MCR-84-596
Contract No NAS8-34938

NASA-CR-171213

Phase 2
Final
Report

September 1984

Power Subsystem Automation Study

MARTIN MARIETTA

MCR-84-596

Contract No. NAS8-34938

Phase 2
Final
Report

September 1984

POWER SUBSYSTEM AUTOMATION STUDY

John C. Tietz
David Lewy
Cynthia Pickering
Ronald Sauers

MARTIN MARIETTA AEROSPACE
DENVER AEROSPACE
P O. Box 179
Denver, Colorado 80201

N85-12299#

FOREWORD

This report presents the results of a study by Martin Marietta Denver Aerospace for the National Aeronautics and Space Administration's George C. Marshall Space Flight Center. The study was the second phase of contract NAS8-34938, Power Subsystem Automation Study. It resulted in the demonstration of prototype "expert system" software for managing one aspect of a simulated space station power subsystem.

CONTENTS

Figure

Page

I. EXECUTIVE SUMMARY

The purpose of phase 2 of the Power Subsystem Automation Study was to demonstrate the feasibility of using computer software to manage an aspect of the electrical power subsystem on a space station. To accomplish this, we developed a software testbed that uses artificial intelligence techniques. This software prototype, known as the energy management expert system (EMES), is a first step toward the long-range objective of developing and demonstrating prototype software to automate such tasks as managing loads, power, and resources, monitoring state of health, and detecting and isolating faults in the space station power subsystem. In this study we also investigated the state of the art in expert systems software and analyzed the applicability of the generic capabilities of the software to other space station subsystems.

An expert system is a computer program that can competently act in the role of a human expert in a narrow field. Such programs are a new development in artificial intelligence; most of the work has been done in the past decade. They differ from conventional software in how they are built, in how they solve problems, and in what problems they solve. However, the most obvious physical difference is that expert systems are usually built with an auxiliary piece of software known as a production system and consist primarily of heuristics--rules of thumb extracted from an expert--encoded in the production system's formalism. These encoded rules are interpreted by the production system and applied opportunistically as the production system sees they are applicable. A number of such systems have been developed, not all of them research projects, and many are of considerable commercial value. With today's technology, prototype or operational expert systems have been written for:

1) Analyzing or diagnosing problems in diesel-electric locomotives, computers, telephone cables, and other equipment and systems;

2) Diagnosing diseases, analyzing electrocardiograms, and advising doctors in administering chemotherapy;

3) Assisting in exploring for mineral deposits and oil, and analyzing oil well data;

4) Assisting in design and analysis of software, including other expert systems;

5) Assisting in solving mathematical problems;

6) Assisting in the design or analysis of integrated circuits, databases, printed circuit cards, and single-board computers and other circuitry;

7) Assisting in job-shop scheduling and in management of manufacturing and large projects;

8) Assisting chemists and geneticists by analyzing protein crystallography data, mass spectrograms and other chemical data, by planning bioengineering experiments involving DNA, by solving certain problems in genetic engineering, and by helping plan organic chemical synthesis;

9) Providing computer-aided instruction;

10) Configuring computers;

11) Assisting in solving water-resource problems;

12) Adjusting signal-processing systems;

13) Analyzing structures;

14) Performing statistical analysis.

EMES is the first expert system ever developed to address the problem of spacecraft energy management. Some of the things EMES does are now handled by algorithmic load shedding. The problem with that approach is that it does not reason about how priorities change with time and circumstances. EMES can also do scheduling, which has traditionally been an expensive human-intensive task.

EMES is also a first step toward building flight software. It provides a baseline from which to build, highlights the major problems such software must address, and illustrates how these problems can be solved.

The work performed under this contract provides NASA a benchmark for estimating the speed and hardware requirements for a flight system. It will also allow NASA to more accurately forecast the size and capability of flight expert system software and determine how much time and effort will be required to design and implement it.

A. DESCRIPTION OF EMES

The EMES software testbed demonstrates that expert system software can manage the allocation of power to the various electrical components of a simulated space station. It does this by sequencing the operation of these components in an attempt to make the best use of available power while meeting basic mission requirements and energy management constraints. EMES also permits graceful degradation of the spacecraft under abnormal conditions.

The EMES program provides onboard automation of energy management under normal, failure, and degraded modes of spacecraft operation. This involves operation of all the housekeeping subsystems and payload equipment that consume power.

EMES operates in three major phases: library development, mission development, and mission execution. Of these, only the mission execution phase involves artificial intelligence. The other two phases set up a problem for EMES to solve, a task that corresponds to premission activities.

During the library development phase, the user defines a "library" of electrical loads with which to develop mission models. For each load, the user specifies a number of attributes, including power consumption and duty cycle. To make defining loads more convenient, the load library initially contains many predefined loads; the user can then add others unique to the mission.

During the mission development phase the user constructs a mission, using the loads in the library. In addition to specifying which loads are to be in the spacecraft during the mission, the user provides information to define the orbit and spacecraft attitude. The system then generates a mission definition file that is ready to be processed by the intelligent portion of EMES.

The final phase is mission execution during which EMES inspects the mission timeline, looking for resource requirements that cannot be met and energy management constraints that have been violated. The expert system modifies the mission timeline and produces a new one in which no constraints are violated. If the power available is suddenly reduced, EMES has the intelligence to reason about which loads can safely be removed and which cannot be removed without loss of data, product, or capability.

While working on the schedule revisions, EMES explains its decisions. Explanation is an important feature of an expert system because heuristics, not algorithms, are used to solve problems. The explanations allow humans to follow the line of reasoning that leads to the revised schedule, either to approve EMES' decisions or to gain confidence in its ability. To aid the human in overseeing the scheduling, EMES provides tabular and graphical displays of the information it uses in making decisions. The user obtains this information by selecting options from a set of menus.

The EMES program was designed to run on a Digital Equipment Corporation VAX-11/7xx-series computer under the VMS operating system. It requires Franz Lisp and Eunice as support software. Franz Lisp, created at the University of California at Berkeley, is an interpreter for a dialect of the LISP computer language. Eunice is a software package (produced by the Wollongong Group, Inc.) that adapts Franz Lisp to the VMS operating system.

EMES also requires the HAPS production system, a product of Martin Marietta Denver Aerospace. This software interprets the "rules" on which EMES is based.

Although EMES requires these support software packages, their use is completely transparent to the user. The user invokes EMES by typing the VMS operating system command "EMES." EMES then uses these other packages internally with no effects visible to the user.

If the user wishes to modify the rule base, he will also need two additional software packages:

1) Liszt, a LISP compiler that is compatible with Franz Lisp, created at the University of California at Berkeley;

2) HAPSZT, the companion compiler of HAPS, a product of Martin Marietta Denver Aerospace.

The EMES software requires approximately 75 megabytes of disk space. The computer needs a minimum of three megabytes of memory for reasonably efficient operation, and a "working set" of at least 2000 pages should be allocated to the user. For best performance, use of the computer by others should be restricted while EMES is running.

All communication between EMES and the user can take place through a Digital Equipment Corporation model VT-100 terminal (or equivalent), but full use of the program's graphics capability will require a line printer in addition to the terminal.

B. STUDY CONCLUSIONS AND RECOMMENDATIONS

The energy management expert system was designed as a demonstration prototype and software testbed. The intent was to demonstrate that expert system technology could be applied to management of a space station power subsystem. The emphasis in its design was therefore not on producing a high-performance piece of production software but on:

1) Determining what heuristics are required;

2) Demonstrating that such a system can work, i.e., produce schedules that experts would agree are reasonable;

3) Determining what constraints such an expert system places on hardware, cost, and time;

4) Demonstrating the capabilities and limitations of such a system.

EMES does produce reasonable answers with the heuristics documented elsewhere in this report. As it runs it explains its reasoning process and we have found this reasoning to be logical. The schedules it produces are reasonable.

The constraints EMES places on hardware, cost, and time are considerably greater than we had imagined. In its current implementation on a VAX-11/750 computer, EMES' performance is slow compared to human experts: it takes hours to perform tasks a human expert could accomplish in a few minutes. Practical constraints on computer time, system software capabilities, and human patience limit it to planning for a mission time period of approximately five hours, and it handles effectively only two or three payloads. These limits are not imposed by the amount of knowledge EMES needs to reason about the problem, and they do not imply that an expert system is impractical for the task. We know why its operation is slow and the reasons are not insurmountable.

First, a major source of inefficiency in EMES comes from design decisions in dividing the logic of EMES between production system rules and functions written in the LISP computer language. For example, a conceptually simple task, i.e., updating the power consumed by loads, was found to require 80 "rule firings." In other words, EMES applied 80 production system rules to accomplish what could have been done in a LISP function probably hundreds of times faster. The conclusion from this is that production system rules should be used sparingly, preferably only for heuristics that are to be applied opportunistically. When a predefined sequence of actions is to be taken, it is probably best to use a single rule that invokes a function in a procedural language to carry them out. If EMES were rewritten with this in mind, it might run two or three times faster.

Second, we designed EMES to run on a VAX computer. LISP-oriented computers on the market will run the same logic approximately five times faster.

Third, EMES could be recoded to maximize efficiency. This would certainly involve replacing some rules with LISP procedures as discussed previously, but it could also mean coding for a different production system or a second generation of the HAPS production system in which it is currently coded. Some expert systems have been improved in speed by recoding in a procedural language, e.g., "C" or Fortran. This could be an effective, although probably an expensive, solution because the size of the program will increase greatly.

In its current implementation, EMES requires support from the HAPS production system, from the Franz Lisp LISP interpreter, from the system software known as Eunice, and from the VMS operating system. Altogether EMES and the support software require the full capability of a VAX computer with three megabytes of memory to operate with anything approaching reasonable efficiency. The size of the program could certainly be reduced dramatically--perhaps by a factor of 10 or more--if EMES were recoded in a compiled procedural language. However, the cost of software development would certainly be higher, perhaps by a factor of 10 for a system of comparable capability.

The energy management expert system (EMES) requires approximately 150 production system "rules" in the HAPS formalism. Because production systems vary in how efficiently they encode heuristics, the same capability might require 300 or more rules in a different formalism. Flight software to perform the same tasks will be more complex because EMES benefits from a number of simplifying assumptions.

Flight software will also probably take on such additional tasks as thorough state-of-health monitoring, energy storage management, and fault detection, isolation, and correction. EMES addresses these problems only to the extent of recognizing when the electric power available is inadequate for scheduled activities and knowing how to ensure that it stores enough energy from the solar arrays during daylight periods. A different expert system developed by Martin Marietta Denver Aerospace that requires approximately 200 rules performs fault isolation for a simulated space station power subsystem.

One might extrapolate from the EMES experience that a flight system might require 1000 to 2000 HAPS rules, each of which could be expected to require as much effort as a small subroutine in a high-level programming language. However, accurately estimating the development effort for a new domain is very difficult. The first problem for the estimator is that expert systems, even small ones, are difficult to build; they are not trivial software systems. Therefore, the building of an expert system will have all the aspects of a large software development effort.

But what complicates estimating most is that initially no one working on the project has a clear picture of what is involved in creating the system. The domain expert knows nothing about building expert systems. Although he knows his own field, he cannot grasp how difficult it may be to encode what he knows. In fact, he may never have thought much about how he goes about making expert decisions. Similarly, the implementers know how to encode knowledge in general, but they do not yet know the complexity of the knowledge they must encode for the current project. They may not even be able to say with conviction that the expert's reasoning can be done in a computer at all. For example, after the project starts they may find that a key element in what the expert does is a notoriously difficult problem that artificial intelligence researchers have struggled with for years.

Because expert systems are new, it is not yet clear how their development can best be managed. Many of the management techniques for conventional software can be applied. For example, the concepts of modular construction and top-down design are applicable. However, new techniques are needed to deal with the fact that a great deal of work must be done before anyone knows the true size of the problem.

The inherent risk in the design of expert system software argues for a two-phase design approach. The first phase is the software analog of hardware breadboarding. During this phase the knowledge engineers work with the domain expert and extract as much knowledge as they can. They formulate rules and test them with a crude software product they plan to throw away when they are done. This phase provides the understanding needed to accurately estimate the amount of work phase 2 will require. When phase 1 ends, they will have an initial set of heuristics; they will know the major data structures required and will know how fast the work can progress.

In the second phase, the project team builds on the work done in the first phase. However, for two reasons they do not attempt to salvage the software developed in the first phase. First, the original software will generally be of poor design, if it can be said to be designed at all. It will have been changed many times and its original structure may be so hidden by patches and alterations that the program has become difficult to understand. At best, it will not be a clean, coherent, unified design. Second, if the implementers know they will have to build on what they produce in phase 1, they will probably not learn as much about the problem as they would otherwise. For example, the building of an expert system requires a great deal of experimentation because it is almost impossible to predict how well a set of rules will solve problems until one sees them in action on a computer. If the implementers spend much of their time keeping the program presentable, they will have less time to experiment. They will also be more reluctant to experiment because of the work it will entail.

Finally, quality-control constraints may make it difficult for them to experiment with multiple versions of potentially deliverable software or with software fragments they may want to use to test an idea. This doesn't mean that they should not be encouraged to document their work or that they should work without planning. But they should plan to throw the first system away.

When preparing to build an expert system, it is crucial that a reliable, thoughtful expert be found. It is hard to overestimate the importance of this because the working relationship between the expert and the knowledge engineer is key to the development of the expert system.

The ideal expert will be enthusiastic about the project, openminded, and able to tolerate frustration well. However, computer experience is not required. Indeed, such experience may be a handicap because he may try to predigest the information he gives the "knowledge engineers" to match the approach he would use in encoding his knowledge. If he does, he is not likely to present the heuristics he really uses, and he is likely to skip or gloss over the subtleties he does not know how to manage in conventional software. It is much better if he lets the knowledge engineers solve the encoding problems.

Enthusiasm and tolerance for frustration are vital because the project will take several months of his time, and for much of that time he will see nothing happening. He will answer countless "stupid" questions and often wonder why he must explain "obvious" conclusions in detail.

Openmindedness is equally important because experts rarely appreciate the amount of work required to encode their knowledge. Some experts will state categorically that their expertise is too subtle to be embedded in a computer program. These experts may of course be right, at least for today's technology, but they will rarely be good judges of the matter and their skepticism does not help the project. In contrast, other experts will greatly underestimate the effort because they do not realize that some tasks easily handled by a three-year-old child are unsolved problems in artificial intelligence. Such an expert may become hostile toward the knowledge engineers who, he feels, are incompetent and are wasting his time. In summary, it takes an open mind to prevent inaccurate expectations from adversely affecting the project.

Often more than one expert will be required to develop a useful system. The builders of the expert system need to be careful of disagreements among experts because this can result in a program that does not properly implement any consistent line of reasoning.

As important as finding a good expert is selecting the problem for which an expert system is to be built. This is difficult because, while many problems are complicated enough to warrant an expert system, most of these are far too complicated. Soberly assessed, expert systems are useful, but the technology is not sufficiently developed for tackling the more difficult problems. Spacecraft load management appears to be a good problem for an expert system because it requires diverse knowledge, flexibility, and speed.

It is important that an adequate knowledge representation system be designed for storing the information and knowledge the expert system will use. A number of well-known production systems are byproducts of the development of an expert system. Although some new production systems have been developed simply because the group wanted to be among the pioneers who have developed one, most were developed because no existing system was sufficiently suited to the new project. It is therefore not safe to assume that an existing production system will be suitable for a proposed expert system.

The designers of an expert system should seriously consider the division between rules and procedural code. Rules should not be used for everything because procedural code is faster for many operations and its purpose is often more transparent. This fact argues for the use of a production system that allows insertion of calls to functions or subroutines in a procedural language. Some production systems make this very difficult, a fact that should be remembered when selecting a production system.

The use of computers especially designed to efficiently implement LISP and related languages should also be seriously considered. Virtual-memory architectures do not appear to be well suited to building large expert systems because expert systems, by their nature, tend to cause a large number of "page faults" on such machines. The result is that the computer may spend more time in shuttling data into and out of memory than in computing.

For example, on a VAX-11/750 computer EMES takes approximately three hours to initialize itself for a seven-orbit schedule. It then takes an hour per orbit to compute the schedule. Our tests showed that LISP-oriented machines operate approximately five times as fast. Moreover, we found that when two users attempted to use an expert system on the VAX computer simultaneously, the running time went up not by a factor of two but by a factor of four. The lesson learned from this is that if a virtual-memory machine is used, it should be populated with as much real memory as possible, and the expert system should be given exclusive use of the machine with as large a memory allocation as possible.

In the area of power subsystem automation with expert systems, we recommend a study to model energy storage management in sufficient detail to allow an accurate estimate of the size and cost of an expert system to perform this function. The study should address both nickel-cadmium batteries and regenerative fuel cells and should consider causal reasoning as well as rule-of-thumb heuristics for managing energy storage.

A fault-isolation expert system should also be interfaced with a breadboard power system to demonstrate its ability to diagnose faults in real time from a hardware-generated telemetry stream. This will provide a level of confidence in the expert system's logic that could not be gained through all-software simulation.

Another area that should soon be investigated is the interface between expert systems for space station and the human operators of such systems. It would be valuable to determine the strengths and shortcomings of graphical and natural-language interfaces and to design and test one or both.

An expert system with some of the capabilities EMES demonstrates could be a useful experiment in its own right. For example, an expert system could be developed to automate some function on a space station despite doubts about its ability to handle the task. Its performance could be evaluated during flight by comparison with the decisions of the human experts who control the space station. During this evaluation, needed improvements could be noted and the expert system could be modified so it could be used for control in a future mission. If the function performed is critical, or if the consequences of a bad decision could be severe, the expert system might be carried as an experiment a number of times.

Although EMES demonstrated management of only one aspect of a space station power subsystem, the same generic capabilities could be put to work in other aspects of managing the power subsystem. For example, an expert system developed under Martin Marietta Denver Aerospace IR&D project D-55R demonstrated the ability of such software to detect and isolate faults in a space station power subsystem and, to a limited extent, find a workaround procedure.

An expert system might provide automatic state-of-health monitoring beyond simple fault detection. For example, it could observe trends in solar-array degradation and battery capacity, provide interpretations and, where possible, corrective actions to prevent failures or avoid operational problems.

Another fruitful area for applying the capabilities EMES demonstrates is in energy storage management. Software for this task will be more complex than EMES, however, because it will have to reason from cause and effect, whereas EMES rules do not require this. Reasoning from cause and effect or fundamental understanding of a system is a relatively new topic in the field of expert systems and the technology is not yet mature. However, the capability can be reasonably expected to be available in the space station time frame.

Organizing these capabilities in one expert system is currently beyond the state of the art. However, it is reasonable to suppose that they could be built into a hierarchy of cooperating experts with a "manager" expert system controlling and organizing the activities of the others. If this is done, the data passed among the expert systems should be minimized and highly structured to avoid design problems. The more the expert systems interact, the more problems can be expected in debugging them. If development of such a system is contemplated, work should begin soon on a prototype system to identify the design problems such an architecture will present.

One issue that needs to be addressed is the fact that expert system software, unlike conventional software, is generally neither "correct" nor "incorrect." Like human judgment, the performance of expert systems is better described by assigning a degree of competence. This means there is always the risk of finding a situation the software is not competent to handle. The minimal-risk approach to placing expert systems on space station to manage the power subsystem is to use them as experimental software that makes recommendations but controls nothing, at least initially. As confidence builds in the system's competence, it could be gradually given increased control over the power subsystem.

EMES demonstrates five generally useful capabilities:

1) Scheduling and revising schedules;

2) Reasoning about priorities that change with time and circumstances;

3) Detection of abnormal situations;

4) Displaying data in the form of tables and graphs;

5) General reasoning ability.

These capabilities might find use in other spacecraft subsystems. For example, the control and display subsystem could use these abilities for "intelligent caution and warning." An expert system could preinterpret the symptoms of abnormal conditions, find possible explanations, and suggest corrective actions to the crew rather than simply presenting raw data. As a minimum, the expert system could prioritize the display of data to emphasize the most important indications. An expert system with reasoning ability could do this more effectively than a simple algorithmic prioritization scheme because it could recognize more subtle patterns in the data, reason about possible causes and implications, and observe trends in data over a period of time.

These abilities might also be put to use in data management. An expert system could screen some kinds of data to prevent storage or transmission of redundant or meaningless data; or it could prepare predigested abstracts of data along with its interpretation of their meaning. These capabilities would reduce the amount of data a space station would have to return to earth, reduce the problems of data storage, and decrease the manpower required to interpret the data.

Some of the capabilities demonstrated by EMES could be useful in various payloads. Payloads such as scientific instruments and technology-development experiments require a large amount of human supervision. An expert system might substitute for some human activities, reducing costs and decreasing the chances of something being overlooked because of fatigue or inattentiveness. However, the payload would have to be chosen with some care because the expert system will itself be expensive to develop. The ideal payload to use an expert system is one that will be used for more than a year, requires intelligent supervision beyond the capability of conventional software, and does not require such human capabilities as development of novel theories, invention of new methods to solve unforeseen types of problems, insight, and intuition. Even when these abilities are required on occasion, an expert system might be able to reduce the human expert's burden.

II. INTRODUCTION

The purpose of this study was to demonstrate that it is feasible to use computer software to manage electric power on a space station. To demonstrate this, we developed a software testbed that uses artificial intelligence techniques. This software, known as the energy management expert system (EMES), is a first step toward the long-range objective of developing and demonstrating prototype software to automate such tasks as managing loads, power, and resources, monitoring state of health, and detecting and isolating faults in the space station power subsystem. This study also investigated the state of the art in such software and analyzed the applicability of the generic capabilities of the software to other space station subsystems.

A. DEFINITION OF AN EXPERT SYSTEM

An expert system is a computer program that can completently act in the role of a human expert in a narrow field. Such programs are a new development in artificial intelligence; most of the work has been done in the past decade. This work has received more attention than other artificial intelligence research because it has produced software of significant commercial value and promises to produce much more.

Although researchers do not completely agree on what distinguishes an expert system from conventional software, a number of differences are readily observed in practice. Many of these differences are more a matter of convenience or custom than necessity. For example, most expert systems are developed with support software known as a "production system." This practice has become so universal that many workers in the field cannot conceive of doing it any other way.

Typically, the programmer specifies the expert system's knowledge to the production system, using a formalism that differs significantly from such conventional computer languages as Fortran, FORTH, Pascal, and LISP. When the production system integrates this knowledge, it becomes a major part of the finished product, saving a great amount of time in software design and coding. Some production systems also come with useful software tools for implementing and testing the expert system.

However, the use of a production system does not make software an expert system. The primary differences between expert systems and conventional software are:

1) How they are built;

2) How they solve problems;

3) What problems they solve.

1. How They Are Built

Expert systems are composed of rules or encoded knowledge painstakingly derived from a series of interviews between the builders of the expert system and one or more human experts in some field. In general, the builders or "knowledge engineers" have little understanding of the field when they start; their expertise is not in the domain of the expert system but in extracting knowledge from experts and transforming it into a form the computer can use.

This contrasts with the development of conventional software because often the designers of such software are either domain experts themselves or at least can scope the task through knowledge of the algorithms required and comparison with similar software with which they are familiar. They may go to experts for algorithms, data, and advice, but typically they do not attempt to embed the experts' whole reasoning process into the software.

This difference could largely disappear for many expert systems in the near future with the development of tools that allow the expert to directly transfer his knowledge to the computer. This trend started with a program called TEIRESIAS [Davis 1977], which acts as an assistant in building expert systems. Recently a software package known as the automated reasoning tool (ART) has been introduced to allow novice engineers to create full-scale expert systems [Williams 1984].

2. How They Solve Problems

Expert systems are often said to differ from conventional software in their use of heuristics, not algorithms, as the backbone of their logic. The chief difference between an algorithm and a heuristic is that an algorithm is guaranteed to produce a desired result. A heuristic isn't; it corresponds to a human "rule of thumb" that usually gives satisfactory results but can sometimes fail. The failure can be the production of a wrong or suboptimal answer, or the heuristic may fail to find an answer at all.

Heuristics are not used as a substitute for efficient and effective algorithms. Rather, they are used when no satisfactory algorithm is known, as is often the case for the complex problems expert systems are designed to solve. For example, optimal load scheduling for a power system is impractical on a computer or even in the human mind. The number of possible solutions increases dramatically with the number of items to be scheduled, and no known algorithm to produce an optimal schedule is much smarter than trial and error. Algorithmic solution therefore takes far too long to be useful for problems of realistic size or practical importance. But human experts can schedule loads reasonably well, even when the problem is large, by using rules of thumb. These rules do not guarantee an optimal schedule; they don't even guarantee a good schedule. But they almost always produce an acceptable schedule if one is possible. Expert systems solve problems by mimicking the methods the experts use.

Although conventional software has made extensive use of heuristics, expert systems tend to use a larger collection of heuristics, and heuristics that are less dependent on one another. They also tend to apply their heuristics more opportunistically, i.e., they examine the situation, find an applicable heuristic, use it, see how this has changed the situation, and repeat the process until either the problem is solved or none of the heuristics is applicable. Conventional software, in contrast, tends to use a small set of closely related heuristics that are applied sequentially or according to some explicitly coded sequencing scheme. Because expert systems are not constrained to a predefined processing sequence, they can often deal with uncertainty and missing information better than conventional software.

Expert systems also tend to search for a solution by attaining goals that are set up during program execution; conventional software tends to have only implicit goals, and these are thought out in advance by the programmer and encoded into the program in a prescribed order.

Finally, in expert systems the emphasis is on domain-specific knowledge rather than on specific techniques and this knowledge is largely qualitative or symbolic, not quantitative. For example, an expert system typically reasons about the relationships among objects. Conventional software is usually more concerned with manipulating objects according to a predetermined processing method.

3. What Problems They Solve

Because expert systems are designed primarily for the implementation of heuristics, the types of problems expert systems solve best differ from the types of problems handled well by conventional software. If an algorithm solves a problem with acceptable speed, or if an exact answer is always required, an expert system should not be built to solve it. However, if there is no known algorithm, or if an answer is required more quickly than an algorithm can provide it, an expert system is worth considering. Expert systems are best used in fields where humans do most of the problem solving because many such domains require educated guessing. Expert systems have been effective in incorporating this ability.

In summary, no clear line separates expert systems from ordinary software, just as there is no clear line between great art and mediocre art. Nevertheless, just as art critics generally concur in classifying a work, the practitioners of artificial intelligence are in reasonable agreement about whether a given program qualifies as an expert system.

B. DESCRIPTION OF EMES

EMES is a prototype expert system computer program designed to demonstrate that such a system can manage the allocation of power to the various electrical components of a simulated space station. It sequences the operation of these components in an attempt to best use the available power while meeting basic mission requirements and energy management constraints. EMES also permits graceful degradation of the spacecraft under abnormal conditions.

This demonstration software illustrates the kinds of tasks an expert system could handle in a real space station scenario. In future applications where automation of various power system functions is expected to play a crucial role, such a system might extend the life of critical power system components as well as reduce the required size of energy storage devices.

The EMES program provides onboard automation of energy management under normal, failure, and degraded modes of spacecraft operation. This involves operation of all the housekeeping subsystems and payload equipment that consume power.

EMES operates in three major phases: library development, mission development, and mission execution. Of these, only the mission execution phase involves artificial intelligence. The other two phases set up a problem for EMES to solve, a task that corresponds to premission activities.

During the library development phase, the user defines a "library" of electrical loads with which to develop mission models. For each load, the user specifies a number of attributes, including power consumption and duty cycle. To make defining loads more convenient, the load library initially contains many predefined loads; the user can then add others unique to the mission.

During the mission development phase the user constructs a mission using the loads in the library. In addition to specifying which loads are to be on the spacecraft during the mission, the user provides information to define the orbit and spacecraft attitude. The system then generates a mission definition file that is ready to be processed by the intelligent portion of EMES.

The final phase is mission execution during which EMES inspects the mission timeline, looking for resource requirements that cannot be met and energy management constraints that have been violated. The expert system modifies the mission timeline and produces a new one in which no constraints are violated. If the power available is suddenly reduced, EMES has the intelligence to reason about which loads can safely be removed and which cannot be removed without loss of data, product, or capability.

While working on the schedule revisions, EMES explains its decisions. Explanation is an important feature of an expert system because heuristics, not algorithms, are used to solve problems. The explanations allow humans to follow the line of reasoning that leads to the revised schedule, either to approve its decisions or to gain confidence in its ability. To aid the human in overseeing the scheduling, EMES provides tabular and graphical displays of the information it uses in making decisions. The user obtains this information by selecting options from a set of menus.

The EMES program was designed to run on a Digital Equipment Corporation VAX-11/7xx-series computer under the VMS operating system. It requires Franz Lisp and Eunice as support software. Franz Lisp, created at the University of California at Berkeley, is an interpreter for a dialect of the LISP computer language. Eunice is a software package (produced by the Wollongong Group, Inc.) that adapts Franz Lisp to the VMS operating system.

EMES also requires the HAPS production system, a product of Martin Marietta Denver Aerospace. This software interprets the "rules" on which EMES is based.

Although EMES requires these support software packages, their use is completely transparent to the user. The user invokes EMES by typing the VMS operating system command "EMES." EMES then uses these other packages internally with no effects visible to the user.

If the user wishes to modify the rule base, he will also need two additional software packages:

1) Liszt, a LISP compiler that is compatible with Franz Lisp, created at the University of California at Berkeley;

2) HAPSZT, the companion compiler of HAPS, a product of Martin Marietta Denver Aerospace.

The EMES software requires approximately 75 megabytes of disk space. The computer needs a minimum of three megabytes of memory for reasonably efficient operation, and a "working set" of at least 2000 pages should be allocated to the user. For best performance, use of the computer by others should be restricted while EMES is running.

All communication between EMES and the user can take place through a Digital Equipment Corporation model VT-100 terminal (or equivalent), but full use of the program's graphics capability will require a line printer in addition to the terminal.

C. BENEFITS EMES PROVIDES

The development of EMES accomplished several things important to NASA. It is the first expert system ever developed to address the problem of spacecraft energy management. Currently some of the things EMES does are handled by algorithmic load shedding. The problem with that approach is that it does not reason about how priorities change with time and circumstances. EMES can also do scheduling, which has traditionally been an expensive human-intensive task.

Second, EMES is a first step toward building flight software. It provides an initial set of heuristics, highlights the major problems such software must address, and illustrates how those problems can be solved.

Finally, EMES provides a benchmark for estimating the speed and hardware requirements for a flight system. It will also allow NASA to more accurately forecast the size and capability of flight expert system software and determine how much time and effort will be required to design and implement it.

III. CONCLUSIONS AND RECOMMENDATIONS

A. EMES PERFORMANCE

The energy management expert system (EMES) was designed as a demonstration system and software testbed. The intent was to demonstrate that expert system technology could be applied to management of a space station power subsystem. The emphasis in its design was therefore not on producing a high-performance piece of production software but on:

1) Determining what heuristics are required;

2) Demonstrating that such a system can work, i.e. produce schedules that experts would agree are reasonable;

3) Determining what constraints such an expert system places on hardware, cost, and time;

4) Demonstrating the capabilities and limitations of such a system.

EMES does produce reasonable answers with the heuristics documented elsewhere in this report. As it runs it explains its reasoning process, and we have found this reasoning to be logical. The schedules it produces are reasonable.

The constraints EMES places on hardware, cost, and time are considerably greater than we had imagined. In its current implementation on a VAX-11/750 computer, EMES' performance is slow compared to human experts. It takes hours to perform tasks a human expert could accomplish in a few minutes. Practical constraints on computer time, systems software capabilities, and human patience limit it to planning for a mission time period of approximately five hours, and it handles effectively only two or three payloads. These limits are not imposed by the of the amount of knowledge EMES needs to reason about the problem, and they do not imply that an expert system is impractical for the task. We know why its operation is slow, and the reasons are not insurmountable.

First, a major source of inefficiency in EMES comes from design decisions in dividing the logic of EMES between production system rules and functions written in the LISP computer language. For example, a conceptually simple task, i.e., updating the power consumed by loads, was found to require 80 "rule firings." In other words, EMES applied 80 production system rules to accomplish what could have been done in a LISP function probably hundreds of times faster. The conclusion from this is that production system rules should be used sparingly, preferably only for heuristics that are to be applied opportunistically. When a predefined sequence of actions is to be taken, it is probably best to use a single rule that invokes a function in a procedural language to carry them out. If EMES were rewritten with this in mind, it might run two or three times faster.

Second, we designed EMES to run on a VAX computer. LISP-oriented computers on the market will run the same logic approximately five times faster.

Third, EMES could be recoded to maximize efficiency. This would certainly involve replacing some rules with LISP procedures as discussed previously, but it could also mean coding for a different production system or a second generation of the HAPS production system in which it is currently coded. Some expert systems have been improved in speed by recoding in a procedural language, e.g., "C" or Fortran. This could be an effective, though probably an expensive, solution because the size of the program will increase greatly.

In its current implementation, EMES requires support from the HAPS production system, from the Franz Lisp LISP interpreter, from software known as Eunice, and from the VMS operating system. Altogether EMES and the support software require the full capability of a VAX computer with three megabytes of memory to operate with anything approaching reasonable efficiency. The size of the program could certainly be reduced dramatically--perhaps by a factor of 10 or more--if EMES were recoded in a compiled procedural language. However, the cost of software development would certainly be higher, perhaps by a factor of 10 for a system of comparable capability.

B. PLANNING FOR AND MANAGING FUTURE EXPERT SYSTEMS

EMES requires approximately 150 production system "rules" in the HAPS formalism. Because production systems vary in how efficiently they encode heuristics, the same capability might require 300 or more rules in a different formalism. Flight software to perform the same tasks will be more complex because EMES benefits from a number of simplifying assumptions.

It is also likely that flight software will take on such additional tasks as thorough state-of-health monitoring, energy storage management, and fault detection, isolation, and correction. EMES addresses these problems only to the extent of recognizing when the electric power available is inadequate for scheduled activities and knowing how to ensure that it stores enough energy from the solar arrays during daylight periods. A different expert system developed by Martin Marietta Denver Aerospace performs fault isolation for a simulated space station power subsystem and requires approximately 200 rules.

One might extrapolate from the EMES experience that a flight system might require 1000 to 2000 HAPS rules, each of which could be expected to require as much effort as a small subroutine in a high-level programming language. However, accurately estimating the development effort for a new domain is very difficult. The first problem for the estimator is that expert systems, even small ones, are difficult to build; they are not trivial software systems. Therefore, the building of an expert system will have all the aspects of a large software development effort.

However, the fact that initially no one working on the project has a clear picture of what is involved in creating the system complicates estimating the most. The domain expert knows nothing about building expert systems. Although he knows his own field, he cannot grasp how difficult it may be to encode what he knows. In fact, he may never have thought much about how he goes about making expert decisions. Similarly, the implementers know how to encode knowledge in general, but they do not yet know the complexity of the knowledge they must encode for the current project. They may not even be able to say with conviction that the expert's reasoning can be done in a computer at all. For example, after the project starts they may find that a key element in what the expert does is a notoriously difficult problem that artificial intelligence researchers have struggled with for years.

Because expert systems are new, it is not yet clear how their development can best be managed. Many of the management techniques for conventional software can be applied. For example, the concepts of modular construction and top-down design are applicable. However, new techniques are needed to deal with the fact that a great deal of work must be done before anyone knows the true size of the problem.

The inherent risk in the design of expert system software argues for a two-phase design approach. The first phase is the software analog of hardware breadboarding. During this phase the knowledge engineers work with the domain expert and extract as much knowledge as they can. They formulate rules and test them with a crude software product that they plan to throw away when they are done. This phase provides the understanding needed to accurately estimate the amount of work phase 2 will require. When phase 1 ends, they will have an initial set of heuristics--they will know the major data structures required and they will know how fast the work can progress.

In the second phase, the project team builds on the work done in the first phase. However, for two reasons they do not attempt to salvage the software developed in the first phase. First, the original software will generally be of poor design, if it can be said to be designed at all. The software will have been changed many times and its original structure may be so hidden by patches and alterations that the program has become difficult to understand. At best, it will not be a clean, coherent, unified design. Second, if the implementers know they will have to build on what they produce in phase 1, they will probably not learn as much about the problem as they would otherwise. For example, the building of an expert system requires a great deal of experimentation because it is almost impossible to predict how well a set of rules will solve problems until one sees them in action on a computer. If the implementers spend much of their time keeping the program presentable, they will have less time to experiment. They will also be more reluctant to experiment because of the work it will entail. Finally, quality-control constraints may make it difficult for them to experiment with multiple versions of potentially deliverable software or with software fragments they may want to use to test an idea. This does not mean that they should not be encouraged to document their work or that they should work without planning. But they should plan to throw the first system away.

When preparing to build an expert system, it is crucial that a reliable, thoughtful expert be found. It is hard to overestimate the importance of this because the working relationship between the expert and the knowledge engineer is key to the development of the expert system.

The ideal expert will be enthusiastic about the project, openminded, and able to tolerate frustration well. However, computer experience is not required. Indeed, such experience may be a handicap, because he may try to predigest the information he gives the "knowledge engineers" to match the approach he would use in encoding his knowledge. If he does he is not likely to present the heuristics he really uses, and is likely to skip or gloss over the subtleties he does not know how to manage in conventional software. It is much better if he lets the knowledge engineers solve the encoding problem.

Enthusiasm and tolerance for frustration are vital because the project will take several months of his time, and for much of that time he will see nothing happening. He will answer countless "stupid" questions and often wonder why he must explain "obvious" conclusions in detail.

Openmindedness is equally important because experts rarely appreciate the amount of work required to encode their knowledge. Some experts will state categorically that their expertise is too subtle to be embedded in a computer program. These experts may be right, at least for today's technology, but they will rarely be good judges of the matter and their skepticism does not help the project. In contrast, other experts will greatly underestimate the effort because they do not realize that some tasks that are easily handled by a three-year-old child are unsolved problems in artificial intelligence. Such an expert may become hostile toward the knowledge engineers who, he feels, are incompetent and are wasting his time. In summary, it takes an open mind to prevent inaccurate expectations from adversely affecting the project.

Often more than one expert will be required to develop a useful system. The builders of the expert system must be careful of disagreements among experts because this can result in a program that does not properly implement any consistent line of reasoning.

As important as finding a good expert is selecting the problem for which an expert system is to be built. This is difficult because although many problems are complicated enough to warrant an expert system, most of these are far too complicated. Soberly assessed, expert systems are useful, but the technology is not developed enough for tackling the more difficult problems. Spacecraft load management appears to be a good problem for an expert system because it requires diverse knowledge, flexibility, and speed.

It is important that an adequate knowledge representation system be designed for storing the information and knowledge the expert system will use. A number of well-known production systems are byproducts of the development of an expert system. Although some new production systems have been developed simply because the group wanted to be among the pioneers who have developed one, most were developed because no existing system was sufficiently well suited to the new project. It is therefore not safe to assume that an existing production system will be suitable for a proposed expert system.

The designers of an expert system should seriously consider the division between rules and procedural code. Rules should not be used for everything because procedural code is faster for many operations and its purpose is often more transparent. This fact argues for the use of a production system that allows insertion of calls to functions or subroutines in a procedural language. Some production systems make this very difficult, a fact that should be considered when selecting a production system.

The use of computers especially designed to efficiently implement LISP and related languages should also be seriously considered. Virtual-memory architectures do not appear to be well suited to building large expert systems because these systems, by their nature, tend to cause a large number of "page faults" on such machines. The result is that the computer may spend more time in shuttling data into and out of memory than in computing.

For example, on a VAX-11/750 computer, EMES takes approximately three hours to initialize itself for a seven-orbit schedule. It then takes an hour per orbit to compute the schedule. Our tests showed that LISP-oriented machines operate approximately five times as fast. Moreover, we found that when two users attempted to use an expert system on the VAX computer simultaneously, the running time went up not by a factor of two but by a factor of four. The lesson learned from this is that if a virtual-memory machine is used, it should be populated with as much real memory as possible, and the expert system should be given exclusive use of the machine with as large a memory allocation as possible.

C. RECOMMENDED FUTURE ACTIVITY

In the area of power subsystem automation with expert systems, we recommend a study to model energy storage management in sufficient detail to allow accurate estimation of the size and cost of an expert system to perform this function. The study should address both nickel-cadmium batteries and regenerative fuel cells and should consider causal reasoning as well as rule-of-thumb heuristics for managing energy storage.

A fault-isolation expert system should also be interfaced with a breadboard power system to demonstrate its ability to diagnose faults in real time from a hardware-generated telemetry stream. This will provide a level of confidence in the expert system's logic that could not be gained through all-software simulation.

Another area that should soon be investigated is the interface between expert systems for space station and the human operators of such systems. It would be valuable to determine the strengths and shortcomings of graphical and natural-language interfaces and to design and test one or both.

IV. ASSUMED SPACE STATION CONFIGURATION

EMES is designed to demonstrate that expert system software can competently manage load scheduling, i.e. the allocation of power to the various electrical components of a simulated space station. In scheduling, EMES does far more reasoning than a simple load-shedding algorithm would because EMES does not assign loads fixed priorities; it reasons about the nature of each load in deciding when it should be scheduled and for how long.

Such detailed reasoning cannot be done without considerable knowledge of the space station configuration, the payloads, and the nature of the power subsystem. We therefore describe here the details of the space station modeled for EMES. We also define the extent to which EMES is aware of these details.

A. CONFIGURATION DETAILS

The contract statement of work called for a "generic" power subsystem, and the software design was not to rely on the details of a specific configuration. This requirement was met. However, to exercise EMES we found it necessary to assemble a set of baseline space station subsystems to allow us to assign power consumption values and establish operating characteristics. It also provided a framework within which to operate various payloads. The specific details of the subsystems are not built into EMES, however. For example, the expert system rule base does not depend on a specific number of power modules, a specific power capability, etc. When it needs specific details, it gets them from a disk file produced by Fortran analysis software that is readily altered with no impact on the logic of the expert system itself.

Because previous work had identified the major space station subsystems, these were selected for the baseline configuration. This configuration consists of the following subsystems and payloads:

1) Commercial payloads;

2) Communication and tracking (CTS);

3) Control and display (CDS);

4) Data management (DMS);

5) Environmental control (ECS);

6) Guidance, navigation, and control (GNCS);

7) Life support (LSS);

8) Electric power (EPS);

9) Science payloads;

10) Technology development payloads;

11) Thermal control (TCS).

The initial approach was to develop a few simple block diagrams to characterize the design of the space station. From this we would develop a set of loads for all the major components. It was soon determined, however, that most of our work would be influenced by the application of payload-type loads rather than manipulating the space station subsystem loads. For this reason, a major shift of emphasis was made to establish a set of reasonable payloads that would be manipulated or scheduled to exercise the EMES model.

The space station subsystems are represented by simplified block diagrams (Fig. IV-1 through IV-9).

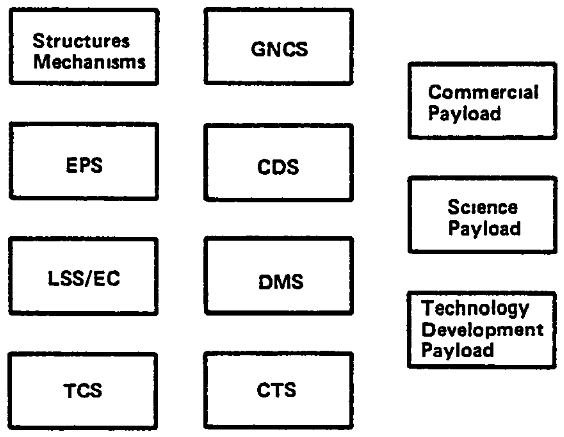


Figure IV-1 Major Space Station Systems

The power subsystem we baselined uses three battery modules, each supported by a solar array module arranged in two large panels. For daylight operation, the subsystem is specified to deliver an average of 34.2 kW per module over life and 32.5 kW at end of life. For night operation, the module uses batteries with a capacity of 30 kWh per module. The estimated depth of discharge is 20%. These design parameters give an estimated battery power capability for each orbit of

(total battery capacity)(depth of discharge)/(dark time),

which equals 36 kW with the parameters cited previously and an assumed dark time of 30 minutes.

Such a power subsystem would support a total day load of at least 97.5 kW and a night load of approximately 36 kW.

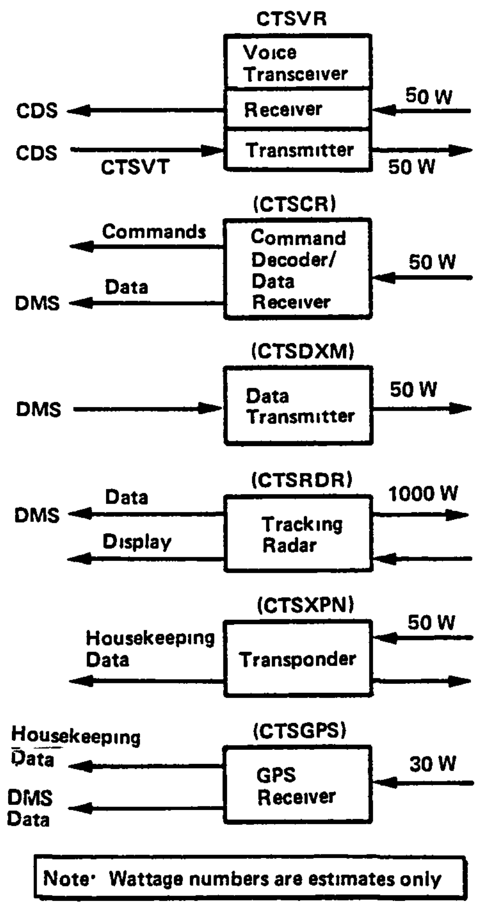


Figure IV-2 CTS Concept Configurations

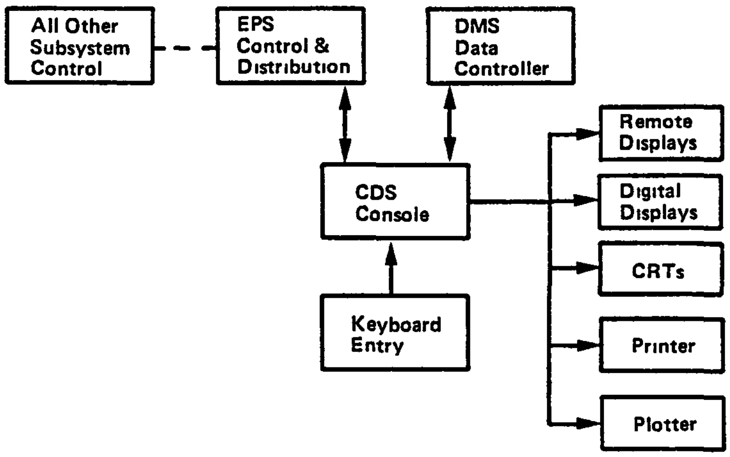


Figure IV-3 CDS Concept Configuration

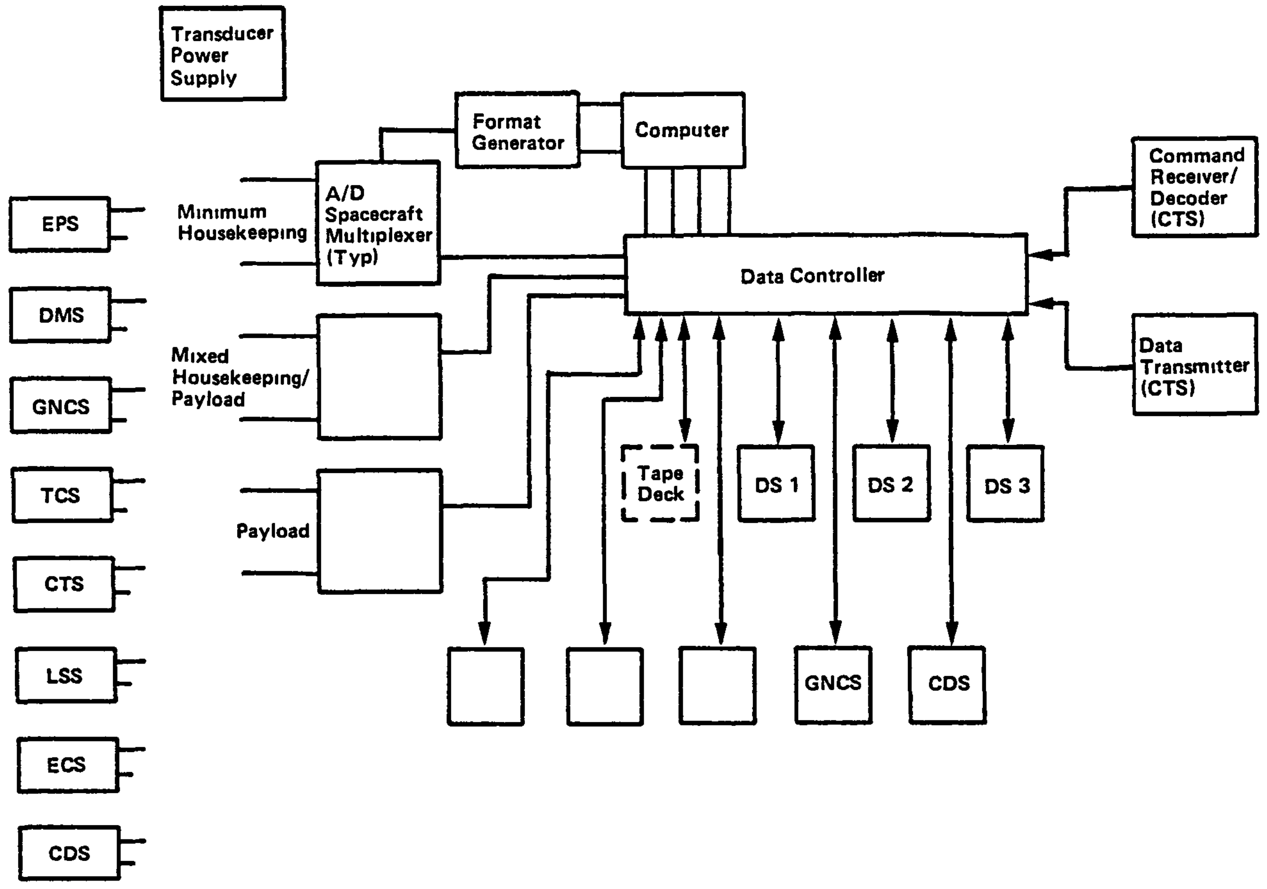


Figure IV-4 DMS Concept Configuration

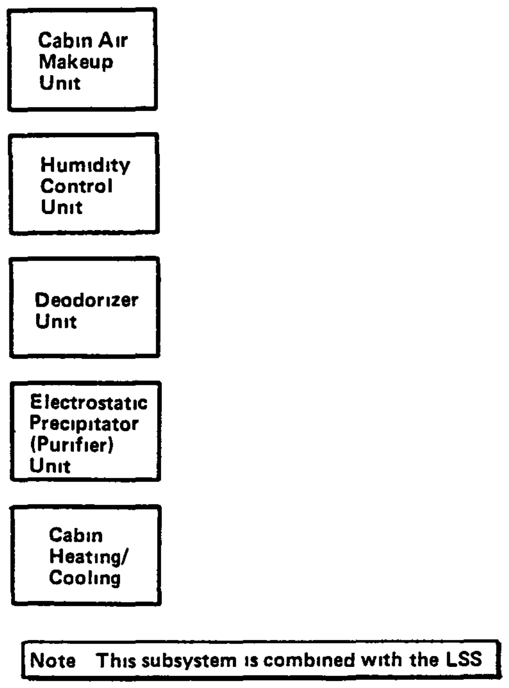

Cabin Air Makeup Unit
Humidity Control Unit
Deodorizer Unit
Electrostatic Precipitator (Purifier) Unit
Cabin Heating/ Cooling
Note This subsystem is combined with the LSS

Figure IV-5 ECS Subsystem Concept Configuration

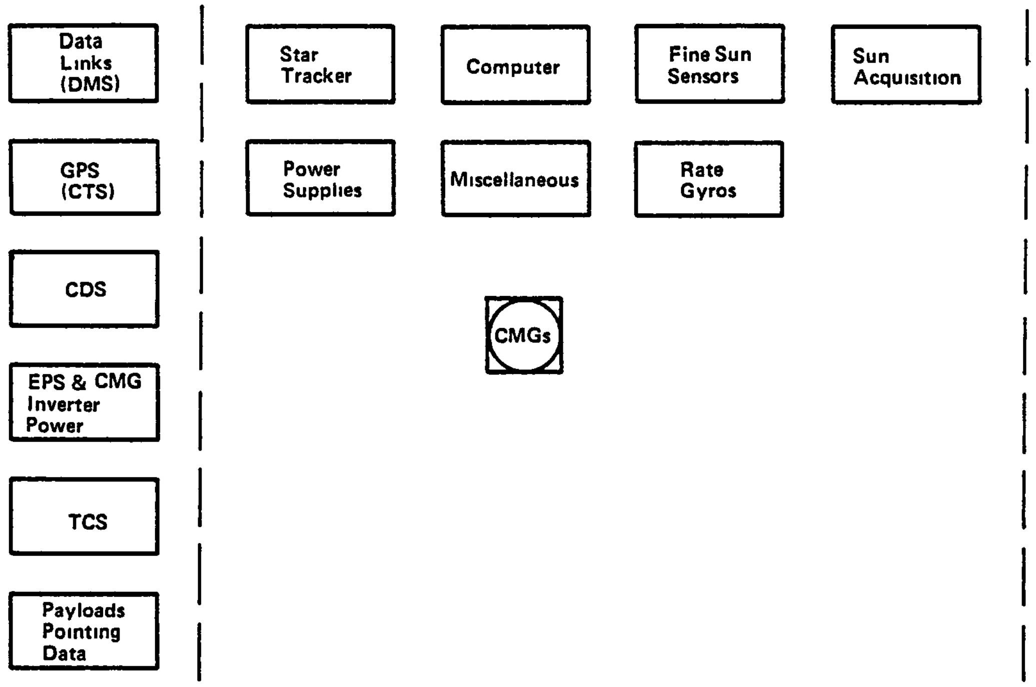


Figure IV-6 GNCS Concept Configuration

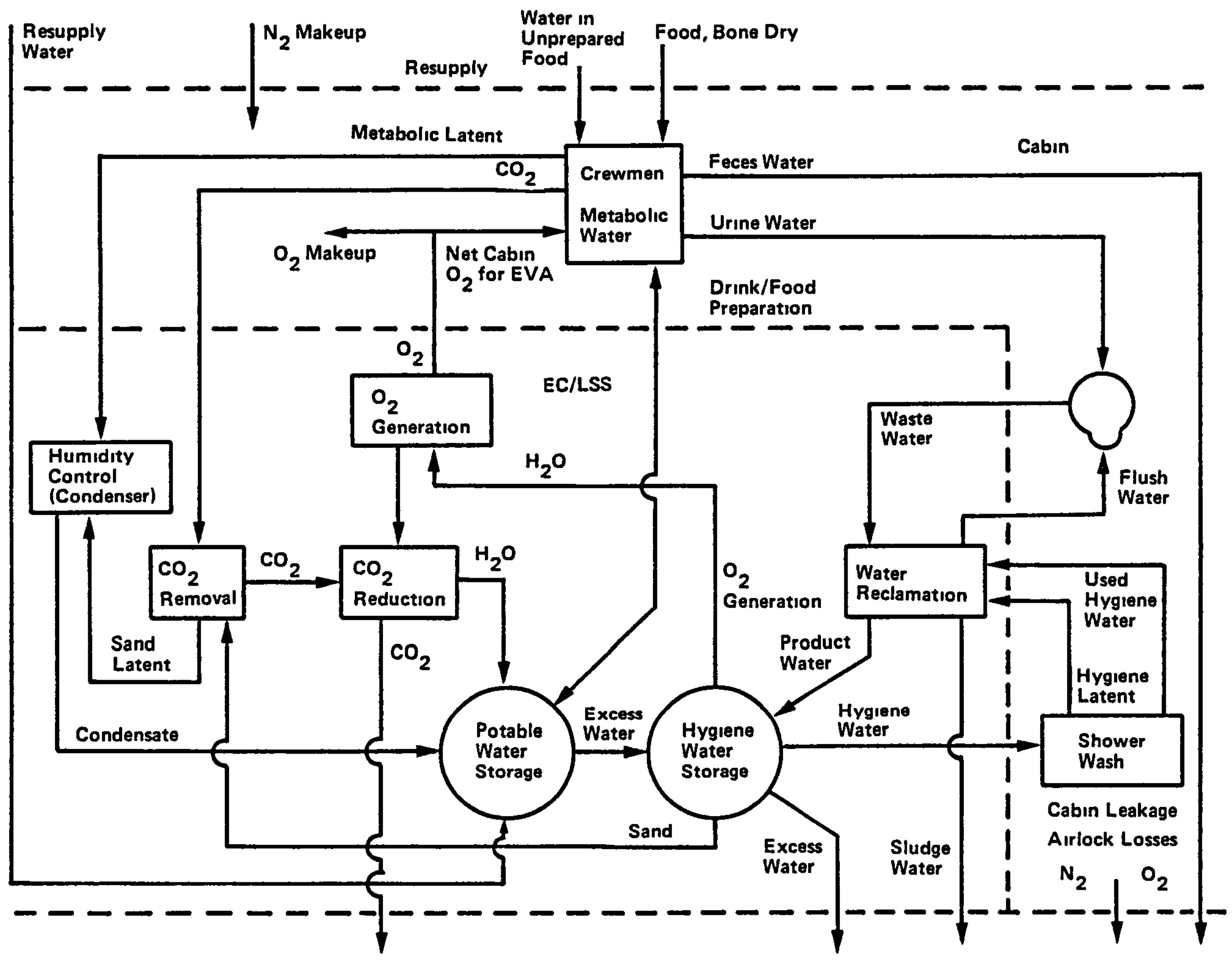


Figure IV-7 LSS Subsystem Concept

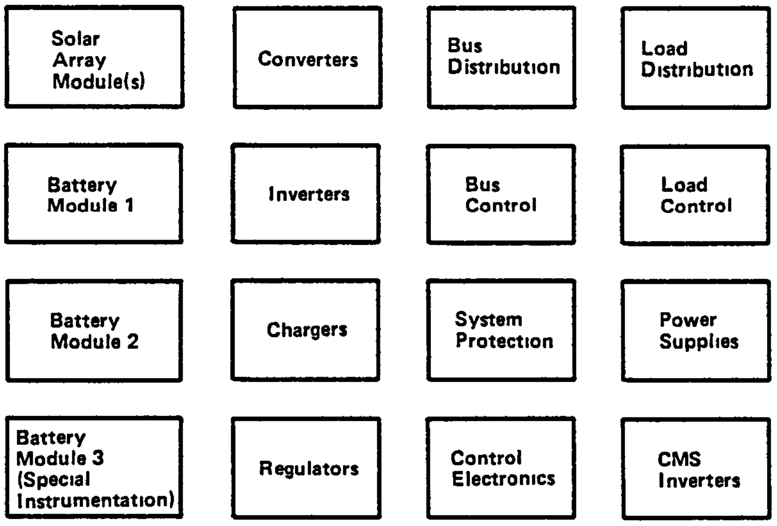


Figure IV-8 EPS Concept Configuration

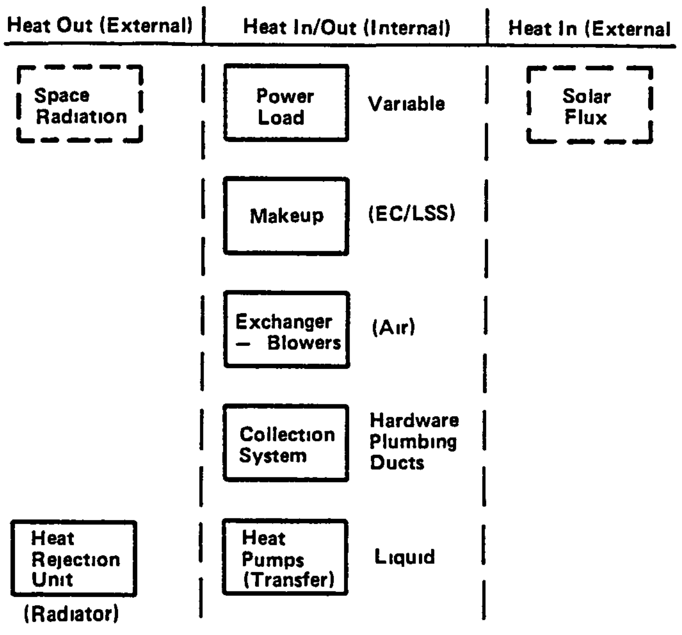


Figure IV-9 TCS Concept Configuration

The aggregate baseline load power consumption is 60 kW for daylight operations and 25 kW for night operations. If the night load is fully used during the eclipse portion of each orbit, the daytime charging load will be somewhat less than the night load because the charge time is approximately twice as long. Specifically, for a 90-minute orbit and a charging efficiency of 85%, the daytime charging load will be 21.18 kW. Then for day payloads, the power available is approximately

(power available) - (baseline load power) - (battery charging power),

which equals 16.32 kW, a rough figure to use for daytime scheduling.

For night payloads the power available is

(power available) - (baseline load power),

which equals 11 kW, a rough figure to use for nighttime scheduling.

For each category of payload, i.e. commercial, science, and technology development, a set of three payloads was identified. These were:

1) Commercial payloads,
    a) Materials processor,
    b) Medical mixer,
    c) Biology cell;
2) Science payloads,
    a) Day mapper,
    b) Sun pointer,
    c) Star pointer;
3) Technology development payloads,
    a) Night mapper,
    b) Laser communication unit,
    c) Battery management expert system.

The payloads were "designed" with a variety of characteristics that would constrain scheduling. These constraints created a number of realistic problems for intelligent decision-making in EMES. Although the names of the payloads are arbitrary, they were chosen to help the user grasp their intended functions. To this extent they may be considered "generic" payloads, and a host of other types could have just as easily been selected.

The major characteristics selected for the payloads are:

1) Some payloads operate exclusively during daylight, some exclusively at night, and others either day or night;

2) Some require continuous power, while others may have the power interrupted without harm;

3) Some require a warmup period, and some require an active or passive cooldown period;

4) The payloads differ in duty cycle from short to long;

5) Some require pointing. This may involve a maneuver prior to activation, continuous pointing, pointing at a specified time, and/or pointing at a specified target.

Although practical payloads have many other characteristics that could be considered, this set provides a good cross-section of typical constraints.

The user of EMES can change the payload power consumption values and other characteristics by answering a series of questions EMES asks. This allows the user to investigate the performance of EMES with various levels of problem difficulty. The nominal power requirements of the loads are tabulated.

Name	Power Required, kW
Materials processor	12
Medical mixer	2
Biology cell	1.4
Day mapper	8
Sun pointer	10
Star pointer	6
Night mapper	5
Laser communications unit	11
Battery management expert system	4

B. CONFIGURATION GROUND RULES

In a demonstration system with a limited development time, it is not practical to consider all the problems of flight software for a real space station. A number of ground rules were therefore developed to aid in defining and bounding the problem:

1) For normal operations, all subsystems are free of hardware and software failures;

2) Any redundancy, switchover, crossfeeds, etc are not included;

3) All subsystems are near-ideal state-of-the-art equipment. They are sophisticated and can perform all necessary tasks, including complex data management, precise voltage regulation for all buses, precise guidance and navigation, target acquisition and lock on target, multiple axis control, flexible communications, and maintenance of all life support requirements and space station integrity for long life;

4) The number of crew members is not a factor in this study;

5) All space station loads are generic representations only and may be changed as required;

6) Day prime power is from solar arrays;

7) Night prime power is from batteries;

8) The power system size may be changed as required;

9) Power distribution losses are absorbed in the baseline loads;

10) There are no special charging components in the power subsystem;

11) Thermal control provides for and maintains the required space station thermal balance at all times;

12) The life support subsystem and environmental control provide all required life support functions and an ideal working environment;

13) Ground stations or relay links are available at all times for command or data uplink or downlink;

14) Voice communication links are available at all times;

15) The space station is collecting housekeeping data at all times for storage or downlink;

16) The capability exists to handle a broad mix of analog and digital data and a wide range of data rates;

17) Data may be collected, stored, transmitted, and received simultaneously;

18) Selected data may be retrieved and displayed when desired;

19) The guidance, navigation and control subsystem can perform all desired pointing maneuvers and maintain an absolute or relative pointing orientation;

20) Pointing refers to an allocation and may be accomplished by the space station itself or by a platform;

21) Payloads are arbitrarily named and were selected to provide a variety of operating requirements, duty cycles, and constraints;

22) The payload power consumption levels may be adjusted as required;

23) The payload success criteria may be adjusted as required;

24) Any payload may have its success criteria specified by a higher authority.

C. HOW INTIMATELY EMES KNOWS THE CONFIGURATION

Because EMES is a testbed for demonstrating a generic power system management capability, we tried to make it as independent as possible of any specific power subsystem configuration. For example, although EMES needs specific information about how much power is available at different times, the details of calculating this information can be divorced from the reasoning of the expert system itself. Specifically, EMES uses Fortran software to provide a profile of power available during each 6-minute time slot. This software is equivalent to the analysis software a human expert might use in planning a mission. So although EMES reasons about the information in a data file this software produces, it does not know the configuration details the Fortran software needed to produce the data. An entirely new power system could be modeled for EMES without changing the reasoning of the expert system itself; only the Fortran analysis software would need to be changed. Furthermore, if the change is minor, this software can be changed very quickly. For example, the number of power modules is simply a parameter that can be changed by modifying one line in the Fortran software.

Some of the heuristics EMES uses would not be applicable if the power system did not rely on solar power. For example, EMES knows that, when there is a choice, it is better to use power in the daytime than at night because energy storage always involves some inefficiency. A radically different power subsystem, e.g., nuclear, would have a different set of constraints. However, the EMES rules should be suitable for a variety of energy storage methods as long as the principal source of energy is solar power. In particular, we attempted to use rules that would be equally applicable to subsystems using batteries, fuel cells, or flywheels for energy storage.

Much of this generality was achieved by pushing subsystem details into the Fortran analysis software. For example, constraints on depth of discharge are embedded in this software. However, some of the knowledge could not conveniently be divorced from the EMES rule-based logic. For example, to make proper use of its knowledge that nickel-cadmium batteries need reconditioning periodically, an expert system needs more than a table of data; it needs rules for determining when and how to recondition. These rules are closely tied to the choice of energy storage technology but are not readily removed from the scheduling logic. In minimizing this system-specific knowledge, we were sometimes forced to make EMES less intelligent than it might otherwise be.

V. HOW EMES IS USED

A. OVERVIEW OF OPERATION

Before the intelligent portion of EMES begins to run, the user sets up a problem for it to solve by selecting options from menus and answering questions the program asks. The user can then define the characteristics of electrical loads and the constraints these loads place on scheduling. He can also specify a number of mission parameters, including the orbit, the time interval for which EMES is to prepare a schedule, success criteria, and the level of degradation of the power subsystem. This problem-setup phase of EMES operation corresponds to premission activities and does not use the intelligence of the expert system.

After the user has defined the electrical loads and the mission characteristics, he can start execution of the intelligent portion of EMES to schedule the loads.

While producing the schedule, EMES explains its reasoning. Although some production systems have built-in explanation facilities for this purpose, such facilities were not appropriate for EMES because they are designed for expert systems that consult with the user by asking a series of questions and then giving advice. In such programs the consultation is directed by the intelligence of the expert system, and the explanation consists of the rationale for asking each question or drawing each conclusion. In contrast, the EMES question-and-answer operations are completed before the intelligent part of EMES starts working. The explanation facility in EMES is therefore not a part of the "HAPS" production system used to implement the program. Instead, it is a comment routine activated during the execution of certain key rules. Though this routine is not part of the production system, it is not unique to EMES and can be used by any expert system implemented with the HAPS production system.

After EMES has scheduled the loads, the user can request any of four types of graphical reports--power capability, load profile, power margin, or battery depth of discharge. The user can direct the program to print the graphs on paper or display them on the computer terminal.

B. DETAILS OF DIALOG WITH THE USER

EMES contains separate modules for library development, mission development, and mission execution. The first two correspond to ground activities before the mission; the latter simulates the flight software.

The load library development module allows the user to construct and maintain a set of load definitions with which to develop mission models. Loads have various user-specifiable attributes, e.g., power consumption and duty cycle. The library initially contains many predefined payloads, and the user is permitted to add to these at any time. The library also contains a set of predefined subsystem loads. These loads are necessary to maintain normal spacecraft operation and are automatically integrated into the spacecraft timelines. Loads of this type include guidance, navigation and control, thermal control, and life support and environmental control among others.

The mission development module allows the user to construct a mission using the loads in the system load library. To construct a mission, the user lists the information required for orbit and attitude definition and then specifies the set of loads he wishes on board the spacecraft during the mission. The software uses this information to generate a mission definition file that can be processed by the EMES mission execution module to simulate a mission.

After the user has defined the loads and the mission, the mission execution module can be used. This is the module that simulates the flight software. It inspects the mission timeline, looking for any resource requirements that cannot be met and energy management constraints that have been violated. It then modifies the mission timeline and produces a new one in which no constraints are violated. If the mission definition does not initially violate any power constraints, EMES schedules the payloads requested during mission definition.

The user selects the module he needs by using a dispatching module called the top-level executive. This module directs the user through library development, mission development, and mission execution. It is started by typing the command "EMES" after the VMS operating system prompt:

```
    $ EMES
```

When EMES starts, it presents to the user the following top-level menu.

```
    Top-Level Executive.

         1. Load Library Development.
         2. Mission Development.
         3. Mission Execution.
         4. Exit EMES.

    Menu Selection:
```

This menu is used to select the major function to be performed. If the value selected is "4," which corresponds to the "Exit EMES" command, control is returned to the VMS operating system. The other possible values are "1," "2," or "3," which invoke the library development, mission development and mission execution submodules, respectively. These submodules return control to the top-level executive when they have completed their tasks, allowing the user to select another function.

1. Load Library Development

The load library development module allows the user to create and maintain the payload definitions used in developing mission models and in maintaining and updating the baseline loads' power consumption in four modes of operation--normal, degraded, severely degraded, and emergency. For payload development, four basic development functions are provided: define a load, remove an old load definition, list loads in the library, and show the definition of a load. The load library development module is activated by selecting "1" when the library development menu is displayed:

```
    Menu Selection: 1

    Load Library Development.

        1. Define a new load.
        2. Redefine baseline load power consumption.
        3. Remove an old load definition.
        4. List loads in the library.
        5. Show baseline load power consumption.
        6. Show the definition of a load.
        7. Return to top-level menu.
```

a. Baseline Load Upkeep - The load library development module allows the user to edit and view the system-defined baseline loads. If the user selects option "5" from the load library development menu, corresponding to the "Show baseline load power consumption" command, another menu will appear:

```
    Send output to ...

        1. The Terminal.
        2. A Disk File.
        3. The Line Printer.

    Menu Selection:
```

This gives the user the option of viewing the baseline load information directly on the terminal, having it printed out, or placing it in a disk file. This menu appears whenever the user wishes to view any type of load or mission information.

If the user wishes to change the baseline loads' power consumption for the four levels of degradation, option "2" should be selected. The energy management expert system will step through all four modes of all seven baseline loads. The system will prompt the user as to what the new power level should be. The power levels for the baseline loads in the system library at present are:

BASELINE-LOADS	MODE1	MODE2	MODE3	MODE4
Communication-and-Tracking	100	90	80	70
Control-and-Display-System	1600	1500	1450	1400
Data-Management-System	4000	3500	3000	2500
Guidance-Navagation-and-Control	3300	3000	2900	2800
Life-Support-Systems/ Environmental-Control	6000	6000	5000	5000
Electrical-Power-System	2100	1800	1700	1600
Thermal-Control-System	8000	7500	7000	6000
TOTAL	25100	23390	21130	19370

The following is an example of the system prompts for baseline power consumption editing:

```
Baseline Load Power Consumption for Degraded Operation
-------------------------------------------------------------------


Mode 1 power level is power required during normal operating
conditions.
Modes 2, 3, and 4 are power consumption levels for degraded
operating conditions where mode 2 is less than mode 3 is less than
mode 4.


Type the power consumption in watts for
Communication-and-Tracking during mode 1: 100

Type the power consumption in watts for
Communication-and-Tracking during mode 2: 90

Type the power consumption in watts for
Communication-and-Tracking during mode 3: 80

Type the power consumption in watts for
Communication-and-Tracking during mode 4: 70
```

b. <u>Define a Payload</u> - If the user wishes to define a new payload to be maintained by the load library, option "1" should be selected at the load library development menu level. This will cause another menu to be displayed:

```
Load Library Development.

     1. Define a new load.
     2. Redefine baseline load power consumption.
     3. Remove an old load definition.
     4. List loads in the library.
     5. Show baseline load power consumption.
     6. Show the definition of a load.
     7. Return to Library Development Menu.

Menu Selection: 1
```

The user will then be prompted for the name of the load. The name must be one word, although hyphens may be added for clarity. In the following example the name "TEST-LOAD" is used to illustrate the load definition process.

```
Defining a New Load.
------------------------------------------------------------------

Name of this load: TEST-LOAD
```

The user is asked what the power consumption is for the defined load. If the user enters a value of less than 3000 watts, the soon-to-follow bus-connection menu will not display the choice "High Power Bus." On the other hand, if the user inputs a value of more than 3000 watts, the menu will not give the option of "Low Power Bus." If the value selected is exactly 3000 watts, High Power, Low Power, and Critical Power will all appear on the menu.

```
Power consumption (watts): 3000
Note that selecting the Critical Bus implies
that TEST-LOAD is a critical load.
```

The next thing EMES prompts for is the bus connection for the load. If the user selects "1" (Critical Bus), the load is considered "continuous." (This term is defined below.)

```
Which bus is TEST-LOAD connected to?

     1. Critical Bus.
     2. Low Power Bus.
     3. High Power Bus.

Menu Selection:
```

EMES then asks to what category the payload belongs. The user should then choose the most appropriate selection.

```
    To which category does TEST-LOAD belong?

        1. Commercial Payload.
        2. Science Payload.
        3. Technology Development Payload.

    Menu Selection:
```

The next series of questions ask how long the payload should run. All values entered must be multiples of 6 minutes. The first question asks if any warmup time is required for the payload. The user should simply press the "return" key if the question is not applicable.

```
    Type the warmup time required for
    TEST-LOAD, in minutes.
    Enter the value as a positive integer that is a multiple of 6.

    If TEST-LOAD requires no warmup period,
    then type [return].

    Preconditioning Power Period (minutes):
```

EMES next asks for the duration of the main power duty cycle. This will usually mean the amount of time the load operates when it is collecting its data, producing its resource, etc. Again, the value must be a multiple of 6 minutes. This value is not optional.

```
    Type the duty cycle of TEST-LOAD, in
    minutes, which should be a positive integer
    and a multiple of six.

    Main Power Duty Cycle (minutes):
```

User-defined loads may have an active or passive cooldown period, also referred to as the postconditioning period. The user can enter a value for either active or passive cooldown periods but never both. If neither passive nor active apply in the definition, the user need only press the "return" key.

```
    Type the cooldown power period for TEST-LOAD, in minutes.
    Enter the value as a positive integer.

    If TEST-LOAD requires no cooldown
        time, then type [return].

    Post Power Period (minutes):
    Type the passive cooldown power period for TEST-LOAD, in minutes.
    Enter the value as a positive integer that is a multiple of 6.
```

```
If TEST-LOAD has no passive cool down
time, then type [return].

Passive Cool Down Period (minutes):
```

The next series of questions requires several definitions for reference. The following definitions refer to the data or product being collected or produced during main duty cycle:

1) Continuous - Operation of this load must be continuous for the duration of the duty cycle or the data/product already collected will be lost;

2) Interruptable - Operation of this load may be interrupted before the duty cycle is completed without loss of data/product.

The following definitions refer to the payloads:

1) Restartable - The payload is not damaged if power is interrupted, and the payload can be restarted at another time;

2) Nonrestartable - The payload is damaged if the power to the load is interrupted and cannot be restarted at another time.

The questions continue:

```
If TEST-LOAD requires continuous power each
time it operates, then select 'continuous'.
Otherwise, select 'interruptable'.

The power to TEST-LOAD...

    1. must be continuous.
    2. can be interrupted.

Menu Selection:


If TEST-LOAD cannot be restarted after an
interruption, then select '2'.
Otherwise, select '1'.

The power to TEST-LOAD...

    1. can be restarted.
    2. cannot be restarted.

Menu Selection:
```

The following three questions inquire about the success criteria for the payload being defined. A success criterion is defined as the amount of time a payload must operate to achieve the intended success level. EMES allows the user to define three success levels for each payload. The success criteria are defined in minutes and must be a multiples of the load's duty cycle. For example, TEST-LOAD has a 60-minute duty cycle. If total success means running the load three times, the user would enter "180" (three times 60). Partial success could be "120," (two times 60), and minimum success could be "60." The only other restriction on success criteria values is that the "minimum" success value should never be greater than the "partial" success value, which, in turn, should never be greater than the "total" success value. They can be equal.

```
Success Criteria for TEST-LOAD :

Enter the time in minutes required for minimal,
partial, and total success of this payload.
Valid entries must be multiples of 60.

Total success time must be an integral multiple of 60.

Total success (minutes):

Partial success time must be an integral multiple of 60.

Partial success (minutes):

Minimal success time must be an integral multiple of 60.

Minimal success (minutes):
```

The next set of questions requests the knowledge needed if the payload requires any pointing. If the payload does not require any pointing, the user responds "no" to the following question. Otherwise a series of questions pertaining to pointing maneuvers will be asked.

```
Does TEST-LOAD require target acquisition?

Target Acquisition Control Axis Specification
```

EMES then asks whether this load requires one axis, two axes, or three axes of control to perform its pointing task.

```
The Axes of Control needed for Target Acquisition by TEST-LOAD
are...

    1. One Axis
    2. Two Axes
    3. Three Axes

Menu Selection:
```

The program continues by asking whether the load needs to be repointed after it is finished with one run or whether it can be immediately re-run.

```
    Does TEST-LOAD require repointing after each duty cycle?
```

The next question asks whether the load must run only at nighttime or only during daytime. If the question is not applicable, the user should select option "3."

```
    Operating Time Constraints

    If operating time is not a constraint for TEST-LOAD, then 'Either'
    should be selected.


    TEST-LOAD must  operate during the...

        1. Daytime
        2. Nighttime
        3. Either day or night

    Menu Selection:
```

The following questions inquire into the nature of the product and its use. The appropriate response is indicated by selecting the proper integer value. Or, by selecting "3" ("other"), the user can enter a selection other than those listed on the menu. This selection must be a single word.

```
    This load produces:

        1. Data or Information
        2. Physical Product
        3. Other

    Menu Selection:


    The Product of this load meets a:

        1. General Need
        2. Specific Need
        3. Other
```

```
Menu Selection:


The urgency for the Product of this load is such that:

The Result of this load is needed:

    1. Immediately
    2. In the Near Future
    3. Sometime in the Future

Menu Selection:


Select one or any combination of the listed categories.
If product use is not described above, then 'Other' should be
selected.


The Product of this load is useful to:

    1. Medicine
    2. Biology
    3. Science
    4. Military
    5. Space
    6. Commerce
    7. Other or Combination of the Above

Menu Selection:

The Resource of TEST-LOAD provides:

    1. Earth Knowledge
    2. Universe Knowledge
    3. Scarce Products
    4. Product to Relieve Human Suffering
    5. Other or Combination of the Above

Menu Selection:
```

After EMES asks the last question, it lists all the information it has just collected and asks the user whether it is correct. If it is not correct, the whole load definition process is repeated.

```
The power consumption of TEST-LOAD is 3000 watts.
TEST-LOAD is connected to the high-power bus.
TEST-LOAD is a Commercial Payload.
The preconditioning power period of TEST-LOAD is 6 minutes.
The duty cycle of TEST-LOAD is 60 minutes.
TEST-LOAD requires no cooldown time.
The passive cooldown time for TEST-LOAD is 6 minutes.
The total duty cycle of TEST-LOAD is 60 minutes.
The power to TEST-LOAD can be interrupted.
The power to TEST-LOAD can be restarted.
The total success time required for TEST-LOAD is 180 minutes.
The partial success time required for TEST-LOAD is 120 minutes.
The minimum success time required for TEST-LOAD is 60 minutes.
TEST-LOAD requires one-axis of control for target acquisition.
TEST-LOAD does not require repointing after each duty cycle.
TEST-LOAD requires target acquisition.
TEST-LOAD operates during the day.
TEST-LOAD produces output that is data-information.
The product of TEST-LOAD meets a general need.

[MORE]

The result of TEST-LOAD is needed immediately.
The product of TEST-LOAD is useful to a medicine application.
TEST-LOAD provides information on the-earth.

Is this information correct? Y
```

Note: The word "[MORE]" displayed at the bottom of the screen means that there is more information to be displayed and the program is waiting for the user to press the "return" key before displaying the rest.

Finally EMES will ask for a single-word file name, which the user must supply.

```
Entering this load into the load library...
What file will contain this load definition? TL
```

c. <u>List Loads in the Library</u> - If the user chooses selection "4" from the load library development menu, EMES displays a list of the loads it is maintaining. First, however, it asks the user to select one of three display options:

```
Load Library Development.

    1. Define a new load.
    2. Redefine baseline load power consumption.
    3. Remove an old load definition.
    4. List loads in the library.
    5. Show baseline load power consumption.
    6. Show the definition of a load.
    7. Return to Library Development Menu.

Menu Selection: 4

Send output to ...

    1. The Terminal.
    2. A Disk File.
    3. The Line Printer.
```

```
Menu Selection: 1


Load Library Index.
------------------------------------------------------------

Load Name            Definition File      Definition Date
---------          ---------------      ---------------
BATTERY-MANAGEMENT-EXPERT-SYSTEM
                      load-lib:BMES.def    Wed Jul 11 09:18:51 1984
DAY-EARTH-MAPPER      load-lib:DEM.def     Tue Jul 10 16:09:40 1984
LASER-COMMUNICATION-UNIT
                      load-lib:LCU.def     Wed Jul 11 09:14:42 1984
MEDICAL-MIX           load-lib:medmix.def  Mon JUl 9 17:39:56 1984
MATERIALS-PROCESSOR   load-lib:matproc.def Mon JUl 9 0:0:00 1984
BIOLOGY-CELL          load-lib:biocell.def Mon jul 9 0:0:00 1984
NIGHT-EARTH-MAPPER    load-lib:NEM.def     Wed Jul 11 09:08:18 1984
STAR-POINTER          load-lib:STP.def     Tue Jul 10 16:18:05 1984
SUN-POINTER         load-lib:SP.def        Tue Jul 10 16:13:16 1984
TEST-LOAD           load-lib:TL.def        Thu Jul 12 14:11:17 1984

[MORE]
```

d. <u>Remove an Old Load Definition</u> - The load library development module also allows the user to remove previously defined payloads. The user must make selection "3" from the load library development menu. EMES will then prompt the user for the name of the load to be removed.

```
Load Library Development.

    1. Define a new load.
    2. Redefine baseline load power consumption.
    3. Remove an old load definition.
    4. List loads in the library.
    5. Show baseline load power consumption.
    6. Show the definition of a load.
    7. Return to Library Development Menu.

Menu Selection: 3


Name of load to Delete:
```

The user then enters the name of the load he wishes to delete. If the user does not want to delete a load but has already entered the delete module, he may type "Control-Z," i.e., the letter Z pressed while the "CTRL" key is held down). EMES will return the user to the load library development menu without making any changes.

e. Show the Definition of a Load - The user may choose to view the definition of a previously defined load by selecting option "6" from the load library development menu. EMES will then ask the user for the name of the load to be viewed and how the information should be presented.

2. Mission Library Development

The mission development module allows the user to create and maintain mission definitions. Within this module, the user can define a new mission, remove an old mission definition, list all the missions in the library, generate mission reports, or show the definition of a specific mission. The mission development module is activated by selecting "2" from the library development menu. This results in the display of another menu:

```
    Menu Selection: 2

    Mission Development.

        1. Create a mission definition.
        2. Remove an old mission definition.
        3. List missions in the library.
        4. Show the definition of a mission.
        5. Generate a mission report.
        6. Return to top-level menu.

    Menu Selection:
```

a. Define a New Mission - To define a new mission to be executed by EMES at a later time, the user should select option "1" from the mission library development menu:

```
    Mission Development.

        1. Define a new mission.
        2. Remove an old mission definition.
        3. List missions in the library.
        4. Show the definition of a mission.
        5. Return to Library Development Menu.


    Menu Selection: 1
```

EMES will then ask for the name the user wishes to assign to the mission. As with previous names, this name must be a single word but may be hyphenated for clarity:

```
    Name of this mission: TESTMISSION
```

The series of questions EMES asks next are used to define the orbit. Orbit information is used, in turn, to calculate the dark time and light time for the spacecraft. The questions list the valid range for data the user is to supply. When the range values are "real" numbers in the Fortran sense of the word (i.e., they contain decimal points or the letter "E" indicating an exponent), EMES expects the user to supply an answer in that form:

```
    $ENTER LONGITUDE OF INITIAL ASCENDING NODE IN DEGREES.
    $SUGGESTED VALUE FOR SPACE-STATION IS 28.5.
    $THE VALID RANGE IS 0.0 to 360.0 :

    $ENTER ORBIT INCLINATION (IN DEGREES) IN EARTH EQUATORIAL COORDINATE
    $SYSTEM.
    $SUGGESTED VALUE FOR SPACE-STATION 23.5.
    $THE VALID RANGE IS 0.0 to 89.99 :

    $ENTER THE ALTITUDE OF THE ORBIT (NAUTICAL MILES).
    $SUGGESTED VALUE FOR SPACE-STATION IS 250.0.
    $THE VALID RANGE IS 0.0 to 40000.0 :
```

When entering the date, the form should be month, date, and year, separated by the "/" character. Each number should have two digits. If the month number or date is less than 10, the user should add a leading zero to make a two-digit number, e.g., "06/05/61". One-digit values ("6/5/61") are not accepted:

```
    $ENTER DATE OF ASCENDING NODE--IN THE FORM--MM/DD/YY :
```

When entering the time of the ascending node, the form should be 01:10.00. Remember that there are only 24 hours in a day and 60 minutes in an hour.

```
    $ENTER TIME OF ASCENDING NODE WITH HOURS AS AN INTEGER AND MINUTES
    $AS A REAL NUMBER--IN THE FORM--HH,MM.MM :

    $ENTER THE FIRST ORBIT AT WHICH CALCULATIONS SHOULD START.
    $THE VALID RANGE IS 1 TO 9999 :

    $ENTER THE NUMBER OF ORBITS FOR WHICH CALCULATIONS SHOULD BE MADE.
    $THE NUMBER OF ORBITS MUST BE NO MORE THAN 24 :
```

After these questions have been properly answered, the user can view the orbit definition:

```
    Send output to ...

        1. The Terminal.
        2. A Disk File.
        3. The Line Printer.
```

```
Menu Selection: 1

Mission Definition:
------------------------------------------------------------------


Orbit 1 starts at 510 and ends after 600
minutes of elapsed mission time.

In Orbit 1 the sun sets at 540 and rises after 570
minutes of elapsed mission time.


In Orbit 1 the battery power capability is 31157 watts.

At time 510 the solar power capability is 124568 watts.
At time 516 the solar power capability is 112478 watts.
At time 522 the solar power capability is 106723 watts.
At time 528 the solar power capability is 103500 watts.
At time 534 the solar power capability is 101531 watts.
At time 570 the solar power capability is 124568 watts.
At time 576 the solar power capability is 112478 watts.
At time 582 the solar power capability is 106723 watts.
At time 588 the solar power capability is 103500 watts.
At time 594 the solar power capability is 101531 watts.
At time 600 the solar power capability is 100262 watts.
At time 606 the solar power capability is 99416 watts.
At time 612 the solar power capability is 98840 watts.
At time 618 the solar power capability is 98442 watts.
At time 624 the solar power capability is 98164 watts.

Orbit 2 starts at 600 and ends after 690
minutes of elapsed mission time.

In Orbit 2 the sun sets at 630 and rises after 660
minutes of elapsed mission time.


In Orbit 2 the battery power capability is 31181 watts.

At time 660 the solar power capability is 124568 watts.
At time 666 the solar power capability is 112478 watts.
At time 672 the solar power capability is 106723 watts.
At time 678 the solar power capability is 103500 watts.
At time 684 the solar power capability is 101531 watts.
At time 690 the solar power capability is 100262 watts.
At time 696 the solar power capability is 99416 watts.
At time 702 the solar power capability is 98840 watts.
At time 708 the solar power capability is 98442 watts.
At time 714 the solar power capability is 98164 watts.

Orbit 3 starts at 690 and ends after 780
minutes of elapsed mission time.
```

In Orbit 3 the sun sets at 720 and rises after 750 minutes of elapsed mission time.

In Orbit 3 the battery power capability is 31205 watts.

At time 750 the solar power capability is 124568 watts.
At time 756 the solar power capability is 112478 watts.
At time 762 the solar power capability is 106723 watts.
At time 768 the solar power capability is 103500 watts.
At time 774 the solar power capability is 101531 watts.
At time 780 the solar power capability is 100262 watts.
At time 786 the solar power capability is 99416 watts.
At time 792 the solar power capability is 98840 watts.
At time 798 the solar power capability is 98442 watts.
At time 804 the solar power capability is 98164 watts.

Overall-Mission starts at time 510.
The Overall-Mission ends at 1050 minutes.

EMES also lists the baseline loads and their power consumptions:

Communication-and-Tracking is a base-line load.
The power consumption of Communication-and-Tracking during mode 1 is 100.
The power consumption of Communication-and-Tracking during mode 2 is 90.
The power consumption of Communication-and-Tracking during mode 3 is 80.
The power consumption of Communication-and-Tracking during mode 4 is 70.

Control-and-Display-System is a base-line load.
The power consumption of Control-and-Display-System during mode 1 is 1600.
The power consumption of Control-and-Display-System during mode 2 is 1500.
The power consumption of Control-and-Display-System during mode 3 is 1450.
The power consumption of Control-and-Display-System during mode 4 is 1400.

Data-Management-System is a base-line load.
The power consumption of Data-Management-System during mode 1 is 4000.
The power consumption of Data-Management-System during mode 2 is 3500.
The power consumption of Data-Management-System during mode 3 is 3000.
The power consumption of Data-Management-System during mode 4 is 2500.

```
Electrical-Power-System is a base-line load.
The power consumption of Electrical-Power-System during mode 1 is
2100.
The power consumption of Electrical-Power-System during mode 2 is
1800.
The power consumption of Electrical-Power-System during mode 3 is
1700.
The power consumption of Electrical-Power-System during mode 4 is
1600.


Guidance-Navigation-and-Control-System is a base-line load.
The power consumption of Guidance-Navigation-and-Control-System
during mode 1 is 3300.
The power consumption of Guidance-Navigation-and-Control-System
during mode 2 is 3000.
The power consumption of Guidance-Navigation-and-Control-System
during mode 3 is 2900.
The power consumption of Guidance-Navigation-and-Control-System
during mode 4 is 2800.

Thermal-Control-System is a base-line load.
The power consumption of Thermal-Control-System during mode 1 is
8000.
The power consumption of Thermal-Control-System during mode 2 is
7500.
The power consumption of Thermal-Control-System during mode 3 is
7000.
The power consumption of Thermal-Control-System during mode 4 is
6000.

Life-Support-and-Environmental-Control is a base-line load.
The power consumption of Life-Support-and-Environmental-Control
during mode 1 is 6000.

The power consumption of Life-Support-and-Environmental-Control
during mode 2 is 6000.
The power consumption of Life-Support-and-Environmental-Control
during mode 3 is 5000.
The power consumption of Life-Support-and-Environmental-Control
during mode 4 is 5000.
```

EMES will then ask the user if he wishes to schedule any previously defined loads in this mission. The following example illustrates how to insert a load into a mission.

```
Mission Definition.

    1. Request a load to be scheduled.
    2. List the loads which have been requested.
    3. Return to Mission Development Menu.
```

```
Menu Selection: 1

Name of this load: TEST-LOAD

Would you like to see the load definition?N
```

The user is given the option of when he would like the various runnings of the payload. If the user wants the payload to run at a certain time, he says so here.

```
Options for duty cycle scheduling

    1. Request specific start times for all duty cycles.
    2. Request start time windows for all duty cycles.
    3. Schedule each duty cycle individually.
    4. Start times for all duty cycles can be anytime.

Menu Selection: 4
```

The user is also given the option of specifying when he wishes his specified success criteria to be complete. Therefore the user can request to have minimum success (e.g., 1 run) completed a quarter of the way into the mission, partial success (e.g., 2 runs) half way through the mission and total success by the end of the mission.

```
Do you wish to specify mission success criteria for this load?Y

Minimum success (%) : 25

Partial success (%) : 50

Total   success (%) : 100

Minimum success for TEST-LOAD should be accomplished within 25
% of the mission.
Partial success for TEST-LOAD should be accomplished within 50
% of the mission.
Total success for TEST-LOAD should be accomplished within 100
% of the mission.

Minimum : 25%
Partial : 50%
Total   : 100%
```

This menu will continue to appear until the user selects the option to return to the mission development menu. EMES will then enter the mission into the mission library after requesting and being given a file name.

```
Mission Definition.

    1. Request a load to be scheduled.
    2. List the loads which have been requested.
    3. Return to Mission Development Menu.
```

```
Menu Selection: 3

Entering this mission into the mission definition library... What
file will contain this mission definition? TEST
```

b. Generate a Mission Report - The "generate mission report" module allows the user to generate graphical reports summarizing various important characteristics of a mission. Four types of reports are permitted: power capability, load profile, power margin, and battery depth of discharge (DOD).

```
Generate Mission Reports.

    1. Report Mission Power Capability.
    2. Report Mission Load Profile.
    3. Report Mission Power Margin.
    4. Report Mission Battery DOD.
    5. Return to Mission Development Menu.
```

c. Report Mission Power Capability - The "report mission power capability" module allows the user to obtain a graphic representation of the power available for each 6-minute time slot for the duration of the mission. The user is prompted for the name of the mission definition file. The system looks for the file in the mission library and reads its contents. All solar array capability and battery capability data items are summarized to obtain the solar array output profile for each daytime period and the battery capability for each nighttime period. The solar array and battery degradation factors are taken into account only if they are present in the mission definition (that is, only if this mission definition has already been processed by the EMES mission execution module). Otherwise, degradation factors are assumed to be zero. The data are shown graphically. The user has the option of showing the data on the terminal, sending them to a disk file, or sending them to the line printer:

```
Generate Mission Reports.

1. Report Mission Power Capability.
2. Report Mission Load Profile.
3. Report Mission Power Margin.
4. Report Mission Battery DOD.
5. Return to Mission Development Menu.

Name of mission: today-show

*************************
```

```
          Send output to ...

          1. The Terminal.
          2. A Disk File.
          3. The Line Printer.

          Menu Selection: 1


      30    39     48     58     67     77    86     95     105    114
     ----------------------------------------------------------------------
636  !                                                                   *
642  !                                                          *
648  !                                                     *
654  !                                                   *
660  !*
666  !*
672  !*
678  !*
684  !*
690  !*
696  !                                                                   *
702  !                                                          *
708  !                                                     *
714  !                                                   *
720  !                                                  *
726  !                                                 *
732  !                                                *
738  !                                                *
744  !                                                *
750  !                                                *
756  !*
762  !*
768  !*
774  !*
780  !*
786  !                                                                   *
792  !                                                          *
798  !                                                     *
804  !                                                   *
810  !                                                  *
816  !                                                 *
822  !                                                *
828  !                                                *
834  !                                                *
840  !*
846  !*
852  !*
858  !*
864  !*
870  !*
876  !                                                                   *
882  !                                                          *
888  !                                                     *
894  !                                                   *
```

```
 900 !                                                  *
 906 !                                                 *
 912 !                                                *
 918 !                                                *
 924 !                                                *
 930 !*
 936 !*
 942 !*
 948 !*
 954 !*
 960 !*
 966 !                                                                   *
 972 !                                                          *
 978 !                                                     *
 984 !                                                   *
 990 !                                                  *
 996 !                                                 *
1002 !                                                *
1008 !                                                *
1014 !                                                *
1020 !*
1026 !*
1032 !*
1038 !*
1044 !*
1050 !*
1056 !                                                                   *
1062 !                                                          *
1068 !                                                     *
1074 !                                                   *
1080 !                                                  *
1086 !                                                 *
1092 !                                                *
1098 !                                                *
1104 !                                                *
1110 !*
1116 !*
1122 !*
1128 !*
1134 !*
1140 !*
1146 !                                                                   *
1152 !                                                          *
1158 !                                                     *
1164 !                                                   *
1170 !                                                  *
1176 !                                                 *
1182 !                                                *
1188 !                                                *
1194 !                                                *
1200 !*
1206 !*
1212 !*
1218 !*
1224 !*
```

```
1230 !                                                                     *
1236 !                                                            *
1242 !                                                       *
1248 !                                                     *
1254 !                                                    *
1260 !                                                   *
1266 !                                                  *
```

After the graph has been displayed, the "generate mission reports" menu is displayed again. This process is continued until the user explicitly selects the option to leave the submodule.

The "report mission load profile" module allows the user to obtain a graphic representation of the power consumed by loads for each 6-minute time slot for the duration of a mission. The system looks for the file in the mission library and reads its contents. If the mission has not yet been processed by the EMES mission execution module, the system will inform the user and return to the "generate mission report" menu. Otherwise, a menu will appear inquiring as to where the user wishes to have the graph displayed.

The "report mission power margin" module allows the user to obtain a graphic representation of the power margin for each 6-minute time slot for the duration of a mission. Again, the mission must be processed by the EMES mission execution module before the requested information can be displayed. Otherwise the user is informed that the mission has not been processed.

The "report mission battery DOD" module allows the user to obtain a graphic representation of the battery depth of discharge for each 6-minute time slot for the duration of a mission. The procedure for displaying this report graphically is the same as used with the power margin and load profile modules. Note that only the power capability module can generate a graphics display without prior processing by the EMES mission execution module.

3. Mission Execution

The mission execution module of EMES corresponds to the flight software portion of the expert system. It can inspect a mission timeline, schedule any unscheduled mission events or load requests, and ensure that no resource-requirement or energy-management constraints have been violated. The user is prompted for the name of the mission to be executed:

```
Mission Execution
-------------------------------------------------------------------


Type the name of an existing mission which may or may not already be
executed.

Name of mission: TESTMISSION
```

The user is asked what percentage of solar array degradation is to be assumed for the mission:

```
Type solar array degradation between 0 and 1.00 :  0.
```

The user is next asked for the time he would like the degradation, if any, to begin. This time must be between the start time and end time of the mission.

```
Type start of degradation between 636 and 1266 :     636
```

Similarly, for battery degradation, EMES asks:

```
Type battery degradation between 0.0 and 1.00 :  0.

Type start of degradation between 636 and 1266 :     636
```

The user is then prompted for the current mission time. This can be either mission start time of the earliest degradation time, if one exists. Current mission time can never be after the earliest degradation.

```
Type current mission time between 636 and 1266:
```

The user is next asked for a new name for the mission.

```
Create a new name for the executed mission.
Name of this mission:
```

As the expert system is examining the timeline, it will print out its decisions to the terminal. EMES will then list all the loads it has scheduled in this particular execution and list their start times. Finally, the user is asked for the name of a file it should use to store the executed mission. The mission report menu will also appear so that the user can graphically see aspect s of the mission. The use of this menu has been described previously.

```
EMES Scheduled Loads
-------------------------------------------------------------------


Entering this mission into the mission definition library...
What file will contain this mission definition:
```

VI. IMPLEMENTATION DETAILS

A. MAJOR DESIGN DECISIONS

EMES is divided into three separate parts: problem setup, execution without interaction, and report generation. This design was not selected arbitrarily. It simplifies operation because the user does not need to know very much about the expert system to use it. It also made the logic of the expert system's intelligence independent of the problem-setup and reporting functions so it can be understood more readily and be more easily extracted for other uses than would be the case if it were intertwined with the other operations.

For similar reasons, we did not give EMES a natural-language interface to the user. Designing and implementing such an interface would have diluted the study effort because it would have required as much development time as the intelligence of the expert system. We also realized that natural language would be of very little benefit in this system for two reasons. First, nearly all the interfacing with the user takes place before the intelligent portion of EMES starts to run. The interface would therefore not really be with the expert system itself but with a database editor. Second, without detailed prompts and a thick user's manual, the user would not know what kinds of things to tell the program in natural language. An interface that could handle anything he asked for would be totally impractical; it would dwarf the work on the load-scheduling logic. And a less sophisticated interface would probably frustrate the user by constantly telling him that it did not understand his commands.

One of the objectives of the study was to demonstrate a simplified state-of-health monitoring and fault detection capability. In EMES these functions are limited to detecting that the power available is inadequate for the scheduled activities. The decision to limit EMES capabilities in this area was not taken lightly. Initially we considered integration of EMES with the Martin Marietta Denver Aerospace fault isolation expert system. However, we found there were more difficulties in doing so than we had initially realized, and there was less to be gained than we originally believed. The reasons were:

1) The two expert systems require different reasoning models of the power subsystem. The fault isolation program needs to know the configuration in great detail because it attempts to isolate the faulty component. On the other hand, the energy management program has no need for these details. It does, however, need to predict power available as a function of time, which the fault isolation system does not need to do, and, to make its predictions efficiently, it must use a simplified modeling of the power subsystem. As a result, if the programs are to be joined, the connection must be at a very high level of abstraction, e.g., an executive that selects one or the other;

2) The two expert systems use different kinds of expertise. The fault isolation system uses the knowledge of an electrical engineer or technician; the energy management system uses the knowledge of a systems expert. Such different forms of expertise should not be merged at a low level because the resulting program would not implement the coherent line of reasoning of either expert;

3) The two expert systems work with different time scales. The fault isolation system reasons about the current situation; the energy management system plans for times hours into the future. Again, linking the low-level internal reasoning processes is unwise.

This leaves the possibility of a high-level link. In a flight situation, such a link between the two programs might be useful. For example, the fault isolation program could detect a fault and then invoke the energy management program to plan around the problem. It could also tell the energy management system how much degradation is involved.

However, in the laboratory implementation we were designing, the fault isolation system does not run continuously waiting for a fault to occur as it would in flight. Rather, it checks simulated measurements that reflect a static situation. We therefore believed that little would be gained by linking these programs. But there was much to lose. If the programs were linked, the resulting program would be larger and more complex than the sum of the two individual programs because the designers of each program would have to know details of the other program. This would result in an increased development time and cost or a less thorough treatment of both problems. It would also result in a greatly increased testing and debugging time because both programs would have to be run to thoroughly test either. Finally, it would make the logic of each program harder to understand or to apply in another context.

B. EMES' INTELLIGENT FUNCTIONS

Scheduling is the most basic operating mode of the intelligent portion of EMES. EMES takes as input the mission power profile, i.e., solar array and battery capabilities for each time slot during the mission. The power consumption of each of the spacecraft housekeeping subsystems (each summarized in a total baseline power rating) is incorporated in this power profile. EMES also accepts a set of requests for loads to be scheduled into the timeline, along with the loads' operational constraints and mission success criteria. Scheduling rules consider the amount of power available over time, permitting graceful degradation under abnormal modes of operation.

EMES attempts to schedule the loads so the mission success criteria are satisfied as well as possible, while abiding by energy management constraints that optimize power allocation and extend the life of critical power subsystem components. The rule-based nature of EMES allows the scheduling of load requests to be triggered by data. EMES can enter the scheduling mode as a result of load requests generated either by the user or by other EMES rule modules.

EMES also has a "descheduling" mode. Descheduling is required when a change in the expected power profile occurs because of power subsystem degradation, resulting in insufficient power to operate all of the loads scheduled for execution. In this case, EMES must reason about the priorities of the payloads and other subsystem components to determine which loads should be removed from the schedule. If possible, these loads are then placed elsewhere in the timeline.

The final major operating mode is the handling of contingency situations. These situations arise when the spacecraft is operating with a degraded power subsystem, resulting in a shortage of power to run the scheduled loads. However, the contingency mode is more complex than simple degradation because the expert system does not know about the degradation in advance, i.e. the degradation occurs suddenly during the mission, resulting in the need to turn off loads already in operation.

This means that EMES must reason about the effects of interrupting an operating load on the overall level of mission success. Such reasoning is important because a badly timed interruption of power may cause the loss of an important set of data or destroy a product being manufactured. In fact, in extreme cases a payload may be destroyed or rendered inoperable for the duration of the mission. EMES must consider all these factors in deciding which loads to deschedule.

In addition to the three major modes of operation, EMES provides an additional function, timeline optimization, that may get triggered into operation at various times. Optimization occurs when EMES recognizes that a legal mission timeline is not optimal because of, for example, energy management constraints or the level of mission success achieved. Optimization rules interact with the other EMES rules to make the energy management process more intelligent.

1. Scheduling

The scheduling module is the most fundamental of all modules in the expert system. It is activated when there are load requests in the mission definition file that have yet to be scheduled. It takes as input a set of load requests and a power profile, and results in a scheduled timeline.

The scheduling control structure (Fig. VI-1) is algorithmic and starts with the first day or night of the first orbit, iteratively analyzing each day and night until the end of the mission. The actual scheduling of payloads that occurs within this process, however, consists of two sets of rule-based heuristics.

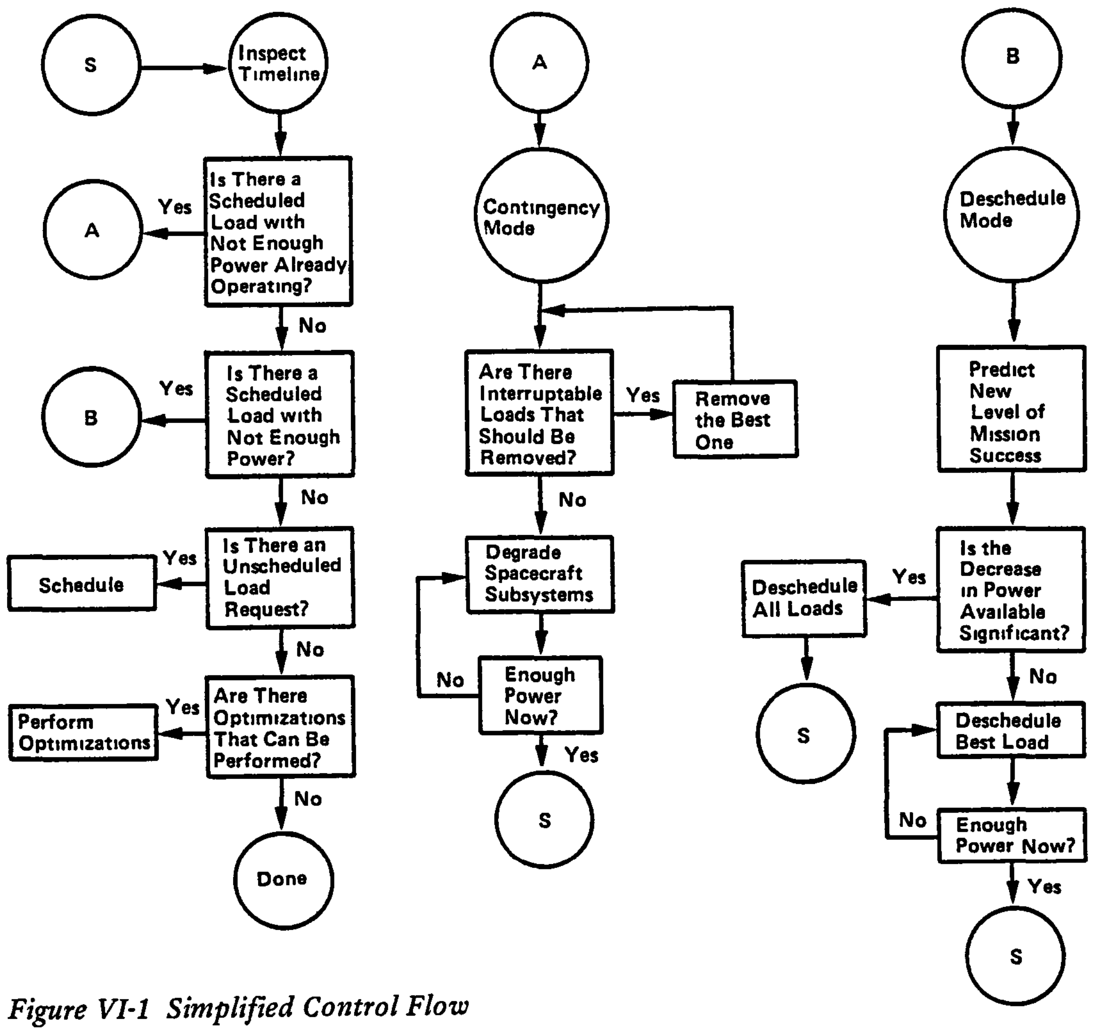

S
Inspect Timeline
Is There a Scheduled Load with Not Enough Power Already Operating?
Yes
A
No
Is There a Scheduled Load with Not Enough Power?
Yes
B
No
Is There an Unscheduled Load Request?
Yes
Schedule
No
Are There Optimizations That Can Be Performed?
Yes
Perform Optimizations
No
Done
A
Contingency Mode
Are There Interruptable Loads That Should Be Removed?
Yes
Remove the Best One
No
Degrade Spacecraft Subsystems
Enough Power Now?
No
Yes
S
B
Deschedule Mode
Predict New Level of Mission Success
Is the Decrease in Power Available Significant?
Yes
Deschedule All Loads
S
No
Deschedule Best Load
Enough Power Now?
No
Yes
S

Figure VI-1 Simplified Control Flow

The first set makes a list of all the loads desirable to be scheduled during each day and night period and then intelligently selects the best load to lay into the timeline. To decide which loads are desirable, it initially looks at which loads can run during that period. For example, day payloads can only operate during the day and night payloads can operate only at night. EMES also considers other operational constraints. For example, no two payloads that require pointing maneuvers for orientation can operate at the same time.

When the selection process is complete, the list of desired payloads is sent to the second set of heuristics, which intelligently chooses the best load to schedule next. This decision also considers energy management rules, efficient use the power, and scheduling urgency. For example, if a specific payload is not scheduled immediately, it may not have another chance to be scheduled later in the mission. After examining all of the desirable payloads, EMES determines the best load to schedule next during the same period. When little or no unallocated power remains for scheduling loads in this period, EMES recognizes this fact and begins scheduling the next day/night period.

Each time an optimum load is chosen during this scheduling process, the instantiation* mechanism incorporates the payload into the timeline and updates information about the amount of power the new load requires. This process repeats, working with the same day/night period, until EMES does not believe there is an optimal load to schedule. The scheduling module then starts working with the next day/night period.

2. Descheduling

The descheduling module is activated when EMES discovers a period in which there is insufficient power for the loads assigned. This is usually the result of using a previously scheduled timeline with a new, degraded power profile. To accommodate this power shortage, EMES must perform intelligent descheduling.

Descheduling can occur in two ways. If the level of degradation is small, EMES removes selected loads from the timeline. On the other hand, if the power shortage is severe, EMES deschedules the entire timeline. When a load is descheduled, it is put back onto the scheduling list and is treated once again as a load request. The reason for the dual mode of operation is that descheduling is not as efficient as scheduling. If much descheduling is required, it is most efficient to erase the whole timeline and reschedule under the new degraded power profile. This also leads to a timeline that makes better use of the power available.

When the power problem is relieved, EMES exits the descheduling module, and rescheduling of the descheduled loads is attempted.

3. Contingencies

The contingency mode is activated when loads are in operation and, because of some malfunction, e.g., solar array or battery degradation, there is no longer enough power to support them. EMES distinguishes this problem from the degraded mode that the descheduling module handles by checking the current time of the mission. If the current time is the same time as that of the power shortage, EMES recognizes that the space station is in a state of contingency. Otherwise, EMES enters the previously discussed descheduling mode.

In the contingency mode, EMES must use a set of transition heuristics to alleviate the power problem without damaging any of the payloads while, if possible, retaining all of the data and products already collected. The first thing EMES looks for in the overloaded time slot are interruptable loads. These loads can be turned off without loss of any of the product or data already collected.

* Instantiation of a rule is application of the rule with a specific set of data.

If turning these loads off does not solve the problem, EMES cuts power to certain subsystem loads to accommodate payloads that must continue operating. The tradeoff that EMES reasons about here is that of the astronauts' comfort (subsystem loads) versus support of the payloads. EMES will will cut back power from the subsystem loads only to accommodate payloads in operation that can't be turned off without damage to the payload or product. Thus, subsystems are never cut back in the normal scheduling mode. The subsystem modes will return to the normal operating mode as soon as possible. When this occurs, EMES is no longer in the contingency mode, and any further power problems will be handled by the descheduling module.

C. THE CONCEPTS OF "RULES" AND "GOALS"

1. The IF/THEN Approach and HAPS Formalism

The heuristics or rules of thumb used in EMES are encoded in a formalism called the HAPS production system language. This language differs from conventional languages in several ways, but the most fundamental difference is that the programmer does not specify the order in which instructions are to be executed. In fact, the statements of a program in this language can be rearranged or even shuffled with no effect on the way the program executes.

Most procedural languages provide a number of statement types. For example, Pascal provides an "IF/THEN/ELSE" type, a "DO WHILE" type, a "REPEAT UNTIL" type, a "DO CASE" type, and a number of others to declare variables, perform arithmetic, and control the sequence of operations.

The fundamental statement type in the HAPS production system language is the "IF/THEN" statement used to express a rule. Despite the apparent resemblance between this type and the "IF/THEN" type in Pascal, the meaning of the word "IF" in this statement differs radically from the meaning in Pascal. In Pascal, it means "If the specified condition is true, perform the following task before executing the next instruction; otherwise go on to the next instruction." In the HAPS formalism, the meaning of "IF" is close to the meaning of the English word "whenever." If we ignore for the moment the concept of "goals," which will be introduced shortly, the HAPS "IF" means "Any time you notice that the following condition is true for any set of data you find in memory, you might want to perform the following action." Thus, while Pascal and other procedural languages require the programmer to specify the order in which things are done, the HAPS formalism does not. The production system software decides what sequence will be used. Furthermore, the sequence is determined as the program executes, not in advance.

At any given time during program execution, more than one "IF/THEN" statement (rule) may be applicable. HAPS must then decide which rule to apply or "fire." The process of making this decision is called "conflict resolution." Conflict resolution may also be required when a single rule could be used with any of several sets of data. HAPS must pick one set.

Most of the conditions tested in the "IF" parts of HAPS rules are the existence or nonexistence of data items in so-called "working memory," and most of the actions in the "THEN" parts are insertion of data items into working memory or removal of such items. For example, the HAPS rule fragment

```
    IF:      (a b c)
    THEN:    (x y z)
```

says that if the item "(a b c)" is ever found in working memory, the program should insert the item "(x y z)" into working memory. This particular rule is not very useful, because if it is ever used ("fired") there is nothing to turn it off again. The action of the rule does nothing to invalidate the condition in the "IF" part of the rule.

Another reason the rule is not very useful is that it tests for only one specific data item. The following rule fragment is more practical:

```
    IF:      (a b =c) :1
    THEN:    (*remove :1)
             (x y =c)
```

This rule says that any time the program finds a three-element list that starts with the symbols "a" and "b" in working memory, it should remove it and insert in its place a three-element list that starts with the symbols "x" and "y." The third element of the new list equals the third element of the list that was originally in memory. This element will be a constant, which may be a number, a symbol, or an entire list enclosed in parentheses. Because this rule will fire regardless of the value of the third list element, it is far more general than the original rule. And because it removes the triggering data item from memory, the rule will not fire continuously; eventually it will run out of lists starting with "a" and "b."

HAPS knows that the third item is a variable because its name starts with the character "=." A rule may contain a number of such variables. Each variable is assigned a value when HAPS finds a data item that matches the remaining symbols in the list in which it is embedded. However, the value of such a variable will be consistent throughout the rule. For example, if a variable appears in two lists in the "IF" part of a rule, it is given a value when HAPS matches the first list with an item in memory. The variable then acts as a constant with that value in subsequent lists. This process of associating a constant value with a variable is called binding, and a variable that has been given a value is said to be bound. For example, the rule fragment

```
    IF:   (a b =c)
          (q r =c)
      THEN: ....
```

means, "Whenever working memory contains a three-element list that starts with the symbols "a" and "b," and there is also a three-element list that starts with "q" and "r," and the third element of the first list matches the third element of the second list,...." There may be many lists in working memory that satisfy these conditions. HAPS may apply the rule to any or all of them. For example, if working memory contained the lists "(a b f)" and "(q r f)," the production system could bind the variable "=c" to the symbol "f." Whis would produce the specific instance or "instantiation" of the rule:

```
    IF:    (a b f)
           (q r f)
    THEN: ....
```

Similarly, if working memory also contained the lists "(a b m)" and "(q r m)," the production system could produce the instantiation

```
    IF:    (a b m)
           (q r m)
    THEN: ....
```

When multiple data sets satisfy a rule's "IF" part, HAPS may apply the rule to them in any order.

A rule can also test for the absence of a data item from working memory. Any list in the "IF" part of a rule can be preceded by a minus sign to signify a list that cannot be in working memory if the rule is to apply. For example,

```
    IF:    (a b =c)
          -(m =c q)
    THEN: ....
```

means, "Whenever you find a list of three elements that starts with 'a' and 'b,' and there is no list of the form '(m =c q),' where '=c' stands for the third element of the first list,...."

Sometimes it is useful to use a variable as a mere place holder for an item whose value we do not care about. To specify such a variable, the HAPS formalism uses a question mark. Thus,

```
    IF:   (a b ?)
    THEN: ....
```

means, "Whenever working memory contains a three-element list that starts with the symbols 'a' and 'b,' ...." The value of the third item in the list is not bound.

Every rule must be applied in the context of a "goal." Only rules that bear on the currently active goal are considered as candidates to execute or "fire." Rules can also suspend direct work on the current goal by setting up subgoals that must be achieved to accomplish the larger goal. Similarly, they can reverse this operation by declaring a goal to have been successfully achieved, sending the production system back to a suspended higher level goal.

The ability to create and manipulate goals allows EMES to use its heuristic knowledge in the proper problem-solving context. It also makes the expert system easier to modify and extend because it promotes modularity in the rule base, increases code readability, and decreases debugging time.

The main goals used in EMES are:

1) Initialize timeline: Initialize the mission timeline with power capabilities and day/night periods, and allocates baseline loads;

2) Inspect timeline: Inspect timeline for needed scheduling and for degradation and contingency situations;

3) Schedule: The top-level goal for scheduling a load;

4) Handle power shortage: The goal for degraded mode of operation;

5) Handle contingency: The goal for the situation where there is not enough power and a load is already running;

6) Schedule initializations: Initialize for scheduling;

7) Determine desired loads: Determine a set of desired loads for scheduling from the set of all loads;

8) Determine optimum loads: Determine one optimum load for scheduling;

9) Schedule a load: Schedule a particular load;

10) Drop baseline: Lower the baseline. The baseline is the base level of loads needed for spacecraft operation. The levels run from normal to survival mode;

11) Unschedule a load: The goal for unscheduling a load;

12) Calculate pointer opportunity.

Several additional goals are used for "bookkeeping" and do not directly relate to the heuristics a human expert would use to solve the problem:

1) Update legal slots;

2) Update load opportunity;

3) Update load opportunity by success;

4) Update active slot;

5) Update power consumed;

6) Update pointer opportunity;

7) Update pointer opportunity by success.

HAPS allows the programmer to specify the LISP functions to be executed in either the "IF" part or the "THEN" part of a rule. The syntax of the LISP invocation is identical to that used in normal LISP programming except that LISP function names are preceded with an asterisk. Bound variables can be passed to LISP functions and the functions can, in turn, invoke other functions or even Fortran code.

A rule from EMES illustrates how these principles work in practice. EMES has a set of rules that allow it to determine which of the major modes of operation should be entered. To recognize the contingency scheduling mode, it uses a heuristic that can be stated in plain English:

```
IF:    I am inspecting a mission timeline,
       AND there is any time when more power is scheduled to be
           consumed than is available,
       AND EITHER the current mission time equals the time of the
           power shortage
       OR a load that is now running will encounter the power
           shortage before it is scheduled to finish running,
THEN:  Enter contingency mode.
```

In HAPS notation, this heuristic becomes the formal rule:

```
; Recognize contingency mode
  (INIT-PRODUCTION recognize-contingency
    CONTEXT:
       (OBJECT:  inspect-timeline)
    IF:
       ; more power used than is available
       (power-available      =t1      =watts1)
       (power-consumed       =t1      =watts2)
       (*lessp               =watts1  =watts2)

       (current-mission-time  =t2)

           ;mission time is same as time of power problem
       OR [(*eq   =t1    =t2)

           ;a load is running and there is a power problem
           AND [(scheduled  ?  (=t3 =t4))
               -(*lessp         =t2 =t3)
                (*lessp         =t2 =t4)
               -(*lessp         =t1 =t3)
                (*lessp         =t1 =t4)]]
    THEN:
       ;notify user we have a contingency situation
       (*explain ([CR] "A contingency situation has been"
           "recognized!" [CR] "Entering handle contingency mode"
           [CR]))

       (GOAL
           OBJECT:        handle-contingency
           MISSION-TIME:  =t2))
```

Notice that this rule is identified as an "init" production. This means that the rule will be given priority over other rules or "productions" within this rule's goal context. Specifically, if HAPS is trying to determine which of several rules to fire, it will not consider any ordinary productions if it can fire any "init" productions. Similarly, regular productions have priority over so-called "default" productions. The programmer can use this feature of the HAPS formalism to avoid the inefficiency that might result from random rule selection. In this example it prevents HAPS from attempting to produce a schedule it will later have to throw away.

A second thing to notice about this rule is that the value bound to the variable "=t2" is passed as a private object to the "handle-contingency" goal context under the name "MISSION-TIME." This information can be used much as if it were in working memory, but it is automatically erased after the goal "handle-contingency" has been satisfied. This feature of the HAPS formalism is analogous to the passing of parameters to subroutines in procedural languages.

2. Benefits of a Production System

A production system is not required to implement expert systems or even to use the IF/THEN approach to programming described previously. However, they do provide advantages in:

1) Speed of implementation - Production system languages allow the programmer to ignore such low-level implementation details as how testing of the "IF" conditions of rules is done, how binding of variables is accomplished, how private memory for goals is allocated, etc;

2) Transparency of code - Experience has shown that experts who do not know computer programming can quickly learn to understand rules written in a production-system language. The same cannot be said of programs written in "C," LISP, Fortran, or Pascal;

3) Speed of operation - A straightforward implementation of a rule-based program in a conventional procedural language is very inefficient. Consider, for example, the logic shown here in pseudocode,

```
For each rule
    For each combination of data items in working memory that
    matches the constants in the rule [
        Bind the variables in the rule to the constants that are in
        corresponding positions in the data-item elements;
        If the condition in the "IF" part of the rule is true with
        this set of bindings THEN
            Perform the actions in "THEN" part of rule;
    ]
```

With 200 data items and an average of four items involved in the "IF" part of each rule, the program's inner loop could test over a billion data combinations for each rule. With the number of rules EMES has, the program could test over 200 billion rule/data combinations before finding an applicable data set and rule to "fire." This approach is patently impractical for a useful expert system.

The clever programmer can do a number of things to speed execution of the program. For example, he can maintain a list of eligible rules, updating this list each time the contents of memory are altered. However, in doing so, he is creating a special-purpose production system whether he realizes it or not. The decision he then faces is not whether to use a production system but whether to write a new one or use one that already exists.

If he elects to write a new one, he must also decide whether to make the production system and rule base separate, distinct, recognizable entities or to merge them. Most implementers have chosen to keep them separate because this makes the rule base easier to understand and allows reuse of the production system in future projects. It also makes the rule base far easier to modify. This is perhaps the greatest virtue of a separate production system because a typical expert system requires many revisions of the rule base before it performs properly. An interwoven production system would require manual revision of all the tricks used to speed rule selection, dramatically increasing the time required to debug the program. However, writing a general-purpose, reusable production system is a large undertaking, and a special-purpose production system may be considerably faster because the overhead of generality is removed and domain-specific knowledge can easily be built into the rule-selection process.

The programmer might also abandon the "IF/THEN" approach entirely. For problems that do not require extensive opportunistic application of rules, it is practical to use conventional languages and design practices. Some would deny that the resulting product constitutes an expert system. It will also probably be less able to cope with incomplete information than a rule-based system. However, the issue should not be whether researchers in artificial intelligence would certify it as an expert system. Rather, the issue should be whether it meets the needs of the user. The nature of EMES tasks did require opportunistic rule application, although perhaps not to the extent we used it.

3. Why HAPS was Chosen Over Other Production Systems

Many production systems are available for developing expert systems. Among the better known are OPS5, KAS, ROSIE, EMYCIN, and MRS. In looking at these, we found none that combined the speed and flexibility we believed we needed to implement EMES. For example, OPS5 is a general-purpose knowledge representation and inference system. It derived from its predecessor, OPS4, which was developed for an expert system to configure VAX computer systems. OPS5 is mature and very fast. But it is not very flexible because:

1) OPS5 restricts the tests in the "IF" portion of rules to arithmetic comparisons of two numerical values ("greater than," "less than," etc) and to tests of equality or inequality. Similarly, the "THEN" part of a rule is restricted to a set of actions OPS5 provides for. Furthermore, OPS5 makes no provision for interfacing an expert system with Fortran, which EMES needs for analyzing the power available over time, times of local sunrise and sunset, and similar calculations. HAPS allows all the tests and actions allowed in OPS5 and any function that can be written in LISP or Fortran as well;

2) With OPS5, information stored in the expert system's database must be represented as simple lists with no embedded lists. This restriction would make knowledge representation in EMES both difficult to implement and difficult to decipher. HAPS allows an arbitrary list structure.

Although the MRS (metalevel representation system) environment has the flexibility of HAPS, we found HAPS to be approximately 30 times faster than MRS in a benchmark test comparing the two. Because EMES can take hours to solve some problems with HAPS, it is totally impractical to use a production system this much slower.

Other production systems investigated had similar problems in speed, flexibility, or features that were not well suited to the nature of EMES.

Some of the features of these other production systems could have been useful, however. For example, unlike such production systems as KAS and EMYCIN, HAPS does not have a built-in uncertainty model. Such a model might have been useful, although we did not find it necessary to implement one in EMES. Similarly, some production systems come with a full development environment to assist in the creation of expert systems. Such an environment could have shortened the time required to implement EMES.

4. How HAPS Works

a. Background and Motivation for HAPS - As expert systems move into broader and more complex domains, new constraints on their design come into play:

1) The systems must be able to handle larger rule bases because an expert system's breadth of knowledge and level of expertise both depend on the number of rules in its rule base. If a system is to handle more complex domains at a higher level of expertise, the size of the rule base must increase;

2) The system must be able to handle a much larger working memory set. One reason for this is that when the domain of an expert system becomes larger, more domain-specific knowledge is required;

3) Some systems will have to operate in real time. Many new applications, including automation of satellite subsystems, will require the processing of data streams in real time. These domains also often require problem solving in time-critical situations so the expert system must be flexible enough to consider the constraints imposed by the availability and distribution of scarce system resources during the problem-solving process.

Such constraints imply that efficiency concerns will become increasingly important in the design of future expert systems. This demands the development of new tools for construction of the expert systems subject to these constraints.

Most of the research concerning the efficiency of production systems focuses on pattern matching, the process by which patterns from the conditional ("IF") portions of rules are compared to data items in working memory. When all of the conditionals of a rule or "production" are true for a specific set of data, the production may be "instantiated," or fired with that set of data. At any given time, several rules might be applied and a large number of data items might be used with each. The set of all possible production instantiations is known as the conflict set.

Pattern matching to produce or maintain this set is the most time-consuming operation the production system must perform and is therefore the system bottleneck. The most widely known and most efficient of the pattern-matching algorithms is the Rete match algorithm [Forgy 1982]. This algorithm takes advantage of two important characteristics of expert systems:

1) Pattern similarity - Because productions are tested against the same set of data items, many of the patterns will have similar characteristics so at least some of the matching for many of the patterns can be done simultaneously;

2) Temporal redundancy - The contents of working memory change slowly over time. In any given recognize/act cycle, a few data items may be added and a few may be modified or removed, but most remain the same. Thus pattern-matching information can be saved from cycle to cycle with only a few modifications.

Taking advantage of these concepts requires the compilation of production patterns into a "discrimination network." Data elements entering working memory are sent through the network starting at its "root." The conflict set is modified at the network's terminal nodes. These principles lead to the Rete match algorithm. With this algorithm the execution time for a single firing is very insensitive to the number of data items in working memory, which is why it is used in the interpreters of most state-of-the-art production systems.

Introducing goal-directedness into a production system makes writing of programs for that system easier. For example, OPS5 provides an alternative conflict resolution strategy that facilitates means-ends analysis [Forgy 1981]. Other systems are explicitly goal-directed [Sauers and Farrell 1982]; the system builds an explicit hierarchy of goals during execution. At any given time, such a system is focusing on a single goal, and the system objective is to find a way to achieve that goal.

Conceptually, productions in a goal-directed system are expressed in the form:

```
IN a given goal context,
IF some set of conditions is true,
THEN perform some set of actions.
```

The use of an explicit goal hierarchy has many advantages. From the user's standpoint, goal-directed programs are easier to construct because a production is prohibited from firing unless it is applicable to solving the specific subtask at hand. Equally important, however, is the increase in runtime efficiency that can be obtained by incorporating goal-directedness into a system design. This efficiency comes about two ways.

First, because restricting productions to firing in a particular goal context forms natural partitions in the rule set, a set of equivalence classes can be defined. Productions in the same class are all ones that fire in the same goal context. During execution, the interpreter knows which class of productions is relevant to achieving the current goal, and only productions in that class need be considered. This means that no class of productions needs to be processed until it becomes relevant to the problem-solving process.

Similarly, the nature of the goal hierarchy can be exploited in the design of the working memory structure. Often during operation of a goal-directed system, data items are inserted into working memory that are relevant only to achieving a given goal. Once a method for achieving that goal has been determined, these data items are no longer needed.

This characteristic can be exploited to improve efficiency by introducing hierarchical levels of working memory. Data elements can be declared local to a particular goal; when a goal is achieved, its local data elements disappear. Such a scheme allows the working memory structure to grow hierarchically along with the goal structure. This is important because it permits all processing resulting from the creation of a given data element to occur only within a limited local environment.

While the notion of goal-directedness is by no means novel, the incorporation of an explicit goal hierarchy into a production system architecture is important in terms of efficiency because it provides two key capabilities:

1) Productions can be integrated into the problem-solving process in such a way that system resources are spent in processing only productions that apply to the solution of the current subtask;

2) A hierarchical working memory scheme can be introduced, allowing for more efficient management of the large, domain-dependent knowledge bases future expert systems will use.

An explicit goal hierarchy also allows for the construction of expert systems with a much more general control structure, which more closely models the problem-solving processes of a human expert in a dynamic environment.

b. Production Hierarchies - Much research has been done on the nature of the knowledge contained inside a production. This research has led to a distinction between productions and metaproductions. In general, a standard production represents a piece of expert knowledge specific to a given domain. A metaproduction contains metaknowledge, i.e., knowledge about the system's knowledge and how to use it. The use of such knowledge allows a system to make such high-level decisions as which of a set of solution paths is most likely to lead to the best answer to the problem at hand.

Usually metaproductions have precedence over regular productions because of the nature of the knowledge encoded in the metaproductions. It is generally preferable to make high-level decisions concerning how the system will attempt to solve a problem before considering the minute details of the solution itself.

This idea has been important in the design of state-of-the-art hierarchical planning systems [Stefik 1980]. A hierarchical planner first produces an abstract representation of a plan to accomplish a task. Then, as more constraints are considered, this plan gradually becomes increasingly detailed until the final plan is produced.

These concepts can be generalized to produce the notion of production hierarchies, in which productions are grouped in sets--productions are placed in the same set if they are similar according to specified criteria, e.g., level of knowledge represented or level of detail of problem solution produced.

Now suppose the interpreter can fetch selected rule sets into the environment at execution time. The system starts with some initial rule sets provided by the user. These rules can recognize situations that require additional rule sets, and these can be loaded into the environment and declared local to a given goal. The resulting production hierarchy can grow as new levels of subproblems are identified.

This is best illustrated through an example. Suppose an expert system has the task of troubleshooting some malfunction in a satellite system. Initially only general problem-solving procedures and high-level troubleshooting rules reside in the environment. One production might notice that the malfunction was caused by a loss of power, and might suggest focusing on the power subsystem. As the malfunction becomes isolated to a smaller subset of possible faults, the interpreter may call in a production set specific to the power subsystem of a particular satellite, a production set specific to solar arrays, and even a production set particular to the environmental causes of solar array failure.

One advantage of this scheme is that it provides an efficient way to manage large rule bases. Individual groups of productions can reside in separate source files and on different physical devices. A group of productions does not need to reside in memory until it is needed. Also, the addition of a new production set is often triggered by the system's attention to a particular goal. In these instances, not only can the goal hierarchy be used as a framework for building production hierarchies, but it can also serve as a framework for dismantling them. If a production set was brought into memory in response to the creation of a particular goal, it can be removed from memory when that goal has been achieved.

Adding an hierarchical production scheme to the expert system environment works well in conjunction with the goal-directed partitioning strategy discussed earlier. The resulting system is one in which, conceptually, a library of production sets relevant to different problem-solving tasks is available. Several sets are selected during system execution, and the goal-directed nature of the system guides the search through these selected sets. Together these techniques provide an efficient mechanism for managing large rule sets.

c. Alternative Memory Structures - The efficiency of a data representation is usually measured along two dimensions: space and time. We have already considered the space efficiency of working memory and a memory management scheme based on hierarchical levels of working memory has been described. We still need to address the time efficiency of the operations performed on working memory.

The operations usually performed on working memory are updating its contents and pattern matching against the individual data elements. The Rete match algorithm previously discussed efficiently implements these operations.

Recall that the Rete algorithm exploits pattern similarity and temporal redundancy in the database. Temporal redundancy is therefore critical to its efficiency. Imagine a situation, however, in which a set of data items changes frequently, e.g., during every recognize/act cycle. Each time any data element is updated, all the production instantiations in the conflict set that depend on that data item must be removed or tagged invalid. Then the new value of the data item must be matched again to form the set of valid instantiations.

This is clearly inefficient, yet this is precisely the situation that exists in a real-time environment. Real-time systems must deal with such data as health and status information, links to other real-time information, and feedback from sensor systems. Such data may change hundreds of times in the interval between production cycles.

One solution to the resulting efficiency problem is to provide additional, globally accessible memory structures in addition to standard working memory. A variety of these structures has been implemented in HAPS, including system attributes, arrays, and tables. Pattern matching must now occur in two stages. Matching against standard working memory remains a data-driven process, i.e., matching is done at the time the database changes. Matching against alternative memory structures, however, must be performed dynamically at instantiation time, i.e., at the start of each recognize/act cycle.

In addition to solving some of the problems of processing real-time data, this scheme simplifies the interface with other software systems. This permits the development of expert systems consisting of many components, not all of which are rule-based. Finally, this scheme allows for creation of separate match procedures for each data type so in future systems the idiosyncrasies of each memory structure can be identified and exploited in the same way the Rete algorithm takes advantage of the temporal redundancy in standard working memory.

d. Conflict Resolution - Often during the running of a rule-based program, the production system finds either that any of several rules could be selected next or that any of several sets of data could be used in instantiating one or more rules. The procedure it uses to pick a single instantiation of one rule is called conflict resolution. Two strategies are commonly used for this--elimination strategies and selection strategies. As the name suggests, an elimination strategy rules out certain alternatives. Selection strategies are then used to choose among the remaining alternatives. An example of an elimination strategy is refraction, which rules out production instantiations that have already fired in the past. A common selection strategy is specificity, i.e, favoring specific productions over more general ones.

These standard conflict resolution strategies have an important flaw--they are generally unaware of the characteristics of the system environment and therefore cannot respond to changes in the environment. This is especially true of characteristics that affect system efficiency. An intelligent system should be able to maintain a set of performance statistics over time and use them in selection strategies to improve efficiency.

Suppose, for example, a large expert system has been installed and has been operating in the same environment for a long time. Assume also that a statistical summary of past system performance is available. Now, in a certain context, several productions are candidates to fire, each representing a different approach to the solution of the problem at hand. One production may have a history of taking a long time to run, and another may have a history of seldom leading to a good solution. It might be desirable to eliminate these productions immediately. Of the remaining rules, some might be more likely to lead to long-term solutions than others, and it may be desirable to select these first. Thus conflict resolution can be used to help allocate resources to tasks with the greatest probable payoff.

Conflict-resolution strategies can also address the constraints of real-time operation by enabling productions to alter conflict-resolution strategies in critical situations. For example, suppose an expert system is given a limited amount of time to solve a critical problem. If time begins to run out, the program could consider only productions that always produce answers quickly. Although this may provide only a short-term failure workaround, the time-critical nature of the situation will have disappeared, and the system will then have more time to pursue a more permanent solution.

Finally, many systems require that some operations (in this scheme, for example, the matching against alternative memory structures) be performed at instantiation time. The conflict resolution process usually assumes that it is given a valid conflict set--a set of production instantiations all of which have all their conditionals satisfied. Here, however, the system has at instantiation time only a list of candidates for the conflict set. These candidates are not valid members of the conflict set until they are found to satisfy the tests to be performed at instantiation time.

This suggests that a type of meta-conflict-resolution procedure should be used. Meta-conflict-resolution strategies can consider such statistics as cost for instantiation and can eliminate some candidate instantiations before they are tested for validity. For example, if the system is performing under a time-critical condition, it is reasonable to immediately eliminate candidate instantiations that would require a great deal of some expensive processing, e.g., inferencing, to enter the conflict set.

e. Control Strategies - Most production systems use the same general control structure; this is the recognize/act cycle. In goal-directed systems, an additional issue must be addressed. The system has a hierarchy of goals to achieve, and it must choose one goal as the focus of attention in each cycle.

One of the most common search strategies for this is a depth-first search. In this strategy, a goal is pursued until it is achieved, or, if it sprouts subgoals, until all of its subgoals have been achieved. An alternative strategy is breadth-first search, in which all of the subgoals of a particular goal are expanded one level before any deeper expansion of any subgoal occurs.

These search strategies are adequate for some applications, but they have an inherent flaw--neither can respond to changes in the system environment so neither can take into account the particular characteristics of the problem being solved.

This flaw adversely affects system efficiency. For example, consider a system that has two ways to achieve a particular goal, each of which requires the achievement of a different subgoal. If the system does not have enough information to select the more efficient method and uses a blind depth-first search strategy, it may waste large amounts of some scarce resource, e.g., time, pursuing one goal when pursuing the other would have led to an immediate solution.

Because this problem is closely related to those we discussed previously in analysis of conflict resolution, it seems appropriate here to apply techniques traditionally use for conflict resolution, i.e., the use of selection and elimination strategies. We can use system performance statistics to produce a more effective search of the goal hierarchy, thereby allowing system resources to be applied in the directions where they will most likely produce desirable results.

For example, to solve the search problem one might rule out the pursuit of goals that have already consumed more than a specified amount of a given system resource. Alternatively, a selection strategy may be used that pursues those branches of the goal tree that have produced the largest amount of new information for each unit of some selected resource. Similarly, these control strategies can be used to prevent the pursuit of goals that have failed under similar circumstances in the past and to prevent infinite recursion.

Control strategies work well in conjunction with the explicitly represented goal hierarchy. For example, the goal may be achieved through either of two different sets of subgoals. The system can begin to pursue one solution path and then decide, through the use of a control strategy that monitors the depth of subgoal expansion, that it might be more advantageous to switch to an alternative solution path. The resulting control structure is one that can recover from situations human experts could avoid, yet which could not be handled with more traditional production system architectures.

f. Inference - Another problem directly related to memory management is the need for inference. This need can be demonstrated quite easily. Suppose some pattern in the conditional portion of a production does not match. There are at least two ways this can occur:

1) The data item being tested for represents a proposition that is false in the current context;

2) The proposition is true in the given context but is not explicitly represented in working memory.

In the latter situation, we may be able to infer the truth of the desired data element from other data items that are explicitly represented. In this case, an interpreter that does not permit inferencing will disallow the execution of what we would like to be a valid production instantiation.

Many knowledge representation schemes provide automatic inferencing capabilities [Genesereth, Greiner and Smith 1980]. In these schemes, inferencing is performed at the time the database is queried. Such inference mechanisms usually require the representation of some form of metaknowledge. However, these schemes are not currently applicable in the production system scenario, primarily because production systems cannot take advantage of such properties as temporal redundancy unless the data items are explicitly represented in memory.

For this reason, inferencing in a production system is an expensive operation. One method for increasing its efficiency is to modify the inference scheme as follows:

1) The individual patterns on which inferencing is permitted are tagged. This ensures that inferencing is permitted only on those clauses for which it is desirable;

2) Inferencing is not performed when data elements required for instantiation of a production are explicitly represented in memory;

3) Inferencing is delayed until instantiation time and is only executed for instantiations that have passed the meta-conflict-resolution process.

These modifications allow a cost to be associated with any given inference mechanism so potential instantiations that would require extensive calculations to test their validity can be eliminated by meta-conflict-resolution strategies.

Finally, it has been shown that the inference procedures required to derive new data elements are not uniform over all data types [Fox 1979] so it is wise to let the user define external inference routines and indicate which types of data element these procedures are designed for.

Combining these ideas results in a simple inference scheme that provides many of the advantages of automatic inference mechanisms without imposing an unnecessary strain on system resources.

g. How HAPS Addresses These Issues - These considerations have led to the design of a new production system architecture known as HAPS, the hierarchical, augmentable production system architecture. HAPS is a goal-directed system that allows both hierarchical levels of working memory and the dynamic construction of production hierarchies. HAPS also provides predefined global memory types designed to facilitate the implementation of large expert systems for real-time situations.

The system also provides modular, modifiable sets of control strategies and conflict-resolution strategies, making the system responsive to changes in its environment. These strategies take into account cost estimates, the history of system statistics, and the availability of scarce system resources to more effectively guide the problem-solving process.

User-declared inference procedures are also handled, and cost estimates of operations performed at instantiation time can be included in meta-conflict-resolution strategies.

Finally, the system is equipped with a sophisticated production compiler, HAPSZT, which is designed to increase the overall level of system efficiency.

D. RULES USED IN EMES

EMES uses a large collection of heuristics to schedule the various loads. These do not always correspond one-for-one with the rules in HAPS formalism, although the agreement is close. Production systems differ considerably in their expressive power, and what can be said in one rule in one production system's formalism may require two or three rules in another formalism. None can match English in expressive power. In EMES, 77 heuristics are expressed in approximately 150 rules. In some cases it was necessary to use several rules to accomplish what is described here in one plain-English rule of thumb.

"Initialize the Timeline" is the top-level goal of EMES, i.e., the goal that is active when EMES starts to run. Several of EMES' heuristics, grouped under this goal context, are used to initialize the expert system:

1) Calculate day power available with degradation - When solar degradation has been introduced into the system for time t, calculate the power available at time t by applying the degradation to the solar array capability at time t;

2) Calculate day power available without degradation - If there is no solar degradation at time t, the power available at time t is simply the solar array capability at time t;

3) Calculate night power available with degradation - When battery degradation has been introduced into the system for time t, calculate the power available at time t by applying the degradation to the battery capability at time t;

4) Calculate night power available without degradation - If there is no battery degradation at time t, the power available at time t is simply the battery capability at time t;

5) Calculate day/night periods - For every orbit in the mission, designate a period whose interval is greater than or equal to time1 and less than time2 for the day portions and the night portions of the orbit. Here we are separating the orbits into day and night periods and identifying when these periods start and end;

6) Calculate day/night period spanning orbits - A contiguous day/night period might have been erroneously declared to be two separate periods if the period overlaps orbits. Look for such cases and make one contiguous period out of two periods. More precisely, if the end time of a day period equals the start time of another day period, combine the two periods into a single day period beginning at the start of the first period and ending at the conclusion of the second period;

7) Initialize total day/night - If the total period for the day and/or the total period for the night hasn't been initialized, then initialize the total period for the day and night to zero;

8) Calculate total day/night - Update the total period for the day or the night by adding the duration of a particular day or night period to the current total period. Day and night totals are calculated separately;

9) Initialize total baseline power consumed - If the total baseline power consumed has not been initialized, initialize it to zero;

10) Calculate total baseline power consumed - If the total baseline power consumed is initialized, calculate the total baseline power consumed as the sum of the power consumed by all the baseline loads for the current operating mode;

11) Determine survival mode - Establish the survival mode for minimal baseline load operation as mode 4;

12) Initialize power consumed to total baseline power consumed - After all baseline loads have been scheduled, calculate and remember the baseline power consumed for each 6-minute interval as follows. For each time t that marks a 6-minute interval and for the applicable baseline mode of operation (normal, degraded, etc), the total baseline power consumed at time t is the sum of the power consumptions of all baseline loads;

13) Determine baseline mode - If the baseline operating mode has not been established for time t, let the baseline mode be the mode for which the difference between the total baseline power consumed and the power available at time t is the least but greater than zero;

14) Not enough power for baselines - If not enough power is available at time t to support the total baseline power consumed, recommend using a different mission profile to generate a timeline;

15) Cannot operate mission in normal mode - If the baseline operating mode is not mode 1 (the normal operating mode), inform the user that the mission cannot be supported in the normal manner and suggest trying to process a different mission profile;

16) Ready to inspect - If all initializations are completed, insert the goal "inspect the timeline."

The following heuristic forecasts the required success level for the mission and applies in the goal context "forecast":

17) Determine forecasted success goal for the mission - If we haven't determined the success goal for the mission, insert into working memory the information that the required success level is "total success." Then inform the user and declare forecasting successful.

In a flight system this heuristic would certainly be replaced with a less arbitrary one.

The following four heuristics, which apply in the goal context "inspect-timeline," are used to inspect the timeline and recognize "contingency" situations before the degraded or scheduling mode is entered:

18) Recognize contingency - If there is any time when more power is scheduled to be consumed than is available and either the current mission time equals the time of the power problem or a load is running during the power problem, enter the contingency mode;

19) Recognize power shortage - If there is a time when more power is schedules to be consumed than is available and heuristic (18) does not apply, enter the degraded mode;

20) Do Scheduling initializations - If the scheduling mode has not been entered for this mission definition yet but you are about to do so and scheduling has not previously been initialized, initialize it and then go directly into the scheduling mode;

21) Do scheduling - If there are load requests that haven't been scheduled yet, and there are no recognized power shortages in the mission, enter the scheduling mode.

The following heuristics, which apply in the goal context "sched-init," are used to initialize scheduling:

22) Initialize forecast - If the forecasted success goal has not been determined, insert the goal that enables entrance into the forecasting stage, which predicts achievable success level;

23) Initialize load opportunity - For every load request, and for the total, partial, and minimum success levels, find the dc time, determine the mission start and end, and calculate the load opportunity as the dc time divided by the difference between the mission start and end times;

24) Initialize pointer opportunity - For every pointer load request, and for the total, partial, and minimum success levels, initialize the pointer opportunity for the period, i.e., day or night, in which the pointer load operates. This is done by inserting the goal to calculate pointer opportunity;

25) Initialize legal slots - For every load request, initialize the legal slot for the load as the entire mission. Then insert the goal to update legal slots by applying constraints, e.g., two operations of the same load cannot run concurrently, two pointer loads may not operate simultaneously, etc;

26) Initialize active slot - Initialize the active slot to the current mission time.

Heuristic (27) handles scheduling and is applicable under the goal context "schedule":

27) Choose a load to be scheduled - If there is still time remaining in the mission, find the active slot, determine if there are any unscheduled load requests that could be scheduled in the slot, and insert the goals to determine desired (legal) loads and, from the desired loads, determine the optimum load. The optimum load is the load that will be scheduled.

The following four heuristics determine desired loads to schedule. These rules apply in the goal context "determine-desired-loads":

28) Desired load for inclusive legal slots - Determine which loads can be scheduled in the active slot by collecting all legal loads and attaching the earliest possible start time. A legal load is one whose duty cycle is within an active slot time duration;

29) Desired continuous load for encompassing legal slot - Pick a desirable load when the legal slot encompasses the active slot and the load under consideration is a continuous load, i.e., cannot be turned off once it is turned on. The recommended start time for such a load is the start time of the active slot;

30) Desired load for legal slot after start of active slot - Determine a desired load whenever the legal slot starts after the start of the active slot and stops after the end of the active slot but the duty cycle of the load still fits within the active slot time period;

31) No desired loads selected - If no load can be legally scheduled for the current active slot, update the active slot.

The following heuristics pick the optimum load for scheduling from a list of desired loads, i.e., ones that can be scheduled during a particular period. Picking the optimum load involves ranking, pruning, and choosing the event of the optimum load and the time to schedule that event. These rules apply in the goal context "determine optimum loads":

32) Establish ranking, selecting, and event choice subgoals - If the context is "determine optimum load," insert the goals to rank loads, pick the optimum load, and choose an event of the optimum load to schedule.

Three heuristics apply in the goal context "rank for scheduling" and assign each load a rank for used in scheduling:

33) Rank according to pointer opportunity ratio - If a desired load is a pointer and the pointer opportunity ratio for the forecasted mission success is greater than or equal to 64%, rank the load higher than any load that is not a pointer. The pointer opportunity ratio is defined in heuristic (61);

34) Rank according to opportunity ratio - If loads A and B are desired loads to schedule, and if load A's opportunity ratio is greater than load B's for the forecasted mission success, rank load A higher than load B for opportunity ratio. This ratio is defined in heuristic (58). Each load may have a different ratio;

35) Rank according to power consumption - If loads A and B are desired loads to schedule, and if load A consumes more power than load B, rank load A higher than load B for power consumption.

The following heuristics apply in the goal context "pick optimum load for scheduling," and their function is just what the goal name implies. They pick an optimum load from a set of desired loads, which are ranked for scheduling via various justifications:

36) Pick the one that's best - If the optimum load hasn't been chosen yet, the desired load that is ranked higher than any other desired load is the optimum load;

37) Pick a pointer from many - If there are pointers that are desired loads and an optimum load hasn't been chosen yet, arbitrarily choose one of the pointers as the optimum load for scheduling;

38) Prune off all nonpointers - If there is a pointer that is a desired load, and the pointer opportunity ratio is greater than 65%, no nonpointer load is to be considered for scheduling;

39) Prune opportunity versus power consumption - Prune off opportunity ranking if a load's opportunity ratio is less than 75% and power consumption is a lower ranking for that load. In other words, don't worry about opportunity ratio if the ratio is less than 75%, because power is a bigger concern with ratios this small;

40) Prune multi-outranked loads - Prune off nonpointer loads that are outranked by more than one load;

41) Only one desired load - If there is only one desired load, it has to be the optimum load for scheduling.

The next group of heuristics applies in the goal context "find time slot for scheduling," and, as the goal name suggests, they define a time slot to be used for scheduling loads:

42) Use the earliest time for start - Pick an event (period of operation) of the optimum load and use the recommended start time for scheduling. The recommended time may be user-defined or set to a default value as determined by other heuristics;

43) Start at user-defined start - If the user specified a time to start the load to be scheduled and the time is in the scheduling period, that is the time to schedule the event, i.e., the "recommended time" of heuristic (42);

44) Start at user-defined starting window - If the user specified a starting window for the load to be scheduled, and the window is in the scheduling period, the beginning of the window is the time to schedule the event;

45) Use the desired start time - If the user hasn't requested a starting time for the optimum load, use the start time recommended when the load was chosen as a desired load;

46) Only one event can be used - If several events for a load request have been instantiated and heuristic (47) does not apply, choose one event that has not been scheduled, schedule it, and delete the others. Selection is done arbitrarily;

47) Pick the event with the earliest time - If there is more than one optimum event, then pick the event with the earliest recommended start time as the optimum event for scheduling.

The following heuristics record the of power consumed during the mission. They apply in the goal context "update power consumed":

48) Increase-power-consumed violation - For a specific mission time, if the power consumption of the load being scheduled, when added to the current power consumption, will cause the total power consumption to be greater than the power available at the mission time in question, inform the user that there is not enough power available at that mission time;

49) Increase power consumed - For a specific mission time, add the power consumption of the load being scheduled to the current power consumption to generate the new "power consumed" data item. Remove the old "power consumed" data item for the mission time in question.

The following two heuristics, which reinstate legal slots, apply in the goal context "reinstate legal slots":

50) Reinstate Legal Slots - If there is a load request for which a legal slot does not exist, create the legal slot and update the load opportunity;

51) Reinstate legal slot cleanup - Make sure that the reinstated legal slot is continuous. If a load has more than one legal slot and these two legal slots overlap, generate a new legal slot using the minimum start time and he maximum end time. Then remove the old legal slots and update the load opportunity.

The following heuristics are used to update legal slots and apply in the goal context "update legal slots":

52) Continuous load constraints - If the legal slot for a continuous load is smaller than the total duty cycle for that load, remove the legal slot and update the load opportunity;

53) Day/night constraints - If a load may only operate during the night but the legal slot for that load overlaps a day period, remove the legal slot and generate a new legal slot for the load with day times eliminated. Also update the load opportunity. Likewise, if a load may only operate during the day, but the legal slot for that load overlaps a night period, remove the legal slot, generate a new legal slot for the load with night times eliminated, and update the load opportunity;

54) Load constraints - Make sure that the same load does not "operate simultaneously with itself." If a duty cycle of a load is already scheduled during a time that overlaps with the current legal slot for the load, remove the legal slot and update the load opportunity;

55) Resource constraints - If a load consumes more of a resource than is available during the legal slot for the load, remove the legal slot and update the load opportunity;

56) Pointer constraints - Make sure that no two pointers are scheduled to operate simultaneously. If load A is a pointer and is scheduled during a time that overlaps with the legal slot for load B, which is also a pointer, remove the legal slot and update the load opportunity for load B;

57) Power constraints - If the power consumed exceeds the power available when we hypothetically schedule a load at time t and time t falls within the legal slot for the load, remove the legal slot and update the load opportunity for the load.

The following heuristics update load opportunity and apply in the goal context "update load opportunity":

58) Update load opportunity ratio - For each load success specification, update the opportunity ratio for the load. That is, update the opportunity ratio for total, partial, and minimum levels of success. The opportunity ratio is calculated by dividing the amount of data collection time still required to meet a particular success level by the amount of time remaining in which these data can be collected;

59) Tell the user about success accomplishments - If all the data collection required to attain a particular level of success for a load has been completed, inform the user that the success criteria have been met;

60) Tell the user about success failures - If, in order to reach the required success level, more data-collection time is required than remains in the mission, inform the user that the success level will not be met.

The following heuristic calculates and updates the pointer opportunity ratio. It applies in the goal context "calculate-pointer-opportunity":

61) Calculate pointer opportunity - Calculate the pointer opportunity ratio for the specified day or night periods for each of the success levels: total, partial, minimum. For every load request, determine how much data collection time is required for each load that operates in the specified day/night period and has not been accounted for. Add the data collection time to the current pointer opportunity data collection time. Then calculate the pointer opportunity ratio by dividing the new data collection time by the time available in the mission for collecting the data. Finally, account for the load, remove the old pointer opportunity data item, and add the new ratio to working memory.

These rules apply in the goal context "update pointer opportunity":

62) Update pointer opportunity by success levels - For each success level--total, partial, and minimum--and for every load that is a pointer, update the current pointer opportunity ratio by subtracting the data collection time the pointer load requires from the present data collection time and decreasing the present availability by the amount of time used by the pointer load. Remove the old pointer opportunity ratio data item and add the new pointer opportunity to working memory;

63) Explain pointer opportunity - Pointers that operate during the "day/night" must operate "x minutes" to reach total success. "y minutes" are available to operate these loads;

64) Tell user about pointer success failures - If more data collection time is required than is available, report to the user that day or night pointers do not have enough time to meet success criteria;

The following heuristics handle degradation. They apply in the goal context "handle power shortage":

65) Deschedule interruptable load - If an interruptable load is scheduled at a time when power available is less than power consumed, unschedule the load;

66) Deschedule continuous load - If a continuous load is scheduled at a time when power available is less than power consumed, remove the load from the schedule;

67) Deschedule baseline mode - If no loads are scheduled at a time when power available is less than power consumed, reduce the baseline load power requirements by dropping the baseline mode of operation by one.

The following heuristics, which are used in the goal context "drop baseline mode," reduce baseline load power requirements:

68) Drop baseline modes at time t - If not in survival mode, i.e., the lowest possible baseline power consumption, and the baseline mode has not already been dropped for time t, drop baseline mode by one and insert a goal to update power consumed for time t;

69) Baseline already in survival mode - If baseline power consumption is at the survival level, i.e., it can't be reduced any further without endangering the mission, abort the mission.

Following are the heuristics that handle contingency situations. These rules apply in the goal context "handle-contingency":

70) Recognize earliest contingency - Recognize that any times in the mission when power available is less than power consumed are power problems. Then locate the time of the earliest power problem in the mission;

71) Immediate contingency workaround with interruptable load - If an interruptable load is scheduled during a power problem, unschedule that load;

72) Immediate contingency workaround with continuous but no interruptable load - If a continuous load is scheduled during the power problem and there are no interruptable loads, drop the baseline load power consumption levels at the time of the power problem;

73) Immediate contingency workaround and no interruptable or continuous loads - If no interruptable or continuous loads are scheduled during a power problem, reduce baseline power requirements for the remainder of the mission;

74) Inevitable contingency workaround with interruptable load - If an interruptable load is on and will run into a power problem, unschedule the load;

75) Inevitable contingency workaround with continuous not on but no interruptable load - If no interruptable loads are scheduled and there is a continuous load that is not on yet but will run into the power problem if turned on, don't turn on the continuous load, unschedule it;

76) Inevitable contingency workaround with continuous on but no interruptable load - If no interruptable loads are scheduled and a continuous load is running during the power problem, reduce baseline load power requirements during the power problem;

77) Inevitable contingency workaround, no interruptable no continuous - If there are no loads on during a power problem at current mission time, reduce baseline load power requirements for the remainder of the mission.

1. Possible Improvements

EMES was designed as a software testbed, not as a finished product. A number of improvements would increase its "intelligence." For example, although the user provides criteria for various degrees of success, the program always attempts to achieve total success. Additional rules could be added to forecast the degree of success it might achieve, resulting in better schedules under degraded and contingency conditions.

Another potential improvement relates to EMES' treating all payloads as being of equal importance. This simplification results from any firm guidelines for setting priorities. In a real space station, this is unlikely to be the situation. For example, some payloads will cost many times more than others.

These improvements would come largely from increased knowledge of the constraints placed on scheduling; they do not materially affect the sophistication of EMES' reasoning process.

However, we have identified one improvement that would improve the program's reasoning ability. After EMES places a load on the timeline, it does not remove it unless it discovers a power problem. In contrast, a human expert might notice that an improvement could be made to a schedule he was developing. For example, he might move a load he had placed near the start of the timeline after noticing that another load was particularly hard to place. Similarly, he might notice that one of the loads is far easier to place than the others and might give other loads priority, knowing that the easy load can be used as a gap filler later. This kind of mental backtracking and learning by observing are not built into EMES. Nor would they be easy to add. On the other hand, they could produce superior schedules.

We would not recommend that EMES be modified to add these improvements. Although EMES was built with a modular design to facilitate future development and it could be expanded, doing so would require intimate familiarity with its rules and structure and with the HAPS production system. We believe that more can be learned with the same amount of effort by studying other areas where expert systems could be applied.

If an EMES-like expert system is to be designed for use on a space station, the major design issues should be thoroughly investigated again because much more is known now than when this project started about space station and the capabilities and limitations of expert systems for energy management. The constraints on operational software will also be far different from the constraints on a laboratory testbed or prototype like EMES.

2. Dependency on Spacecraft Configuration

In designing EMES, we attempted to minimize dependency on the details of our specific space station model so mistakes in predicting the ultimate space station configuration would not result in a fundamentally flawed rule base. For example, we avoided rules that reason about the details of managing energy storage because we did not know whether to plan for batteries or regenerative fuel cells. Similarly, we made the payloads as generic as possible, allowing the user to characterize them according to a number of parameters. The EMES rules then reason about the parameters rather than about the idiosyncrasies of precisely defined payloads that may or may not fly.

As a result, the changes EMES would require to adapt it to another (or more specific) configuration are primarily the addition of rules rather than the removal or revision of rules. The details of these changes, however, will be difficult to predict until the space station design is well along.

E. FORTRAN MODULES

EMES uses Fortran modules in much the way a human expert would use computer programs to automate such tasks as analyzing the orbit, determining power availability as a function of time, and drawing graphs of certain quantities. While these functions aid the reasoning process by supplying information, they are not in themselves part of the reasoning process.

We wrote these portions of EMES in Fortran because that language is better suited to algorithmic computation than a production system. Furthermore, portions of Fortran programs were available to use as models in writing these portions of EMES.

Appendix B presents the details of the algorithms used in these modules. In general, we did not attempt to provide as accurate a set of software tools for EMES as a human expert might want for a flight situation. For example, the orbit calculations use only a first-order approximation in accounting for the oblateness of the earth. Similarly, the calculations for solar array efficiency use simplified formulas for both temperature change and the variation of efficiency with temperature. These approximations are entirely adequate for demonstrating EMES' reasoning ability, and uncertainties in the mission model produce far larger errors than the simplifications in the algorithms. However, a flight system is likely to require considerable refinement in these modules.

VII. FUTURE WORK IN EXPERT SYSTEMS

A. STATE OF THE ART IN EXPERT SYSTEMS

Expert systems have built an impressive record in a short time. The work started with DENDRAL, an expert system for determining the structure of molecules from mass-spectrometer data. This project began in 1965, and was received with some skepticism when it was introduced in 1967: "It sounds like good chemistry, but what does it have to do with A[artificial] I[intelligence]?" [Michie 1979].

Since then a number of such systems have been developed, not all of them research projects. Many are of considerable commercial value. With today's technology, prototype or operational expert systems have been written for:

1) Analyzing or diagnosing problems in diesel-electric locomotives, computers, telephone cables, and other equipment and systems;

2) Diagnosing diseases, analyzing electrocardiograms, and advising doctors in administering chemotherapy;

3) Assisting in exploring for mineral deposits and oil, and analyzing oil well data;

4) Assisting in design and analysis of software, including other expert systems;

5) Assisting in solving mathematical problems;

6) Assisting in the design or analysis of integrated circuits, databases, printed circuit cards, and single-board computers and other circuitry;

7) Assisting in job-shop scheduling and in management of manufacturing and large projects;

8) Assisting chemists and geneticists by analyzing protein crystallography data, mass spectrograms and other chemical data, by planning bioengineering experiments involving DNA, by solving certain problems in genetic engineering, and by helping plan organic chemical synthesis;

9) Providing computer-aided instruction;

10) Configuring computers;

11) Assisting in solving water-resource problems;

12) Adjusting signal-processing systems;

13) Analyzing structures;

14) Performing statistical analysis.

The majority of today's expert systems use shallow reasoning, i.e., they apply heuristics, but they do not "understand" the basic principles underlying the problem they are solving. For example, an expert system to analyze a piece of electronic equipment may know such rules of thumb as might be found in a user's manual under "In Case of Problems," e.g., "If you are just starting to troubleshoot, check the indicator lights and the power-supply voltages." In fact, the rules used by the expert system will likely go well beyond these in detail. However, the program will not know or apply such fundamental principles as cause-effect relationships, Ohm's law, Kirchoff's laws, etc. If it uses Ohm's law at all, it may know it only as a formula for computing current when it knows voltage and resistance. It will not realize that the same equation can be used to compute any of these three quantities when it knows the other two.

This kind of expert system is coming, and there are a few examples today, but they are still the exception. The reasons are easy to state:

1) "Deep reasoning," "causal reasoning," and "reasoning from first principles" are the buzzwords of recent research in expert systems, but they represent a new field that has not had time to mature. It will take time for effective techniques to be developed and for these to become widely known;

2) This kind of system involves much more work than a shallow-reasoning system.

The term "shallow reasoning" should not be taken as a pejorative, however. Such reasoning has proved adequate for producing expert systems for a wide variety of applications.

Another limitation of current expert systems is a very limited ability to learn. Typically, such programs do not learn new methods of solving problems--new rules--although some adjust parameters through experience to improve performance. This, too, is an active area for research. One of the best known is VEXED [Schindler 1984], developed at Rutgers University for very-large-scale integrated circuit design. This system is designed to add rules of its own invention by generalizing feedback from users.

Human experts know their limitations far better than expert systems do. When an expert system is presented with a problem that is close to the limits of its competence, it performs badly but provides no indication that anything is wrong. When human beings know something, they also know that they know it, and when they don't know something, they typically are aware that they are ignorant. But building this kind of self-awareness into a computer program is very difficult and is an issue that has largely been ignored in the design of today's expert systems.

Most of today's expert systems are restricted to narrow domains so that one expert can supply all the rules. Handling knowledge from multiple experts is very difficult and is referred to as the "scattered expert" problem. What makes it a problem is that different experts approach a problem different ways, and the rule base can easily end up with rules that contradict each other or are at least incompatible.

A related problem occurs when there is no expert. Curiously, it is easier to write a program to outperform most human experts in a narrow domain than to write one to do what a three-year-old child can do. The developer of an expert system runs the risk of finding a classic unsolved problem of artificial intelligence embedded in the expertise he is trying to capture. This risk is minimized when the expert is called an expert primarily because he knows a lot, not because he is better than the average person in applying ordinary human skills.

Expert systems are not well suited to performing tasks for which adequate algorithmic solutions are known because the algorithmic solution will invariably be faster. This is more an issue of practicality than a fundamental limitation, and an expert system might be useful in, for example, selecting an appropriate algorithm or assisting in formulating a problem properly for algorithmic solution.

Some of the limitations of today's expert systems can be attributed to the limitations of today's hardware. This problem is the driving force behind Japan's "Fifth Generation" project, which aims to develop computers that are orders of magnitude faster than those now available for running expert system programs.

A final limitation on expert systems is that they can take a long time to develop. While new expert systems can be developed quickly in fields where others have been developed, e.g., for medical diagnosis, attacking a new field is difficult. Similarly it is difficult to develop expert systems for a field that is changing rapidly. For example, an expert system for electronic circuit design could become obsolete before it is debugged if the implementers embed the part numbers of today's integrated circuits in the rule base instead of providing a means to update through a parts list. Even so, it would be difficult to allow for breakthroughs like microprocessors, which make major changes in the constraints on practical designs.

This is a potential problem for space station, because its design features may change during the design of the expert systems it is to employ. Fortunately, design tools are being developed to greatly reduce the time required to implement these programs. Some such tools are already available, and there is a great economic incentive to develop more.

EMES has the same limitations that characterize most of today's expert systems. It does not learn new problem-solving techniques on its own, although it could be modified periodically to add intelligence, and it cannot reason about cause and effect. However, it knows a lot about the payloads aboard the simulated space station--their power consumption, success criteria, interruptability, pointing requirements, etc--and it knows how to create schedules that meet these requirements. These are the primary characteristics of today's first-generation expert systems: possession of rare knowledge and the ability to apply it in solving problems.

B. APPLICABILITY OF EXPERT SYSTEMS TO POWER SUBSYSTEMS

EMES demonstrated management of only one aspect of a space station power subsystem, but the same generic capabilities could be put to work in other aspects of managing the power subsystem as well. For example, an expert system developed under Martin Marietta Denver Aerospace IR&D project D-55R demonstrated the ability of such software to detect and isolate faults in a space station power subsystem and, to a limited extent, find a workaround procedure.

An expert system might provide automatic state-of-health monitoring beyond simple fault detection. For example, it could observe trends in solar-array degradation and battery capacity, provide interpretations and, where possible, corrective actions to prevent failures or avoid operational problems.

Another fruitful area for applying the capabilities EMES demonstrates is in energy storage management. Software for this task will be more complex than EMES, however, because it will have to reason from cause and effect, whereas EMES rules do not require this. As we discussed previously, reasoning from cause and effect or fundamental understanding of a system is a relatively new topic in the field of expert systems, and the technology is not yet mature. However, the capability can be reasonably expected to be available in the time frame of space station.

Organizing these capabilities in one expert system is currently beyond the state of the art. However, it is reasonable to suppose that they could be built into a hierarchy of cooperating experts with a "manager" expert system controlling and organizing the activities of the others. If this is done, the data passed among the expert systems should be minimized and highly structured to avoid design problems. The more the expert systems interact, the more problems can be expected in debugging them. If development of such a system is contemplated, work should begin soon on a prototype system to identify the design problems such an architecture will present.

One issue that needs to be addressed is the fact that expert system software, unlike conventional software, is generally neither "correct" nor "incorrect." Like human judgment, the performance of expert systems is better described by assigning a degree of competence. This means there is always the risk of finding a situation the software is not competent to handle. The minimal-risk approach to placing expert systems on space station to manage the power subsystem is to use them as experimental software that makes recommendations but controls nothing, at least initially. As confidence builds in the system's competence, it could be gradually given increased control over the power subsystem.

C. APPLICABILITY OF GENERIC CAPABILITIES OF EMES TO OTHER SPACECRAFT SUBSYSTEMS

EMES demonstrates five generally useful capabilities:

1) Scheduling and revising schedules;

2) Reasoning about priorities that change with time and circumstances;

3) Detection of abnormal situations;

4) Displaying data in the form of tables and graphs;

5) General reasoning ability.

These capabilities might find use in other spacecraft subsystems. For example, the control and display subsystem could use these abilities for "intelligent caution and warning." An expert system could preinterpret the symptoms of abnormal conditions, find possible explanations, and suggest corrective actions to the crew rather than simply present raw data. As a minimum, the expert system could prioritize the display of data to emphasize the most important indications. An expert system with reasoning ability could do this more effectively than a simple algorithmic prioritization scheme because it could recognize more subtle patterns in the data, reason about possible causes and implications, and observe trends in data over a period of time.

Similarly, these abilities might be put to use in data management. An expert system could screen some kinds of data to prevent storage or transmission of redundant or meaningless data; or it could prepare predigested abstracts of data along with its interpretation of their meaning. These capabilities would reduce the amount of data space station would have to return to earth, reduce the problems of data storage, and decrease the manpower required to interpret the data.

Some of the capabilities demonstrated in EMES could be useful in various payloads. Currently such payloads as scientific instruments and technology-development experiments require a large amount of human supervision. An expert system might substitute for some human activities, reducing costs and decreasing the chances of something being overlooked because of fatigue or inattentiveness. However, the payload would have to be chosen with some care because the expert system will itself be expensive to develop. The ideal payload to use an expert system is one that will be used for more than a year, requires intelligent supervision beyond the capability of conventional software, and does not require such human capabilities as development of novel theories, invention of new methods to solve unforeseen types of problems, insight, and intuition. Even where these abilities are required on occasion, an expert system might be able to reduce the human expert's burden.

In addition, an expert system with some of the capabilities EMES demonstrates could be a useful experiment in its own right. For example, an expert system could be developed to automate some function on a space station despite questions about its ability to handle the task. Its performance could be evaluated during flight by comparison with the decisions the human experts who control the space station. During this evaluation, the improvements needed could be noted, and the expert system could be modified so it could be used for control in a future mission. If the function performed is critical, or if the consequences of a bad decision could be severe, the expert system might be carried as an experiment a number of times.

VIII. REFERENCES

B. Buchanan and E. Feigenbaum: "DENDRAL and META-DENDRAL: Their Applications Dimensions." Artificial Intelligence 11, 1978, pp 5-24.

Randall Davis: "Interactive Transfer of Expertise: Acquisition of New Inference Rules." Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 22-25, 1977.

C. L. Forgy: "On the Efficient Implementation of Production Systems." PhD Thesis, Carnegie-Mellon University, 1979.

C. L. Forgy: OPS5 User's Manual. Department of Computer Science, Carnegie-Mellon University, 1981.

C. L. Forgy: "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern-Match Problem." Artificial Intelligence 19, 1982, pp 17-37.

M. S. Fox: "On Inheritance in Knowledge Representation." Proceedings of the Sixth International Joint Conference on Artificial Intelligence, 1979, pp 282-284.

M. R. Genesereth, R. Greiner, and D. E. Smith: MRS Manual. Stanford Heuristic Programming Project, Memo HPP-80-24.

J. McDermott: "R1: A Rule-Based Configurer of Computer Systems." Artificial Intelligence 19, 1982, pp 39-88.

D. Michie, ed.: Expert Systems in the Microelectronic Age. Edinburgh University Press, Edinburgh, Scottland, 1979, p 7.

R. Sauers and R. Farrell: GRAPES User's Manual. Technical Report ONR-82-3, Carnegie-Mellon University, 1982.

M. Schindler: "Artificial Intelligence Begins to Pay Off with Expert Systems for Engineering." Electronic Design 32:16, August 9, 1984, pp 133-134.

E. Shortliffe: H1, Computer-Based Medical Consultations: MYCIN. American Elsevier Publishing Company, New York, NY, 1976.

M. J. Stefik: "Planning with Constraints." PhD Thesis, Stanford University, 1980.

Chuck Williams: "Software Tool Packages the Expertise Needed to Build Expert Systems," Electronic Design 32:16, August 9, 1984, pp 154+.

APPENDIX A - ENERGY MANAGEMENT EXPERT SYSTEM SOFTWARE REQUIREMENTS DOCUMENT

The following pages contain the software requirements document originally submitted to Marshall Space Flight Center in December 1983. This appendix is included in this report to satisfy the requirements of the contract statement of work.

I. INTRODUCTION

The Energy Management Expert System (EMES) is a rule-based expert system designed to demonstrate management of the power resources on a spacecraft. The expertise applied in building EMES is that of a payload operations director who must decide whether the various loads onboard a spacecraft should be operated and, if so, when the best time is to operate each.

The loads that make up the spacecraft configuration must be managed, as well as system experiments and other miscellaneous payloads. The information the director has available includes the orbit-by-orbit power capability of the spacecraft, a set of requests for the operation of various loads during particular time windows, and a set of mission events that must be performed. The director must attempt to schedule each load requested in a way to preclude violation of a set of energy management constraints.

EMES must be able to make load management decisions normally made by the payload operations director. In this context, load management consists of determining a mission timeline (the operational sequence of the various loads) that avoids violation of operational constraints and that meets the efficiency requirements for the use of available power.

For example, EMES must use the power produced by the solar arrays during the daytime portion of an orbit more than the power drawn from the batteries during eclipse portions of the orbit. More battery power can be used immediately after reconditioning than under normal conditions; less battery power can be used toward the end of battery life, etc. Standardly, as many loads as possible will be scheduled for the daylight portions of each orbit. These are only a few of the heuristics that govern the behavior of EMES.

EMES is intended as a demonstration software package illustrating the feasibility of using expert system technology in conjunction with standard automation techniques to provide a general increase in the efficiency of spacecraft power utilization, regardless of power capability.

II. OVERALL FUNCTIONAL REQUIREMENTS

The energy management expert system must have the following capabilities:

1) Be able to provide the required electric power to the user loads under normal and degraded modes of power system operation;

2) Be able to optimize the use of available power;

3) Be able to determine the sequence of operation of all electrical equipment in conjunction with basic mission operation activities and requirements;

4) Be able to extend the life of critical power system components, such as the batteries, through the use of energy management heuristics.

The EMES software must be applicable to manned and unmanned spacecraft. It shall be designed to a large photovoltaic power system with multiple power modules similar to that to be found on board a space station. The designed power capability of the spacecraft power system will be 75 kilowatts.

The EMES system shall be designed with future real-time applications in mind. The initial version of EMES shall use an adequate simulation of the capability of the photovoltaic power system, and shall be designed to facilitate future modifications to allow an interface with the appropriate sensors in a real hardware power system. For example, the eventual system would be able to calculate solar array capability from several voltage and current measurements at the operating point, and the total array energy or sunlight average array power would then be determined from this peak power profile.

EMES shall also be designed with the following future capabilities in mind:

1) An onboard EMES would enable autonomous spacecraft operation for an extended length of time in the event of a sudden change in power capability during which ground intervention is not possible;

2) EMES has the potential for significantly enhancing ground mission planning capability and reducing human-intensive tasks;

3) EMES can reduce the total required battery weight of a system via reduction of nighttime energy demand through appropriate bus load control;

4) The extension of battery life is possible via minimization of battery depth of discharge and operating temperatures;

5) EMES can satisfy other subsystem operational constraints as they are defined.

The EMES software will be designed so the priorities of loads to be scheduled in the mission timeline can be reasoned about dynamically. EMES must also be able to reason about the relationship between loads and mission events so it can make intelligent decisions about what effect turning off a given load will have on the overall mission success.

For demonstration purposes, EMES must allow the user to conveniently specify the desired mission timeline. This means that the user must be able to specify various mission events that will occur during the timeline, as well as the loads that should operate during specified time windows. The user must be able to specify the constraints on the operation of loads and between mision events and loads.

III. SYSTEM ENVIRONMENT REQUIREMENTS

The EMES software will operate in a LISP environment on a VAX running the VMS operating system. The decision to use LISP was based on its generality, knowledge representation capabilities, and ability to interface with other required software. The specific dialect of LISP required for running EMES is Franz-LISP. This dialect runs on either a VAX-11/750 or VAX-11/780. Both a compiler and an interpreter for Franz Lisp are required. Also, Eunice (the UNIX emulator) is recommended because it generally facilitates compiling LISP. The ideal operating system environment is VMS with Eunice facilities.

Because software modules that perform heavily calculation-oriented engineering functions will be implemented in Fortran, a VAX Fortran compiler is required.

The actual rule base, the core of the energy management system, will be implemented in HAPS. HAPS is an interpreter for rule-based systems developed at Martin Marietta that runs in a Franz Lisp environment. Thus, the environment described above is sufficient to ensure the ability to execute HAPS.

IV. REPORTING

The following documentation shall be produced for the energy management expert system:

1) Software Design Description Document - The software design description document shall contain the design details for each EMES module. It will provide a complete technical description of the functions performed by each module, the structure of each module, and the control flow through the major system components;

2) Software Test Plan - The software test plan will define the scope of tests required to ensure that the software meets all applicable technical, operational, and performance specifications. It will establish acceptance criteria for the program and identify each level of testing;

3) EMES User's Manual - The user's manual will contain scope of application, program limitations, and all other operating constraints. The document will provide explanations for the operation of all software modules. All I/O to and from the user shall be documented either in the form of a sample run or an operational description.

APPENDIX B - ENERGY MANAGEMENT EXPERT SYSTEM DETAILED DESIGN SPECIFICATION

The following pages contain the software design description originally submitted to Marshall Space Flight Center in December 1983. This appendix is included in this report to satisfy the requirements of the contract statement of work.

I. INTRODUCTION

This document presents the Detailed Design Specification for the Energy Management Expert System (EMES) software designed under contract with NASA Marshall Space Flight Center (MSFC). The energy management system is intended to permit onboard automation of energy management under normal, failure, and degraded modes of spacecraft operation. This is a system-level spacecraft supervisory function because it involves the operation of all housekeeping subsystem and payload equipment that consume power.

The energy management software is intended to provide electric power to the loads of a spacecraft to optimize the use of available power. It must determine the sequence of operation of all electrical equipment on board the spacecraft with respect to basic mission event requirements. It is also intended to permit graceful degradation of the spacecraft under abnormal conditions. Such a system would extend the life of critical system components such as the batteries, as well as reduce the required size of energy storage devices.

The primary focus of this phase of the effort is to demonstrate the feasibility of using expert system technology in an application such as the energy management task. The intent is not for expert system technology to replace traditional automation, but to work in conjunction with traditional automation on tasks that are currently human-intensive.

The remainder of this document provides an overview of the design of the energy management expert system, followed by a detailed design of each of the system software components. A familiarity with the EMES Requirements Document, which contains an overview of the functional and performance requirements of the energy management software, is assumed. For information regarding the planned testing and evaluation of the EMES software, the energy management expert system test and evaluation plan (Appendix C) should be consulted.

II. DESIGN OVERVIEW

Overall operation of the EMES consists of three major phases--library development, mission development, and mission execution. The software design of EMES reflects this division.

The library development phase allows the user to construct and maintain a set of load and event definitions with which to develop mission models. Two system libraries exist--the load-library, which holds definitions of the various loads in the spacecraft, and the event-library, which holds definitions of mission event types that can be scheduled into the mission timeline. Both loads and event types have various user-specifiable attributes (for example, the power consumption of a load). The system libraries are initialized to contain many predefined loads and event types, and the user is permitted to modify these libraries at any time.

The mission development phase allows the user to construct a spacecraft mission, using the loads and event types in the system libraries. To construct a mission, the system must perform several operations. First, the system must initialize the spacecraft configuration, using the spacecraft-configuration-file, which contains information about the spacecraft that does not change between missions (e.g., the set of all subsystem loads always required for operation of the spacecraft). Next, the user must specify the set of loads to be on board the spacecraft during this mission (those not critical to spacecraft operation but that can change from mission to mission). This completes the definition of the spacecraft configuration.

The user is now permitted to supply event requests (events to be scheduled into this mission such as stationkeeping), and load requests (requests to operate a load at particular times during the mission). Finally, the user provides information required for orbit and attitude definition. The system now uses all of the above information to generate a mission definition file ready to be executed by EMES.

The final phase of operation is mission execution during which EMES is executed on a particular mission definition. The system inspects the mission timeline, looking for resource requirements that cannot be met or energy management constraints that have been violated. The expert system modifies the mission timeline and produces a new timeline that does not violate mission constraints.

Because the EMES software will operate in a LISP environment, all of the software described in this document will be developed in LISP unless otherwise noted. The decision to use LISP was based on its generality, knowledge representation capabilities, and ability to interface with other required software.

Modules that perform engineering functions or that are heavily calculation-oriented are encoded in Fortran. This decision was based on the efficiency of numeric computation in Fortran.

The actual rule base, which is the core of the EMES, will be implemented in HAPS. The decision to use HAPS for the development of the rule base was based on its knowledge representation capabilities, powerful rule formalism, efficiency with respect to the large amounts of knowledge to be processed, and its ability to interface with other software systems.

The remainder of this document provides a detailed description of EMES design. The system is divided into its component software modules, and each is described individually. The data required to interface the various system components are described. Where disk files are required to support the software components, they are also described.

III. TOP-LEVEL EXECUTIVE

Overall operation of EMES is controlled by the top-level executive. This module is responsible for directing the user's activity through each of the major system functions--library development, mission development, and mission execution.

The Top_Level_Executive provides the user with simple menu-driven access to any of the three major system modules. This will be implemented as a call to the Select_Menu_Entry module, with Menu_Descriptor equal to Top Level_Menu. Top_Level_Menu is a global constant, with value

```
    ("Top Level Executive."
        (Library_Development . "Library Development.")
        (Mission_Development . "Mission Development.")
        (Mission_Execution . "Mission Execution.")
        (Exit . "Exit EMES."))
```

Select_Menu_Entry is called repeatedly until the value returned is Exit, in which case the EMES halts and control is returned to the VMS operating system. Other possible values returned are Library_Development, Mission_Development, and Mission_Execution. In each of these cases, a call to the named major system submodule is made.

IV. LIBRARY DEVELOPMENT

The Library_Development module is responsible for the construction, development, and maintenance of the system libraries. The system requires two libraries: a load library, which contains definitions of all loads known to the system, and an event library, which contains definitions of the different types of mission events known to the system. The user has full control over the development of each library; that is, the user is able to both access and modify the contents of each library.

The Library_Development module provides the user with a menu-driven interface to two major library development submodules--the Load Library Development module, and the Event Library Development Module. This will be implemented as a call to the Select_Menu_Entry module, with Menu_Descriptor equal to Library_Development_Menu. Library_Development_Menu is a global constant, with value

```
    ("Library Development."
        (Load_Library . "Load Library Development.")
        (Event_Library . "Event Library Development.")
        (Exit . "Return to Top Level Menu."))
```

Select_Menu_Entry is called repeatedly until the value returned is Exit, in which case control is returned to the top-level executive. Other possible values returned are Load_Library and Event_Library. In each of these cases, a call to the appropriate submodule is made.

A. LOAD LIBRARY DEVELOPMENT

The Load_Library_Development module allows the user to maintain a set of load definitions for use in the development of mission models. Four basic development functions are provided: Show Load Index, Show Load Definition, Define Load, and Delete Load. Each of these functions is performed by a different Load Library submodule.

The Load_Library_Development module provides the user with a menu-driven interface to the four named load library submodules. This will be implemented as a call to the Select_Menu_Entry module, with Menu_Descriptor equal to Load_Library_Development_Menu. Load_Library_Development Menu is a global constant, with value

```
    ("Load Library Development."
        (Define_Load . "Define a new load.")
        (Delete_Load . "Remove an old load definition.")
        (Load_Index . "List loads in the library.")
        (Show_Load . "Show the definition of a load.")
        (Exit . "Return to Library Development Menu."))
```

Select_Menu_Entry is called repeatedly until the value returned is Exit, in which case control is returned to the library development module. Other possible values returned are Load_Index, Show_Load, Define_Load, and Delete_Load. In each of these cases, a call to the appropriate submodule is made.

The load library will be located in the directory "load_lib:". Thus, the VMS logical name load_lib: must be initialized to point to the load library directory. The EMES_system: directory contains the load_library_index_file, which serves as an index to all of the loads currently defined in the system. An index entry for each load exists, with a pointer to a file in the load_lib: known as the load_library_definition_file, which contains the actual definition for that load.

1. Define Load

The Define_Load module allows the user to define a new load to the system. Each new load requires a Load_Definition_File in the load_lib:, and also requires an entry in the load_library_index_file.

The Define_Load module leads the user through each step required to define a new load. The following interaction between the system and the user occurs when defining a new load. First, the system asks about the characteristics common to all loads.

```
    Name of this load:
```

The user must input the name of this load. The name must be an atomic symbol, and must not be a previously defined load or a previously defined event type.

```
    Power consumption (watts):
```

The user must input the peak power consumption of this load, in watts. The power consumption must be an integer, in the range 1 .. max_power consumption.

```
    Which bus is this load connected to?

       1. Critical Bus.
       2. Low Power Bus.
       3. High Power Bus.

    Menu Selection:
```

The user must type the number corresponding to the name of the power system bus that this load gets connected to when it is included in a mission. The selection process is implemented through a call to the Select_Menu_Entry module, which will return one of the following:

critical: This load gets connected to the critical bus.
low-power: This load gets connected to the low power bus. Only loads whose peak power consumption is in the range 1 .. max_low power_load may be connected to the low power bus.

high-power: This load gets connected to the high power bus. Only loads whose peak power is in the range min_high_power_load .. max_power_consumption may be connected to the high power bus.

Note that loads do not actually become connected to spacecraft buses at this point they merely get definitions in the load library.

```
    Duty Cycle (minutes):
```

The user must type the duty cycle of this load. Legal inputs are

[n], where [n] is a positive integer, which is the number of minutes this load is turned on each time it is used.
[return], which means that this load does not have a known duty cycle. This is used to describe loads such as heaters, which are thermally controlled, or loads that are on for the duration of the mission.

The next selection is made only for noncritical loads.

```
    The operation of this load ...

        1. must be continuous.
        2. can be interrupted.
        3. can be restarted.

    Menu Selection?
```

The user must type the number corresponding to the choice that best describes the operation of this load. The selection process is implemented through a call to the Select_Menu_Entry module, which will return one of the following:

continuous: No operation of this load resulting from a single load request can be interrupted without damage to the load or to other resources used by this load.
interruptable: The operation of this load can be interrupted before the duty cycle is completed without damage to the load or other resources.
restartable: The operation of this load may be interrupted and restarted later in the mission. Thus, a single request for this type of load may result in turning the load on at several different times during the mission.

Critical loads are assumed to require "continuous" power when they are in operation.

```
    To which category does this load belong?

         1. Commercial Payload.
         2. Communication and Tracking.
         3. Control and Display.
         4. Data Management.
         5. Environmental Control.
         6. Guidance, Navigation, and Control.
         7. Life Support.
         8. Power System.
         9. Science Payload.
        10. Technology Development Payload.
        11. Thermal Control.
    Menu Selection:
```

The user must type the number corresponding to the choice that best describes the category of this load. The selection process is implemented through a call to the Select_Menu_Entry module, which will return one of commercial-payload, communication-and-tracking, control-and-display, data-management, environmental-control, guidance, life-support, power-system, science-payload, technology-development-payload, or thermal-control.

Some loads (for example, the flight computer or a heater) provide a resource (such as compute-power or heat) to be used by other loads in the system. For this reason, the system needs to ask about resources.

Each of the above inputs results in the creation of a new data item to be inserted into the working memory of the expert system. At this point, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
    The power consumption of load-12 is 500 watts.
    Load-12 is connected to the low-power bus.
    The duty cycle of load-12 is 10 minutes.
    The power to load-12 must be continuous.
    Load-12 is a technology-development-payload.

    Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process starting with the input of the power consumption of the load.

```
    Does this load provide any resources?
```

The user types a yes/no answer. If the user responds with a "yes," the system proceeds to ask for the name and quantity of each resource.

```
    Type the resources provided by this load.
    Type each resource on a separate line, followed by the quantity of
    that resource provided.
    Type [return] when done.

    Resource:
```

The user now proceeds to describe the resources provided by this load. On each line, the user types the resource name (an atom), optionally followed by the quantity of that resource (which defaults to 1). Legal resource names are given in the system_resource_list. If the resource cannot be found in this list, the user is given a warning.

Each resource results in the creation of a "resource-provided" data item. When all resources have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
Load-12 provides 12 units of compute-power.

Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process of getting the resources provided by this load.

Next, the system asks about resources required by this load.

```
Does this load require any resources?
```

The user types a yes/no answer. If the user types "yes," the system proceeds to ask the user for the name and quantity of each resource.

```
Type the resources required by this load.
Type each resource on a separate line, followed
by the quantity of that resource required.
Type [return] when done.

Resource:
```

The user now proceeds to describe the resources required by this load. On each line, the user types the resource name (an atom), optionally followed by the quantity of that resource (which defaults to 1). Legal resource names are given in the system_resource_list. If the resource cannot be found in this list, the user is given a warning.

Note that "power" should not be treated as a resource by the user because the system will later do that automatically. Thus if the user refers to "power" or "energy" as a resource, the system will print an error message and ignore that reference.

Each resource results in the creation of a "resource-required" data item. When all resources have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
Load-12 requires 2 units of compute-power.

Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process of getting the resources required by this load.

```
Do any loads conflict with this load?
```

The system requires a yes/no answer. If the user responds with "yes," this means there are loads that cannot operate while this one is operating. In this case, the system asks for the name of each conflicting load.

```
Type the loads that conflict with this load.
Type each conflicting load on a separate line.
Type [return] when done.

Load name:
```

The user types each conflicting load on a separate line. Each load must be an atom. If the load is not the name of a load already defined in the system library, the user is given a warning message.

Each conflicting load results in the creation of a "conflicts" data item. When all loads have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
Load-1 conflicts with load-12.
Load-3 conflicts with load-12.

Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process of getting the loads that conflict with this load.

This completes the definition of a new load to the system. The system now enters the new load definition into the load library.

```
Entering this load into the load library...

What file will contain this load definition?
```

The user must type the name of the file that will contain the new definition. The file name must not be the name as a file currently containing a load definition. The file will reside in the load_lib: directory.

All of the information in a load definition file is represented as a set of data items to be inserted into the working memory of the EMES. An entry for this load also gets inserted into the load_library_index_file.

When this process is complete, the system will respond with

```
Done.
```

and the Define_Load module exits. Control is returned to the Load Library Development module.

2. Delete Load

The Delete_Load module allows the user to remove the definition of a load from the load library. The user is prompted for the name of the load to be deleted, which must be a load currently defined in the library. The entry for that load in the load_library_index_file is then removed, and the file containing the definition of that load in the load library is deleted.

3. <u>Show Load Index</u>

The Show_Load_Index module allows the user to get a listing of all of the loads currently defined in the system. For each defined load, the user is shown the name of the load, the name of the file that contains the load definition, and the date when the load definition was last modified. All of this information can be obtained from the load_library index_file. The Direct_User_Output module provides the user the option of showing the load index on the terminal, sending it to a disk file, or sending it to the line printer.

4. <u>Show Load Definition</u>

The Show_Load_Definition module allows the user to get a listing of all of the information in the definition for a load in the load library. The user is prompted for the name of the load whose definition is to be shown. This must be a load currently defined in the library, and the system reads the contents of the load_library_definition_file for that load. Each piece of information in the load definition is a data item to be inserted into working memory. The Translate_Data_Item module is used to print an English description of each of these data items. The Direct_User_Output module provides the user the option of showing the load definition on the terminal, sending it to a disk file, or sending it to the line printer.

5. <u>Load Library Index File</u>

The load_library_index_file (currently loadlib.idx) is the master file for the load library and serves as an index to all of the loads that have definitions in the system. This is a LISP text file, containing the load_index_list. The load_index_list is a LISP list (initially nil) serving as an index to the individual load definitions. This list contains one entry for each load currently defined in the system, listed in alphabetical order according to load name. Each entry is of the form

    (load-name definition-file definition-date),

where

    load-name: Is an atom representing the name of this load;
    definition-file: Is the name of the file containing the
        definition of this load;

    definition-date: Is an atom representing the date when this
        load definition was created.

The load_index_list gets updated whenever a new load is defined or an old load is deleted.

6. Load Library Definition Files

Each load defined in the load library has a load definition file. The load definition file contains all of the information required to define this particular load to the EMES. All information in a load definition file is represented as a separate list (in the LISP sense) to be inserted into working memory and processed by EMES during mission execution.

The following are examples of the types of data that may be contained in a load definition file:

```
(power-consumption vacuum 1000)
(bus-connection vacuum low-power)
(subsystem vacuum environmental-control)
(duty-cycle vacuum 5)
```

For more information on the representation of a load to the system, see Section VI.A.1.

B. EVENT LIBRARY DEVELOPMENT

The Event_Library_Development Module allows the user to maintain a set of event-type definitions for use in the development of mission models. Four basic development functions are provided: Show Event Index, Show Event Definition, Define Event, and Delete Event. Each of these functions is performed by a different Event Library submodule.

The Event_Library_Development module provides the user with a menu-driven interface to the four named event library submodules. This will be implemented as a call to the Select_Menu_Entry module, with Menu_Descriptor equal to Event_Library_Development_Menu. Event_Library_Development_Menu is a global constant, with value

```
("Event Library Development."
    (Event_Index . "List event types in the library.")
    (Show_Event . "Show the definition of an event type.")
    (Define_Event . "Define a new event type.")
    (Delete_Event . "Remove an old event type definition.")
    (Exit . "Return to Library Development Menu."))
```

Select_Menu_Entry is called repeatedly until the value returned is Exit, in which case control is returned to the Library Development module. Other possible values returned are Event_Index, Show_Event, Define Event, and Delete_Event. In each of these cases, a call to the appropriate submodule is made.

The event-type library will be located in the directory "event_lib:." Thus, the VMS logical name event_lib: must be initialized to point to the event library directory. The EMES_system: directory contains the event_library_index_file, which serves as an index to all of the event types currently defined in the system. An index entry for each event type exists, with a pointer to a file in the event_lib: known as the event_library_definition_file, which contains the actual definition for that event type.

1. Define Event

The Define_Event module allows the user to define a new event type to the system. Each new event type requires an Event_Definition_File in the event_lib:, and also requires an entry in the event_library_index file.

The Define_Event module leads the user through each step required to define a new event type. The following interaction between the system and the user occurs when defining a new event type. First, the system asks for the name of this event type.

```
Name of this event type:
```

The user must input the name of the event type being defined. The name must be an atomic symbol, and must not be a previously defined load or a previously defined event type.

```
Is this a major mission phase?
```

The user must tell the system whether or not this event is a major mission phase. The main difference between mission phases and other events is that two different mission phases cannot occur simultaneously. The system requires a yes/no response.

```
Are there any subevents that occur during this event?
```

The user must tell the system whether or not any subevents occur during this event. The system requires a yes/no answer. If the user responds with 'yes', then the system continues with

```
Type the events which occur during this event.
Type each subevent on a separate line.
Type [return] when done.

Subevent Name:
```

The user then types the names of the subevents. Each subevent name must be an atomic symbol and cannot be a previously defined load name. If the subevent is not yet defined in the system, the user is given a warning message.

Each subevent results in the creation of a "during" data item. When all subevents have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
Event-12 must occur during event-1.
Event-13 must occur during event-1.
Event-14 must occur during event-1.

Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process of getting the subevent names.

```
Are there any loads which must operate during this event?
```

The user must tell the system whether or not any loads that must operate during this event. This does not include loads that have already been declared as operating during one of the subevents. The system requires a yes/no answer. If the user responds with "yes," the system continues with

```
Type the loads that operate during this event.
Type each load on a separate line.
Type [return] when done.

Load Name:
```

The user then types the names of the loads. Each load name must be an atomic symbol and cannot be a previously defined event type. If the load is not yet defined in the system, the user is given a warning message.

Each load results in the creation of a "during" data item. When all loads have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
Load-3 must operate during event-1.
Load-4 must operate during event-1.
Load-5 must operate during event-1.

Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process of getting the load names.

Now, if any subevents or required loads have been specified, the system asks the user:

```
Would you like to place any additional constraints
on the scheduling of these loads or subevents?
```

The user must tell the system whether or not any additional scheduling constraints are desired. The types of constraints allowable are constraints such as "event-1 must occur before event-2." The system requires a yes/no answer. If the user responds with "yes," the system continues with

```
Type the desired scheduling constraints.
Type each constraint on a separate line.
Type [return] when done.

:
```

The user then types the scheduling constraints. The process of getting the constraints from the user is implemented through a call to the Get Event_Constraints module, which returns a list of new data items representing scheduling constraints on the events in this event type. When all constraints have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
Load-3 must operate before load-5.
Load-4 conflicts with load-5.
Event-1 and Load-4 occur simultaneously.

Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process of getting the scheduling constraints.

```
Do any events conflict with this event?
```

The user must tell the system whether or not any other events conflict with the event being defined (that is, cannot occur during this event). Note that two major mission phases are automatically conflicting so these do not need to be specified. The system requires a yes/no answer. If the user responds with "yes," then the system continues with

```
Type the names of conflicting events.
Type each event on a separate line.
Type [return] when done.

Event Name:
```

The user then types the names of conflicting events. Each event name must be an atomic symbol and cannot be the name of a previously defined load. If the event is not an event in the event library, the user is given a warning. Each conflicting event results in the creation of a "conflict" data item. When all conflicting events have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
Event-1 conflicts with event-2.

Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process of getting the conflicting event types.

```
Are there any loads that cannot operate during this event?
```

The user must tell the system whether or not any loads conflict with the event being defined (that is, cannot operate during this event). The system requires a yes/no answer. If the user responds with "yes," the system continues with

```
    Type the names of conflicting loads.
    Type each load on a separate line.
    Type [return] when done.

    Load Name:
```

The user then types the names of conflicting loads. Each load name must be an atomic symbol, and cannot be the name of a previously defined event type. If the load is not a load in the load library, then the user is given a warning. Each conflicting load results in the creation of a "conflict" data item. When all conflicting loads have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. For example,

```
    Load-17 conflicts with event-1.
    Load-16 conflicts with event-1.

    Is this information correct?
```

If the user does not confirm that this information is correct, the system repeats the process of getting the conflicting loads.

This completes the definition of a new event type to the system. The system now enters the new event type definition into the event library.

```
    Entering this event type into the event library...
    What file will contain this event definition?
```

The user must type the name of the file that will contain the new definition. The file name must not be the name of a file currently containing an event definition. The file will reside in the event_lib: directory.

All of the information in an event type definition file is represented as a set of data items to be inserted into the working memory of the EMES. An entry for this event type also gets inserted into the event library_index_file.

When this process is complete, the system will respond with

```
    Done.
```

and the Define_Event module exits. Control is returned to the Event Library Development module.

2. Delete Event

The Delete_Event module allows the user to remove the definition of an event type from the event library. The user is prompted for the name of the event type to be deleted, which must be an event type currently defined in the library. The entry for that event type in the event_library_index_file is then removed, and the file containing the definition of that event type in the event library is deleted.

3. Show Event Index

The Show Event Index module allows the user to get a listing of all of the event types currently defined in the system. For each defined event type, the user is shown the name of the event type, the file that contains the event-type definition, and the date when the event type was defined. The user has the option of showing the event-type index on the terminal, sending it to a disk file, or sending it to the line printer.

4. Show Event Definition

The Show_Event_Definition module allows the user to get a listing of all of the information in the definition for an event type in the event library. The user is prompted for the name of the event whose definition is to be shown. This must be an event type currently defined in the library, and the system reads the contents of the event_library_definition_file for that event type. Each piece of information in the event-type definition is a data item to be inserted into working memory. The Translate_Data_Item module is used to print an English description of each of these data items. The Direct_User_Output module provides the user the option of showing the event definition on the terminal, sending it to a disk file, or sending it to the line printer.

5. Event Library Index File

The event_library_index_file (currently eventlib.idx) is the master file for the event library, and serves as an index to all of the event types that have definitions in the system. This is a LISP text file containing the event_index_list. The event_index_list is a LISP list (initially nil) serving as an index to the individual event type definitions. This list contains one entry for each event-type currently defined in the system, listed in alphabetical order according to event type. Each entry is of the form

```
(event-type definition-file definition-date),
```

where

event-type: Is an atom representing the name of this event type;
definition-file: Is the name of the file containing the definition of this event type;

definition-date: Is an atom representing the date when this event type definition was created.

The event_index_list gets updated whenever a new event type is defined or an old event type is deleted.

6. Event Library Definition Files

Each event type defined in the event library has an event definition file. The event definition file contains all of the information required to define this particular event type to the EMES. All information in an event definition file is represented as a separate list (in the LISP sense) to be inserted into working memory and processed by EMES during mission execution.

The following are examples of the types of data that may be contained in an event definition file:

```
(mission-phase event-1)
(during load-1 event-1)
(during load-2 event-1)
(conflict load-1 load-2)
```

For more information on the representation of an event to the system, see Section VI.A.2.

V. MISSION DEVELOPMENT

The Mission_Development module allows the user to develop mission definitions for processing by the EMES. Three basic mission development functions are provided--Define Mission, Show Mission Definition, and Generate Mission Reports. Each of these functions is provided by a different Mission Development submodule.

The Mission_Development module provides the user with a menu-driven interface to the three named mission development submodules. This will be implemented as a call to the Select_Menu_Entry module, with Menu_Descriptor equal to Mission_Development_Menu. Mission_Development_Menu is a global constant, with value

```
("Mission Development."
    (Define_Mission . "Create a Mission Definition.")
    (Show_Mission . "Show the Definition of a Mission.")
    (Report_Mission . "Generate a Mission Report.")
    (Exit . "Return to Top Level Menu."))
```

Select_Menu_Entry is called repeatedly until the value returned is Exit. Other possible values returned are Define_Mission, Show_Mission, and Report_Mission. In each of these cases, a call to the appropriate submodule is made.

A. DEFINE MISSION

The Define_Mission Module is responsible for allowing the user to construct mission models to be processed by the EMES using the event types and loads defined in the system libraries. The development of a single mission requires the specification of several pieces of information by the user, as well as the processing of several system-defined parameters.

First, the user must specify the definition of the orbit of the spacecraft. This information is required so EMES can determine periods of daylight and eclipse for each orbit in the mission timeline. This phase of mission development is handled by the Orbit_Definition module.

Next, the system must determine the power capability of the solar arrays for each daytime interval during the mission, and the power capability of the batteries for each nighttime interval. This phase of mission development is handled by the Power_Capability module.

Now, the mission timeline must be specified. The mission timeline consists of the desired sequence of mission events to occur during the mission, along with any additional loads that are to be operational at various points in the mission (for example, miscellaneous experiments). Two types of events get scheduled. The first is a set of load requests that are constant for each mission. These are requests for critical loads and other data that define the spacecraft configuration to the system. These definitions are handled by the Initialize_Mission_Timeline module.

Next, the user specifies the parts of the mission timeline that are unique to this mission. This includes the scheduling of mission events (along with their subevents and required loads), plus any miscellaneous load requests that may be desired (for example, miscellaneous experiments). This phase of mission development is handled by the Define_Mission_Timeline module.

Finally, the Assemble_Mission_Definition module is used to take all of the information that defines this mission and to create from it a Mission Definition File. This file contains the mission definition in a form that can be read in and processed by the rule base of the EMES.

For the purposes of demonstration, the mission timelines processed by EMES cover time periods 1 to max_num_orbits in length (currently 24 orbits, or approximately 36 hours maximum). This decision was made so realistic energy management decisions could be reached in a reasonable time frame. We do not feel that this is a significant limitation for the feasibility demonstration.

The mission definition process maintains a Mission_Definition_Data list, which is a list of all of the data items that make up the definition. When the mission definition is complete, the data items in this list are used as the contents of the mission definition file. Several of the mission definition submodules make updates to this list. The mission definition submodules are described in the following subsections.

1. Orbit Definition

The purpose of the Orbit_Definition module is to calculate the amount of time per orbit the spacecraft is in the earth's shadow over the duration of the mission. The Orbit_Definition module obtains the required input parameters from the user through the use of the Input_Orbit_Data module. The Eclipse_Sun_Profile module performs the actual calculations and sends the output to the Day_Night_Cycles_File. The Orbit_Definition module, along with its submodules, will be coded in Fortran because of the computational nature of the task.

Input Orbit Data - The Input_Orbit_Data module gets the parameters necessary to generate the orbit definition for a given mission. The following interaction between the system and the user occurs. All month, day, year, hour, and minute parameters are Greenwich Mean Time (GMT) unless otherwise noted.

    Longitude of initial ascending node (degrees):

The user inputs the value of LONG_INIT, the longitude of the initial ascending node in a rotating equatorial, earth-centered coordinate system. The units are degrees. The type is REAL and the range is 0.0 to 360.0.

    Month of initial ascending node (1..12):

The user inputs the value of GMON_INIT, the month of the year corresponding to the initial ascending node. The type is INTEGER, and the range is 1 to 12.

    Day of initial ascending node (1..31):

The user inputs the value of GDAY_INIT, the day of the month corresponding to the initial ascending node. The type is INTEGER, and the range is 1 to 31.

    Year of initial ascending node (0..99):

The user inputs the value of GYEAR_INIT, the last two digits of the year corresponding to the initial ascending node. The type is INTEGER, and the range is 0 to 99.

    Hour of initial ascending node (0..23):

The user inputs the value of GHOUR_INIT, the hour of the day corresponding to passage at the initial ascending node. The type is INTEGER, and the range is 1 to 23.

    Minutes of initial ascending node (0.0 .. 59.99):

The user inputs the value of GMIN_INIT, the minutes of the hour corresponding to passage at the initial ascending node. The type is REAL, and the range is 0.0 to 59.99.

    Inclination of the orbit (degrees):

The user inputs the value of INC_EQU, the inclination of the orbit in an earth equatorial coordinate system. The units are degrees. The type is REAL, and the range is 0.0 to 89.99.

Altitude of the orbit (kilometers):

The user inputs the value of ALT, the altitude of the orbit. The units are kilometers. The type is REAL, and the range is 0.0 to 40000.0.

Initial orbit in this mission:

The user inputs the value of ORB_START, the first orbit at which calculations are to begin. One orbit is defined as the time between passage of the spacecraft through consecutive local noons. The first orbit is the first local noon following the initial ascending node. The type is INTEGER and the range is 1 to 9999.

Total number of orbits in this mission:

The user inputs the value of NUM_ORB, the total number of orbits for which the calculations should be made, beginning at ORB_START. The type is INTEGER, and the range is 1 to max_num_orbits.

This completes definition of the orbit parameters. The input data are summarized on the user's terminal, and the user is asked for confirmation. If the data are incorrect, the input process is repeated.

Eclipse/Sun Profile - The Eclipse_Sun_Profile module uses the input orbit definition data to calculate the amount of time per orbit the spacecraft is in the earth's shadow over the duration of the mission. A simplified analytical technique previously developed for analyzing lighting conditions [Brown 1969] is the basis for this module.

For the purposes of EMES development and demonstration, the following criteria apply. The spacecraft is assumed to be in a low-earth circular orbit. Therefore, the penumbra can be assumed nonexistent and the sun's rays are parallel to the earth-sun line. This causes the shadow to become a circular cylinder whose diameter is the mean diameter of the earth. These assumptions introduce an error in eclipse duration of a few seconds. The orbit altitude is assumed to be constant and not decay over the mission time. Only first-order earth oblateness correction is included in computing the regression rate of the ascending node.

The Eclipse_Sun_Profile module shall be implemented as follows:

1. A set of initial conditions is generated.

```
Initial day of the year (days):
    IF (GMON_INIT.EQ.1) THEN DY_INIT=GDAY_INIT
    DOY_INIT=DAY_PER_MON(GMON_INIT)+GDAY_INIT

Correct for leap year:
    IF (MOD(GYEAR_INIT,4).EQ.0.AND.GMON_INIT.GT.2)
        THEN DOY_INIT=DOY_INIT+1
```

```
Initial hour of the day (hours):
    HOD_INIT=GHOUR_INIT+GMIN_INIT/60.0

Orbit radius (kilometers):
    RADIUS_ORBIT=RADIUS_EARTH+ALT

Orbit period (hours):
    PERIOD=TWO_PI*SQRT(RADIUS_ORBIT**3/GRAV)/SPH

Time between local noons (hours):
    DELTA_NOON=1.0/(1.0/PERIOD-1.0/DPY)

Orbit inclination with respect to equator (radians):
    INC=INC_EQU*RPD

Longitude of initial ascending node in rotating, earth
equatorial system (radians):
    LONG=LONG_INIT*RPD

Ascending node of the initial orbit in inertial, ecliptic
coordinate system (radians):
    SIGMA=RDOT_EARTH*(DOY_INIT+HOD_INIT/HPD-82.859)
SIGMA must be forced between 0 and TWO_PI.
    OMEGA=LONG+SIGMA+HOD_INIT*ROT_RATE
OMEGA must be forced between 0 and TWO_PI.

Time to first local noon (hours):
    ALPHA=(HOD_INIT-12.0)*ROT_RATE-LONG
ALPHA must be forced between 0 and TWO_PI.
    COS_A=SIN(INC)*COS(LONG)
    A=ACOS(COS_A)
    THETA_NOON=ASIN(SIN(LONG)/SIN(A))
Place THETA_NOON between 0 and TWO_PI and in the same quadrant
as ALPHA.
    HOD_NOON=HOD_INIT+THETA_NOON/TWO_PI*PERIOD
    DOY_NOON=DOY_INIT
If HOD_NOON is greater then or equal to HPD, then subtract HPD
from HOD_NOON and increment DOY_NOON by 1.0.

Update the orbit number, day of year, hour of day, and right
ascension for starting orbit.
    DELTA_ORB=ORB_START-1
    DELTA_HOURS=DELTA_ORB*DELTA_NOON
    OMEGA=OMEGA-TWO_PI*J2*COS(INC)*
          (RADIUS_EARTH/RADIUS_ORBIT)**2*DELTA_HOURS/PERIOD
OMEGA must be forced between 0 and TWO_PI
    DOY_NOON=DOY_NOON+DELTA_HOURS/HPD
    HOD_NOON=HOD_NOON+MOD(DELTA_HOURS,HPD)
    ORB=ORB_START
```

2. The output parameters are calculated.

Current local noon (days):
```
    DAY=DOY_NOON+HOD_NOON/HPD
```

Calendar date of local noon (via subroutine CAL_DATE):
```
    CALL CAL_DATE (GYEAR_INIT,DAY,GMON_NOON,GDAY_NOON,GYEAR
    NOON, GHOUR_NOON,GMIN_NOON)
```

Time from initial conditions (days):
```
    DELTA_TIME=DAY-DOY_INIT-HOD_INIT/HPD
```

Revolution number (REV is type INTEGER):
```
    REV=DELTA_TIME*HPD/PERIOD
```

Time of last ascending node (days);
```
    DAY_NODE=DOY_INIT+HOD_INIT/HPD+REV*PERIOD/HPD
```

Calendar date of last ascending node (via subroutine CAL_DATE):
```
    CALL CAL_DATE (GYEAR_INIT,DAY_NODE,GMON_NODE,GDAY_NODE,
         GYEAR_NODE, GHOUR_NODE,GMIN_NODE)
```

Inclination of orbit plane to ecliptic (radians):
```
    GAMMA=ACOS(0.91706009*COS(INC)+0.39874902*SIN(INC)*COS(OMEGA)
```

Right ascension of ascending node in ecliptic plane (radians):
```
    SIN_SIGMA0=SIN(OMEGA)*SIN(INC)/SIN(GAMMA)
    COT_SIGMA0=0.91706009/TAN(OMEGA)-0.39874902/
         (SIN(OMEGA)*TAN(INC))
    SIGMA0=ASIN(SIN_SIGMA0)
```
If COT_SIGMA0 and SIN_SIGMA0 are of opposite signs then SIGMA0=PI-SIGMA0. Now force SIGMA0 between 0 and TWO_PI.

Earth position in inertial, ecliptic coordinate system for satellite at local noon (radians):
```
    SIGMA=RDOT_EARTH*DAY-82.859
```
SIGMA must be forced between 0 and TWO_PI.

Sun-plane incident angle (radians):
```
    BETA=ASIN(SIN(GAMMA)*SIN(SIGMA-SIGMA0))
```

Shadow angle (radians):
```
    THETA_ECLIPSE=ACOS((1.0/COS(BETA))*
         SQRT(1.0-(RADIUS_EARTH/RADIUS_ORBIT)**2)))
```

Dark time (hours):
```
    DARK_TIME=THETA_ECLIPSE/PI*PERIOD
```

Light time (hours):
```
    LITE_TIME=DELTA_NOON-DARK_TIME
```

Calendar date of sunset (via subroutine CAL_DATE):
```
    DAY_SET=DAY+LITE_TIME/MPD/2.0
    CALL CAL_DATE (GYEAR_INIT,DAY_SET,GMON_SET,GDAY_SET,GYEAR
    SET, GHOUR_SET,GMIN_SET)
```

Calendar date of sunrise (via subroutine CAL_DATE):

```
    DAY_RISE=DAY_SET+DARK_TIME/MPD
    CALL CAL_DATE (GYEAR_INIT,DAY_RISE,GMON_RISE,GDAY_RISE,
         GYEAR_RISE,GHOUR_RISE,GMIN_RISE)
```

3. Write output parameters to disk file and print out. Note it is necessary to convert angle parameter units from radians (used in internal computations) to degrees (used on the output).

4. Return execution control to Executive Module if ORB is greater then ORB_START+NUM_ORB.

5. Increment parameters for the next orbit.

   Update time:

   ```
       HOD_NOON=HOD_NOON+DELTA_NOON
   ```

   If HOD_NOON is greater then or equal to HPD, then subtract HPD from HOD_NOON and increment DOY_NOON by 1.0.

   Update the ascending node location:

   ```
       OMEGA=OMEGA-TWO_PI*J2*COS(INC)*
            (RADIUS_EARTH/RADIUS_ORBIT)**2*DELTA_NOON/PERIOD
   ```

   Update orbit number:

   ```
       ORB=ORB+1
   ```

6. Repeat calculations starting at Step 3.

7. The CAL_DATE subroutine executes as follows.

   Assume the argument sequence -

   ```
       (YEAR0,DOF,MON,DAY,YEAR,HOUR,MIN)
   ```

   where DOF, MIN, and TEMP are REAL*8 and all other parameters are type INTEGER

   ```
   TEMP=DOF
   YEAR=YEAR0

   K=0
   IF (MOD(YEAR,4).EQ.0) K=1
   TEMP=TEMP-365.0-K
   IF (TEMP.GE.0.0) YEAR=YEAR+1 and repeat these four statements

   TEMP=TEMP+365.0+K
   MON=TEMP/29.0+1
   IF (MON.EQ.13) MON=12
   IF (MON.NE.1) THEN
       IF (TEMP.LE.DAY_PER_MONTH(MON-1)) THEN MON=MON-1
   ENDIF
   IF (K.EQ.1.AND.TEMP.GT.31.AND.TEMP.LT.61) MON=2

   IF (MON.GT.1) THEN TEMP=TEMP-DAY_PER_MONTH(MON-1)
   IF (MON.GT.2) TEMP=TEMP-K
   DAY=TEMP
   TEMP=(TEMP-DAY)*HPD
   HOUR=TEMP

   MIN=(TEMP-HOUR)*60.0
   ```

The Eclipse_Sun_Profile module will create the Day_Night_Cycles_File, which contains all of the information relevant to the orbit definition. A report file showing the user all of the information contained in the disk file shall also be generated in a tabularized format.

Day/Night Cycles File - The Day_Night_Cycles_File is a data file generated by the Eclipse_Sun_Profile module, containing all of the orbit data required for calculation of the solar array capability data. The Day Night_Cycles_File shall be a temporary sequential file, with variable record size. The contents of the file shall be as follows.

Record 1 - This record contains the orbit initial conditions used in the generation of the day/night cycle data.

LONG_INIT :	Longitude of the initial ascending node in a rotating equatorial, earth-centered coordinate system. The units are degrees. The type is REAL and the range is 0.0 to 360.0.
GMON_INIT :	Month of the year corresponding to the ascending node. The type is INTEGER and the range is 1 to 12.
GDAY_INIT :	Day of the month corresponding to the ascending node. The type is INTEGER and the range is 1 to 31.
GYEAR_INIT :	Last two digits of the year corresponding to the ascending node. The type is INTEGER and the range is 0 to 99.
GHOUR_INIT :	Hour of the day corresponding to passage at the ascending year. The type is INTEGER and the range is 0 to 23.
GMIN_INIT :	Minutes of the hour corresponding to passage at the ascending node. The type is REAL and the range is 0.0 to 59.99.
INC_EQU :	Inclination of the orbit in an Earth equatorial coordinate system. The units are degrees. The type is real and the range is 0.0 to 89.99.
ALT :	Altitude of the orbit. The units are kilometers. The type is REAL and the range is 0.0 to 40000.0.
ORB_START :	The first orbit at which calculations are to start. One orbit is defined as the time between passage of the satellite through consecutive local noons. The first orbit is the first local noon following the initial ascending node. The type is INTEGER and the range is 1 to 9999.
NUM_ORB :	The number of orbits for which the calculations should be made beginning at ORB_START. The type is INTEGER and the range is 1 to 9999.

Records 2 through the end - One record is written for each orbit of the spacecraft.

ORB :	Orbit number of current local noon. The type is INTEGER and the range is 1 to 9999.
DAY :	Day of the year referenced to midnight December 31st. The units are days. The type is REAL and the range is 0.0 to 365.9999.
GMON_NOON :	Month of the year corresponding to the current local noon. The type is INTEGER and the range is 1 to 12.
GDAY_NOON :	Day of the month corresponding to the current local noon. The type is INTEGER and the range is 1 to 31.
GYEAR_NOON :	Last two digits of the year corresponding to the current local noon. The type is INTEGER and the range is 0 to 99.
GHOUR_NOON :	Hour of the day corresponding to the current local noon. The type is INTEGER and the range is 0 to 23.
GMIN_NOON :	Minutes of the hour corresponding to the current local noon. The type is REAL and the range is 0.0 to 59.99.
REV :	Revolution number. One revolution is defined as the time between consecutive ascending nodes. The type is INTEGER and the range is 0 to 9999.
GMON_NODE :	Month of the year corresponding to the previous ascending node. The type is INTEGER and the range is 1 to 12.
GDAY_NODE :	Day of the month corresponding to the previous ascending node. The type is INTEGER and the range is 1 to 31.
GYEAR_NODE :	Last two digits of the year corresponding to the previous ascending node. The type is INTEGER and the range is 0 to 99.
GHOUR_NODE :	Hour of the day corresponding to the previous ascending node. The type is INTEGER and the range is 0 to 23.
GMIN_NODE :	Minutes of the hour corresponding to the previous ascending node. The type is REAL and the range is 0.0 to 59.99.
BETA :	Sun-plane incident angle. The units are degrees. The type is REAL and the range is -90.0 to +90.0.

GMON_SET :	Month of the year corresponding to the next sun set. The type is INTEGER and the range is 1 to 12.
GDAY_SET :	Day of the month corresponding to the next sun set. The type is INTEGER and the range is 1 to 31.
GYEAR_SET :	Last two digits of the year corresponding to the next sun set. The type is INTEGER and the range is 0 to 99.
GHOUR_SET :	Hour of the day corresponding to the next sun set. The type is INTEGER and the range is 0 to 23.
GMIN_SET :	Minutes of the hour corresponding to the next sun set. The type is REAL and the range is 0.0 to 59.99.
GMON_RISE :	Month of the year corresponding to the next sun rise. The type is INTEGER and the range is 1 to 12.
GDAY_RISE :	Day of the month corresponding to the next sun rise. The type is INTEGER and the range is 1 to 31.
GYEAR_RISE :	Last two digits of the year corresponding to the next sun rise. The type is INTEGER and the range is 0 to 99.
GHOUR_RISE :	Hour of the day corresponding to the next sun rise. The type is INTEGER and the range is 0 to 23.
GMIN_NODE :	Minutes of the hour corresponding to the next sun rise. The type is REAL and the range is 0.0 to 59.99.
DARK_TIME :	The amount of dark time on the current orbit. The units are hours. The type is real and the range is 0.0 to 30.0.
LITE_TIME :	The amount of light time on the current orbit. The units are hours. The type is real and the range is 0.0 to 30.0.

2. Power Capability

The purpose of the Power Capability module is to determine the amount of power available from the solar arrays during the daylight portions of each orbit in the mission, and the power available from the batteries during the nighttime portions of each orbit. These functions are performed by the Solar_Array_Capability and Battery_Capability modules, respectively. Because of the computational nature of these tasks, the Power_Capability module, along with all of its submodules, will be implemented using Fortran.

Solar Array Capability - The purpose of the Solar_Array_Capability module is to calculate the solar array power available for each 6-minute time interval as a function of the position of the earth in its orbit around the sun. For the purposes of this demonstration, it is assumed that the solar arrays are gimbaled so they are always in the solar inertial (SI) mode regardless of the attitude mode of the spacecraft. This assumption is realistic because space stations will probably use this design and it allows for accurately calculating the solar power for the purposes of energy management.

For this simulation, the power capability of the solar arrays during the daylight portions of the orbit is approximated as a function of the solar intensity and the temperature of the solar array panels.

$$\text{POWER\_SOLAR} = c1 \, \frac{\text{INTENSITY\_SOLAR}}{\text{TEMP\_SOL\_ARRAYS}} + c2$$

As the array sections heat up, they become less efficient, and the power capability decreases. The solar array capability during the nighttime portions of each orbit is assumed to be zero.

The temperature of the solar array panels during the daytime is modeled as an exponential function increasing over time. The following temperature constraints are assumed:

Initial Solar Array Temperature:
T_init = -90 degrees C
= 183 degrees K.

Final Solar Array Temperature:
T_final = Temp. @ approx. 55 minutes
= 100 degrees C
= 373 degrees K.

Average Solar Array Temperature:
T_avg = 39 degrees C
= 312 degrees K.

Using these constraints, solar array temperature as a function of time is approximated by

TEMP_SOL_ARRAYS = 373-190*EXP(-(t-t0)/18)

Solar array temperature as a function of time is depicted in Figure V-1.

Solar intensity is a function of the distance of the earth from the sun:

$$\text{INTENSITY\_SOLAR} = k1 + \frac{k2}{R^{**}3}$$

$$= 135 \text{ +/- } 3.5 \text{ milliwatts/cm**2}$$

Because of the negligible effect of distance, solar intensity is assumed to be a constant 135 milliwatts per square centimeter.

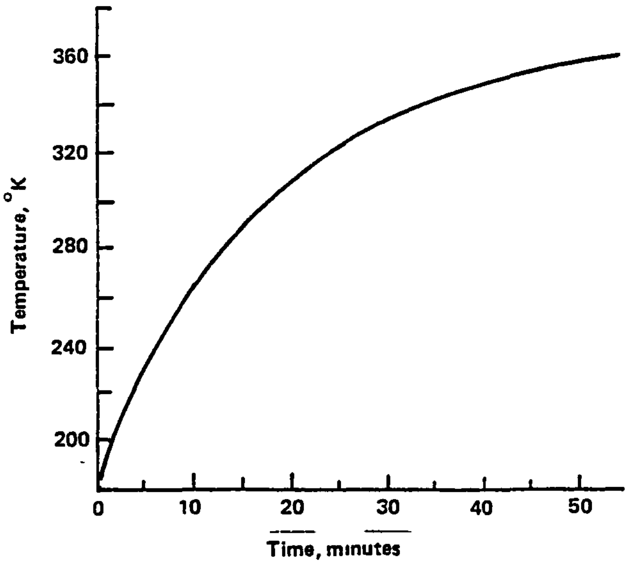


Figure V-1 Solar Array Temperature vs Time

The following constraints on solar array capability are assumed:

Final Solar Array Power:
P_final = Power @ approx. 55 minutes
= 32.5 kilowatts per module.

Average Solar Array Power:
P_avg = 34.2 kilowatts per module.

Using these constraints, solar array power as a function of temperature is approximated by

$$\text{POWER\_SOLAR} = \frac{3243.3}{T} + 23.8$$

This results in the following model of solar power with respect to time:

$$\text{POWER\_SOLAR (kilowatts)} = \frac{3243.3}{373-190*\text{EXP}(-(t-t0)/18)} + 23.8$$

for each of Num_Power_Modules solar array sections, where t is the time (in minutes), and t0 is the time (in minutes) of the start of the current daylight period. Solar array capability with respect to time is depicted in Figure V-2.

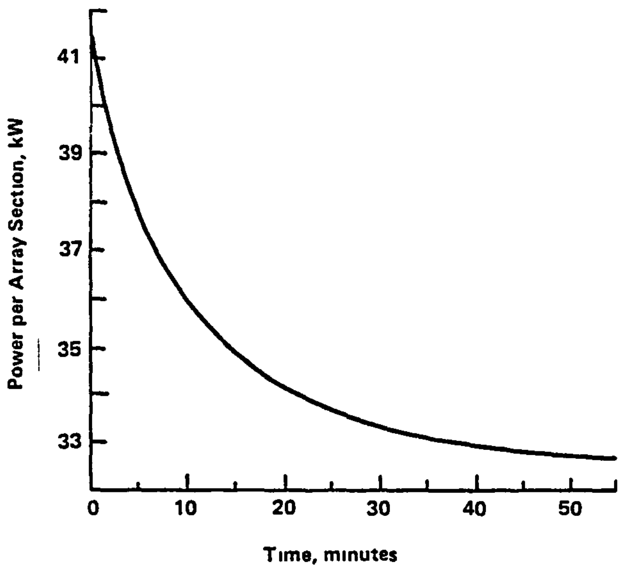


Figure V-2 Solar Array Capability vs Time

The Solar_Array_Capability module will create the Solar_Power_Profile File, which contains solar array capability data for each 6-minute interval during the daylight portions of each orbit. A report file showing the user all of the information contained in the disk file shall also be generated in a tabularized format.

<u>Solar Power Profile File</u> - The Solar_Power_Profile_File is a data file generated by the Solar_Array_Capability module, containing solar array capability data for each 6-minute time interval during the daylight portions of each orbit. No data are given for the solar array capability during the nighttime because these values are all assumed to be zero.

The Solar_Power_Profile_File shall be a temporary sequential file, with variable record size. There shall be one record for each 6-minute interval during the daylight portion of each orbit during the mission. The contents of each record shall be as follows.

ORB :	Orbit number of the current local noon. The type is INTEGER and the range is 0 to 9999.
DAY :	Day of the year referenced to midnight December 31st. The units are days. The type is REAL and the range is 0.0 to 365.9999.
GMON_NOON :	Month of the year corresponding to the current local noon. The type is INTEGER and the range is 1 to 12.
DAY_NOON :	Day of the month corresponding to the current local noon. The type is integer and the range is 1 to 31.
GYEAR_NOON :	Last two digits of the year corresponding to the current local noon. The type in INTEGER and the range is 0 to 99.
GHOUR_NOON :	Hour of the day corresponding to the current local noon. The type is INTEGER and the range is 0 to 23.
GMIN_NOON :	Minutes of the hour corresponding to the current local noon. The type is REAL and the range is 0.0 to 59.99.
MISSION_TIME:	Time value of this 6-minute interval relative to the start of the mission. The units are minutes. The type is integer, and the range is 0 to 2500.
POWER_SOLAR:	The solar power capability during this 6-minute interval. The units are watts. The type is REAL and the range is 0.0 to 100000.0.

Battery Capability - The purpose of the Battery_Capability module is to determine the battery capability during the nighttime periods of each orbit. The nature of this problem is much different from that of solar array capability. The maximum power capability of a battery is usually very high, and it is undesirable for a battery to be operating at its maximum power capability. However, operating at this power level is also unnecessary, and the power capability of the batteries is always much greater than the amount of power consumed by loads at any given time. Therefore, reasoning about the maximum power capability of the batteries during the nighttime is meaningless.

If the EMES is to make intelligent decisions about battery management, it must reason about such constraints as energy balance and battery depth-of-discharge criteria. If battery-DOD criteria are not violated, we can draw as much power from the batteries as we need to as long as there is enough energy during the daylight to recharge the batteries to a fully charged state.

Several problems arise, however. First, battery depth-of-discharge is not a fixed value. Batteries may be able to tolerate a higher depth of discharge when they are new, or for limited periods during certain mission phases. Also, not exceeding an allowable battery DOD is not a sufficient constraint if maximum battery life is desired.

For these reasons, reasoning about battery capability should be used only as a rough measure to guide the scheduling of loads. Afterwards, the mission timeline can be optimized through the use of energy management heuristics. Thus, the purpose of the battery capability module is to provide a rough estimate on the amount of power to be provided by the batteries during the eclipse portion of each orbit. EMES must use this only as a guideline; that is, the battery capability might not be used for any given time period, or it might be exceeded as a result of the application of rules that override the guideline by applying more intelligent heuristics.

Therefore the purpose of the Battery_Capability module is to estimate the maximum power consumption of the loads in the mission which are operating at night. Because this is an estimate, we need one value per eclipse period. The battery model assumes the following constraints:

Maximum Battery Capability:
CAP_BATT = 30,000 watt-hours per battery

Number of Power Modules:
Num_Power_Modules = 3

Estimated Battery DOD:
DOD_EST = 20 percent.

Thus, the estimated battery power capability for each orbit is

$$\text{POWER\_BATT (watts)} = \frac{\text{CAP\_BATT}}{\text{DARK\_TIME}} \text{(Num\_Power\_Modules)(DOD\_EST)}$$

where DARK_TIME is the length of the eclipse portion of this orbit (in hours).

The Battery_Capability module will create the Battery_Power_Profile File, which contains battery capability data for the eclipse portion of each orbit for the duration of the mission. A report file showing the user all of the information contained in the disk file shall also be generated in a tabularized format.

Battery Power Profile File - The Battery_Power_Profile_File, a data file generated by the Battery_Capability module, contains battery capability data per orbit for the duration of the mission. Battery capability for the daylight portion of the orbit is assumed to be zero.

The Battery_Power_Profile_File shall be a temporary sequential file with variable record size. There shall be one record for each orbit during the mission. The contents of each record shall be as follows.

ORB :	Orbit number of the current local noon. The type is INTEGER and the range is 0 to 9999.
DAY :	Day of the year referenced to midnight December 31st. The units are days. The type is REAL and the range is 0.0 to 365.9999.

GMON_NOON :	Month of the year corresponding to the current local noon. The type is INTEGER and the range is 1 to 12.
GDAY_NOON :	Day of the month corresponding to the current local noon. The type is integer and the range is 1 to 31.
GYEAR_NOON :	Last two digits of the year corresponding to the current local noon. The type in INTEGER and the range is 0 to 99.
GHOUR_NOON :	Hour of the day corresponding to the current local noon. The type is INTEGER and the range is 0 to 23.
GMIN_NOON :	Minutes of the hour corresponding to the current local noon. The type is REAL and the range is 0.0 to 59.99.
POWER_BATT :	The battery power capability for this orbit. The units are watts. The type is REAL and the range is 0.0 to 100000.0.

3. Initialize Mission Timeline

The Initialize_Mission_Timeline module has the responsibility of initializing the contents of the mission definition to reflect the spacecraft configuration. The initial contents of the mission definition are as follows.

First, the overall mission is defined as an event, called Overall_Mission, whose start-time is the beginning of the mission, and whose end-time is the end of the mission.

```
(event Overall_Mission nil)
(start-time Overall_Mission 0)
(end-time Overall_Mission [end-time])
```

These data items are added to the initially empty Mission_Definition Data list through use of the Add_Data_Item module.

Next, the system loads that define the spacecraft configuration are added to the mission. These are loads that are an integral part of the spacecraft, and must be specified for all missions. These loads can be found in the spacecraft_configuration_file (config.sys) located in the EMES_system: directory. This file contains one entry, referred to as the Spacecraft_Configuration_Data list, which is a list of the form

```
(([load-name-1] [start-time-1] [end-time-1])
([load-name-2] [start-time-2] [end-time-2])
               . . .
([load-name-3] [start-time-3] [end-time-3]))
```

Here, each [load-name] is the name of a load in the initial configuration, [start-time] is the time this load should be scheduled to turn on, and [end-time] is the time when this load should be turned off (or the special symbol "mission-end," which means the end of this mission). Note that none of these loads should have a "duty-cycle" data item in their definition.

The Initialize_Mission_Definition module informs the user that it is initializing the system configuration, and then, for each entry in the Spacecraft_Configuration_Data list, performs the following:

1. Builds a load request, using the Request_Load module. This module returns a (data . Replace_List) pair, where 'data' is the list of data items which make up the definition of this load request (these get added to Mission_Definition_Data), and Replace List is an A-list of ([generic-event] . [instantiation]) pairs, which tells us how the various load-names were instantiated.

2. Creates a new data item of the form

   (during [new-event] Overall_Mission),

   where [new-event] is the name of the event which was just scheduled.

3. Echoes each new data item to the user, using the Translate_Data Item module, so that the user can see the definition of the spacecraft configuration.

When all spacecraft configuration load requests have been made, the user is informed that the initial configuration definition is complete. The Mission_Definition_Data list now contains all data items that make up the initial configuration. The Initialize_Mission_Definition module returns control back to the Define_Mission module.

4. Define Mission Timeline

The Define_Mission_Timeline module allows the user to schedule arbitrary mission events into the load timeline, and to request the operation of various loads. The user is provided with three capabilities: Request Event, which allows the insertion of an event into the timeline, Request Load, which allows the specification of a load to be operated during the mission, and Get Event Constraints, which allows the user to specify constraints on the scheduling of these loads and events. Each of these functions is provided by a separate submodule. The Define Mission module provides the user with a menu-driven interface to each submodule. This will be implemented as a call to the Select_Menu_Entry module, with Menu_Descriptor equal to Define_Mission_Menu. Define_Mission_Menu is a global constant with value

```
("Mission Definition."
    (Event . "Request an event to be scheduled.")
    (Load . "Request a load to be scheduled.")
    (Constraints . "Place constraints on the scheduling of events.")
    (show-events . "List the events which have been requested.")
    (show-loads . "List the loads which have been requested.")
    (Exit . "Return to Mission Development Menu."))
```

Select_Menu_Entry is called repeatedly until the value returned is Exit, in which case control is returned to the Define Mission module. Other possible values returned are Event, Load, Constraints, show-events, and show-loads.

The Define_Mission module maintains two global lists:

Current_Event_List: This is the current list of events that have been requested. Note that these are not event types, but instantiations of event types.

Current_Load_List: This is the current list of loads that have been requested. Note that these are not load names, but instantiated load requests.

If the user requests an event, the system prompts for the name of the event type being requested.

```
Event type:
```

The event type must currently be defined in the event library. The user is then asked for the time the event should be scheduled.

```
Time to be scheduled:
```

Legal inputs are

1) [return]: The user does not wish to specify a time.

2) [n]: A positive integer that specifies the time the event should begin, relative to the start of the mission. [n] should not be greater than the start of the mission.

3) [n1] [n2]: A time window. The event should start sometime between [n1] and [n2], which are each positive integers that represent times relative to the start of the mission.

The event is instantiated through a call to the Request_Event module, which returns a (data-list . Replace_List) pair. Data-list is a list of data items that make up the instantiation of this event, and Replace List tells us how the various event names were instantiated. The user is shown each new data item through the use of the Translate_Data_Item module, and is asked for confirmation. If the data are not correct, this event definition is ignored. Otherwise, the new data items are added to the Mission_Definition_Data list, and the Current_Event_List and Current_Load_List are updated to reflect the new event defined (plus any subevents or load requests that are part of this definition). Also added to the Mission_Definition_Data list are data items of the form

```
(during [event] Overall-Mission)
(start-time [event] [start-time])
(end-time [event] [start-time])
```

"Start-time" and "end-time" data items are added only if the user has specified start and end times for this event. Here, "[event]" is the name of the instantiation of this event type.

If the user requests that a load be scheduled, then the system prompts for the name of the load being requested.

```
Load name:
```

The load name must currently be defined in the load library. The user is then asked for the time the event should be scheduled.

```
    Time to be scheduled:
```

Legal inputs are as described previously for event requests.

The load is instantiated through a call to the Request_Load module, which returns a (data-list . Replace_List) pair. Data-list is a list of data items that make up the instantiation of this event, and Replace List tells us how the various event names were instantiated. The user is shown each new data item through the use of the Translate_Data_Item module, and is asked for confirmation. If the data are not correct, load request is ignored. Otherwise, the new data items are added to the Mission_Definition_Data list, and the Current_Load_List is updated to reflect the new load request. Also added to the Mission_Definition_Data list are data items of the form

```
    (during [load] Overall-Mission)
    (start-time [load] [start-time])
    (end-time [load] [start-time])
```

"Start-time" and "end-time" data items are added only if the user has specified start and end times for this load. Here, [load-name] is the name of the instantiation of this load request.

If the user asks to add a new set of scheduling constraints, the system responds with

```
    Type the desired scheduling constraints.
    Type each constraint on a separate line.
    Type [return] when done.

    :
```

The user then types the scheduling constraints. The process of getting the constraints from the user is implemented through a call to the Get Event_Constraints module, which returns a list of new data items representing scheduling constraints on the events in this event type. When all constraints have been input, the system shows the user the translation of each new data item, using the Translate_Data_Item module, and then asks for confirmation. If the data are incorrect, these constraints are ignored. Otherwise, the Mission_Definition_Data list is updated.

If the user requests to see the list of events to be scheduled, the system shows the current value of Current_Events_List. For example,

```
    Events to be scheduled in this mission:

        Rendezvous-and-docking:Event-1
        Station-keeping:Event-1
        Materials-experiment-sequence:Event-1
```

Similarly, if the user requests to see the list of loads to be scheduled during this mission, the system shows the current value of Current Loads_List.

When the user is done, the "exit" selection is made, and the Define_Mission_Timeline module returns control to the Define_Mission module. The Mission_Definition_Data list now contains definitions of all loads and events to be scheduled during this mission.

Request Event - The Request_Event module reads in the Event_Definition File for an event type in the event library, and instantiates that event type to form a new event to be scheduled. The Request_Event module requires the following parameters:

event_name: The name of the event type whose definition is to be read.
event_index_list: This is the event type index, obtained from the event_library_index_file.
load_index_list: This is the load index, obtained from the load_library_index_file.

If the event_name cannot be found in the event_index_list, an error message is generated informing the user that the resulting mission definition will be faulty.

If the event_name is found in the event_index_list, the corresponding event definition file is read. Now, the event gets instantiated. For each data item in the definition, we perform the following:

1. If the data item is of the form

   (event event_name) or
   (mission-phase event_name)

   (this will always be the first data item in the definition), then it is instantiated using the Instantiate_Data_Item module, which will generate a new event name to be the name of this event. Instantiate_Data_Item will also create an initial Replace_List, which must be maintained for the duration of the definition of this event.

2. If the data item is of the form

   (during [event] [name])

   where [name] is the name given to the instantiation of this event name, then [event] is the name of a subevent of this event. In this case, Request_Event is called recursively to define this event. Then, the "during" event is instantiated with the Instantiate_Data Item.

3. If the data item is of the form

   (during [load] [name])

   where [name] is the name given to the instantiation of this event name, [load] is the name of a load to operate during this event. In this case, the Request_Load module is called in order to define this load. Then, the "during" event is instantiated with the Instantiate Data_Item.

4. All other data items are translated using the Instantiate_Data Item module.

When all data items have been instantiated, the Request_Event module returns. The value returned is

    (data . Replace_List),

where "data" is the list of data items created when defining this event, and 'Replace_List' is the instantiation list created by the calls to Instantiate_Data_Item.

Request Load - The Request_Load module reads in the Load_Definition_File for a load in the load library, and instantiates that load to form a new load request to be scheduled. The Request_Load module requires the following parameters:

load_name: The name of the load whose definition is to be read.
event_index_list: This is the event type index, obtained from the event_library_index_file.
load_index_list: This is the load index, obtained from the load_library_index_file.

If the load_name cannot be found in the load_index_list, an error message is generated informing the user that the resulting mission definition will be faulty.

If the load_name is found in the load_index_list, the corresponding load definition file is read. Now, the load gets instantiated. For each data item in the definition, we perform the following:

If the data item is of the form

    (load load_name)

(this will always be the first data item in the definition), it is instantiated using the Instantiate_Data_Item module, which will generate a new load request name to be the name of this load request. Instantiate_Data_Item will also create an initial Replace_List, which must be maintained for the duration of the definition of this load request.

All other data items are translated using the Instantiate_Data_Item module. When all data items have been instantiated, the Request_Load module returns. The value returned is

    (data . Replace_List),

where "data" is the list of data items created when defining this load request, and 'Replace_List' is the instantiation list created by the calls to Instantiate_Data_Item.

Get Event Constraints - The Get_Event_Constraints module allows the user to specify constraints on the relationships between the various events to be scheduled. The Get_Event_Constraints module requires the following parameters:

Major_Event_Name: The name of the event currently being defined.
Subevent_Names: The names of any subevents of this event.
Event_Load_Names: Loads to be operated during this event.

The system prompts the user with

```
Type the desired scheduling constraints.
Type each constraint on a separate line.
Type [return] when done.

:
```

The user then types the scheduling constraints. Each constraint is read, and then parsed using the Parse_Constraint_Form module. If this module returns non-nil, then the value returned is a new data item representing a constraint on the scheduling of two events. The Get_Event Constraints module returns a list of these scheduling constraints.

5. Assemble Mission Definition

The Assemble_Mission_Definition module is responsible for putting together all of the information calculated during the definition of this mission, and for using it to generate a new Mission_Definition_File.

The Assemble_Mission_Definition module begins with Mission_Definition Data, which will contain all of the data that define the events and load requests to be scheduled in the mission timeline. The first type of information that needs to be added is the data that define the orbits in this mission. The system informs the user that the orbit definition is beginning:

```
Defining orbit data for this mission...
```

Now, the orbit definition data items are created, using the following process:

1) The system calls the Init_Orbit_File routine;

2) The system calls Get_Next_Orbit number, which returns ORB, the number of the next orbit in the mission. If ORB is negative, there are no more orbits and we are done;

3) The system calls Get_Orbit_Start_Time, which returns the time at which this orbit begins (in minutes), relative to the start of the mission;

4) The system calls Get_Orbit_End_Time, which returns the time at which this orbit ends (in minutes), relative to the start of the mission;

5) A data item of the form

```
(orbit ORB window (start-time end-time))
```

is created and added to the Mission_Definition_Data list.

6) The system calls Get_Sun_Start_Time, which returns the time at which the daylight portion of this orbit begins (in minutes), relative to the start of the mission;

7) The system calls Get_Sun_End_Time, which returns the time at which the daylight portion of this orbit ends (in minutes), relative to the start of the mission;

8) A data item of the form

    (orbit ORB daytime (start-time end-time))

   is created and added to the Mission_Definition_Data list;

9) Go back to step 2.

As each new data item is created, it is echoed to the user using the Translate_Data_Item routine.

The next type of information that needs to be added is the data that define the solar array capability for the daylight portions of each orbit. The system informs the user that the definition of solar array capability is beginning:

    Defining solar array capability for this mission...

Now, the solar array capability data items are created, using the following process:

1) The system calls the Init_Solar_File routine;

2) The system calls Get_Next_Solar_Time, which returns MISSION_TIME, the next 6-minute interval in this mission, which is an integer relative to the start of the mission. If the MISSION_TIME is negative, there are no more time slots and we are done;

3) The system calls Get_Solar_Power, which returns POWER_SOLAR, the solar array capability during this 6-minute interval;

4) A data item of the form

    (solar-array-capability MISSION_TIME POWER_SOLAR)

   is created and added to the Mission_Definition_Data list;

5) Go back to step 2.

As each new data item is created, it is echoed to the user using the Translate_Data_Item routine.

Finally, the data that define the battery capability for the nighttime portion of each orbit needs to be defined. The system informs the user that the definition of battery capability is beginning:

    Defining battery capability for this mission...

Now, the battery capability data items are created, using the following process:

1) The system calls the Init_Battery_File routine;

2) The system calls Get_Next_Battery_Time, which returns ORB, the next orbit number in this mission. If ORB is negative, there are no more time slots and we are done;

3) The system calls Get_Battery_Power, which returns POWER_BATT, the battery capability during this orbit;

4) A data item of the form

   (battery-capability ORB POWER_BATT)

   is created and added to the Mission_Definition_Data list;

5) Go back to step 2.

As each new data item is created, it is echoed to the user using the Translate_Data_Item routine.

This completes the entire mission definition. The system prompts the user for a file name:

```
Mission Definition Completed.
What file name will store this definition?
```

The user inputs the desired file name. The system writes the contents of the Mission_Definition_Data list to the named file, which will be in the mission_dir: directory. The system responds with

```
Done.
```

The Assemble_Mission_Definition module halts, and returns control back to the Define_Mission module.

The following sections describe the functions required to interface LISP to the machine-readable data files created by the Fortran engineering modules. (LISP can read only text files.)

Orbit Data Interface - The following Fortran-coded routines are required to allow the EMES system to create LISP data items from the machine-readable Day_Night_Cycles_File:

Init_Orbit_File: This function opens the Day_Night_Cycles_File for input. The function returns 1 if successful, and 0 otherwise.

Get_Next_Orbit_Number: If we are at the end of the Day_Night_Cycles_File, -1 is returned. Otherwise, this function reads the next record from the input file. All data in this record are shared between this and all of the remaining Orbit Data Interface routines through the use of a Fortran common block.

```
      COMMON/ORB_DATA/ ORB,DAY,GMON_NOON,GYEAR_NOON,GHOUR_NOON,
     1  GMIN_NOON,REV,GMON_NODE,GDAY_NODE,GYEAR_NODE,GHOUR_NODE,
     2  GMIN_NODE,BETA,GMON_SET,GDAY_SET,GYEAR_SET,GHOUR_SET,
     3  GMIN_SET,GMON_RISE,GDAY_RISE,GYEAR_RISE,GHOUR_RISE,
     4  GMIN_NODE,DARK_TIME,LIGHT_TIME
```

The Get_Next_Orbit_Number function returns the value of ORB.

Get_Orbit_Start_Time: Calculates the start time of this orbit relative to the start of the mission, using the REL_TIME module, and returns the result.

Get_Sun_Start_Time: Calculates the start time of the daylight portion of this orbit relative to the start of the mission, using the REL_TIME module, and returns the result.

Get_Sun_End_Time: Calculates the end time of the daylight portion of this orbit relative to the start of the mission, using the REL_TIME module, and returns the result.

Solar Power Data Interface - The following Fortran-coded routines are required to allow the EMES system to create LISP data items from the machine-readable Solar_Power_Profile_File:

Init_Solar_File: This function opens the Solar_Power_Profile_File for input. The function returns 1 if successful, and 0 otherwise.

Get_Next_Solar_Time: If we are at the end of the Solar_Power_Profile_File, then -1 is returned. Otherwise, this function reads the next record from the input file. All data in this record are shared between this and all of the remaining Solar Power Data Interface routines through the use of a Fortran common block.

```
      COMMON/SOLAR_DATA/ ORB,DAY,GMON_NOON,GDAY_NOON,GYEAR_NOON,
     1  GHOUR_NOON,GMIN_NOON, MISSION_TIME,POWER_SOLAR
```

The Get_Next_Solar_Time function returns the value of MISSION_TIME.

Get_Solar_Power: Returns POWER_SOLAR.

Battery Power Data Interface - The following Fortran-coded routines are required to allow the EMES system to create LISP data items from the machine-readable Battery_Power_Profile_File:

Init_Battery_File: This function opens the Battery_Power_Profile_File for input. The function returns 1 if successful, and 0 otherwise.

Get_Next_Battery_Time: If we are at the end of the Battery_Power_Profile_File, then -1 is returned. Otherwise, this function reads the next record from the input file. All data in this record is shared between this and all of the remaining Battery Power Data Interface routines through the use of a Fortran common block.

```
      COMMON/BATTERY_DATA/ ORB,DAY,GMON_NOON,GDAY_NOON,GYEAR_NOON,
     1  GHOUR_NOON,GMIN_NOON,POWER_BATT
```

The Get_Next_Battery_Time function returns the value of ORB.

Get_Battery_Power: Returns POWER_BATT.

B. SHOW MISSION DEFINITION

The Show_Mission_Definition module allows the user to get a listing of all of the information in the definition for a mission. The user is prompted for the name of the file that contains the definition of the mission to be shown. The system looks for the file in mission_dir:, and reads the contents of the mission definition file. Each piece of information in the mission definition is a data item to be inserted into working memory. The Translate_Data_Item module is used to print an English description of each of these data items. The Direct_User Output module provides the user the option of showing the mission definition on the terminal, sending it to a disk file, or sending it to the line printer.

C. GENERATE MISSION REPORTS

The Generate_Mission_Reports module allows the user to generate reports summarizing various important characteristics of a mission. Several different types of reports are permitted: Power Capability, Event Status, Load Profile, Power Margin, and Battery DOD. Each of these report types is generated by a different Generate_Mission_Reports submodule.

The Generate_Mission_Reports Module provides the user with a menu-driven interface to the various mission report submodules. This will be implemented as a call to the Select_Menu_Entry module, with Menu_Descriptor equal to Mission_Report_Menu. Mission_Report_Menu is a global constant, with value

```
("Generate Mission Reports."
    (Power . "Report Mission Power Capability.")
    (Event . "Report Mission Event Status.")
    (Load . "Report Mission Load Profile.")
    (Margin . "Report Mission Power Margin.")
    (DOD . "Report Mission Battery DOD.")
    (Exit . "Return to Mission Development Menu."))
```

Select_Menu_Entry is called repeatedly until the value returned is Exit. Other possible values returned are Power, Event, Load, Margin, and DOD. In each of these cases, a call to the appropriate submodule is made.

1. Show Power Capability

The Show_Power_Capability module allows the user to obtain a graphic representation of the power available for each 6-minute time slot for the duration of a mission. The user is prompted for the name of the mission definition file. The system looks for the file in mission_dir: and reads its contents. All solar-array-capability and battery-capability data items are summarized to obtain the solar array output profile for each daytime period, and the battery capability for each nighttime period. If no such data items exist, the user is notified. The solar array and battery degradation factors are taken into account only if they are present in the mission definition (that is, only if this mission definition has already been processed by the expert system). Otherwise, degradation factors are assumed to be zero. The data are shown graphically using the Graph_Mission_Data module. The Direct_User_Output module provides the user the option of showing the data on the terminal, sending it to a disk file, or sending it to the line printer.

2. Show Event Status

The Show_Event_Status module allows the user to obtain a tabular summary of all of the event requests in a mission definition. This is useful for mission definitions that have not yet been processed by the EMES. The user is prompted for the name of the mission definition file. The system looks for the file in mission_dir: and reads its contents. For each event request, the user is shown, in tabular form,

1) The name of the event request;

2) Whether this is a mission phase, a load request, or an arbitrary event;

3) The load or type of event requested;

4) Whether or not this event has already been scheduled;

5) The time of this event (a time relative to the start of the mission if this event has already been planned, or the time window in which this event must be planned).

The Direct_User_Output module provides the user the option of showing the event status on the terminal, sending it to a disk file, or sending it to the line printer.

3. <u>Show Load Profile</u>

The Show_Load_Profile module allows the user to obtain a graphic representation of the power consumed by loads for each 6-minute time slot for the duration of a mission. The user is prompted for the name of the mission definition file. The system looks for the file in mission_dir: and reads its contents. First, the system looks for load requests that have not yet been scheduled. If any are found, this mission file has not been processed yet by the EMES. Otherwise, all resource-consumed data items that represent power consumed by a load are summarized to obtain the load profile as a function of time. If no such data items exist, the user is notified. The data are shown graphically using the Graph_Mission_Data module. The Direct_User_Output module provides the user the option of showing the data on the terminal, sending it to a disk file, or sending it to the line printer.

4. <u>Show Power Margin.</u>

The Show_Power_Margin module allows the user to obtain a graphic representation of the power margin for each 6-minute time slot for the duration of a mission. The user is prompted for the name of the mission definition file. The system looks for the file in mission_dir: and reads its contents. First, the system looks for load requests that have not yet been scheduled. If any are found, this mission file has not been processed yet by the EMES. Otherwise, all resource-available data items that represent power margin are summarized to obtain the power margin as a function of time. If no such data items exist, the user is notified. The data are shown graphically using the Graph_Mission_Data module. The Direct_User_Output module provides the user the option of showing the data on the terminal, sending it to a disk file, or sending it to the line printer.

5. <u>Show Battery DOD</u>

The Show_Battery_DOD module allows the user to obtain a graphic representation of the battery depth-of-discharge for each 6-minute time slot for the duration of a mission. The user is prompted for the name of the mission definition file. The system looks for the file in mission_dir: and reads its contents. First, the system looks for load requests that have not yet been scheduled. If any are found, this mission file has not been processed yet by the EMES. Otherwise, all battery-DOD data items are summarized to obtain the battery depth-of-discharge as a function of time. If no such data items exist, the user is notified. The data are shown graphically using the Graph_Mission_Data module. The Direct_User_Output module provides the user the option of showing the data on the terminal, sending it to a disk file, or sending it to the line printer.

VI. MISSION EXECUTION

The Mission_Execution module allows the EMES to inspect a mission timeline, schedule any unscheduled mission events or load requests, and make sure that no resource requirements or energy management constraints have been violated.

The mission execution process proceeds as follows:

1) The user is prompted for the name of the file containing the mission definition. The system looks for this file in the mission_dir: directory;

2) The user is required to input solar array and battery degradation constants. Each of these numbers is expressed as a percentage. Total solar array power available for consumption by the loads is equal to a fraction of the output of the solar arrays at any given time, depending on the amount of solar array degradation. Battery degradation is used in a similar fashion. These constants are useful for simulating a spacecraft in a degraded mode;

3) All of the information defining the current mission is inserted into the working memory structure of the expert system. The expert system is given the top-level goal of inspecting the mission timeline, and execution begins;

4) The expert system inspects the mission timeline, checking for inconsistencies or constraints that have been violated. For more information on the operation of the expert system, see Section B;

5) When the expert system finishes, the modified mission timeline is output to a new file, which now contains a new mission definition with no inconsistencies.

One mode of operation of the energy management expert system expected to be common is as follows:

1) The user runs the expert system on a given mission definition, assuming a spacecraft operating under normal conditions;

2) The expert system produces a new, consistent mission timeline;

3) The user runs the expert system again, using the newly created mission definition, but with the spacecraft operating in a degraded mode.

In this way, the user can see the decisions the expert system makes to compensate for a loss of power in the spacecraft.

The following sections describe several important aspects of the design of the energy management expert system (EMES).

A. KNOWLEDGE REPRESENTATION

Knowledge representation decisions are important in any artificial intelligence application. They involve deciding which knowledge is required by the system to do reasoning, as well as how it is best represented.

One of the key factors being considered in the EMES is the representation of values that change over time. The time granularity of information in EMES is 6 minutes; events that occur less than 6 minutes apart are indistinguishable as far as time. All information in EMES that changes over time is given a time tag; that is, it is given a property that indicates the time period over which that information is true. All times are given in minutes and are relative to the start of the mission. Each time tag can take one of two forms:

1) [n], where [n] is a positive integer representing some 6-minute planning interval. This means that the tagged event is true only for that instance in time (which is actually a 6-minute interval);

2) ([min] [max]) is an ordered pair of positive integers. This format is used to represent arbitrary time intervals. [min] is the beginning of the time interval and [max] is the end of the time interval. Both values are rounded to the nearest 6-minute time slot.

Note that the 6-minute granularity of time in the expert system is not hardwired into the encoded rules--it is encoded in the representation of declarative knowledge residing in the mission definition file. For example, solar array capability is given for every 6-minute time slice. The expert system cannot reason about solar array capability any more precisely.

The load management problem is characterized to the expert system as a scheduling problem. The system is presented with events to be scheduled (mission events, load requests) and a set of constraints between events. This allows us to use the same set of rules to reason about constraints concerning mission phases and constraints between load requests. The problem becomes an energy management problem by encoding further constraints on the scheduling of events and the allocation of resources. For example, one constraint on the allocation of power (a resource) to various loads (consumers of power) is that battery depth-of-discharge should be minimized (an energy management constraint).

The advantage of characterizing the problem in these terms is that the problem is generalized and can now handle the allocation of resources other than energy. This makes the system more realistic. For example, other scarce resources that can be reasoned about are astronauts and computer time. This makes the system more realistic in its scheduling of the loads that require these resources (in addition to power).

All knowledge input to the expert system is encoded as a single piece of LIST structure (in the LISP sense) of arbitrary length and complexity. To ensure consistency of representation, the system modules that define loads and events create data items that abide by certain representation conventions. The rules in the rule base are designed to recognize these conventions. The following sections describe of the major data types (conventions) used throughout EMES.

1. Representation of Loads

The following data items are used to describe a load to the EMES. Data items of the form

```
(load [load])
```

declare a load to the system. For each load, the expert system requires a data item of the form

```
(power-consumption [load] [value])
```

which states that the peak power consumption of the load with name [load] is [value] watts. To declare which bus in the power system the load is connected to, a data item of the form

```
(bus-connection [load] [bus])
```

is required, where [load] is the name of the load, and [bus] is the bus to which the load is connected (either critical, low power, or high power). The data item

```
(duty-cycle [load] [cycle])
```

is used to describe loads that have a known duty cycle, where [cycle] is the number of minutes the load will be turned on for each load request. Loads that do not have a known duty cycle, or that need to be on throughout the entire mission, do not require a duty-cycle data item. The data items

```
(continuous [load])
(interruptable [load])
(restartable [load])
```

are used to tell whether the power to the load must be continuous (for a given load request), whether the load can be interrupted, or whether the load can be interrupted and then restarted again. Note that these data items are disjoint (one and only one of these exists for each load). The data item

```
(subsystem [load] [system])
```

indicates which spacecraft subsystem the load resides in, or, for experiments, what the nature of this experiment is (science payload, commercial payload, or technology development payload).

Finally, data items of the form

```
(producer [load] [resource] [quantity])
```

are used to indicate that a [load] is the producer of a given [quantity] of some named [resource], and data items of the form

```
(consumer [load] [resource] [quantity])
```

are used to indicate that a [load] is the consumer of a given [quantity] of a named [resource].

Note that these data items only describe some of the general features characteristic of a wide variety of loads. Much of the information used in load management will be more subtle information that is idiosyncratic to a particular load or class of loads. This type of information is represented as procedural knowledge peculiar to the load management task, and resides directly in the rule base itself.

2. Representation of Events

The following data items are used to represent events and relationships between events. Data items of the form

```
(mission-phase [event-type])
(event [event-type])
```

distinguish between event types. Event-types tagged with "mission-phase" are major mission events (such as stationkeeping or rendezvous and docking), only one of which may be scheduled in any particular time slot. All other event types are tagged with "event." These are usually groups of loads that execute in some sequence relative to each other to perform a specified task.

Data items of the form

```
(event [event] (load-request [load]))
(event [event] (mission-phase [event-type]))
(event [event] (event [event-type]))
```

are used to distinguish between the various types of event instantiations the system must reason about. Events of type "load-request" are requests for the operation of some load. Events of type "mission-phase" are instances of a mission phase. All other events are instantiations of some other event type. In these data items, [event] is always an atomic name assigned to an event that is unique to this event in the current mission.

Data items of the form

```
(start-time [event] [time])
(end-time [event] [time])
```

tell the system when a particular event is scheduled to begin or end. Remember that the [time] may either be a single time slot or a time window.

A scheme for representing the relationships between events has been derived from the work of [Allen 1983]. This scheme is concerned with reasoning about events with respect to time. The data items we will be using to represent relationships between events are of the following form:

(conflict [event1] [event2])	: The two events must not occur in the same time interval.
(simultaneous [event1] [event2])	: The two events occur at exactly the same time.
(before [event1] [event2])	: [event1] occurs before [event2].
(after [event1] [event2])	: [event1] occurs after [event2].
(precedes [event1] [event2])	: [event1] immediately precedes [event2].
(follows [event1] [event2])	: [event1] immediately follows [event2].
(starts [event1] [event2])	: The start of [event1] is the same as the start of [event2], but [event1] finishes first.
(finishes [event1] [event2])	: The end of [event1] is the same as the end of [event2], but [event2] starts first.
(during [event1] [event2])	: [event1] occurs sometime during [event2].
(overlaps [event1] [event2])	: The occurrences of the events overlap.

The above data items are all used to perform temporal reasoning (reasoning with respect to time) on the events to be scheduled. Some may represent original (user-defined) constraints on load or event requests; others are created during the process of reasoning about the placement of events in the timeline.

3. Representation of Simulation Data

The expert system requires an internal representation of the simulation data produced by the engineering modules. This section describes the data items that provide the desired internal representation.

Data items of the form

    (orbit [n] window ([time1] [time2]))

describe the beginning ("[time1]") and end ("[time2]") of each orbit number

("[n]"). Here, [time1] and [time2] must be atomic times (and not time intervals). Similarly, data items of the form

    (orbit [n] daytime [window])

describe the daylight time interval for each orbit.

The maximum solar array capability for the solar arrays is given by data items of the form

    (solar-array-capability [time] [quantity])

One data item of type solar-array-capability is given for each 6-minute time slot during the daytime for the duration of the mission. Solar array capability during the nighttime is assumed to be zero. The data item

    (solar-array-degradation [percent])

is used to calculate how much of the total solar array power is available for use by the loads. This data item is also useful for describing a spacecraft with a given percentage of solar array cells inoperable.

Similarly, the maximum battery capability is given by data items of the form

    (battery-capability [orbit-number] [quantity])

One data item of type battery-capability is given for the nighttime of each orbit for the duration of the mission. Battery capability during the daytime (recharging) is assumed to be zero. The data item

    (battery-degradation [percent])

is used to calculate how much of the total battery capability is available for use by the loads. This data item is also useful for describing a spacecraft with a given percentage of battery cells inoperable.

Finally, data items of the form

    (battery-DOD [time] [percent])

are calculated by the expert system for each 6-minute time slot during the nighttime for the duration of the mission so the system can reason about depth-of-discharge criteria.

4. Representation of Resources

The following knowledge is used to reason about system resources. It is important to remember that power requirements for the various loads and power capability of the batteries and solar arrays are eventually reasoned about as resource problems.

Data items of the form

(resource-provided [producer] [resource] [time] [quantity])

are used to represent sources of system resources. "[producer]" is the entity (for example, a system component) that makes the resource available for consumption. "[resource]" is the resource produced, "[quantity]" is the amount produced, and "[time]" is the time slot for which it is produced (possibly an arbitrary time interval).

Data items of the form

(resource-available [resource] [time] [quantity])

are used to represent the amount of a given resource that is available for consumption at any given time. All "resource-provided" data items cause resources to be available until they are allocated to a consumer of that resource.

Data items of the form

(resource-required [consumer] [resource] [time] [quantity])

are used to represent resource constraints on loads or events. For example, these data items can represent the power requirements of a load, as well as the astronaut requirements for a mission event.

Finally, data items of the form

(resource-consumed [consumer] [resource] [time] [quantity])

are used to signify that a given amount of some resource has been allocated to a given consumer (something that requires that resource) during a specified time period.

Using this representation, the expert system can attempt to satisfy every request for a resource (every data item of type "resource-required") without violating the constraints on the amount of resources available at any given time.

B. EXPERT SYSTEM LOGIC

The EMES takes a goal-directed problem-solving approach to the energy management task. Many factors interact to provide a unique approach to reasoning about energy management, load scheduling, and constraint satisfaction. The most important of these are described in the following subsections.

1. Control Structure

EMES uses the rule-based control structure provided by the HAPS interpreter. Because this control structure is not standardly found in rule-based systems, the HAPS control structure will be described briefly.

The HAPS system provides a goal-directed, forward-chaining control structure for guiding the problem-solving process. This is effected through the repeated application of a set of recognize/act cycles. For each cycle, the following set of operations is performed:

1) Goal selection - At any given time, HAPS has a set of goals to pursue, arranged in a hierarchy. At the beginning of each cycle, this set of goals is examined and one is selected to be the focus of attention for the duration of this cycle;

2) Rule selection - The system determines which rules in the rule base are relevant to solving the current goal. For a rule to be applicable in this cycle, it must have all of its conditionals satisfied in the context of solving the current goal. The set of all applicable rule instantiations is known as the conflict set;

3) Conflict resolution - The system must select one rule from the conflict set to apply in this cycle. A set of conflict resolution procedures is used to choose one rule instantiation, which now becomes the current instantiation;

4) Rule application - The chosen rule is applied by executing the actions it specifies. These actions usually make changes in the system environment, such as creating new goals or inferring new facts about the world.

The system halts when it has achieved all goals in the goal hierarchy. The goal-directed nature of the HAPS control structure allows the EMES to decompose the energy management problem into subproblems in a dynamic fashion. That is, although the system creates different goals for each mission, the behavior of the rule base will not be adversely affected.

For more information on the HAPS system, see Chapter XI, which provides an introduction to HAPS, or consult [Sauers 1984].

2. The Goal Hierarchy

The EMES uses goals to direct the problem-solving process. Goals represent tasks to be performed, problems to be solved, or states to be achieved. When a mission is executed, the expert system is initialized with one top-level goal--to inspect the mission timeline. As rules in the rule base are applied toward the solution of this goal, goals representing subtasks may be created.

Goals in EMES are intended only as a guide to the higher level reasoning processes. For this reason, only a few different types of goals are permitted in EMES (that is, a small number of goals in relation to the number of rules expected in the rule base). The major advantage in designing the EMES system in this fashion is that this type of design results in a loose coupling of rules in the rule base to goals in the goal hierarchy. Briefly, this means that the system is more modular and easily extensible. This notion will be discussed further in Subsection 3.

Because of the relative scarcity of goal types available for the problem-solving process, most of the goal types defined in EMES are general and highly parameterized. The most important EMES goal types are:

```
(GOAL-TYPE
   OBJECT: Inspect-Timeline)
```

The top-level goal given to EMES is of type Inspect-Timeline. When EMES is inspecting a mission timeline, it searches for

1) Events or load requests that have not yet been scheduled;

2) Energy management constraints that have been violated;

3) Scheduling constraints that have been violated;

4) Resource constraints that have been violated;

5) Energy management considerations that have not been optimized;

6) Resources that have not been allocated in an optimal fashion.

The occurrence of any one of these conditions indicates that the current mission timeline is not satisfactory. Thus, the expert system must modify the existing timeline so the offending condition is no longer true.

```
(GOAL-TYPE
   OBJECT: Schedule-Event
   EVENT: =event-name
   TIME1: =early-time
   TIME2: =late-time)
```

Goals of type Schedule-Event are used when EMES needs to schedule an event with an inexact start time (a start time that is a window). Goals of this type require several parameters:

=event-name : the name of the event to be scheduled. This can be the name of a mission phase, a load request, or an ordinary event.
=early-time : the earliest time (relative to the start of the mission) at which this event can begin.
=late-time : the latest time at which this event can begin.

Rules that fire in the context of Schedule-Event goals will consider such factors as availability of resources, scheduling constraints, and energy management constraints.

```
(GOAL-TYPE
   OBJECT: Un-Schedule-Event
   WHY: =justification
   TIME1: =start-interval
   TIME2: =end-interval)
```

Goals of type Un-Schedule-Event are used to remove an event from the timeline. The parameters required are

=justification : a description of the condition which resulted in the need for an event to be rescheduled.
=start-interval : the beginning of the time interval during which the offending condition is true.
=end-interval : the end of the offending time interval.

Note that goals of this type do not specify the event that needs to be unscheduled. This is because the condition that caused the unscheduling requirement may result in a choice of events to be unscheduled, or may require the unscheduling of several events. Examples of the types of justifications possible are:

```
(conflict load-1 event-1)
(resource-exceeded power)
(energy-constraint-violation battery-DOD)
```

The unscheduling of an event usually results in the removal of an event from the mission timeline, and a request for that event to be rescheduled (either during the original start-time window specified by the user, or a modified start-window produced by EMES).

The goals in the goal hierarchy will be pursued at runtime in a depth-first fashion. Thus, the HAPS goal selection strategies will be set appropriately. This decision was made because of the relative scarcity of goal insertion. When a new goal is inserted into the hierarchy, it requires an immediate shift in focus of attention.

3. Conflict Resolution

In a rule-based system, the process by which one rule is selected from the set of all rules that can be applied on any given cycle is known as conflict resolution. Some common conflict resolution strategies are:

refraction: Each instantiation of a given rule can be applied only once.
recency: Rules that match the most recently asserted data items are preferred over rules that match old data items.
specificity: Rules that test more conditions are preferred over rules that test fewer conditions.
randomness: A random rule is preferred over all others.

Most expert systems employ a conflict resolution strategy that is a combination of the above strategies designed to minimize the number of times the randomness strategy needs to be applied.

The conflict resolution process to be used by the EMES will be a sequential application of the following:

refraction: Do not fire the same instantiation of a rule more than once at the same goal.
subsumption: If rule1 is a special case of rule2, then prefer instantiations of rule1 over instantiations of rule2.
randomness: Prefer a random instantiation over all of the others.

With this conflict resolution procedure, the randomness strategy will be the one that most often selects a single rule. This strategy will be used in EMES to ensure loose coupling of rules in the rule base. That is, if more than one rule is able to fire on a given cycle, it usually should not matter as to the order in which the rules are applied. As long as the rule applied is relevant to the current goal, it will be logically correct to fire that rule.

This approach to the design of the EMES system will increase the degree of modularity of rules in the rule base. That is, rules must test sufficient conditions to guarantee the correctness of the rule because we cannot rely on the conflict resolution process to ensure that rules fire in the correct sequence. The advantage of this technique is that the resulting EMES system is readily extensible. New rules can be added to the rule base without harmful side effects on the existing rules. This means we will be able to increase the amount of expert knowledge contained in EMES more easily than in traditional expert systems.

C. THE RULE BASE

Several types of rules required for operation of the EMES have been identified. The most important classes of rules are described in the following subsections.

1. <u>Scheduling Rules</u>

Scheduling rules are rules that ensure scheduling constraints are not violated when a new event is scheduled into the mission timeline. For example, if load-1 is required to operate before load-2, the scheduling rules would ensure this constraint.

Scheduling rules fire in the context of Schedule-Event goals. When an event is to be scheduled, the expert system maintains a set of time intervals during which it is legal to schedule that event. Scheduling rules are designed to update that set of time intervals in cases where the event to be scheduled is constrained by an event already on the timeline.

If more than one scheduling constraint applies to an event, scheduling rules will be designed so the correct set of legal time intervals for the scheduling of this event will be created independent of the order in which the various scheduling rules are applied.

2. <u>Resource Management Rules</u>

Resource management rules ensure that resources are allocated in an optimal fashion. Several types of resource management rules are required. One type recognizes resource management constraints that have been violated. For example, the power requirement of the loads scheduled for a given time interval cannot exceed the power available during that time interval. A second type of resource management rule recognizes situations in which, although the amount of an available resource has not been exceeded, a different allocation of that resource would have been more efficient. For example, it is desirable to minimize the load average of an onboard computer system.

Resource management rules operate on the set of legal time slots available for the scheduling of a load request in much the same way as scheduling rules. If more than one resource management constraint applies to a load request, resource management rules will be designed so the correct set of legal time intervals for the scheduling of that load request will be created independent of the order in which the various resource management rules are applied.

3. <u>Energy Management Rules</u>

Energy management rules ensure that energy management constraints are not violated. Some energy management rules place constraints on the scheduling of loads. For example, loads with a high peak power consumption should not be scheduled during the eclipse period of an orbit. These rules behave in a fashion similar to resource management rules in that they update the set of legal time periods for which a given load can be scheduled. Other energy management rules are optimization rules. For example, if we are trying to optimize battery lifetime, we should minimize battery depth of discharge. Optimization rules place priorities on legal time periods for scheduling a load, and are used to choose the best of all legal scheduling slots.

4. Goal Satisfaction Rules

Goal satisfaction rules determine when an EMES goal has been successful. For example, a goal of type Un-Schedule-Event is successful when the reason for the unscheduling requirement (the justification) is no longer true. Goal satisfaction rules declare the current goal to be successful. This indicates that the system is free to pursue another goal in the hierarchy (that is, change its focus of attention).

5. User Interface Rules

Some EMES rules must interface directly with the user when the expert system must make a decision that is out of the ordinary. For example, if a given load cannot be scheduled in the interval the user originally specified in the mission timeline, the expert system may be able to schedule the load somewhere outside of the specified window. In this case, the user must confirm this decision.

User interface rules are also required for recovery from error situations. For example, the user may want to ignore a load request that requires more of a given resource than other spacecraft components can ever produce.

User interface rules can communicate directly to the user from either the IF: portion of a rule, or the THEN: portion of a rule. In either case, the interface is through an external LISP function call. In instances where the user is required to select among a given set of alternatives, the interface will be through the Select_Menu_Entry module.

6. Examples

This section provides examples of rules that will make up the energy management system. In some cases, the rules have been simplified for the purpose of illustration. Each sample rule is expressed in the HAPS language formalism.

```
(PRODUCTION loads-consume-power
    CONTEXT:
       (OBJECT: Inspect-Timeline)
    IF:
       (event =name (load-request =type))
       (power-consumption =type =watts)
       (start-time =event =interval)
    THEN:
       (resource-required =name power =interval =watts))
```

This rule is one of the rules that allows EMES to reason about power as a resource. It describes a load as a consumer of the power resource. Another rule that allows the system to reason about power as a resource is:

```
(PRODUCTION solar-arrays-provide-power
    CONTEXT:
       (OBJECT: Inspect-Timeline)
    IF:
        (solar-array-capability =time =watts)
        (solar-array-degradation =percent)
    THEN:
        (resource-provided solar-arrays
             power =time (*times =watts (*diff 1.0 =percent))))
```

This rule describes the solar arrays as a producer of the power resource. The amount of power produced by the solar arrays depends on both the maximum solar array capability and the percentage of solar array degradation.

```
(PRODUCTION conflicting-events
    CONTEXT:
       (OBJECT: Inspect-Timeline)
    IF:
       (conflict =event1 =event2)
       (during =event1 =event2)
       (start-time =event1 =time1)
       (start-time =event1 =time2)
    THEN:
       (GOAL
         OBJECT: Un-Schedule-Event
         WHY: (conflict =event1 =event2)
         TIME1: =time1
         TIME2: =time2))
```

This is a simple scheduling rule that recognizes that if two conflicting events have been scheduled during the same time interval, one of them must be unscheduled. Thus, the THEN: portion of this rule sets up a new goal whose object is to Un-Schedule one of the events.

```
(PRODUCTION generic-resource-constraint
    CONTEXT:
       (OBJECT: Schedule-Event
        EVENT: =name)
    IF:
       (event =name (load-request =type))
       (consumer =type =resource =quantity1)
       (legal-time-interval =name (=min =max)) :1
       (resource-available =resource (=time1 =time2) =quantity2)
       (*lessp =quantity2 =quantity1)
       (*greaterp =time1 =min)
       (*lessp =time2 =max)
    THEN:
       (*remove :1)
       (legal-time-interval =name (=min =time1))
       (legal-time-interval =name (=time2 =max)))
```

This is a resource-management rule that fires in the context of a Schedule-Event goal. This rule recognizes that a load cannot be scheduled during a time interval if it requires more of some resource than is available during that time interval. The set of time slots that are legal for the scheduling of this load is updated so it does not include the bad time interval.

VII. SYSTEM SUPPORT MODULES

This section describes selected system support modules that are used as utility functions by many of the individual modules throughout the EMES.

A. ADD DATA ITEMS

The Add_Data_Item function adds a new data item to the Mission_Definition_Data list. The Mission_ Definition_Data list is a global "tconc" list (LISP list structure that permits efficiently adding new elements to the end of the list) that collects mission definition information as it is obtained. The Add_Data_Item function requires two arguments:

Data_Item_to_Add: The data item to be added to the mission model.
Translate_Flag: A boolean flag that tells us whether or not to automatically show new data items to the user.

The Data_Item_to_Add is added to the Mission_Definition_Data only if it is not already there (that is, if this is a new data item). If it is a new data item and Translate_Flag is true, the data item is also shown to the user via the Translate_Data_Item routine. The Add_Data_Item function returns "t" if the data item was added to the Mission_Definition List and returns "nil" otherwise.

B. DIRECT USER OUTPUT

The Direct_User_Output module allows the user to choose which device to send output to. The user may choose to sent output to terminal, to a named disk file, or to the line printer. The user makes his selection through the use of a menu, implemented as a call to Select_Menu_Entry, with Menu_Descriptor equal to

```
("Send output to ..."
    (tty . "The Terminal.")
    (file . "A Disk File.")
    (printer . "The Line Printer."))
```

The Direct_User_Output module returns the port to which the output should be sent. If Select_Menu_Entry returns "tty," the Direct_User Output returns "nil." Other possible value returned by Select_Menu Entry are "file" and "printer." In the case of "file," the user is prompted for the name of the file, the file is opened for output, and the resulting port is returned as the value of Direct_User_Output. In the case of "printer," the file "EMES_printer:emes.tmp" is opened for output, and the resulting port is returned as the value of Direct_User Output. The VMS logical name "EMES_printer:" must be set up to point to the print queue of the desired line printer (for example, "LPA0:").

C. INSTANTIATE DATA ITEM

The Instantiate_Data_Item module is used to transform a data item that describes an event type or a load into a data item that represents a constraint on an event. Several parameters are required:

Data_Item_to_Instantiate: The data item to be instantiated.
Replace_List: A list that tells us which terms in the data item need to be modified when the data item is instantiated. Each member of the Replace_List is of the form

```
(generic-event . instantiated-event),
```

and indicates that the generic load or event-type "generic-event" is now to be instantiated using the name of the new event "instantiated-event."

When a data item is instantiated, all instance-dependent data items (those that can change between instances of an event) must be modified. These data types are currently data items that specify the relationships between events (for example, "after" or "during" data items).

Data items that declare a load or event type to the system must also be instantiated so they create declarations of instantiated events. For example, the data item

```
(load star-lab)
```

would be instantiated to

```
(event EVENT-37 (load-request star-lab))
```

Notice the creation of a new event name, EVENT-37, that is created through a call to the New_Event_Name module. Whenever a new event name is created, the Replace_List will also become modified so it includes the new event name. For example, suppose that, before the creation of EVENT-37, the Replace_List was

```
((life-sciences-lab . EVENT-36)
(materials-processing . EVENT-35)
(solar-optical-telescope . EVENT-34))
```

After the creation of the new event, the Replace_List becomes

```
((start-lab . EVENT-37)
(life-sciences-lab . EVENT-36)
(materials-processing . EVENT-35)
(solar-optical-telescope . EVENT-34))
```

The Instantiate_Data_Item module returns a

```
(data-item . list)
```

pair, where "data-item" is the instantiated value of the input data item, and "list" is the updated Replace_List.

D. NEW EVENT NAME

The New_Event_Name function creates a new event name. One parameter is required: the name of the load or event-type being instantiated. Also, the value of Event_Instantiation_List, which is global to this module, is accessed and modified. The Event_Instantiation_List is of the form

```
(([name1] . [n1])
([name2] . [n2])
     ...
([nameN] . [nN]))
```

Each "[name]" is a load name or event type, and each [n] is the number of times this [name] has been instantiated in this mission. In this way, the new event name generated can be derived from the generic event name and still be unique to this mission.

For example, if the load being instantiated was "star-lab," and the value of Event_Instantiation_List was

```
((life-sciences-lab . 1)
(materials-processing . 3)
(solar-optical-telescope . 1))
```

the new event name generated would be

```
star-lab:Event-1
```

and the value of Event_Instantiation_List would be changed to

```
((star-lab . 1)
(life-sciences-lab . 1)
(materials-processing . 3)
(solar-optical-telescope . 1))
```

The New_Event_Name function returns the name of the new event.

E. REL TIME

The REL_TIME module is a Fortran-coded module that calculates the time of a mission slot relative to the start of the mission. The following parameters are required:

GYEAR_NOON1 :	Last two digits of the year corresponding to the local noon of the initial mission orbit. The type is INTEGER and the range is 0 to 99.
GMON_NOON1 :	Month of the year corresponding to the local noon of the initial mission orbit. The type is INTEGER and the range is 1 to 12.
GDAY_NOON1 :	Day of the month corresponding to the local noon of the initial mission orbit. The type is INTEGER and the range is 1 to 31.
GHOUR_NOON1 :	Hour of the day corresponding to the local noon of the initial mission orbit. The type is INTEGER and the range is 0 to 23.
GMIN_NOON1 :	Minutes of the hour corresponding to the local noon of the initial mission orbit. The type is REAL and the range is 0.0 to 59.99.
GYEAR_NOON2 :	Last two digits of the year corresponding to the local noon of the mission orbit whose time is to be calculated. The type in INTEGER and the range is 0 to 99.
GMON_NOON2 :	Month of the year corresponding to the local noon of the mission orbit whose time is to be calculated. The type is INTEGER and the range is 1 to 12.
GDAY_NOON2 :	Day of the month corresponding to the local noon of the mission orbit whose time is to be calculated. The type is INTEGER and the range is 1 to 31.
GHOUR_NOON2 :	Hour of the day corresponding to the local noon of the mission orbit whose time is to be calculated. The type is INTEGER and the range is 0 to 23.
GMIN_NOON2 :	Minutes of the hour corresponding to the local noon of the mission orbit whose time is to be calculated. The type is REAL and the range is 0.0 to 59.99.

The REL_TIME module calculates the number of minutes between the start time of the mission and the time slice to be determined. An integer representing the number of minutes is returned.

F. SELECT MENU ENTRY

The Select_Menu_Entry Module is a utility that provides a general menu-driven user interface. All software modules that require menu selection from the user will interface with the user through this module.

Select_Menu_Entry is a LISP function of one argument, referred to as Menu_Descriptor. Menu_Descriptor is a (Menu_Name . Menu_Entries) pair, where Menu_Name is a string representing the name of this menu, and Menu_Entries is a list of descriptors for the individual menu choices. Each menu entry is a (Menu_Selection_Name . Menu_Selection_Text) pair. When a call to this module is made, the Menu_Name is printed on the terminal, followed by the possible menu selections. Each Menu_Entry gets a number, after which is printed the corresponding Menu_Selection_Text. Then, the user is prompted for his selection, which must be an integer corresponding to one of the Menu_Entries. Select_Menu_Entry returns the Menu_Selection_Name corresponding to the selected menu entry.

For example, suppose a call to Select_Menu_Entry was made, with Menu Descriptor equal to

```
("Library Development."
    (Load_Library . "Load Library Development.")
    (Event_Library . "Event Library Development.")
    (Exit . "Return to Top Level Menu."))
```

Then, the following is an example of menu-driven interaction with the user:

```
Library Development.

    1. Load Library Development.
    2. Event Library Development.
    3. Return to Top Level Menu.

Menu Selection: 5

? Bad menu selection.
? Menu selection out of range.

Library Development.

    1. Load Library Development.
    2. Event Library Development.
    3. Return to Top Level Menu.

Menu Selection: 3
```

In this case, the Select_Menu_Entry function would return the value "Exit."

G. TRANSLATE DATA ITEM

The Translate_Data_Item routine prints an English description of a data item to be used by the expert system. Three arguments are required:

> Data_Item_to_Translate: The value of the data item to be translated.
> Translate_Context: An atom representing the context in which this data item is being translated. This is used so the same data item can translate several ways, depending on the context value.
> User_Output_Port: The output port where the English translation should be written.

The Translate_Data_Item routine merely does a case analysis on the type of the Data_Item_to_Translate (that is, the first term in the data item) to find the translation procedure. Thus, Translate_Data_Item must have a translation procedure for each type of data item to be translated. If there is no translation procedure for a given data item, no English description of the data item will be printed. Any individual translation procedure can refer to the Translation_Context during the translation process. An unrecognized Translation_Context is ignored, and a default context is assumed.

For example, suppose the Data_Item_to_Translate was

```
(before item-1 item-2)
```

Then, if Translate_Context was "load," the English translation might be

> "Item-1 must operate before item-2."

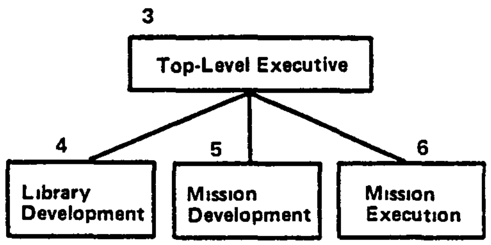
whereas the same data item, in Translate_Context "event," might be

> "Item-1 must occur before item-2."

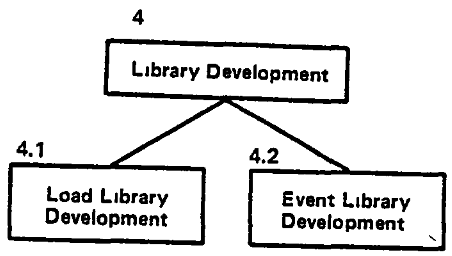
Translate_Data_Item returns 't' if a translation was made, and 'nil' otherwise.

VIII. SYSTEM BLOCK DIAGRAMS

This chapter provides block diagram descriptions of the major software modules in the Energy Management Expert System.



VIII-1 The Top-Level Executive



VIII-2 Library Development Module

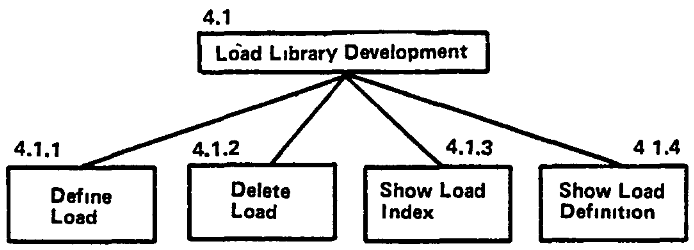
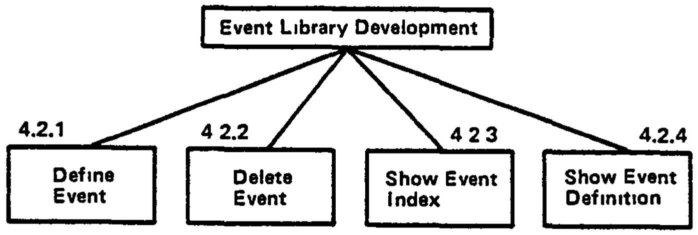
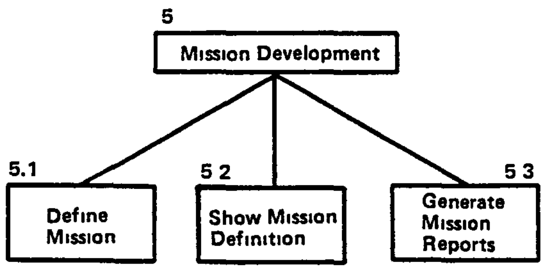


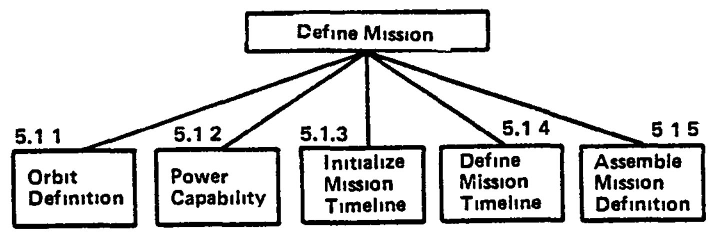
Figure VIII-3 Load Library Development Module



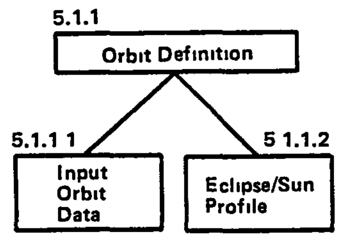
VIII-4 Event Library Development Module



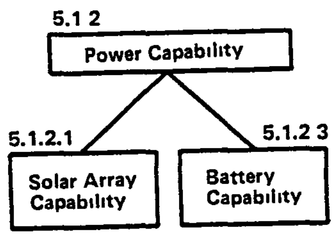
VIII-5 Mission Development Module



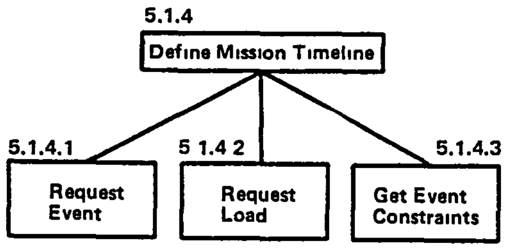
VIII-6 Define Mission Module



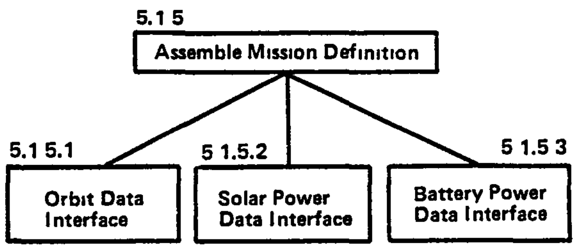
VIII-7 Orbit Definition Module



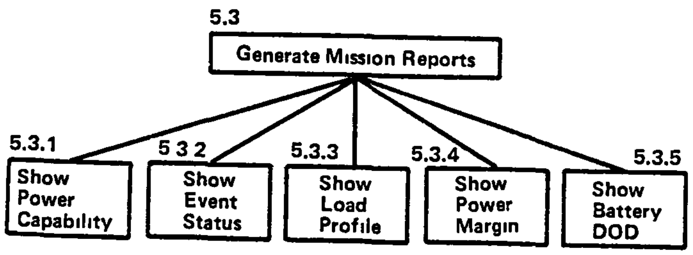
VIII-8 Power Capability Module



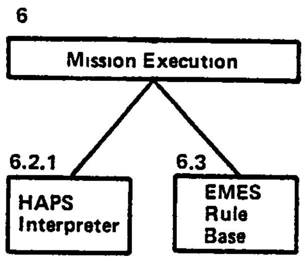
VIII-9 Define Mission Timeline Module


5.1 5
Assemble Mission Definition
5.1 5.1
Orbit Data Interface
5 1.5.2
Solar Power Data Interface
5 1.5 3
Battery Power Data Interface

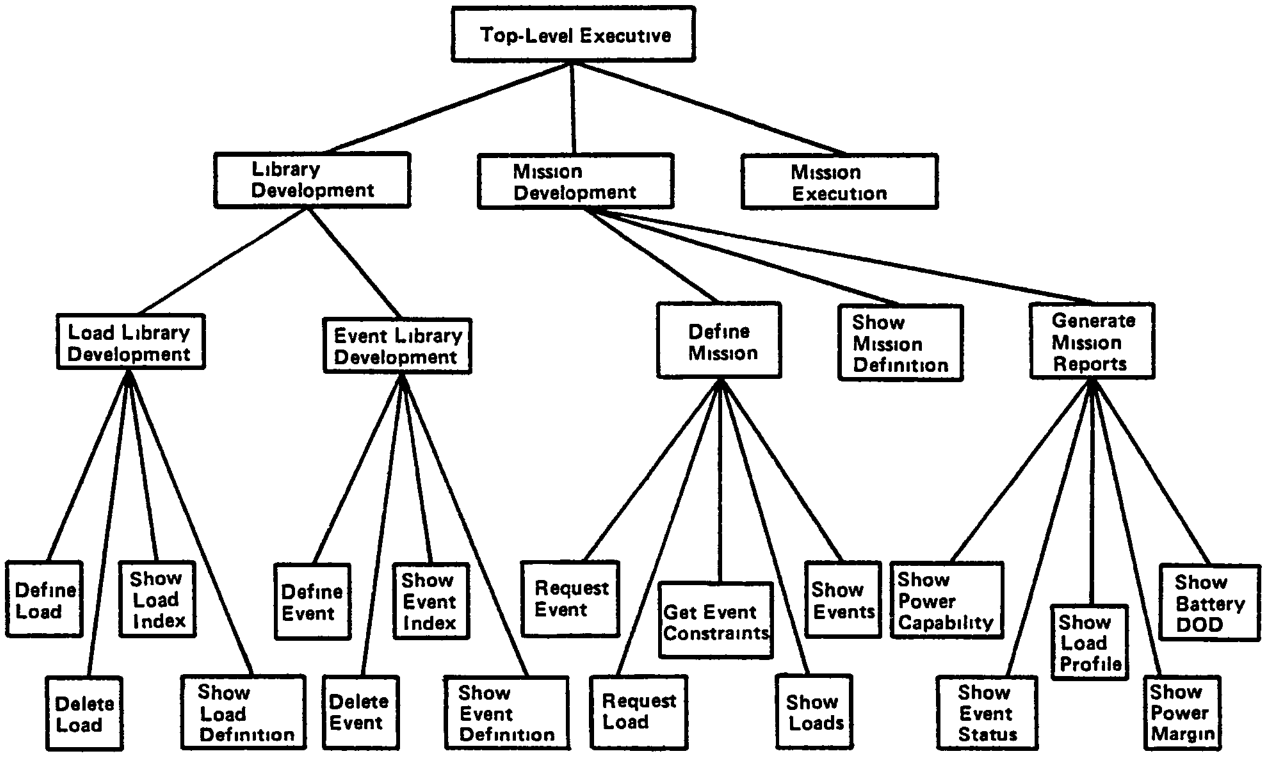
VIII-10 Assemble Mission Definition Module



VIII-11 Generate Mission Reports Module



VIII-12 Mission Execution Module



VIII-13 Menu Structure Overview

IX. FILE STRUCTURE OVERVIEW

The energy management expert system (EMES) will be integrated into the environment of the VMS operating system so the underlying file structure is transparent to the user. The following VMS logical names represent directories used by EMES:

1) EMES_system: This directory contains files referenced by the internal EMES system. The user does not need to directly access these files at any time during execution of EMES;

2) load_lib: This is the load_library_directory that contains files defining all of the various loads known to the system;

3) event_lib: This is the event_library_directory that contains files defining all of the various event types known to the system;

4) mission_dir: This is the directory that contains all of the mission models defined so far. The mission_dir: will be the directory from which the EMES is being run (usually, the user's home directory);

5) EMES_temp: This is the directory that contains all temporary files required during execution of the EMESy. The EMES_temp: directory will be the directory from which the EMES is being run (usually, the user's home directory);

6) EMES_source: This directory contains the source code for the EMES, including the EMES rule base;

7) EMES_main: This is the main EMES directory that contains the executable EMES image and the compiled rule base. It is likely that the EMES_main: directory will be the same as the EMES_system: directory, although this is not required.

The intended directory structure is shown in Figure IX-1.

In addition, the following files are referenced by the EMES:

1) event_library_definition_file: This is the file that contains the definition of an event type in the event library. One such file exists for each defined event type. The name of each of these files shall be of the form '[file_name].def'. Each of the event_library definition_files is a LISP text file, and shall be located in the event_lib: directory;

2) event_library_index_file: This file contains an index of the event types currently defined in the system. This file is a LISP text file, and is only for internal use by EMES. The name of the file shall be eventlib.idx, and the file shall be located in the EMES system: directory;

3) load_library_definition_file: This is the file that contains the definition of a load in the load library. One such file exists for each defined load. The name of each of these files shall be of the form "[file_name].def." Each load_library_definition_file is a LISP text file, and shall be located in the load_lib: directory;

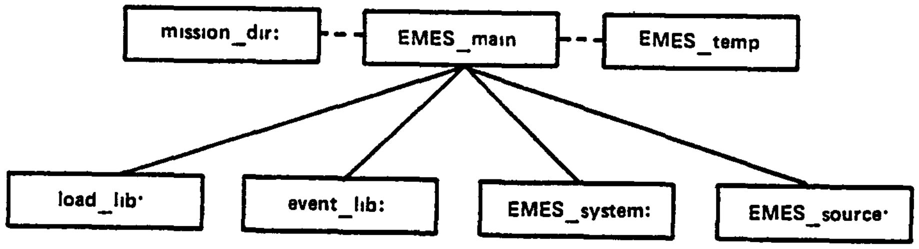


Figure IX-1 EMES Directory Structure

4) load_library_index_file: This file contains an index of the loads currently defined in the system. This file is a LISP text file, and is only for internal use by EMES. The name of the file shall be loadlib.idx, and the file shall be located in the EMES_system: directory;

5) Battery_Power_Profile_File: This file is generated by the Battery Capability module. It contains the battery power profile for a given mission. The Battery_Power_Profile_File will be a machine-readable, temporary, sequential file. The name of this file shall be batdata.dat, and the file shall be located in the EMES_temp: directory;

6) Solar_Power_Profile_File: This file is generated by the Solar Array_Capability module. It contains the solar array power profile for a given mission. The Solar_Power_Profile_File will be a machine-readable, temporary, sequential file. The name of this file shall be soldata.dat, and the file shall be located in the EMES temp: directory;

7) Day_Night_Cycles_File: This file is generated by the Eclipse_Sun Profile module. It contains orbit definition information for a given mission. The Day_Night_Cycles_File will be a machine-readable, temporary, sequential file. The name of this file shall be sundata.dat, and the file shall be located in the EMES_temp: directory;

8) spacecraft-configuration-file: This file contains the information necessary to initialize a mission definition so it contains the initial configuration of the spacecraft. This is a LISP text file. The name of this file shall be config.sys, and the file shall be located in the EMES_system: directory;

9) Mission_Definition_File: This file contains the data that defines a mission to the EMES. One such file exists for each mission definition. The name of a Mission_Definition_File is obtained from the user. Each file shall be located in the mission_dir: directory.

Finally, the following is a list of miscellaneous files that will be provided as part of the EMES:

1) emes.exe. This is an executable LISP image that contains the compiled EMES code. This file will reside in the EMES_main: directory;

2) emes.stb. This is the symbol table required by the emes.exe executable LISP image. This file will reside in the EMES_main: directory;

3.) emesrules.hps. This is the file containing the source code for the EMES rule base. This file will be located in the EMES_source: directory;

4) emesrules.o. This is the compiled version of the EMES rule base. This file will be located in the EMES_main: directory;

5) hapszt.exe. This is an executable version of the HAPS compiler required to compile the EMES rule base. This file will be located in the EMES_system: directory;

6) hapszt.stb. This is the symbol table required by the hapszt.exe executable LISP image. This file will be located in the EMES_system: directory;

7. emes.com. This is a VMS command file that sets up all of the logical names and other miscellaneous symbols required to declare the EMES file structure to the VMS environment. This file should be called from the user's login.com file so that the EMES environment is automatically set up each time the user logs in.

X. LISP OVERVIEW

The LISP programming language is a high-level language frequently used for Artificial Intelligence applications. LISP provides a uniform representation scheme for both programs and data. This allows LISP programs to be manipulated and modified by other LISP programs just as if they were ordinary data.

The LISP language is designed for the manipulation of symbols. The entities manipulated by a LISP program are called symbolic expressions. (LISP programs are also symbolic expressions.) LISP provides two main primitives for the creation of symbolic expressions: atoms and lists. An atom is a sequence of characters, such as

```
symbol-1
has-property or
list-of-expressions
```

Atoms are used as the names of LISP variables, the names of LISP functions, and as symbols representing arbitrary concepts. A list is a sequence of symbolic expressions, enclosed in parentheses. Thus, the following are examples of LISP list structure:

```
(list of expressions)
(list-with-one-atom)
(more (complex (list (((structure))))))
```

The LISP language is a functional language. That is, every program in LISP is implemented through the use of functions that are applied to arguments. LISP provides a method for evaluating symbolic expressions. When an atom is evaluated, it is considered a variable, and the value of that variable is returned as the result of the evaluation. When a list is evaluated, it is considered to be an application of a function. The first item in the list is the name of the function to be applied, and the remaining terms in the list are the arguments the function is applied to. Each argument is evaluated recursively before the function call is made.

For example, one function defined in LISP is "quote," which takes one argument and does not evaluate it. The argument itself is returned. Another LISP function is "car," which returns the first element in a list. Thus, when LISP evaluates the symbolic expression

```
(car (quote (a b c))),
```

it must perform the following sequence of operations:

1) "quote" is a special function that returns its argument without evaluating it. Thus, "(quote (a b c))" is an application of a function that returns the value "(a b c);"

2) "car" returns the first element of a list that, in this case, is the result of the evaluation of the "quote" function. Thus, the "(car ...)" expression returns the value "a."

LISP provides methods for allowing the user to define new functions. These functions can be used just as if they were functions defined by the LISP system. LISP programs are developed by writing a sequence of functions. The programs are executed by calling these functions in a given sequence (one function may call another when it is evaluated).

The LISP programming language is the language of choice in the AI community for several reasons. First, knowledge is easily represented in a symbolic fashion. LISP programs can reason about the world by inspecting symbolic representations of facts that describe the world. Programs can also inspect other programs or write new programs. This mechanism allows a LISP program to learn as it executes. Finally, the uniformity of the LISP formalism allows programs to be constructed quickly, while at the same time encouraging the structured programming style.

XI. HAPS OVERVIEW

HAPS (the hierarchical, augmentable production system) architecture is a sophisticated tool developed by the Martin Marietta artificial intelligence unit to allow the rapid construction of rule-based systems in real-world environments--that is, in uncontrolled environments that require the use of tremendous amounts of both knowledge about the application domain and expert knowledge about problem-solving in that domain.

One of the major advantages this system has over the traditional production system implementations is the notion of goal directedness. HAPS has a separate memory structure called goal memory, which contains a hierarchy of goals the system must achieve. Because all rules must apply in the context of some goal; HAPS rules are expressed in the form

```
IN some particular goal context,
IF a given set of conditions is true
THEN perform this set of actions.
```

The system is initialized with a top goal, and the overall system objective is to achieve this goal. On each cycle, a set of modifiable goal selection strategies is used to select the current goal, which becomes the system's focus of attention for that cycle. When a rule is applied toward achieving a goal, it can declare that goal to be a success or a failure, or it can cause the goal to sprout subgoals.

Another characteristic of HAPS is the ability to construct hierarchically structured levels of working memory. Data items can be declared local to a particular goal. This means they are available for the solution of that goal and its subgoals. When a goal is achieved, there is no longer any need to keep the local data associated with that goal. Thus, working memory is not cluttered with data items no longer needed. This scheme also allows each goal to have its own world model. This permits the simultaneous pursuit of multiple problem solutions that might ordinarily interact with each other to produce inconsistencies.

Similarly, HAPS introduces the notion of production hierarchies. Under this scheme, a rule set can be loaded into the system at runtime and declared local to a particular goal. This rule set is available for the pursuit of that goal and its subgoals. Furthermore, these rule sets can be loaded in by another rule, allowing the production hierarchy to be extremely dynamic. The major advantage of this scheme is that it allows HAPS to function without a decrease in level of performance in very large expert systems because only a small fraction of the entire rule base needs to be processed at any given time.

HAPS is provided with a modifiable set of goal selection strategies (used to select the current goal) and conflict resolution strategies (used to choose between competing rules). Because these strategies are modifiable, the user can tailor the needs of the system to individual applications. These strategies can also be changed by rules in the rule base, allowing the system to modify its behavior in response to changes in its environment.

Finally, HAPS is equipped with a set of alternate memory structures that can be used to store data items in the same way as standard working memory. Examples of types of alternate memory structures are tables and arrays. These structures make HAPS easier to interface with other existing software systems. Also, the operations performed on these structures (for example, pattern matching) are designed to allow HAPS to more easily interface with real-time changing data.

In summary, the HAPS system is equipped with many features that make it applicable to the development of large, sophisticated expert systems in real-world domains.

XII. REFERENCES

[Allen 1983] James F. Allen and Johannes A. Koomen: "Planning Using a Temporal World Model." Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, 1983.

[Brown 1969] R.H. Brown: "Mission Planning As Affected by Sun-Orbit Plane Relationships." AIAA Paper No. 69-129, AIAA 7th Aerospace Sciences Meeting, New York, NY, January 1969.

[Sauers 1984] Ron Sauers and Jonathan Bein: HAPS Reference Manual, Forthcoming Martin Marietta Technical Report, 1984.

APPENDIX C - ENERGY MANAGEMENT EXPERT SYSTEM TEST PLAN

The following pages contain the software test plan originally submitted to Marshall Space Flight Center in December 1983. This software test plan will be used in testing and validating the Energy Management Expert System (EMES). This appendix is included in this report to satisfy the requirements of the contract statement of work.

I. INTRODUCTION

This appendix outlines the test plan that will be used to verify correct operation of the energy management expert system (EMES). The EMES testing will be conducted in several phases:

1) Concept design validation - The detailed design specification for the EMES software will be reviewed to ensure that the system design addresses the functional requirements specified in the EMES requirements document;

2) Engineering software validation - The engineering software used to simulate spacecraft orbit, solar array capability, and battery power capability will be tested and verified;

3) Mission development software validation - The mission development software, which allows the user to develop mission models to be examined by the expert system, will be tested and verified;

4) Rule base validation - The expert system rule base, which encapsulates the knowledge used by the payload operations director in order to perform load and energy management tasks, will be verified;

5) Demonstration - The EMES will be demonstrated at Marshall Space Flight Center (MSFC).

The individual phases in the testing and validation process are described in the following sections of this appendix.

A. CONCEPT DESIGN VALIDATION

During the concept design validation phase, the detailed design specification for EMES will be examined by in-house software engineers and artificial intelligence researchers to ensure that the design meets the functional requirements as described in the EMES Requirements Document. Namely:

1) The software design must give EMES the capability to provide the required electric power to the user loads under normal and degraded modes of power system operation;

2) The design must allow EMES to optimize the use of available power;

3) The design must include a scheme for determining the sequence of operation of all electrical equipment in conjunction with basic mission operation activities and requirements;

4) The design must show how energy management heuristics can be used by EMES to extend the life of such critical power system components as the batteries;

5) EMES must be designed so it can to be extended to handle future concerns such as real-time onboard power system automation;

6) EMES must be able to incorporate the operational constraints of other subsystems as they are defined.

The EMES design is subject to final approval by MSFC.

B. ENGINEERING SOFTWARE VALIDATION

During the engineering software validation phase, the engineering software modules will be tested to ensure that they provide reasonable simulations of spacecraft environment and power capabilities. The specific functions involved are:

1) Orbit data calculation - The EMES engineering software must adequately simulate the orbital configuration of the spacecraft. This will allow accurate calculation of the amount of time per orbit the spacecraft is in the earth's shadow for the duration of the mission;

2) Solar array capability - The EMES engineering software must provide an adequate simulation of the amount of power available from the photovoltaic components of the power system for each time interval during the daylight portions of the mission;

3) Battery power capability - The EMES engineering software must provide a battery model sufficient for the EMES to be able to intelligently reason about energy management constraints and the nighttime power capability for the duration of the mission.

To verify the correctness of the engineering software, a testbed containing a sample orbit configuration shall be developed, and the engineering modules will be executed using the data in this testbed. The results will be represented graphically and inspected by Martin Marietta power systems experts.

C. MISSION DEVELOPMENT SOFTWARE VALIDATION

During the mission development software validation phase, the software designed to build load and event libraries will be tested, along with the software used to construct mission models. This software will be tested along three dimensions:

1) The development software will be tested by Martin Marietta load management experts to ensure that capabilities are provided to accurately describe system loads, mission events, and operational constraints required to specify real-world mission timelines;

2) The development software will be inspected by Martin Marietta software engineers to ensure that the interaction between the system and user is well defined, and that the system is both robust and user-friendly in these interactions;

3) The resulting mission models produced by the development software will be inspected by internal artificial intelligence researchers to ensure that the representation of the mission is both consistent and readily amenable to the rule-based reasoning requirements of the EMES system.

D. RULE BASE VALIDATION

During the rule base validation phase, the EMES rule base will be thoroughly tested to ensure that it behaves in the manner specified in the EMES requirements document. This requires the following process:

1) Design of a realistic mission model by a previously identified human expert in the energy management domain;

2) Execution of EMES on the test mission model;

3) Comparison of resulting modified mission timeline with that produced by the human expert;

4) Modification and augmentation of the rule base to incorporate increasingly intelligent heuristics. Steps 2 through 4 will be iterated until the performance exhibited by EMES is approved by the human expert;

5) Repetition of the above process with a new mission timeline;

6) Repetition of the above process with the spacecraft operating in increasingly degraded modes. The human expert must confirm that EMES allows the system to degrade gracefully with decreasing power capability.

Note that the process of iteratively refining the rule base to incorporate increasing amounts of expert knowledge is standard practice in the development of rule-based expert systems.

E. DEMONSTRATION

The final phase of testing and validation is demonstration at MSFC. The EMES will be ported to the MSFC computing facilities. Extensive testing will be performed to ensure that the behavior of the system is correct in the new computing environment.

The software demonstration will be conducted in two separate phases. First, the EMES software designers will provide a brief demonstration of the major capabilities of EMES using a predefined mission model. EMES will be executed on this mission model twice--once with the spacecraft in a normal mode of operation, and then with the spacecraft in a degraded mode of operation.

Then MSFC will be given the opportunity to execute EMES, creating a new mission model from the load and event libraries built into the system, or using new load or event definitions. The final software demonstration is subject to approval by MSFC.

End of Document