each time period by at most one unit.

2.4.34.2 Modules Called

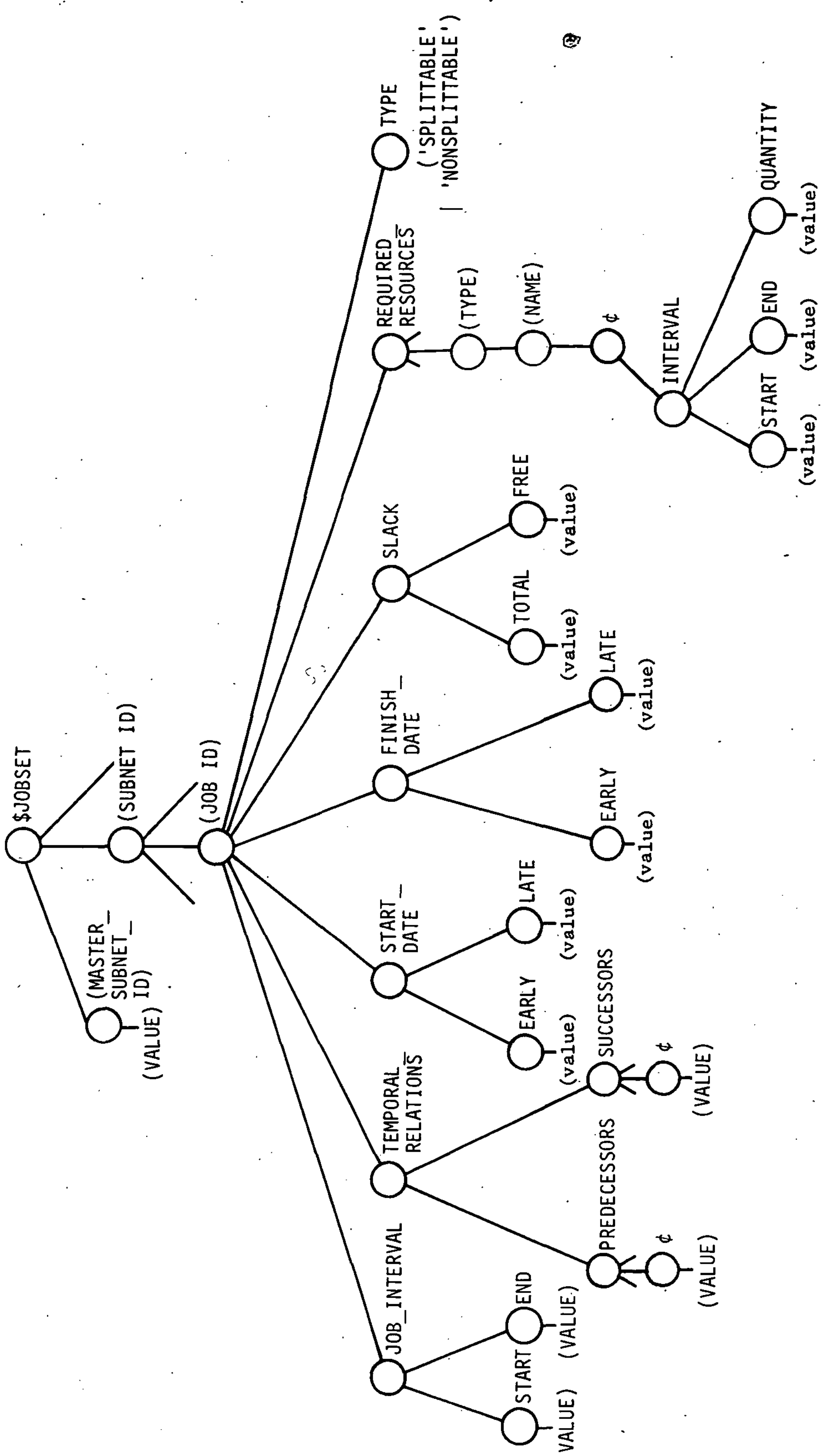
NETWORK_ASSEMBLER

RESOURCE_ALLOCATOR

RESOURCE_LEVELER

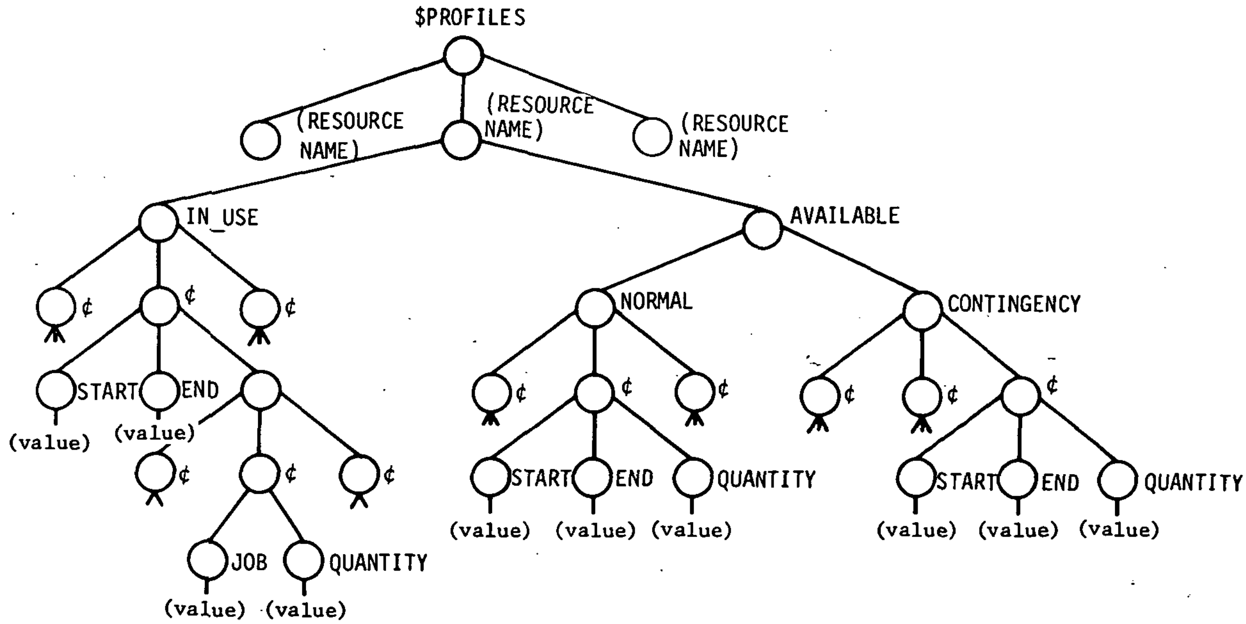
2.4.34.3 Module Input

1) Network, Critical Path Data and Activity or Event Definitions ($JOBSET)

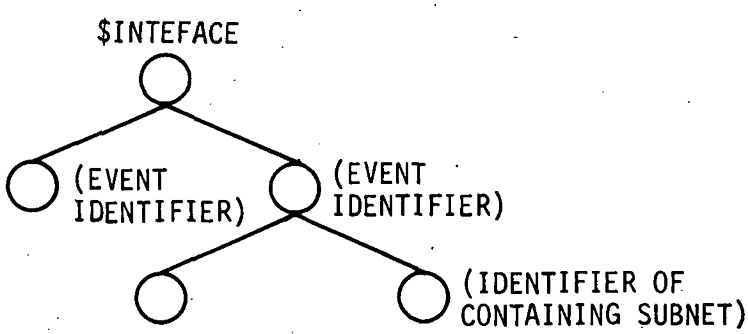


$JOBSET
(SUBNET ID)
(MASTER_ SUBNET_ ID)
(VALUE)
(JOB ID)
JOB_INTERVAL
START
(VALUE)
END
(VALUE)
TEMPORAL_ RELATIONS
PREDECESSORS
¢
(VALUE)
SUCCESSORS
¢
(VALUE)
START_ DATE
EARLY
(value)
LATE
(value)
FINISH_ DATE
EARLY
(value)
LATE
(value)
SLACK
TOTAL
(value)
FREE
(value)
REQUIRED_ RESOURCES
(TYPE)
(NAME)
¢
INTERVAL
START
(value)
END
(value)
QUANTITY
(value)
TYPE
('SPLITTABLE'
| 'NONSPLITTABLE')

2) Resource Definitions ($PROFILES)



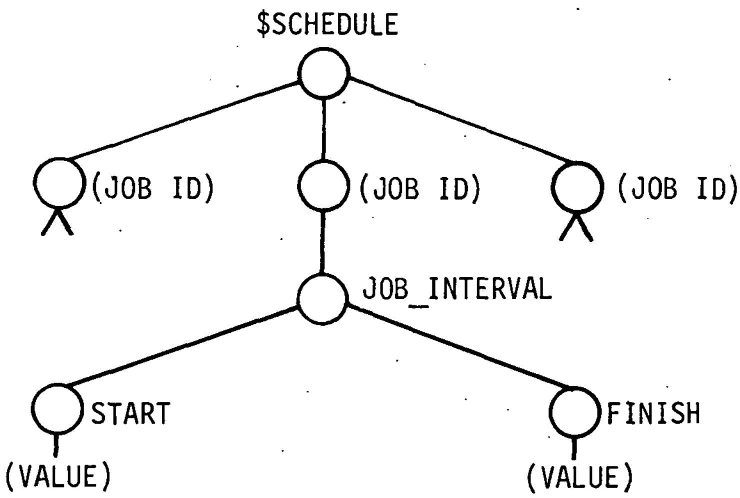
3) Interfacing Event Definitions ($INTERFACE)



4) Resource Leveling Option Indicator (LEVEL)

2.4.34.4 Module Output

1) Resultant Project Schedule ($SCHEDULE)



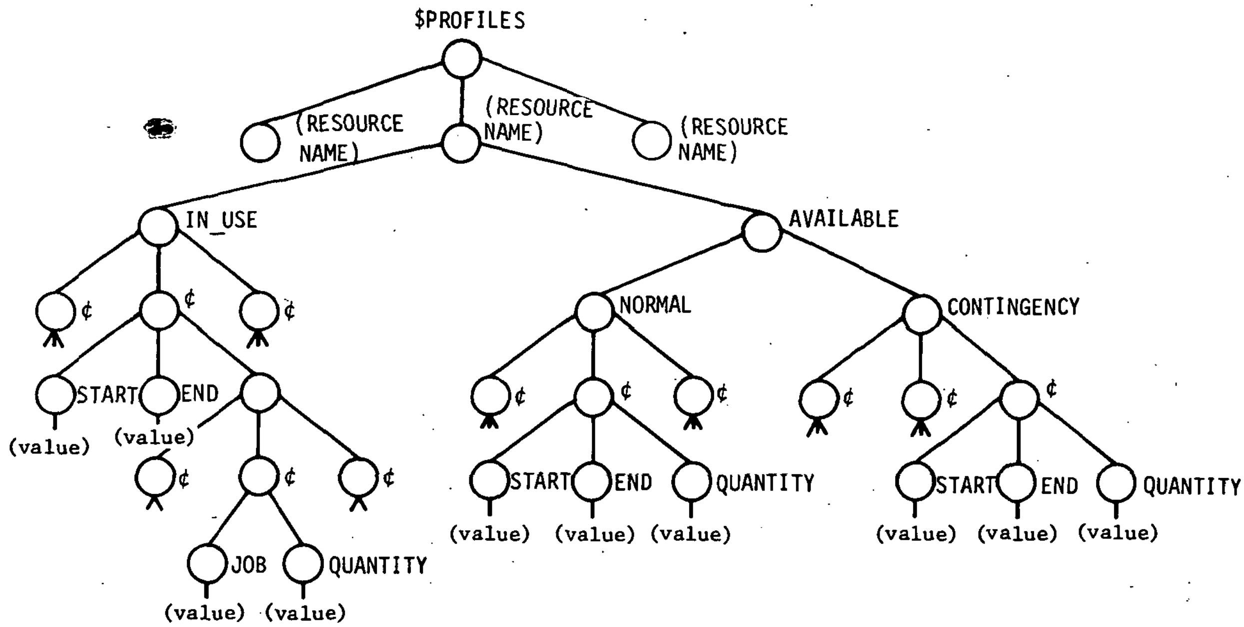
2) Revised Resource Profiles ($PROFILES)

Same as for Module Input.

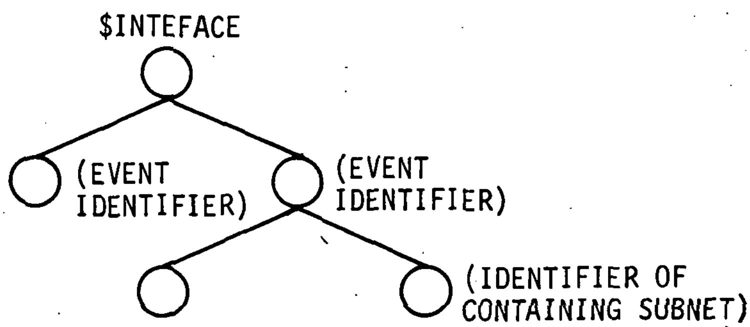
2.4.34.5 Functional Description

The HEURISTIC_SCHEDULING_PROCESSOR serves as an executive procedure for controlling and coordinating the entire heuristic scheduling process. First the network must be built whose activities are to be scheduled sharing the same common resources. By means of a call to the module NETWORK_ASSEMBLER, the user-specified master subnetwork and all of its interfacing subnetworks, as detailed in the interfacing event definitions, are assembled into the desired network. Next, the RESOURCE_ALLOCATOR is called to schedule the activities of the network according to the minimum project duration heuristic procedure described above. Earliest late-start is used as the priority function for each activity.

2) Resource Definitions ($PROFILES)



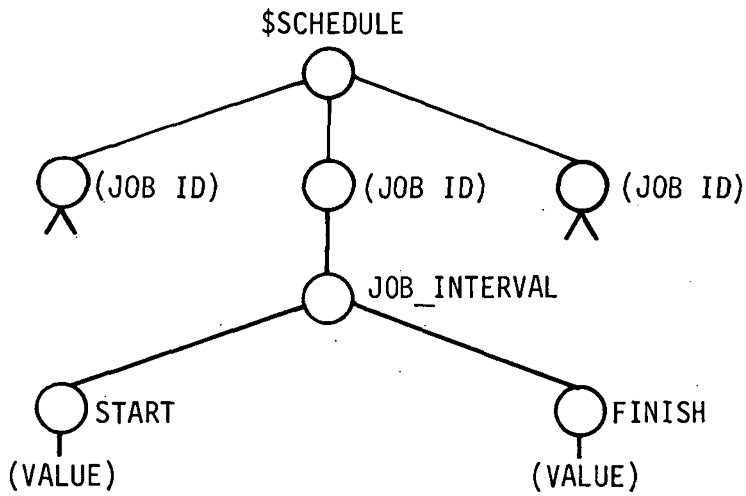
3) Interfacing Event Definitions ($INTERFACE)



4) Resource Leveling Option Indicator (LEVEL)

2.4.34.4 Module Output

1) Resultant Project Schedule ($SCHEDULE)



2) Revised Resource Profiles ($PROFILES)

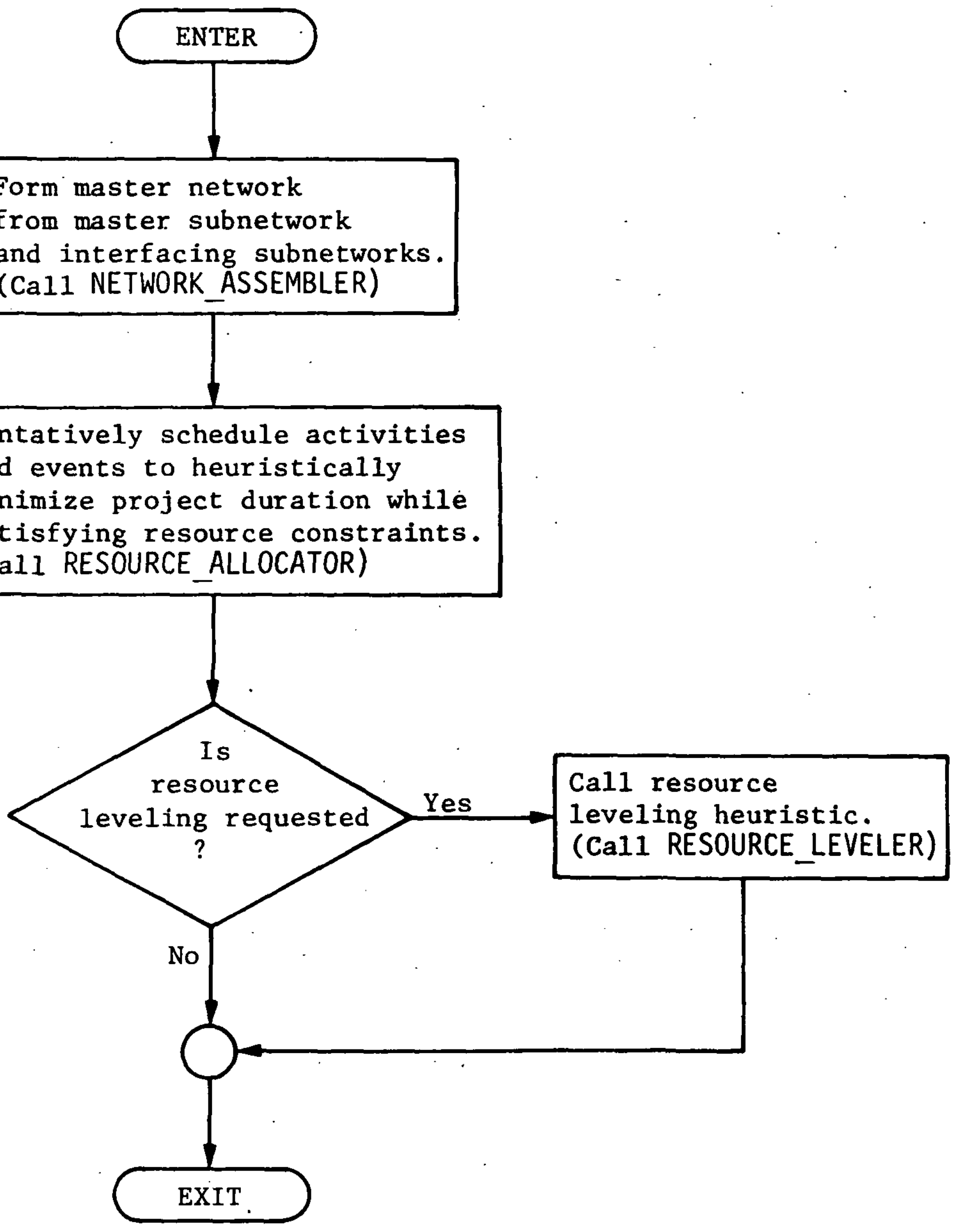
Same as for Module Input.

2.4.34.5 Functional Description

The HEURISTIC_SCHEDULING_PROCESSOR serves as an executive procedure for controlling and coordinating the entire heuristic scheduling process. First the network must be built whose activities are to be scheduled sharing the same common resources. By means of a call to the module NETWORK_ASSEMBLER, the user-specified master subnetwork and all of its interfacing subnetworks, as detailed in the interfacing event definitions, are assembled into the desired network. Next, the RESOURCE_ALLOCATOR is called to schedule the activities of the network according to the minimum project duration heuristic procedure described above. Earliest late-start is used as the priority function for each activity.

If an activity is delayed beyond its late-start date because of a resource shortage, a modifying heuristic is invoked to increase the availability of the deficient resource by a user input contingency threshold. If the user does not request any resource leveling effort by leaving the leveling option indicator, LEVEL, unset, the heuristic scheduling process ends here. Otherwise the module RESOURCE_LEVELER is called to heuristically reduce to a minimum the jumps in the resource utilization rate. The heuristic operates by considering the activities in order of latest scheduled finish. The weighted sum of the resource profiles squares over time is then computed for each possible start time of the activity under consideration within its remaining total float. That start time is selected that minimizes the sum. When all the activities have been considered for delay, the leveling effort is complete and the heuristic scheduling terminates. The simple macrologic for the processor is illustrated in the functional block diagram. More detailed information on the resource allocation and leveling heuristics can be found in the respective specifications for the modules, RESOURCE_ALLOCATOR and RESOURCE_LEVELER.

2.4.34.6 Functional Block Diagram



all the original activities is maintained so that an ordinary predecessor or successor relation can be represented as usual.

References

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

ICT 1900 Series PEWTER (PERT without Tears). ICT Technical Publications Group, London 1967.

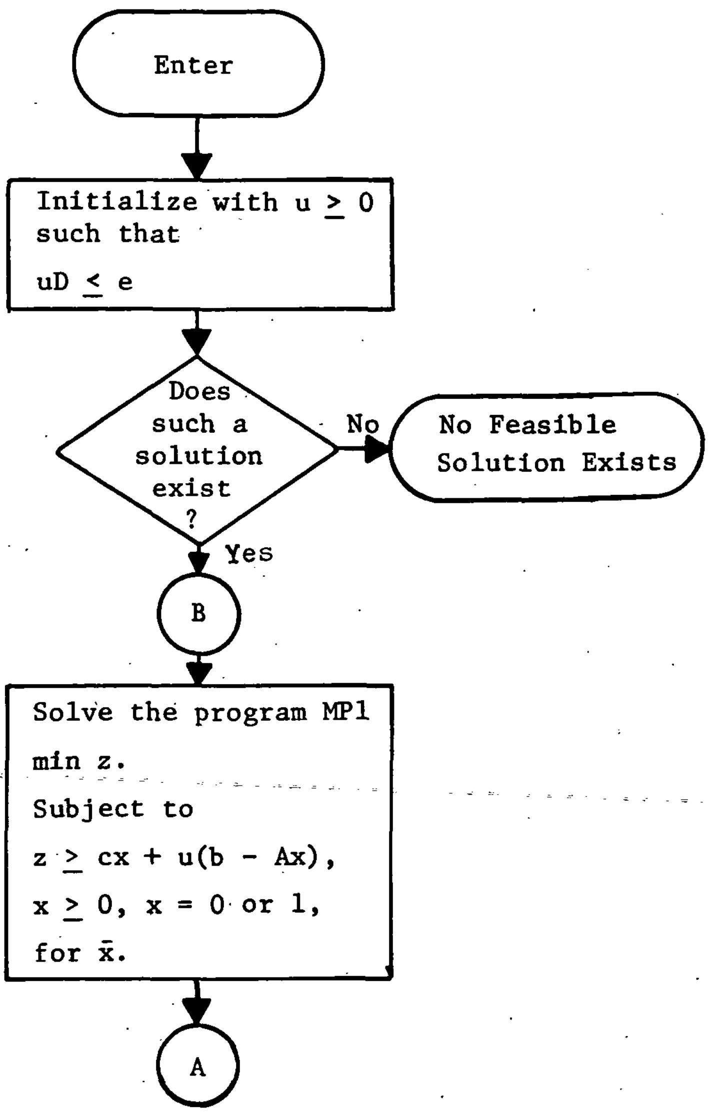
Burman, P. J.: *Precedence Networks for Project Planning and Control.* McGraw Hill, London, 1972.

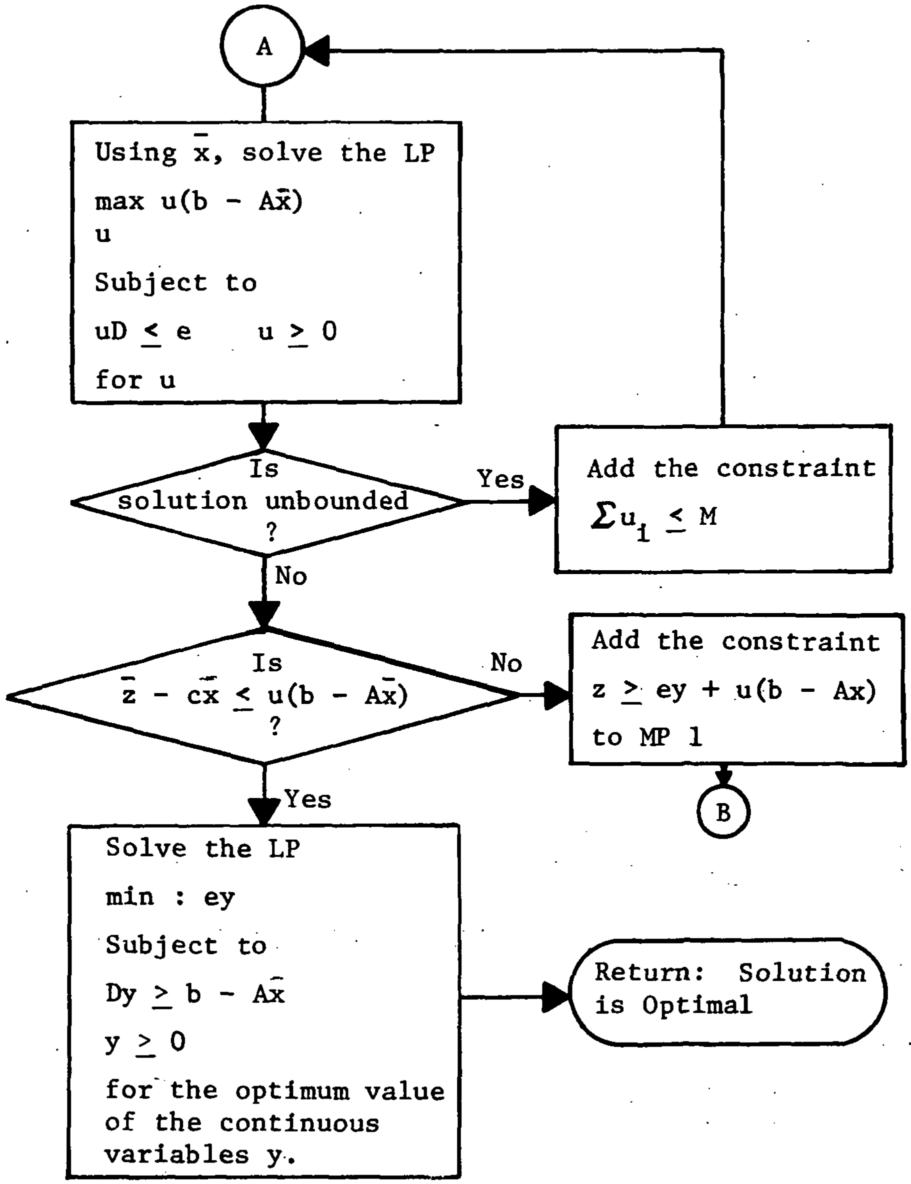
2.4.35 GUB_LP

2.4.35 GUB_LP

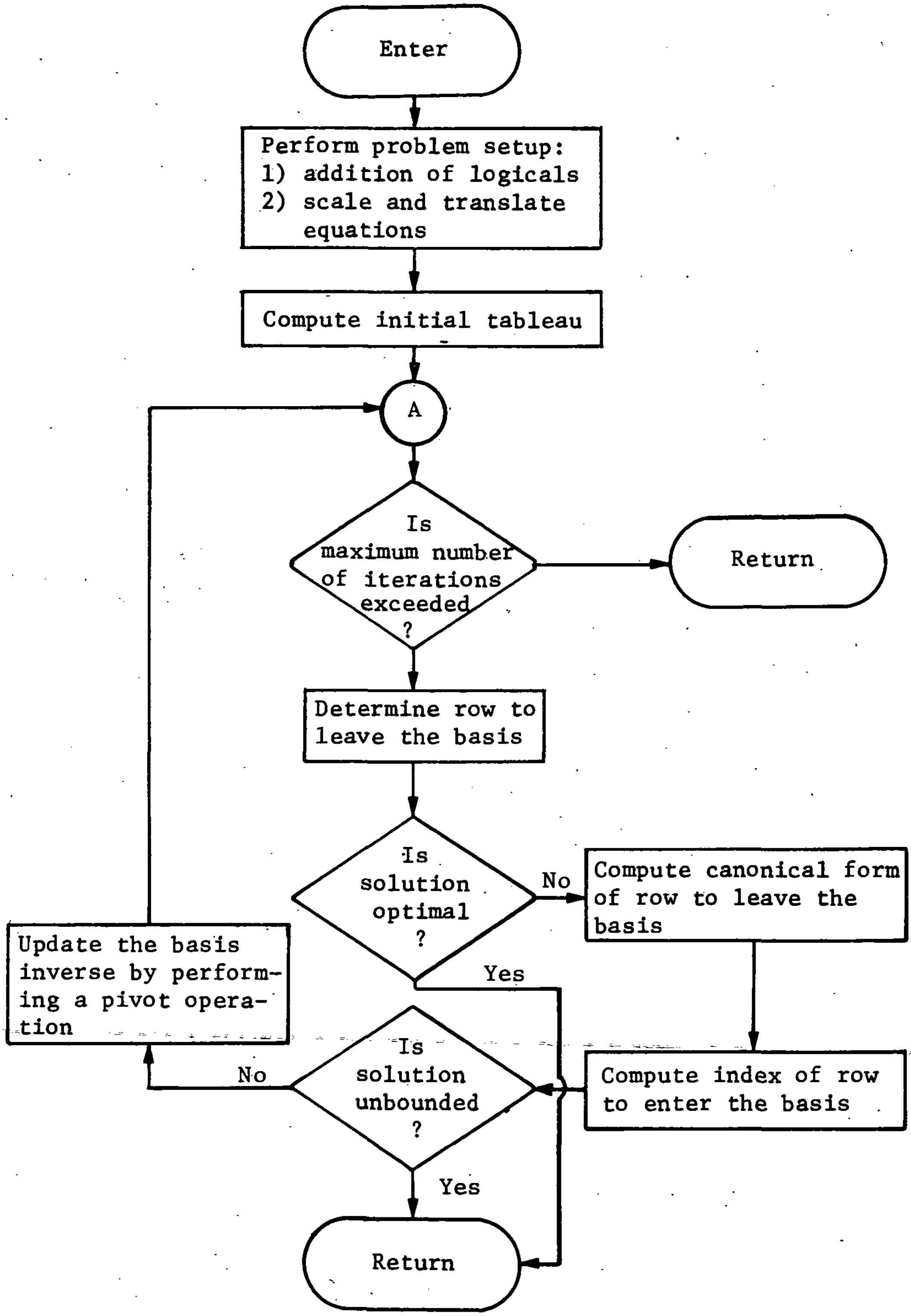
Bender's algorithm makes use of the fact that for given values of x, the problem reduces to an LP whose dual is independent of any particular choice of x. This enables an equivalent program with only one continuous variable to be formulated that can be solved as a subproblem to yield the overall integer solution. A brief description of this approach follows.

2.4.36.6 Functional Block Diagram




A
Using x̄, solve the LP
max u(b - Ax̄)
u
Subject to
uD ≤ e u ≥ 0
for u
Is solution unbounded ?
Yes
Add the constraint
Σu_i ≤ M
No
Is z̄ - cx̄ ≤ u(b - Ax̄) ?
No
Add the constraint
z ≥ ey + u(b - Ax)
to MP 1
B
Yes
Solve the LP
min : ey
Subject to
Dy ≥ b - Ax̄
y ≥ 0
for the optimum value of the continuous variables y.
Return: Solution is Optimal

2.4.38.6 Functional Block Diagram



2.4.38.7 Typical Applications

Dual simplex is generally used as a submodule in other algorithms where the highly specialized advantages of the dual structure can be exploited. For example, dual simplex is used internally in the Benders' decomposition algorithm to solve for the extreme points and rays of the primal problem for a fixed value of the integer variables. The dual is used in this situation because then the constraint set is independent of any particular choice of the integer variables. (For more details, see the description of the Bender decomposition algorithm.) Dual simplex is also used in the Geofferion zero-one algorithm to solve for the strongest surrogate constraint. In both of these examples, dual simplex was used because in the process of solving the master program a subproblem was created that was particularly compatible with the dual algorithm. This is very typical of the situations in which the dual simplex module would be used.

2.4.38.8 Implementation Considerations

A more general dual algorithm could be developed, similar to that described in Ref 3 which handles type 1 variables directly. In this more general setting, the dual algorithm is not the same as the primal simplex applied to the dual problem.

2.4.38.9 References

Lemke, C. E. and Spielberg, K: "Direct Search Algorithms for Zero-One and Mixed-Integer Programming; *Operations Research*, Vol 15, No. 5, 1967.

Lasdon, Leon: "Optimization Theory for Large Systems." *MacMillan Series in Operations Research*. 1970.