MCR-74-314
NAS9-13616

Volume I

Phase 1
Final
Report

September 1974

Study Summary and Overview

Scheduling Language and Algorithm Development Study

(NASA-CR-140281) SCHEDULING LANGUAGE AND ALGORITHM DEVELOPMENT STUDY. VOLUME 1: STUDY SUMMARY AND OVERVIEW Final Report (Martin Marietta Corp.)

N74-337.4

CSCL 09B G3/08

Unclas
50473

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

PRICES SUBJECT TO CHANGE

MARTIN MARIETTA

MCR-74-314
NAS9-13616

Volume I

Phase 1
Final
Report

September 1974

Study Summary and Overview

SCHEDULING LANGUAGE AND ALGORITHM DEVELOPMENT STUDY

DRL T-890
Line Item 3

Approved

John F. Flater

John F. Flater
Program Manager

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

FOREWORD

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

Volume I - Study Summary and Overview

Volume II - Use of the Basic Language and Module Library

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

Appendix - Study Approach and Activity Summary

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

Volume II summarizes the capabilities of the specified scheduling language and the program module library. It is written with the potential user in mind and, therefore, provides maximum insight on how the capabilities will be helpful in writing scheduling programs. Simple examples and illustrations are provided in Volume II to assist the potential user in applying the capabilities of his problem.

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

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

CONTENTS

I. STUDY OBJECTIVE AND REQUIREMENTS

A. STUDY OBJECTIVE

"The objective of the task [study] is to prepare the detailed design of a flexible, user-oriented scheduling language suitable for the resource assignment class of problems represented by the Space Shuttle Transportation System."

This objective for the Scheduling Language and Algorithm Development Study, as quoted from the Statement of Work, was elaborated with a more explicit statement of purpose. This called for design of a scheduling language (software) that can simulate system operation and schedule all resources of an advanced space program, such as the Space Shuttle, where launch frequencies are high, mission objectives are complex, and mission planning and support activities are numerous.

This broad general objective was further amplified by a discussion of problem-related study considerations and a list of language functional requirements in the next section. To establish a proper perspective of the Phase 1 study effort and results, the second chapter of this report explains the motivation for the study with a discussion of "whys" behind the objective and requirements. The third chapter summarizes the main features of the language and applications module library with emphasis on unique study results. Finally, the report concludes with a brief review of expected benefits from use of the language and related problem-solving modules in terms of program implementation efficiency and problem solving capability.

B. STUDY CONSIDERATIONS AND FUNCTIONAL REQUIREMENTS

A number of special considerations related to determining and specifying scheduling applications program modules led to a list of language functional requirements in the Statement of Work. These modules describe the system to be scheduled and mathematically express optimization and resource assignment algorithms used to implement the problem solution/decision logic. A brief discussion of these problem oriented considerations follows.

Many combinations of system resources are used in overall system operation, thus the language must facilitate generation of these feasible combinations. Because any decision or assignment once made affects future decisions, the dynamic aspects of handling many solution combinations, sequential assignments, and constraint sets for decision-tree "pruning" must be considered. These large sets of discrete constraints are inherent in complex operational systems and are usually resource oriented. Some examples are physical dimensions (e.g., payload size and weight, cargo bay parameters, consumables), interface restrictions, performance deficiencies, and mission mode limitations. A language for solution program modules must therefore allow efficient redefinition of data structures and a natural and flexible syntax.

Selected problem solution techniques must schedule system operations optimally over a planning horizon consistent with reliability and availability of planning data. (For Space Shuttle, this may be 100 missions or more.) Such techniques should allow rescheduling for contingencies (e.g. undelivered payload, inoperative payload, or Shuttle) through conflict detection and resolution with minimal effect on future schedules.

The problem-oriented considerations just mentioned became the basis for evolving a number of functional requirements for interfacing the system simulation and optimization modules and facilitating data manipulation and user analyses. These functional requirements state that the final scheduling language design is required to:

- Facilitate generation of feasible resource combinations;
- Have data structures that allow problem redefinition in an efficient manner;
- Provide efficient interface for system simulation modules with assignment and optimization algorithm modules;
- Provide natural and flexible syntax;
- Consider the ease with which an analyst can perform analyses and manipulate problem modules as the prime factor in user-language interfaces;
- Be compatible with data storage and retrieval mechanisms capable of efficiently manipulating a large quantity of information;
- Avoid incompatibility with on-line interactive capability for constraint set manipulation and solution algorithm selection.

Page intentionally left blank

II. STUDY MOTIVATION

The motivation for this study may be explained with a discussion of the "whys" behind the objectives and requirements. The summaries of NASA experience with system scheduling software on previous programs and the factors that contribute to the complexity of scheduling software are presented in this chapter as the basis for development of a software scheduling language.

A. NASA EXPERIENCE WITH SYSTEM SCHEDULING SOFTWARE

A program for system scheduling uses assignment algorithms in various forms to investigate the allocation of system resources. By using feasible combinations of resource elements (e.g. crew, payload, vehicles, time, money, etc) these algorithms make a resource assignment and attempt to optimize some measure of system performance. Examples of this optimization for Space Shuttle include minimization of mission cycle time or minimization of the number of flights needed to orbit given payloads while satisfying system constraints and mission objectives.

NASA has investigated and developed several scheduling programs for the Skylab and Shuttle programs. Among them have been the Experiment Scheduling Program, the Skylab Activity Scheduling Program, and the Operations Simulation and Resource Scheduling Program. These programs were coded in several different digital computer programming languages, such as assembler, FORTRAN, GPSS, SIMSCRIPT, and others. Each program, designed for a specific purpose, experienced severe program redefinition when seemingly minor changes in the system constraints or resource models occurred. The resulting program modification effort was frequently perceived to be increased by programming constraints attributable to the programming language used.

This experience with computer-based scheduling programs led to the conclusion that a more generalized, flexible, and user-oriented scheduling software language tool is needed to simplify development and maintenance of scheduling programs for the Shuttle era. However, determination of what could be done to improve scheduling software required an assessment of the complexity of developing this software before an approach was established.

B. COMPLEXITY OF DEVELOPING SOFTWARE FOR SCHEDULING

Scheduling the operations and resources of the Space Transportation System (STS) will involve a complex mix of tasks. Some individual tasks will be similar to scheduling of other large aerospace programs, but some will be more typical of conventional transportation systems, or even machine shop and construction project scheduling, and resource assignment problems. These and other nonaerospace related scheduling problem solution needs contributed to expression of the objective and functional requirements of the study. Recognition of the broad application and complexity of scheduling software contributes to understanding of the significance of the study results presented in this report.

Software used in a scheduling task will be complicated by the extremely complex activities to be planned and executed in short time intervals as a result of variable and high activity rates or traffic volume. Some resources will enter and leave the system as expendable items; others will be recycled after use. Furthermore, even when the volume of operations is high, the appearance of new resources or the occurrence of special demands cannot, in most cases, be modeled by statistical approximation.

The STS scheduling objectives will be complex composites of minimized development and operating costs, and maximized payload objectives considering complicated system performance and time constraints. Because operations will be scheduled while the Shuttle system is still under development or later undergoing modifications due to technological progress, computer programs will require major changes as the system objectives or operations change. This sensitivity of source computer program code to system model changes also contributes unnecessary complexity to scheduling software development and/or modification.

C. STUDY PREMISE

The premise of this study is that current programming languages make it unnecessarily difficult for the problem analyst to develop and modify scheduling software. Figure 1 illustrates the typical impact of new, complex systems, such as the Space Shuttle, on the development and maintenance cost of scheduling software. Scheduling software development costs increase as the software complexity increases because of traffic volume, recycling of resources, and complex cost-time-performance scheduling objectives. In an environment where the system resources being assigned and the activities being scheduled are frequently altered, or where the scheduling objectives or constraints are often redefined, program maintenance costs caused by program code sensitivity to changes increase rapidly.

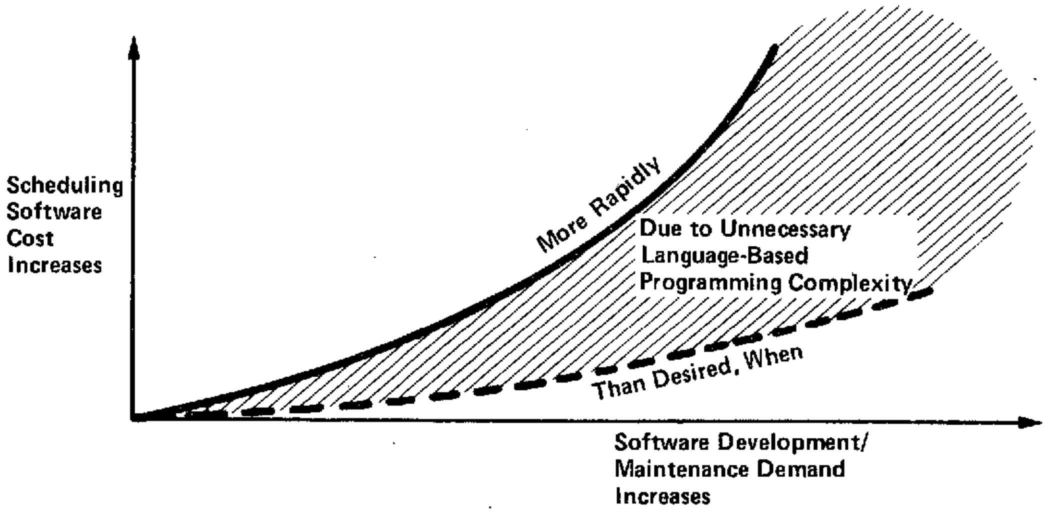


Fig. 1
Impact of System Complexity on Scheduling Software Costs

Unless the programming language permits development of programs that are insensitive to problem changes, maintenance costs (and time) may become excessive. Therefore, a programming language should not only provide high-level capability, but should also be easily readable and understandable to reduce programming time and specialized skills required. Additionally, the language should permit development of programs that are, insofar as possible, independent of problem or application specific details.

When the study was started, no such user-oriented language amenable to solution of resource allocation/scheduling problems existed that:

- Offered substantial reduction of programming and modification times;
- Allowed programs insensitive to problem detail changes;
- Permitted program logic highly independent of problem application;
- Had high-level capability;
- Was easily readable and understandable.

Page intentionally left blank

III. LANGUAGE FEATURES AND UNIQUE STUDY RESULTS

During Phase 1 of the Scheduling Language and Algorithm Development Study, a computer programming language was designed and specified that met the objectives and requirements cited in Chapter I. In addition, a library of modules (subroutines) that take advantage of the special features of the new language to further reduce the impact of the complexities of system characteristics and operations on scheduling software was defined. This chapter summarizes some of the primary language features and unique study results. The reader who desires more detailed information will find the formal specifications for the language and library modules in Volume III and a user's guide in Volume II of this report.

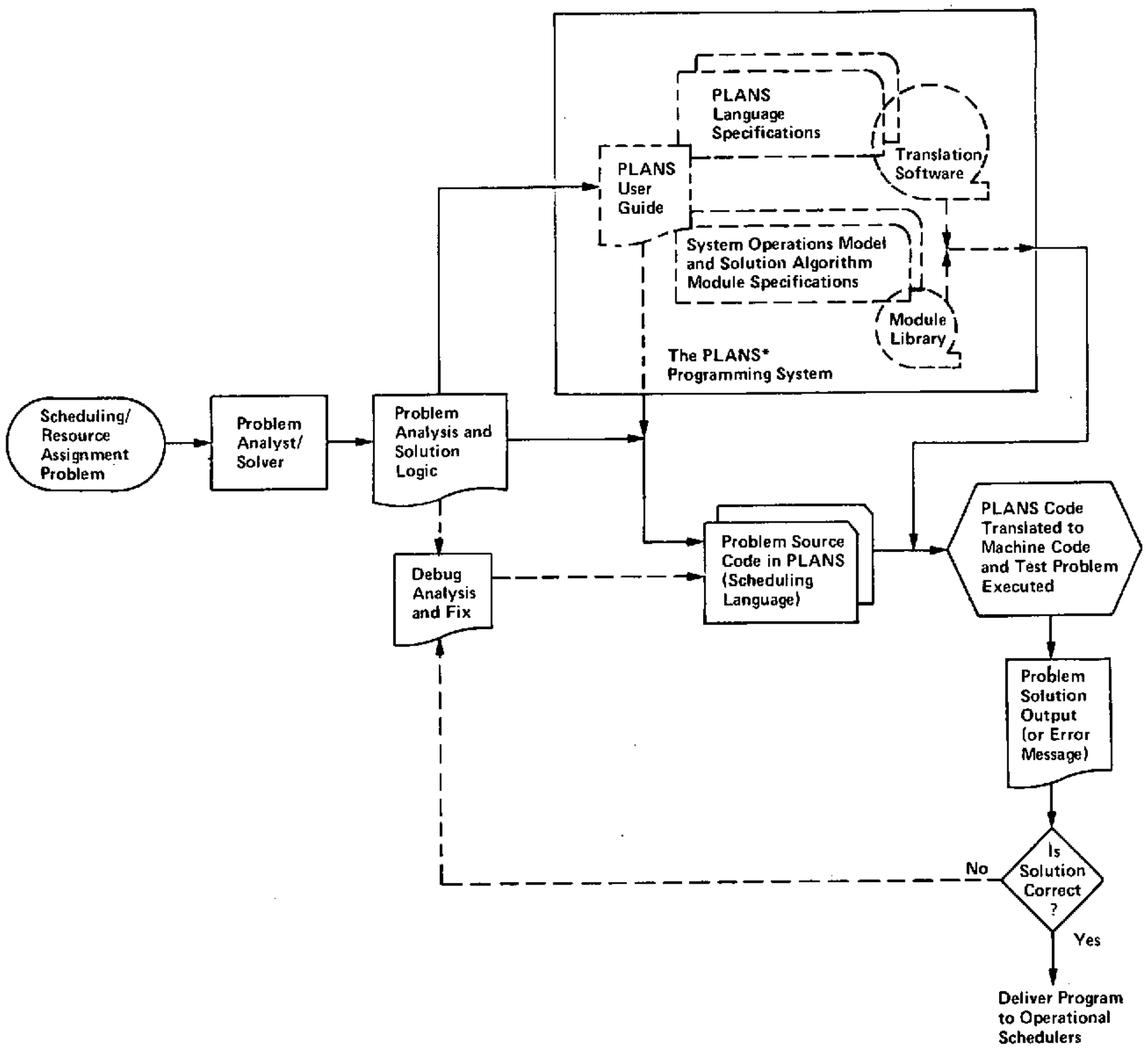
A. A PROGRAMMING LANGUAGE FOR DEVELOPING SCHEDULING PROGRAMS

The final products of this Phase 1 study are detailed functional specifications for a programming language and routine (module) library designed to develop programs related to scheduling or planning problems. These specifications are in Volume III of this report. For conciseness, the language product is called PLANS (Programming Language for Allocation and Network Scheduling). As a programming language, PLANS incorporates a fairly complete set of ordinary arithmetic, transfer-of-control, conditional and iterative statements that are treated essentially in the same way as in the PL/I language. For reasons of early and economical implementation discussed in an Appendix to this set of report Volumes, a PLANS-to-PL/I translator was specified instead of a direct PLANS-to-machine code compiler. Therefore, the basic PLANS syntax and hierarchic block structure are similar to those of PL/I. A summary of unique PLANS features and capabilities are contained in the following pages.

Specifically, this study has not developed complete scheduling system application programs or specified a language in which a user communicates with a scheduling system. As illustrated in Figure 2, PLANS users are assumed to be charged with the design or modification of application programs related to scheduling and resource assignment (allocation), that could be part of a scheduling system. Also, potential users are assumed to have a problem orientation (as opposed to computer programming orientation) that is not limited to aerospace system applications. Thus, a language and associated basic data structure and routines have been specified to provide high-level but flexible programming capability to analysts in a wide variety of scheduling and resource allocation problems.

B. THE PLANS PROGRAMMING SYSTEM

In addition to the mainly conventional arithmetic, transfer-of-control, conditional and iterative statements found in most programming languages, other capabilities were needed if PLANS was to meet the functional requirements previously listed. To make PLANS more usable to the problem analyst, the power of individual statements was increased considerably above that of most general-purpose programming languages. The study showed that increasing correspondence between the individual logical operations of the problem solution and the individual computer program


PLANS Language Specifications
Translation Software
PLANS User Guide
System Operations Model and Solution Algorithm Module Specifications
Module Library
The PLANS* Programming System
Scheduling/ Resource Assignment Problem
Problem Analyst/ Solver
Problem Analysis and Solution Logic
Debug Analysis and Fix
Problem Source Code in PLANS (Scheduling Language)
PLANS Code Translated to Machine Code and Test Problem Executed
Problem Solution Output (or Error Message)
Is Solution Correct ?
No
Yes
Deliver Program to Operational Schedulers

*PLANS = Programming Language for Allocation and Network Scheduling

Fig. 2
Scheduling Language Application Role

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

statements, greatly increased the inherent usability of the language for the user concerned with scheduling/resource allocation problems. However, practical limits to doing this in the basic language must be recognized, because too much individual statement power would unnecessarily reduce the flexibility of the language.

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

In summary, these products are :

1) A high-level programming language for writing scheduling programs that
   - Use typical arithmetic, transfer-of-control, conditional and iterative statements in logic and computational modules,
   - access and manipulate the data structure for problem/module support,
   - define the problem/objectives and manipulate the library modules;
2) A flexible data structure specially suited for describing the operating systems to be scheduled;

3) A library of preprogrammed logic modules to

   - access the data structure for system operations data,
   - implement frequently used scheduling/resource allocation problem solution algorithms.

This programming system meets the prime requirements for (1) substantially reducing software programming and reprogramming times, (2) desensitizing programs to problem changes, and (3) accommodating a wide range of problem types and applications with generic logic codes. A summary of the features of the major components of this programming system is given in the following sections. More detailed descriptions will be found in report Volume II with several example applications.

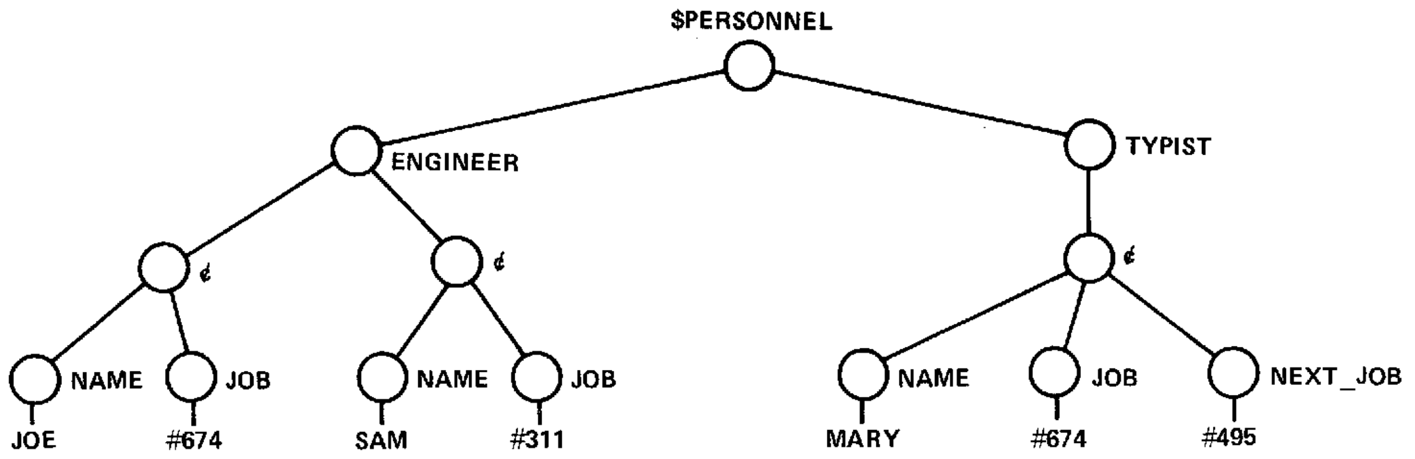
1. The Plans Programming System Data Structure

An initial study task defined a basic structure within which language functional requirements could be developed. Analysis of many types of scheduling problems revealed that all information could be, and most system information is naturally, hierarchically related. Also, time intervals that have a special place in scheduling problems, could be handled without difficulty by using a small number of interval subroutines. Thus, specification of the hierarchical or tree data structure in Volume III as the only structure required for scheduling/resource assignment problems provides logical simplicity and much greater economy of implementation than would result from multiple data types.

The principal difference between PLANS and most other programming languages is that PLANS is oriented primarily toward manipulation of ordered, labeled tree structures as illustrated in Figure 3. Both graphical and textual conventions for representing labeled tree structures were developed. The branching points are called *nodes*, and the *root node* is depicted at the top of the tree. Nodes are represented by circles, and in text by line indentation. The *name* of the tree is shown above the root node. The character $ identifies tree names so the translator software can discriminate them from variable names. Each node has a *label* (character string to the right), but the label may be *null* (indicated by the character ¢, in text and sometimes in graphical format). Nodes with descendants are *nonterminal* nodes and those without are *terminal* nodes. In addition to a label, a terminal node has a *value*, either a character string or a numeric. Graphically, values are depicted below their terminal nodes, and like labels, may also be null.

2. PLANS Language Data Structure Access and Manipulation

Although some other languages have hierarchical data structures, PLANS provides special data access and dynamic manipulation capabilities for its tree-type structure. It can generate and alter tree structures and access the contents of the structures either by key word (label) or by ordinal position (index). The full discussion of the extent and use of these features exceeds the scope of this summary, but the examples here and in Volume II give some indication of their importance.



Note: The character $ is a prefix to identify the label of a data tree root node. Reference to $PERSONNEL in a PLANS program refers to the root node label and all data in the tree; thus

```
$PERSONNEL
  ENGINEER
    ¢
       NAME – JOE
       JOB – #674
    ¢
       NAME – SAM
       JOB – #311
  TYPIST
    ¢
       NAME – MARY
       JOB – #674
       NEXT_JOB – #495
```

In a loose definition $PERSONNEL may be referred to as the data set for PERSONNEL.

The character ¢ indicates a null label.

Fig. 3
PLANS Programming System Data Structure Example

Access Features

Access features are based on the notion that the programmer can "point" to a particular tree node by specifying the tree name, the immediate descendent of the root node it is under, the immediate descendent of *that* node it is under, etc. For example, in Fig. 4a. if the tree name is $PAYLOAD and the name of the payload is TELESCOPE, the analyst could write $PAYLOAD.TELESCOPE to access data about a telescope payload. This is qualification by label. Or, qualification can be done by position, using a subscript notation like some other programming languages. Since PLANS trees are *ordered* trees, the ordering of a node's descendants is significant and unless purposely reordered, the tree structure order remains constant. Thus, the telescope payload might also be referred to as $PAYLOAD(2) if it is the second in order of the first level subnodes of $PAYLOAD.

Other tree structure access statements include the keywords LAST (Fig. 4.a), ALL (Fig. 4.c), FIRST (Fig. 4.b). The update statement, NEXT, can add a new node to a tree without explicitly identifying the node subscript. Each of these may be coupled with conditional algebraic and logical (Boolean) expressions.

Manipulation Features - The primary novel feature of PLANS, its dynamic tree manipulation capability warrants more detailed description. These data tree manipulation statements may be applied to entire trees or tree branches. The basic tree manipulation statement is the assignment statement with fundamental form of $TREE_A = $TREE.B;. This is essentially the same as an

Example features are:

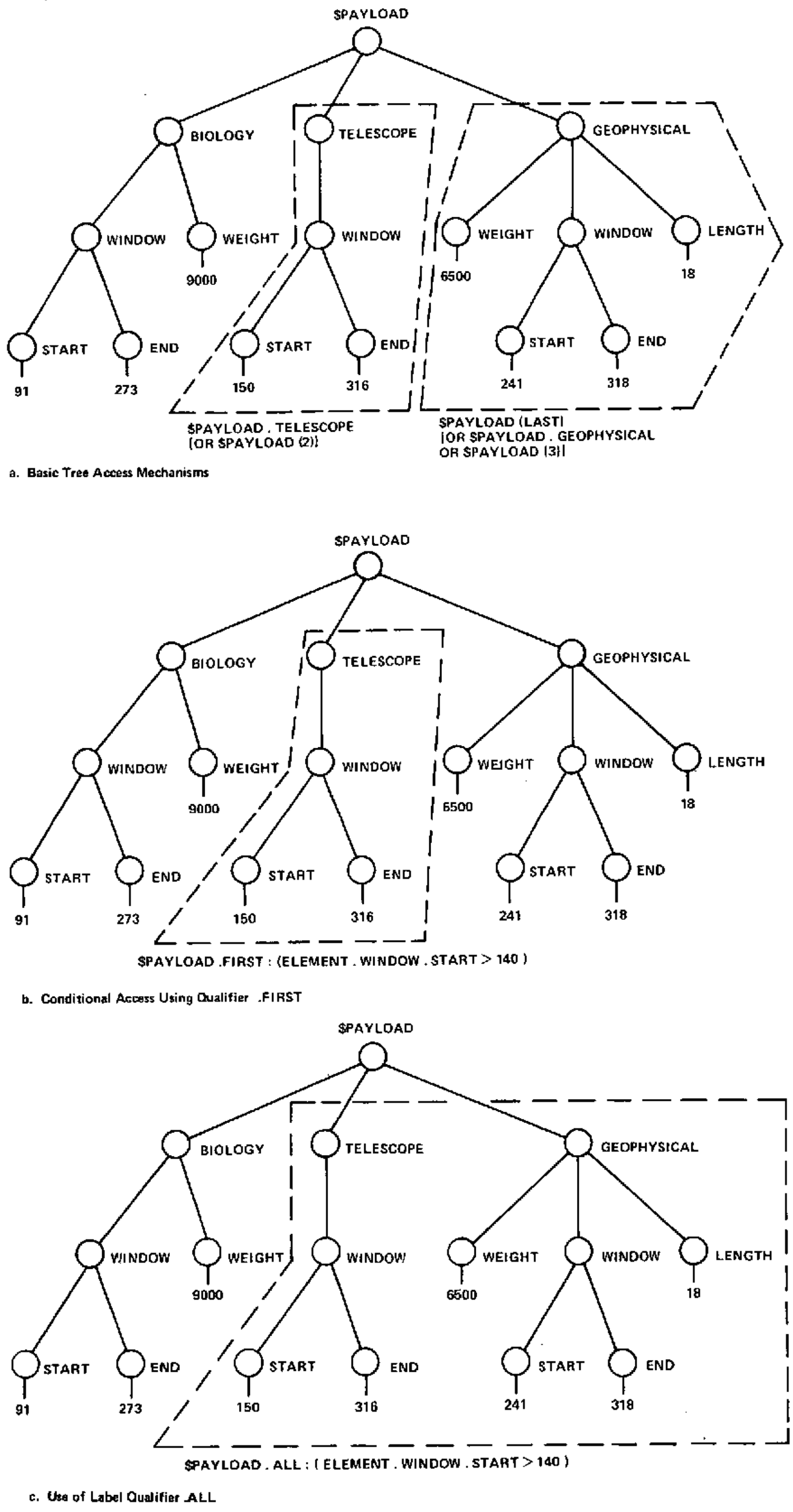

$PAYLOAD
BIOLOGY
TELESCOPE
GEOPHYSICAL
WINDOW
WEIGHT
LENGTH
START
END
9000
6500
18
91
273
150
316
241
318
$PAYLOAD . TELESCOPE (OR $PAYLOAD (2))
$PAYLOAD (LAST) (OR $PAYLOAD . GEOPHYSICAL OR $PAYLOAD (3))
a. Basic Tree Access Mechanisms
$PAYLOAD .FIRST : (ELEMENT . WINDOW . START > 140)
b. Conditional Access Using Qualifier .FIRST
$PAYLOAD . ALL : (ELEMENT . WINDOW . START > 140)
c. Use of Label Qualifier .ALL

Fig. 4
PLANS Language Data Structure Access Feature Example

ordinary arithmetic assignment statement. The "content" of the variable (in this case a tree node) named to the left of the equal sign is destroyed and replaced with a copy of the content (in this case potentially an entire substructure) of the variable (or expression) on the right which is unchanged. Thus, as shown in Fig. 5a. the statement $TREE.B = $TREE.D; replaces the substructure of the tree branch beginning with node B with the substructure of the tree branch beginning with node D in the same tree, while leaving the latter branch intact.

Several other statements, as illustrated in Fig. 5b. through 5e., allow removal of a node with its substructure (basic form: PRUNE $TREE;), removal of a structure from its original tree to be used to replace structure of another tree (basic form: GRAFT $TREE_Y(1) AT $TREE_X(3);), insertion of a copy of a structure in a given place (basic form: INSERT $TREE_Y.C AT $TREE_X(3);), or removal of a structure from its original tree and insertion at a given place in another (basic form: GRAFT INSERT $TREE_Y(1) AT $TREE_X(3);).

The tree structures can be input and output and passed to subroutines as parameters. A provision for implicit type conversion allows tree references to be used in arithmetic expressions (XVAR = $TREE.A.X + 4;) and allows arithmetic expressions in tree contexts ($TREE.B.Y + $TREE.B.Y - 6;). Further explanation and examples of tree manipulation are presented in Volume II of the report.

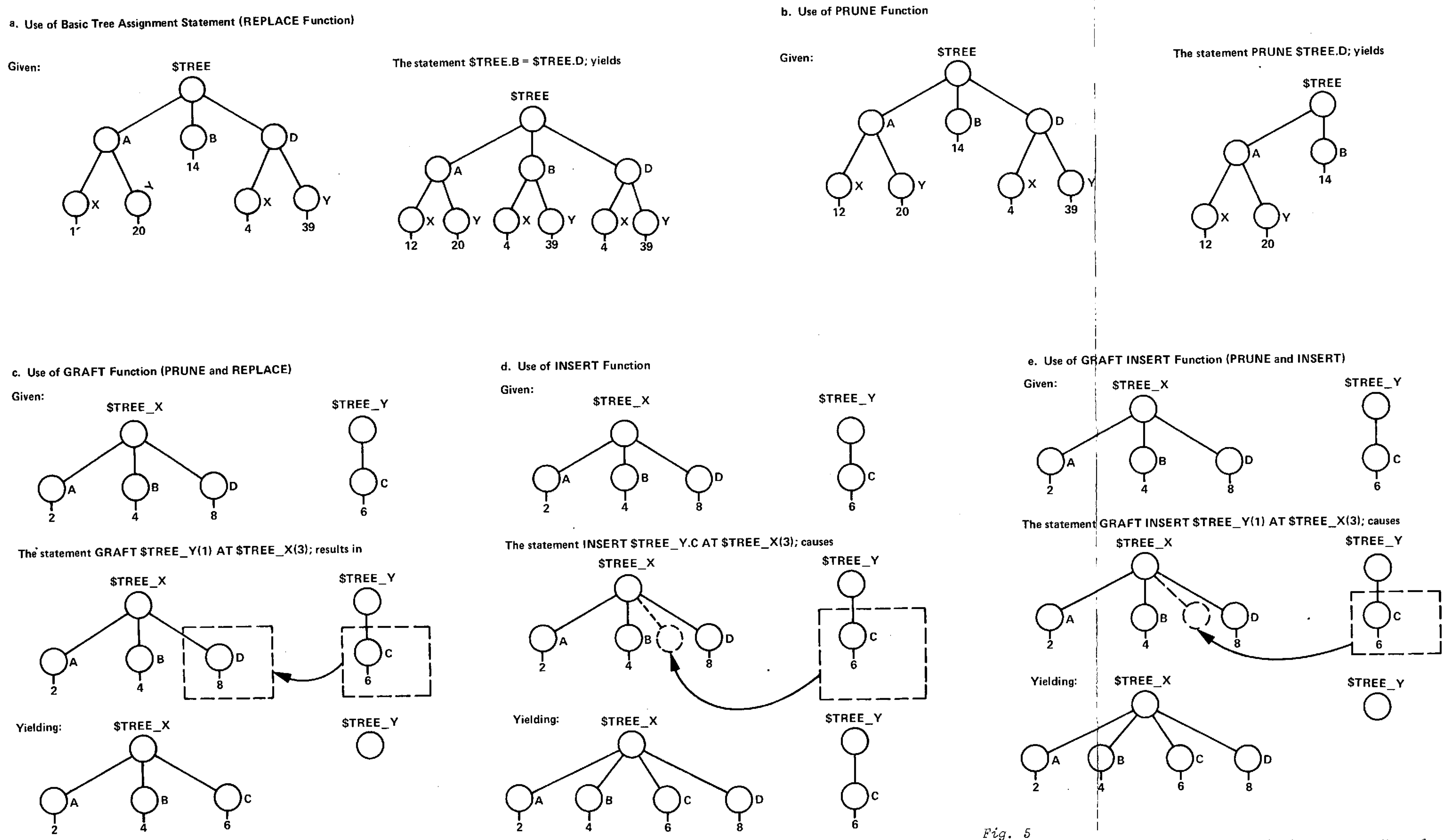

a. Use of Basic Tree Assignment Statement (REPLACE Function)
Given:
$TREE
The statement $TREE.B = $TREE.D; yields
b. Use of PRUNE Function
Given:
The statement PRUNE $TREE.D; yields
c. Use of GRAFT Function (PRUNE and REPLACE)
The statement GRAFT $TREE_Y(1) AT $TREE_X(3); results in
Yielding:
d. Use of INSERT Function
The statement INSERT $TREE_Y.C AT $TREE_X(3); causes
e. Use of GRAFT INSERT Function (PRUNE and INSERT)
The statement GRAFT INSERT $TREE_Y(1) AT $TREE_X(3); causes
$TREE_X
$TREE_Y
A
B
C
D
X
Y
12
20
14
4
39
2
4
6
8

Fig. 5
PLANS Language Data Structure Manipulation Feature Examples

3. The PLANS Programming System Operations Model Modules

Use of the PLANS data structure to describe both the physical nonprocedural and procedural elements (i.e., resources, processor activities, and operations sequences) of the system to be scheduled was analyzed. It was decided that the information could be organized into three tree structures called $RESOURCE, $PROCESS, and $OPSEQ. Standard data structures were defined for each of these system element classifications. The standard data structures that describe the system to be scheduled and the classification of scheduling problems according to functional characteristics, together comprise a scheduling operations model concept that is a unique product of this study. For logical simplicity, the operations model is considered as (1) the operations model data structures and (2) nondecision making data access and manipulation functions required to synthesize a schedule.

The operations model module library is designed in the context of this operations model; thus, its utility is much greater than if it were merely a collection of logical programs. The operations model provides a conceptual framework within which very general scheduling problems can be parametrically described. The analyst who uses this framework will find a large number of routines that can be applied with a minimum of custom-made logic. Since nothing in the basic PLANS language and programming system dictates the use of the operations model, no programming flexibility is lost. Yet, the operations model concept greatly augments the utility of

Preceding page blank

the module library. A few examples of the operations model modules specified in Volume III are summarized in Table 1.

Table 1. Example of PLANS Programming System Operations Model Modules

NAME	FUNCTIONAL SUMMARY
CHECK_EXTERNAL_TEMPORAL_RELATIONS	Identifies temporal constraint violations that would occur if two sets of job assignments were merged. Useful for checking if a potential resource assignment will be consistent with existing resource assignments.
RESOURCE_PROFILE	Determines the profile of a specific resource pool over a given time interval for both a "normal" and "contingency" level of resource. Determines the profile of the assigned portion of the pool and defines the association of jobs making up the usage profile.
NEXTSET	Determines a set of specific resource items to meet the requirements of a job and permit the earliest possible execution of that job. Determines future times the job requirements can be met with any combination of needed resources.
DESCRIPTOR_PROFILE	Determines the descriptors for an item-specific resource that are valid after jobs involving the resources are scheduled. Uses the assignment information in $RESOURCE to determine the descriptor set at a particular time.
UPDATE_RESOURCE	Records the scheduling of a job by writing the assignments in $RESOURCE for all resources used.

4. The PLANS Programming System Problem Solution Algorithm Modules

The module library specified to aid the analyst in programming resource allocation and scheduling logic is unique. Both mathematical programming routines and project scheduling heuristics are included and no single collection of routines is known that incorporates both problem solving approaches. Although the library algorithms are not inventions of this study, a significant amount of analysis verified each one's relevance to realistic scheduling and resource allocation problems. Analyses of state-of-the-art automated scheduling techniques led to the conclusion that both problem modeling methodologies of mathematical programming and heuristic (network and project) scheduling should be included. Each has problem models that have limited generalities; however, others can be solved by building problem-dependent logic. While PLANS can aid this modeling, an analyst will achieve more rapid program development and checkout if he can describe his problem in terms of library modules.

These modules make simple decisions and logical data manipulations based on quantitative criteria easily perceived by the user. Each module specified avoids judgments or logic for which the criteria are open to opinion. For example, no modules assume specific economic models, queue service policies, or criteria for resolving resource alternatives. Also, there are no approximations of dependent variables by polynomials or piece-wise linear functions buried in the logic. These judgmental matters are too problem-dependent and inflexible for an initial library specification. No implication is intended that future modules should be restricted by these criteria, since analyses are underway that will lead to specification of higher-level modules. Examples of the modules currently specified in Volume III are summarized in Table 2.

Table 2. Example of PLANS Programming System Problem Solution Algorithm Modules

NAME	FUNCTIONAL SUMMARY
CRITICAL_PATH_PROCESSOR	Condenses, merges and computes critical path data for a master network. Performs executive function which calls NETWORK_CONDENSER, CONDENSED_NETWORK_MERGER and CRITICAL_PATH_CALCULATOR. Based on network scheduling problem model.
RESOURCE_ALLOCATOR	Allocates resources to jobs to satisfy all resource constraints and heuristically produce a minimum duration schedule. Based on project scheduling problem model.
RESOURCE_LEVELER	Reallocates resources to smooth the usage of resources while maintaining schedule constraints. Based on project scheduling problem model.
HEURISTIC_SCHEDULING_PROCESSOR	Performs both time-progressive resource allocations/job scheduling and resource leveling. Based on project scheduling problem model.
GUB_LP	Solves special-purpose linear programs that arise as simplified models of transportation, distribution, and multi-item scheduling problems. Uses generalized upper bounding LP format.

C. PLANS SCHEDULING PROGRAM LOGIC CONCEPTS

Because of the generic nature of PLANS programming system library modules, it is logical to inquire about use of the modules in a specific problem solution. Several specific scheduling problems were analyzed to identify model/algorithm interface. A typical example is the macrologic shown in Fig. 6 as used to interface the operations model modules and a time-progressive heuristic algorithm like the OSARS (Operations Simulation and Resource Scheduling) program used by NASA JSC/MPAD as a prototype program for building flight schedules.

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

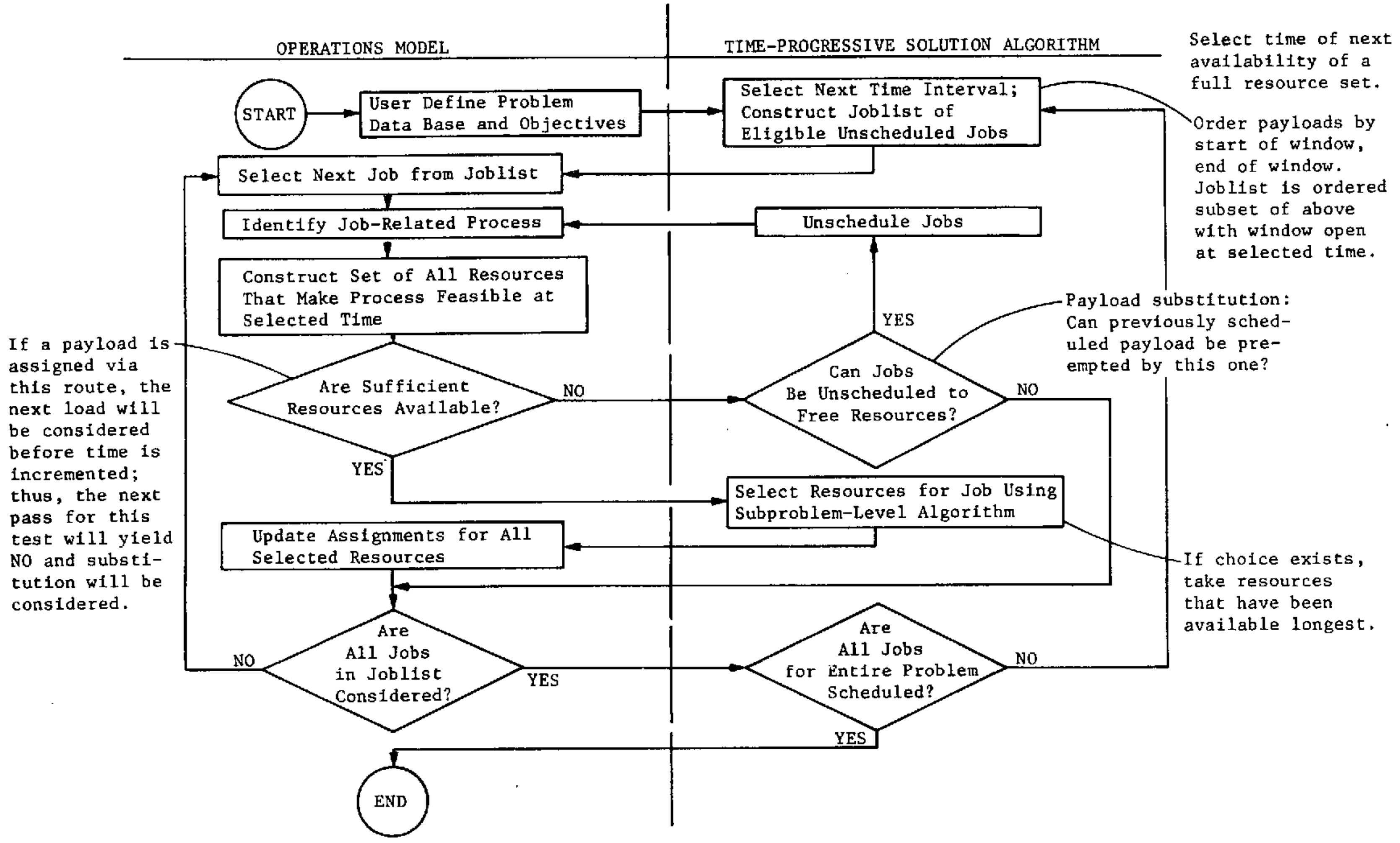

OPERATIONS MODEL
TIME-PROGRESSIVE SOLUTION ALGORITHM
START
User Define Problem Data Base and Objectives
Select Next Time Interval; Construct Joblist of Eligible Unscheduled Jobs
Select time of next availability of a full resource set.
Order payloads by start of window, end of window. Joblist is ordered subset of above with window open at selected time.
Select Next Job from Joblist
Identify Job-Related Process
Unschedule Jobs
Construct Set of All Resources That Make Process Feasible at Selected Time
YES
Payload substitution: Can previously scheduled payload be preempted by this one?
If a payload is assigned via this route, the next load will be considered before time is incremented; thus, the next pass for this test will yield NO and substitution will be considered.
Are Sufficient Resources Available?
NO
Can Jobs Be Unscheduled to Free Resources?
NO
YES
Select Resources for Job Using Subproblem-Level Algorithm
Update Assignments for All Selected Resources
If choice exists, take resources that have been available longest.
NO
Are All Jobs in Joblist Considered?
YES
Are All Jobs for Entire Problem Scheduled?
NO
YES
END

Fig. 6
PLANS Program Logic Concept Example with OSARS Annotation

In this example, the user begins by defining the problem objectives and related constraints in a data base structure ($OBJECTIVES). The operations model executive logic selects the specified systems operations sequence from $OPSEQ and passes it with problem constraints to the algorithm. From the processes in the operations sequence, the algorithm constructs an ordered list of jobs it wishes to schedule. The model recognizes a job as a process for one-time execution of an operation or activity with specific resources, provides algorithm information about resource requirements (from $PROCESS) and resource availability (from $RESOURCE), and receives algorithm job scheduling decisions.

With these conceptual distinctions, a simple approach to developing scheduling/resource allocation program logic evolved. The roles of the model and the algorithm(s) can be seen as a dialog: the algorithm(s) request problem-oriented information about a system and its operations on which to base scheduling decisions and the operations model supplies and updates the data. More detailed discussion of program logic concepts is contained in Volume II of the report.

D. PLANS LANGUAGE SPECIFICATIONS

A unique means for concisely specifying the syntax and semantics of a digital computer programming language was developed to facilitate PLANS implementation. Language *syntax* defines the rules for combining language elements to form language statements. Most programming language syntaxes, including PLANS, can be defined with formal notational techniques such as the often used

Backus-Naur Form. Thus PLANS syntax could be concisely and unambiguously specified with existing techniques; however, the specification of *semantics*, i.e., the meaning of the language elements and statements, presented a different problem. Semantic specifications are frequently written in English text that describes what is supposed to happen when language statements are executed. Implementation from English text specifications has historically encountered difficulties of ambiguity, lack of conciseness, and logical inconsistencies. Therefore, a technique was sought to make PLANS semantic specifications as precise as the syntactic specifications.

A decision was made to embed the semantics into the syntactic specification using a conceptual device, called a pseudomachine, which could respond to simple commands. The semantics of PLANS statements were then uniquely defined in terms of these commands, in conjuction with the syntax of PLANS statement. Thus, the pseudomachine commands that convey the meaning of a PLANS statement, correspond to the syntactical structure and once the statement syntax is recognized, the semantics are known unambiguously. The formal language specifications in Volume III employ this method.

Use of the conceptual pseudomachine commands for embedded semantic specification within the syntactic specification is unique and provides several advantages. First, this technique allows the preparation of formal and concisely written specifications with less effort than previous methods. For example, the HAL/S language specifications required about 250 pages; for PLANS it will be approximately one-tenth of that amount. The technique also avoids implementation errors; thus it saves implementation manpower and time because of the unambiguous meanings. Finally, it is possible to develop a computer program that "emulates" the conceptual pseudomachine. By manually translating PLANS code into the pseudomachine commands, a form of "execution" can be achieved using the emulator. This technique provides a means of debugging PLANS applications programs before the final PLANS translator implementation is complete.

IV. Expected Benefits from Language Use

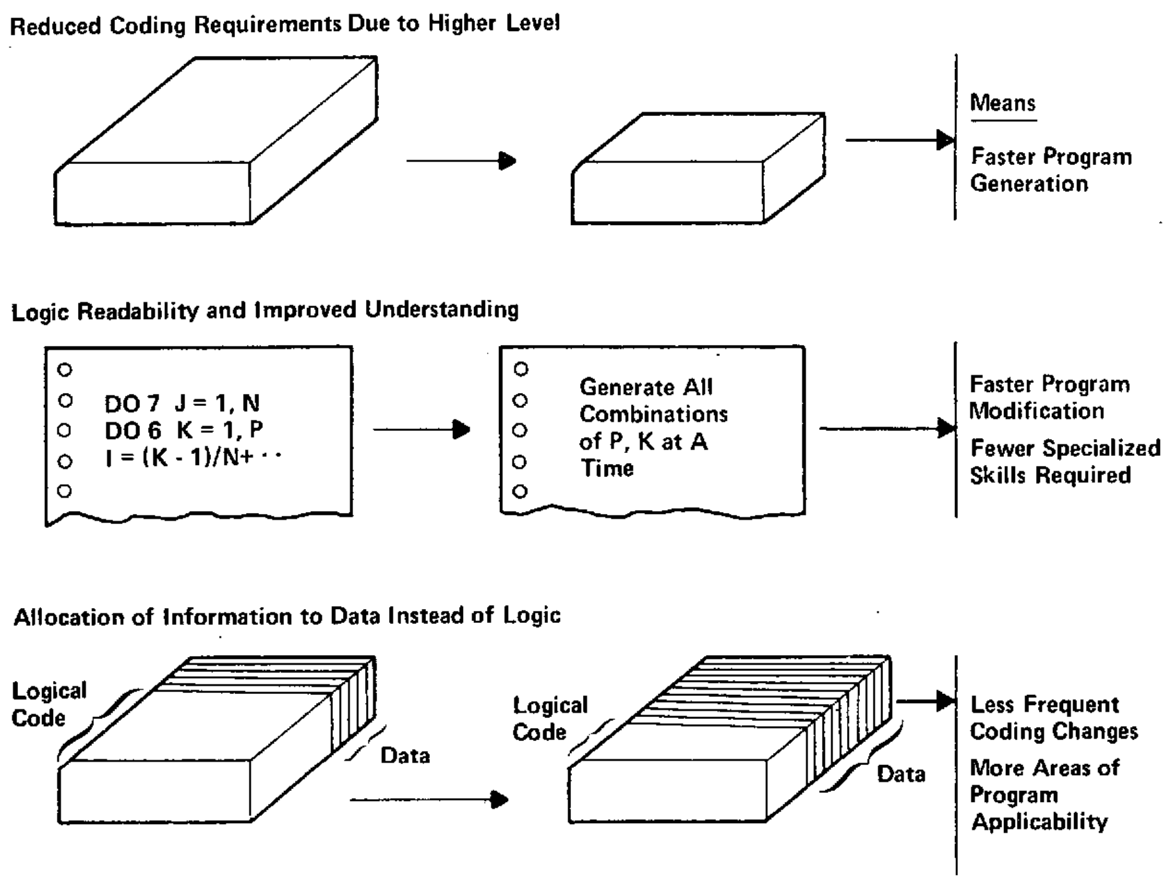
IV. EXPECTED BENEFITS FROM LANGUAGE USE

Because this phase of the scheduling language study has as its output only the functional specifications of the language and a set of library modules plus an assessment of implementation feasibility, determination of benefits to be gained from use of the PLANS Programming System has been made only on a preliminary basis. However, results of a preliminary analysis of the power of PLANS source program code conducted about five months after the start of the study, along with some later assessments of implementation feasibility in which example programs were coded, has provided some encouraging insights into the expected benefits from use of PLANS and the specified modules. Until actual translator implementation during the next phase of the study has progressed to the point where the coded programs can be executed, estimates of the ease of debugging and modifying PLANS programs are based on the PLANS designers' judgment and information presented on the following pages.

A. PLANS METHODS FOR SIMPLIFYING SOFTWARE DEVELOPMENT/MODIFICATION

Appropriate design of a programming language can substantially reduce the complexity of software development or modification. As depicted in Fig. 7, features use the following methods to simplify software development and modification.

High-level language capabilities that are "tailored" to the *class* of problems being programmed will reduce the number of source program statements required to code a problem. Higher-level PLANS statements reduce the programming effort and also



Reduced Coding Requirements Due to Higher Level
Means
Faster Program Generation
Logic Readability and Improved Understanding
DO 7 J = 1, N
DO 6 K = 1, P
I = (K - 1)/N+ · ·
Generate All Combinations of P, K at A Time
Faster Program Modification
Fewer Specialized Skills Required
Allocation of Information to Data Instead of Logic
Logical Code
Data
Logical Code
Data
Less Frequent Coding Changes
More Areas of Program Applicability

Fig. 7
Language-Related Methods for Simplifying Software Development/Modification

provide a programming capability to analysts with less specialized programming skills. Because the language has good readability, i.e. produces statements that make the program logic easier to interpret, the task of modifying programs is also greatly reduced.

Removing the maximum amount of system and problem descriptive information from the program logical code means that many changes in system characteristics can be incorporated by data changes in PLANS programs rather than by executable code changes. Debugging time may also be reduced and the range of applicability of the programs increased by more general logic code. Furthermore, problem models in PLANS that have specific information supplied only as data can be manipulated by a solution algorithm that communicates with the model via data, but not via program code. Thus, many manipulations of the model can be done automatically rather than by an analyst trial-and-error procedure. In summary, use of the PLANS Programming System allows appropriate allocation of problem information, reduces the probability of error, and provides software with maximum flexibility.

B. TRAIL LANGUAGE EFFECTIVENESS EVALUATION

Early in the study after the initial list of language operations and features were identified, it became desirable to test their functional validity. Therefore, a trial syntax was adopted subject to later revision. Complete language statements and programs were coded in this trail language syntax, sometimes called trail PLANS, to evaluate the basic language capabilities in realistic applications. The programming was "synthetic" in the sense that no means existed for translation to machine code for program execution. Three types of programs, including a number of general utility routines for scheduling problems, were programmed, and yielded further valuable insights and requirements for language design.

Two of the programs in this synthetic programming exercise had also been programmed in FORTRAN by other analysts for actual problems. Results are summarized in Table 3. The first of these was a simple program to group payloads for fitting into a Space Shuttle cargo bay. It required 17 statements in trial PLANS compared with 96 statements in the FORTRAN version. The second program coded was the basic logic of the NASA JSC/MPAD OSARS. The coding accomplished using the OSARS functional level flow chart required approximately one trial PLANS statement per block of the flow chart, a total of 48 statements. The FORTRAN version of OSARS requires approximately 600 statements. Of course, some use was made of the general purpose routines, but the trial showed that basic PLANS capabilities make flexible programming possible at a level of coding roughly equivalent to basic logic elements in a functional flow diagram. The final PLANS specification offers still more coding efficiency based on recent programming performed to assess the implementation feasibility of the PLANS Programming System.

Table 3. Trial Language Effectiveness Evaluation

PRELIMINARY COMPARISON DATA	FORTRAN Statements	Trial PLANS Statements
A Simple Payload Grouping Problem	96	17
Basic Heuristic of Operations Simulation and Resource Scheduling Program (OSARS)*	600	48

*PLANS code was generated from functional flowchart and resulted in approximately one PLANS statement per functional block.

C. EXPECTED PROGRAMMING ECONOMIES FROM PLANS USE

Experience gained in the trail PLANS programming and later revision of the trial code into the more efficient, specified version of PLANS, was used to estimate programming economies that might be expected from PLANS. Any such estimate requires assumptions about the use of specified PLANS library modules, and most significantly, the specific problem to be solved; but, based on the trial cases, there are large reductions in the size of the source code deck. In fact, deck size may be only 5 to 20% of comparable FORTRAN, depending upon the specific problem. These reductions are due to efficient use of PLANS features such as the higher-level language statements, tree data structure use and manipulation, dynamic data storage management that eliminates dimension statements, special ordering and combinatorial operations, and the generic operations model and algorithm library modules.

Estimates of manpower required to reach the same level of sophistication as a debugged FORTRAN program show 21 to 70% reductions based on the PLANS trial coding exercise. Some of the savings occur because the higher-level language statements enable coding from functional level instead of detail flow charts. Without actual program execution and debug experience, estimates of debug time savings up to 25% depend upon the assumed degree of use of already debugged library modules as a percentage of the total program code used. This is also highly problem-dependent. Manpower savings due to the problem analyst user-orientation of the language that gives more people programming skills were also considered, because this avoids much analyst interpretation of problems for specialist programmers and subsequent interface efforts. Similar assumptions and efficiencies also apply to program modification and maintenance efforts. In this case, from 4 to 12, or more, code modifications may be possible in a given time period, thus reducing code iterations to hours or days instead of weeks or months.

Finally, other important but less quantifiable benefits can be expected from broad applicability of the PLANS Programming System. PLANS, as indicated by the acronymn, is not only a scheduling language, but a language that is well-suited to the solution of resource allocation problems that are independent of scheduling problems. Its problem and user orientation provides more personnel with problem-solving skills, and the library modules encourage common problem approaches to many problems that can eliminate duplication of effort.