NAG8-717

MARSHALL GRANT

IN-63-CR

332835

p 272

The University of Alabama in Huntsville

Applications of Artificial Intelligence to Mission Planning

FINAL REPORT

for

Mission Analysis Division
Systems Analysis and Integration Laboratory
George C. Marshall Space Flight Center

by
Donnie R. Ford
Stephen A. Floyd
and John S. Rogers
The University of Alabama in Huntsville
Huntsville, AL 35899

Table of Contents

1.0 Introduction

The scheduling problem facing NASA MSFC Mission Planning is extremely difficult for several reasons. The most critical factor is the computational complexity involved in developing a schedule. The problem space is combinatorially explosive. The size of the search space is large along some dimensions and infinite along others. There can be infinite number of choices to assign activities, and a large number of choices of crew assignments to activities. Additionally, the goal of the scheduling process is to produce a "good" schedule. This is ill-specified and encounters a number of often conflicting requirements. These requirements can include efficient use of resources, no time or resource constraint violations, and maximum production during a specified time period. Interrelational requirements between activities, the performance placement of each of the activities, and resource usages can make constraint violations difficult to predict and avoid.

It is because of these and other difficulties that many of the conventional operation research techniques are not feasible or inadequate to solve the problems by themselves. Therefore, the purpose of this research is to examine various artificial intelligence techniques to assist these conventional techniques or replace them entirely.

In June 1988, the Mission Analysis Division of the Systems Analysis and Integration Laboratory of the Marshall Space Flight Center (MSFC) of NASA tasked UAH to study the mission planning activities and how artificial intelligence techniques may benefit these activities. The specific tasks to be performed were (1) identify mission planning applications for object-oriented programming and rule-based programming; (2) investigate interfacing AI dedicated hardware (Lisp machines) to VAX hardware; (3) demonstrate how Lisp may be called from within FORTRAN programs; (4) investigate and report on programming techniques used in some commercial AI shells, such as KEE; and (5) investigate and report on algorithmic methods to reduce complexity as related to AI techniques. The results of this study, the prototype computer

software and their [1]operational instructions were reported to NASA MSFC in the first Interim Report (UAH Research Report JRC 90-07) and presented in the form of an oral presentation in November 1989.

At the conclusion of this oral presentation and during subsequent meetings with the MSFC staff new goals were set for continuing research on the previously defined tasks. These new goals focused on two areas: software and technique. Specific modifications and enhancements to prototype resource allocation software have been incorporated to increase its functionality and performance capabilities. Coupled with the modified software, new Frontier of Feasibility traversing techniques have been developed and evaluated A description of each of the alterations and additions to the prototype software and differing techniques were detailed in the second Interim Report (UAH Research Report JRC 90-48) and were presented to MSFC personnel in the Summer of 1990.

The following is the Final Report for research conducted under NASA Grant NAG8-717. UAH would like to thank the NASA MSFC Mission Planning personnel for thier support and cooperation during the conduct of this research. The contents and conslucions is the sole responsibility of the authors and implies no official position on the part of the National Aeronautics and Space Administration.

2.0 Object-oriented Programming Task

2.1 Task Statement

The purpose of this research was to investigate some of the advantages and disadvantages of using an object-oriented paradigm to assist in solving the scheduling/resource allocation problem that is peculiar to MSFC NASA Mission Planning. This is further targeted to the Space Station effort. In order to assist in this task, the decision was made by UAH personnel to develop a demonstration prototype of the MSFC NASA experiment and payload scheduler using the object-oriented paradigm. This work was conducted by Dave Brown and Dr. Stephen Floyd.

2.2 Task Conditions

The conditions of this task are that the prototype was developed using a Symbolics 3600 machine, that the object-oriented paradigm (Flavors) that is presently supported by this platform was appropriate, and the experiment scheduling experience and data gained from the Spacelab missions was an appropriate starting point for this prototype. Also, this task excluded consideration of between experiment constraints, and focused on within experiment constraints (time and resources).

Because of the newness of the subject, it is appropriate to preface the following sections with a brief introduction to object-oriented programming. Object-oriented programming is becoming popular and important in many areas. This term implies that behavior is associated with objects, usually in the form of code. Thus, each object can possess particular knowledge needed to function in its world. Consequently, programs become a collection of objects

rather than lines of code. Other terms relating to objects are inheritance, message, methods, classes, and metaclasses. Definitions of these terms follow.

Class - a template from which objects are modeled or created. Objects are usually clustered based on behavior, thus a taxonomic relationship can be developed from this. Behavior can be attributed to an individual object or to the class of objects. Classes control the manner in which objects are structured.

Inheritance - the ability of an object to automatically share behavior between classes.

Message - the means by which an object may be requested to perform a certain behavior or action. This is the fundamental control mechanism and is the hallmark for object-oriented programmming.

Method - an actual implementation of a message

Metaclasses - the means for classifying objects and placing them in a hierarchy for inheritance purposes. Metaclasses control the manner in which objects in subclasses are represented.

All of these concepts are needed to have an object-oriented paradigm or programming language.

An object is composed of slots that hold the code and/or information that makes the object unique and a member of a particular class. What is in the slot is called the value of the slot. Thus, the structure of the object is the collection of slots that compose the object. Objects can inherit slots and/or values from the classes that are above it in the hierarchy of inheritance. Consequently, the terms parent and child are used when discussing inheritance. An object can have more than one parent. Also, an object or object class can have behavior that is not inherited.

When an object is created, it inherits its structure from its parents, and is referred to as an instance of a particular class. There are various inheritance mechanisms that control what exactly is passed to children. These can be simple or very sophisticated. These mechanisms are located at various levels of the inheritance hierarchy.

The advantages of using object-oriented programming are varied, but the most cited are (1) information hiding, (2) reuseability of code, (3) restricted visibility, and (4) ease of adding program functionality. Some of the disadvantages are (1) size of the program, (2) no standard language, and (3) training in object-oriented programming.

2.3 Task Approach

The approach taken in this task was to develop a demonstration prototype to test the desirability of object-oriented programming for the scheduling problem. This prototype was developed to handle a subset of the Mission Planning scheduling problem and used experiment data from the Spacelab project. Everything involved in the scheduling process that was modeled in this prototype was represented as objects. The following are the items treated as objects in the prototype:

Resources (durable, consumable, non-depletable)

Crew members

Targets - locations on earth

Attitudes - the orientation of the space vehicle with respect to the earth

Experiments - this includes the general characteristics of an experiment and not specific characteristics of individual experiments

Performance - one complete iteration of an experiment

Step - one operation of a performance. These were divided into startup, normal, and shutdown

timeline - divided into seconds

Other bookkeepping items and the interface for the program were also handled as objects, thus, the program is completely object-oriented.

An interactive resource editor and display mechanism was designed and partially implemented. Currently, the editor handles crew, target, attitude, consumable, and durable resources. The editor allows new resources in these categories to be defined, as well as existing resources to be modified. This includes items, such as quantity available or time period available.

An interactive experiment/performance/step editor has been partially designed. Major work still needs to be done in this area, as most of the functions are stubs.

The heart of the scheduling mechanism has been designed and implemented, but not thoroughly tested. A larger test set of data should be used for a more rigorous test. The data used to test the prototype was some small subsets of Spacelab experiment data. This included eighteen experiments with their associated characteristics.

At present the prototype has the ability to schedule experiment data that has been manually entered into a file structure on the Symbolics. Also, the prototype uses the "front-end loading" scheduling strategy. This means that the first available time that an experiment can be scheduled is used immediately and no other locations are determined as suitable.

Scheduling with respect to this prototype consists of the following steps: (1) selection of an experiment to be scheduled, (2) selection of a time period to begin the first step of a performance of the experiment, (3) determination of start

time for each step, and (4) step scheduling. Determination of the start time for a step consists of an examination of each step, and determination of the earliest and latest start time of the next step. Each step must be examined in order to determine whether the performance can be scheduled at the time period specified. The determination of the start time for the next step is based on duration and delay factors. The mechanism for doing this is essentially a depth first search with backtracking. When a feasible set of times has been identified that satisfies all resource constraints and time constraints for the step, then step scheduling is entered. At this time, resources are decreased, and linkages to objects representing the time periods are made. The portion dealing with the depth-first search with backtracking has been partially implemented but not sufficiently tested.

The prototype should have the ability to automatically schedule the desired number of performances for each experiment, resources permitting, according to several schemes. The user should control which scheme is actually used. This concept was demonstrated in the earlier version; however, these schemes have not been implemented in the latest version of the prototype.

At present, the prototype does not allow for any interaction with the user during the scheduling process. Ideally, interactive scheduling is a desired and necessary feature for the scheduling process. However, the user does have the ability to select an experiment and a time period and attempt to schedule a performance of the experiment to start in the selected time period after a schedule has been generated. Also, the user is allowed to specify a time period, and nominate a list of performances which can be scheduled to start during that time period. The prototype also allows the user to specify an experiment, and nominate a list of time periods in which a performance of that

experiment can be started. In all cases, determination of startup and shutdown steps is accomplished with consideration being given to all other constraints.

Other desirable features for future prototypes that have begun being developed are data entry, automatic scheduling, interactive scheduling, and data output (hardcopy, file, and display). Data Entry will include mechanisms for interactively entering all types of data required , as well as mechanisms to read the data from files. To some extent, yet to be determined, the user will be able to control which data elements are to be interactively entered and which are to be read from files. Currently, input data is thought to consist of experiments, together with their steps, to include startup and teardown steps; resources, with available quantities and time periods (as appropriate); and other mission control data, such as mission duration, desired level of time resolution.

Resources include crew members, targets, attitudes (of the platform), durable goods (those items are available in fixed quantities throughout the mission and are not expended by use), consumable items (those items available initially in some fixed quantity, and which are expended by use, such as quantities of chemicals), and non-depletable items (those items which are generated aboard the platform at some rate, and which may or may not be able to be stockpiled for later use, such as electricity from fuel cells). Resource objects capture how much of each resource is available during each time period (defaulted to 1 per period for each crew member, target and attitude). Non-depletable goods object has not been designed yet.

Experiments are to be represented as a series of steps. Steps are of three varieties -- normal, startup, and shutdown. A performance is an execution of the ordered set of normal steps. The startup steps will be conducted before the performance which occurs first, and the shutdown steps will be executed

after the performance which is conducted last. Note that these are not the same as the first performance scheduled and the last performance scheduled. The automatic scheduling and un-scheduling of startup and shutdown steps is necessary to facilitate interactive scheduling. Currently, an experiment has the following attributes; a name, minimum number of performances to be performed, maximum number of performances to be performed, desired number of performances to be performed (to be used in automatic scheduling), the experiment window (time between start of first step, earliest performance and end of last step of latest performance), and minimum and maximum delay times between performances. Performances include a performance window (similar to experiment window, but dealing with normal steps only). Steps include a maximum and minimum duration, a maximum and minimum delay until next step, and lists of resources required. Additionally, steps include a flag for crew lock-in (that is, when a crew member(s) has been selected to perform a specific step of one performance, that same crew member(s) must perform the same step of all other performances of the experiment). The step also includes the ability to specify subsets of the crew from which members must be selected (independent of crew lock-in). It is recognized that the step must have the ability to be scheduled with respect to some other step of another experiment, but the capture mechanism for this data has not been determined.

Automatic scheduling involves the selection of different strategies and being able to schedule from user specified files. Interacitve scheduling involves adding the ability to interact with the prototype during the actual scheduling of experiments. Finally, data output is the ability to generate various forms of the schedule for the user. This includes hardcopy, file storage, and display. A mechanism to save the input beyond the working session still must be developed. This will not be accomplished until the mechanisms for reading in

data files are completed, as it is intended that the output will have the same format as the input to simplify data loading.

2.4 Task Results

There have been two versions of the prototype scheduling system developed. The latest version has more functioality than the first. The development of these two versions have served to highlight one of the disadvantages of object-oriented programming; that is, that the size of the program becomes extremely large during execution. In treating everything as an object, there is no way to know with any certainty how large the program will become. The main problem in this area stems from the way that the timeline is handled. The timeline was broken down into seconds with each second becoming an object. One can readily see that it does not take a very long time span to cause an enormous number of objects to be created. An associated problem with this is that during the bookkeepping process each time interval must be checked for resources available and other updating functions. Another method of handling the timeline must be developed.

On the other hand, treating the experiments as objects has much potential as a solution to the scheduling problem. More work should be done to determine the appropriate level of grandularity for these objects. That is, should just the experiments be objects or should each step be an object?

3.0 Rule-Based Programming Task

3.1 Task Statement

The purpose of this research project was to develop a research prototype of a system to schedule an experiment payload using the Space Station as a target. The problem used was a very small subset of the payloads for Spacelab. Also, the prototype deals with only two resources. An indirect objective of this research was to study the feasibility of using Knowledge Engineering Environment (KEE) to develop and implement a small prototype scheduler. This work was conducted by Dr. Fan Tseng and Dr. Rajeesh Tyagi.

3.2 Task Conditions

The prototype was built on Symbolics 3620 using Knowledge Engineering Environment (KEE) version 2. Symbolics 3620 is a Lisp machine marketed by Symbolics Incorporated, Cambridge, Massachusetts, and KEE is a commercial knowledge-based system development tool marketed by Intellicorp Incorporated.

3.3 Task Approach

KEE is a set of software tools designed to assist system developers in building their own knowledge-based systems. The main features of KEE include: frames for the representation of knowledge, a rule system for rule-based reasoning, graphics for user interface, and object-oriented programming.

Frame-based representation is a means of representing objects and their attributes. A frame includes all the knowledge about a particular object, stored and organized in a pre-defined manner. The frame is composed of slots (or

fields) that contain specific information relevant to the frame (or object). For example, the frame for a generic experiment may contain four slots as follows:

SLOT	VALUE
Agency	NASA
Duration	20 hours
Power	1200 kilowatts
Runs	1

The prototype scheduler is comprised of three components as shown in Figure 1. The components are: a knowledge base, a model base, and a user interface. The knowledge base possesses information on various experiments and their attributes (like the time needed to run an experiment and peak power consumption during the run). It also contains information on availability of resources needed to run the experiments (like power supply). The model base contains a set of scheduling rules that may be used to develop a schedule for the experiments. And the user interface provides the dialog between the user and the system.

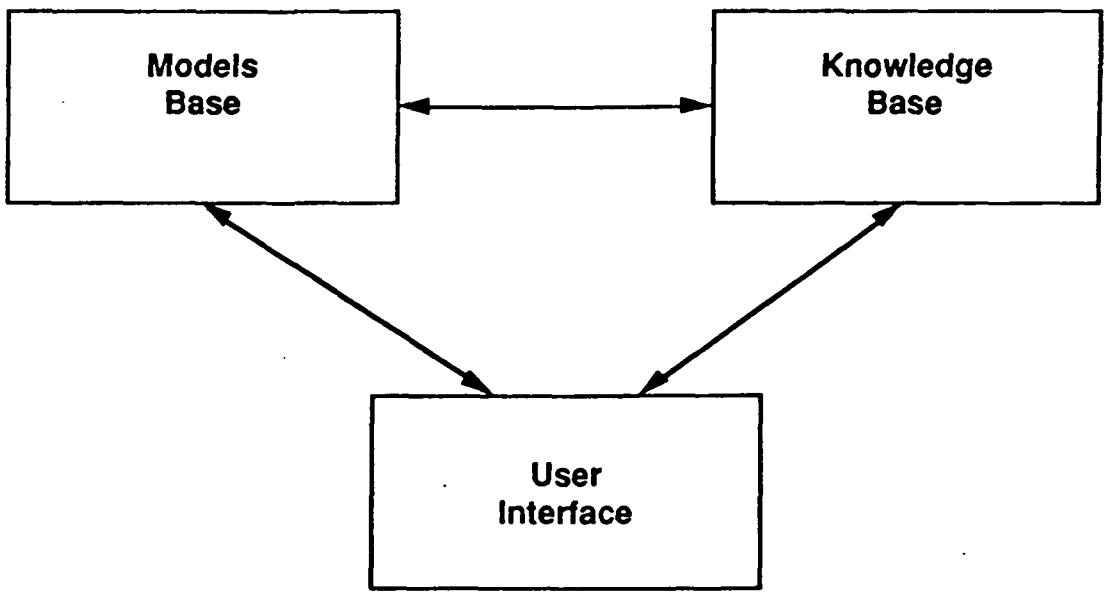


Figure 1. The basic structure of the prototype.

Given the time constraints to complete this study, there wasn't sufficient time to develop a prototype with all the capabilities one would have desired. Since the focus of this research was on the suitability of using KEE for developing the scheduler, it was decided to include only a set of basic features that would be sufficient to allow a comprehensive evaluation of KEE's ability to integrate all the three components mentioned above. Therefore, the knowledge base contains information on only ten experiment and two resources. And a set of four scheduling rules constitute the model base.

The knowledge base is organized in the form of frames. Each experiment is represented by a frame. Each frame consists of slots corresponding to the attributes of the experiment. Figure 2 shows the frame corresponding to an experiment called "Crystal Growth". The experiment is

Frame for Experiment: "Crystal Growth"

SLOT	VALUE
Agency	NASA
Duration	20 hours
Power	1200 kilowatts
Runs	1
Starting Time	--
Ending Time	--

FIGURE 2. An example data structure for the prototype.

sponsored by NASA and is to be run only once. the experiment run requires a power supply of 1200 kilowatts over 20 hours, the duration of the experiment. The starting and ending times for the experiment are to be determined by the scheduling criterion selected by the user to generate the schedule, and are automatically placed in their respective slots. In addition to frames for the experiments, there are two frames for the two resources considered in the prototype, namely, mission length and power supply.

The Model Base contains a set of scheduling strategies that may be used to generate a schedule, based on the objectives and/or requirements of the user. These strategies are: (1) Decreasing Run Time, (2) Increasing Run Time, (3) Decreasing Power Usage, and (4) Increasing Power Usage. These rules have been implemented in the form of Lisp functions which are executed from KEE.

The user interface provides the dialog between the user and the scheduler in the form of windows, menus, and graphical displays. The user controls the execution of the system by specifying the strategy to be used in generating a schedule. The user may also perform what-if analyses. This analysis may use any of the other scheduling rules to provide alternate schedules. It may also be used to evaluate the effects of changing experiment parameters; e.g., varying the duration of an experiment. Any schedule generated will result in starting and ending times for the experiments being placed in their respective slots. It also produces a chart displaying any unused power.

3.4 Task Results

KEE allows for knowledge bases to be created fairly easily using the frames representation. It also displays a pictorial representation of the knowledge base.

Lisp functions can be executed from within the KEE environment. This feature was used to implement the scheduling rules of the prototype. It was observed, however, that KEE was relatively slow to execute any user-written Lisp code.

The user's manuals were very hard to follow for someone using KEE for the first time. No complete example is worked out in the manual, which makes it difficult to get started for a beginner. Unfortunately, for the KEE installed at UAH, none of the demos provided completely worked.

Toward the end, when the prototype was close to completion, a new version of KEE was installed; however, it wasn't fully compatible with the old version and the prototype would not work on it. The people who worked on building this prototype had an extensive software development background, though not specifically with Lisp or KEE. Their experiences with KEE indicate that for someone with such a background, it is not easy to develop a proficiency in using KEE in a short period of time. The knowledge base can be constructed rather easily using KEE. Building a scheduler, however, would necessitate strong programming skills in Lisp since all the scheduling algorithms and the Gantt charts would have to be implemented by the developer in Lisp. When selecting a software tool, one must consider the portability of the software tool, both in terms of transferability to a different hardware system, as well as in terms of conversion to another software system. While it may not be possible to transfer and re-compile Lisp code developed on KEE onto a different hardware/software system, the same cannot be said of the knowledge base developed on KEE. To restate a point mentioned in an earlier section, it was found that execution of Lisp functions in KEE environment is appreciably slower than in operating system environment. The knowledge base developed for the scheduler prototype comprised only a small number of experiments. The response time of the prototype of KEE was not impressive at all. We believe that if the knowledge base were to be expanded to include a more realistic set of experiments, the performance of the prototype would deteriorate even further. In light of the above conclusions, it is recommended that a comprehensive

system like the scheduler not be developed using a commercial expert system tool. Instead, given the current state of the art technology regarding Lisp-based machines, it would be prudent to develop a mainly Lisp-based system. Such a system would be significantly more portable.

4.0 Algorithms for Resource Allocation

4.1 Task Statement

The purpose of this research was to study the feasibility of using an algorithmic approach to provide a solution to the resource allocation problem. The solution to this problem would become the starting point for an experiment scheduler. This primary purpose of the resource allocation problem is to speed up the development of good schedules for the NASA MSFC mission planning process. Also, another purpose is to provide the capability of rapidly evaluating alternative schedules.

4.2 Task Conditions

The conditions of this task were intentionally left open ended. The main constraint was that data from the Spacelab missions be used for testing and developing the algorithms. This data was not actual data but was representative of the types that would need to be handled by the algorithms. The problem size was kept small for development and testing purposes. The other consideration was that performance of the algorithm on the computer should be sufficient to handle an expanded data set. Finally, the Symbolics lisp machine was used to develop the prototype programs and Common Lisp was not strictly utilized.

4.3 Task Approach

There are many subtle differences between scheduling and resource allocation; however, the main difference is basically granularity. Scheduling is more detailed and strictly adheres to any resource or mission constraints than does resource allocation. Resource allocation considers constraints in an

aggregate manner, that is, the area under a curve. The objective of these resource allocation algorithms is to maximize the usage of the area under the curve only. Other relationships and constraints are ignored in this process but are handled by the scheduler.

The algorithms were developed by MSFC Mission Planning personnel or by UAH personnel after consultation with the Mission Planning personnel. There are two that are discussed in this report. These are the Free Expansion Algorithm and the Multiple Pass Algorithm.

The Free Expansion Algorithm was initiated by Mr. James Lindberg of MSFC. It is basically a controlled expansion of a tree where each node represents a combination of experiments. The objective is to find the "best" combination without exceeding the amount of resource available.

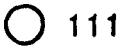
This algorithm requires that a starting point be provided. The first step was to determine the feasibility of the starting point. If the starting point is feasible, then the algorithm is as follows:

(1) Add starting point to feasible solutions
(2) Expand the starting point
(3) Is the point feasible?
    Yes, continue.
    No, prune this branch and choose another point.
(4) Add point to feasible solutions
(5) Expand point
(6) Repeat steps 4 and 5 until all branches are pruned.
(7) Repeat steps 4, 5, and 6 until all branches are pruned.
(8) Stop when the tree is exhausted.

This is the general algorithm; however, some points need to be explained. One is how a point is expanded.

Point expansion is best explained using a simplistic example with illustrations. Assume that there are three experiments and each experiment can

have a maximum of four performances during the mission. Also, assume that the starting point is one performance of each experiment. This can be represented as (111). This makes the graphical illustration easier to use. Thus, the root of the tree is (111) or graphically



To expand this point, certain rules apply. Each child can only have one performance level changed. Also, subsequent children can only change the performance level that was changed to generate them or any successive performance level. Figure 1 illustrates the fist rule using the assumed root node. Here each performance level is changed to create three new nodes or children. This is also referred to as a generation or level when considered in aggregate. This is fairly simple and straight forward; however, the second rule is not as apparent.

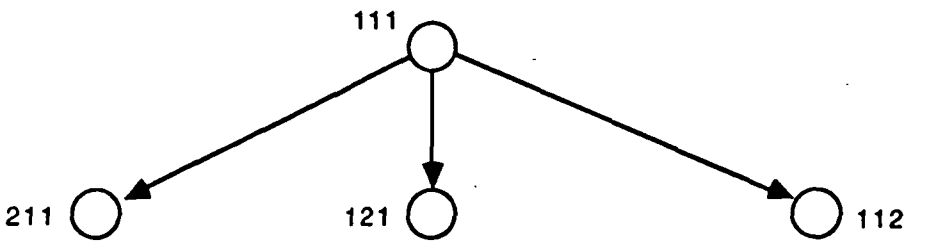


Figure 1. Rule 1 of expansion of a point.

This is illustrated in Figure 2. Here a portion of the tree in Figure 1 is used to illustrate the second rule. The first child (211) of the starting point is used and expanded. The expansion produces three children. Because this point was created by changing the first performance level, all the performance levels can be changed to create children. If the second child is considered,

then only the second and third performance levels can be changed. Thus, the further right a node is in the tree, the less children it can have. Or stated another way, the majority of the children will occur in the left-most branch of the tree. Figure 3 illustrates this point very well.

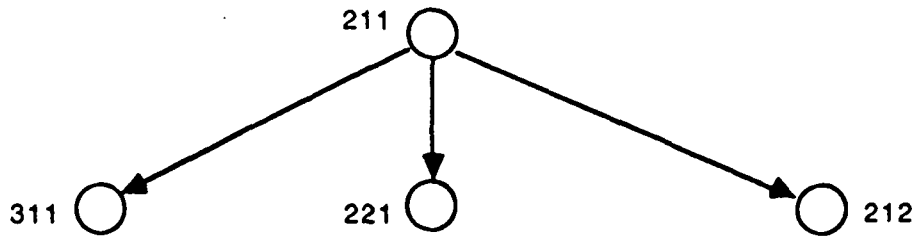


Figure 2. Rule 2 of expansion of a point.

Using rules one and two will generate a very neat and orderly tree. This allows the tree to be searched in an orderly fashion for the points of infeasibility. When all branches are searched and each point of infeasibility is established, then the frontier of feasibility is established. This is important for the decision maker when alternative solutions are a requirement.

The final rule for expansion is that the performance level can be changed by only one performance at a time. This is not as important as the other two rules and it has been found that it may be better to relax this rule at times. More research needs to be done in this area.

Using this algorithm, a model was developed and functions were written on paper; however, none of these were encoded nor tested on a computer. It is believed that this algorithm has some potential, but it was determined that other algorithms may be more appropriate. The reason for this is that this algorithm will conduct an exhaustive search of the tree. This is an unacceptable process due to the amount of time required to search a tree that represents a realistic

data set. Thus, work was stopped on this algorithm and a new algorithm was developed.

The new algorithm was also initiated by Mr. James Lindberg of MSFC and is called the Multiple Pass Algorithm. The first pass is made with the objective being to allocate resources to the minimum number of performances required for each experiment. The second pass is made to fill-in any empty spaces with extra performances of the experiments.

This algorithm requires that the minimum number of performances for each experiment be provided with the data set. Also, the time per performance, the power required, and mission duration are given. From this information, a prioritized list based on power required is generated. The list is in descending order of power required. The algorithm is as follows:

(1) remove the first experiment from the list.

(2) allocate the resource to this experiment beginning at time zero.

If amount available is $\geq$ amount needed, continue.

If amount available is $<$ amount needed, go to (5).

(3) create a new time interval using the duration of the experiment.

(4) update the amount of resource available.

If resource available at this point is zero, then go to (5),

If resource available is greater than zero, then go to (1).

(5) Move to next available time interval.

(6) Repeat steps 1 - 4 until list is exhausted.

The objective of this algorithm is to maximize the resource usage at all the time intervals. Once the first pass is completed, all the experiments are placed back on the experiment list and each time interval is searched for unused resource. At each time interval that has resource available, the experiment list is checked to find an experiment that can fit in this interval.

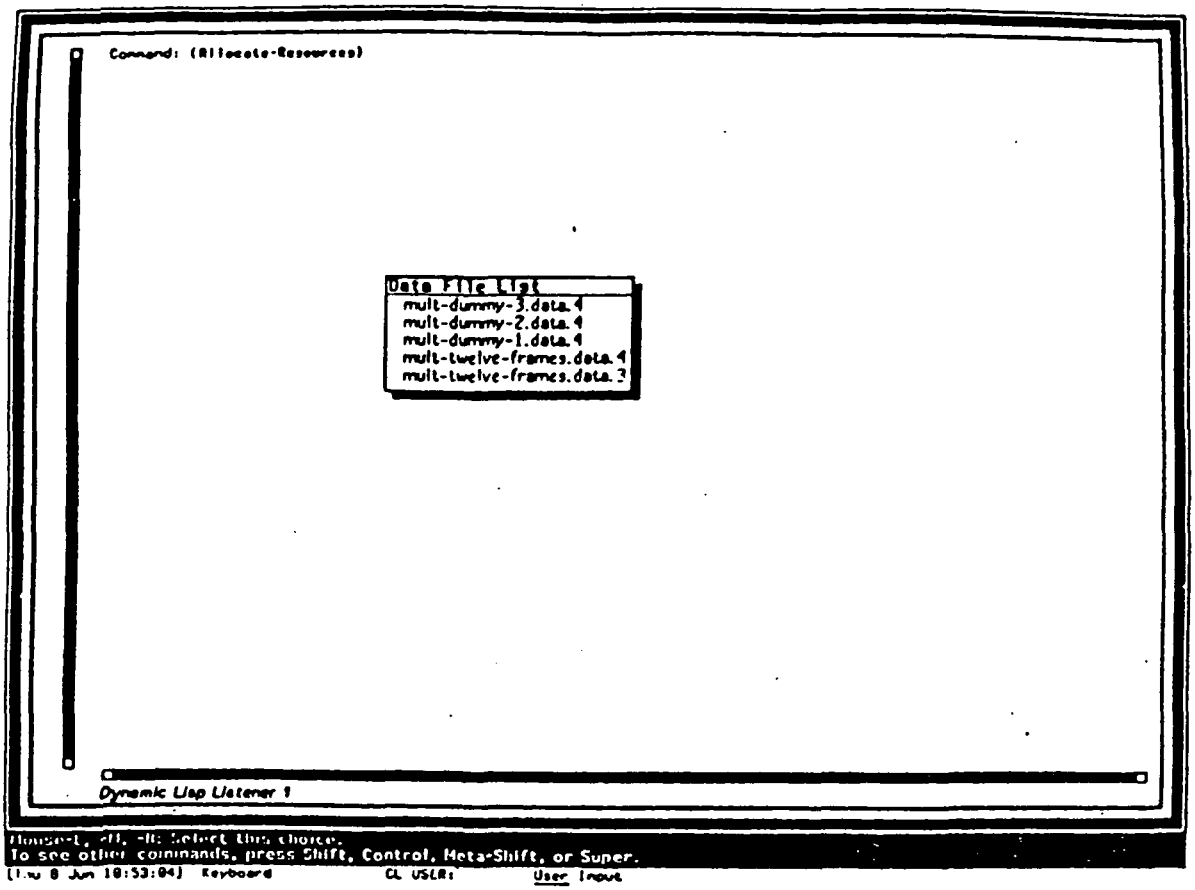
Multiple performances of an experiment can be allocated; however, single performances of multiple experiments are preferred.

The best graphical representation for this algorithm is a Gantt chart. The best representation of this algorithm on the computer is an association list. There are two versions of this algorithm: (1) the Mulitple Pass-Single Resource, and (2) the Multiple Pass-Multiple Resource. Both of these algorithms were implemented on the Symbolics machine using Common Lisp. Also, the Multiple Pass-Single Resource algorithm was transported to VAX Common Lisp. See Appendix B the Symbolics code listing of the Mulitple Pass-Multiple Resource Algorithm, Appendix C for a VAX code listing of the Multiple Pass-Single Resource Algorithm, and Appendix D for a Symbolics code listing of the Multiple Pass-Single Resource Algorithm.

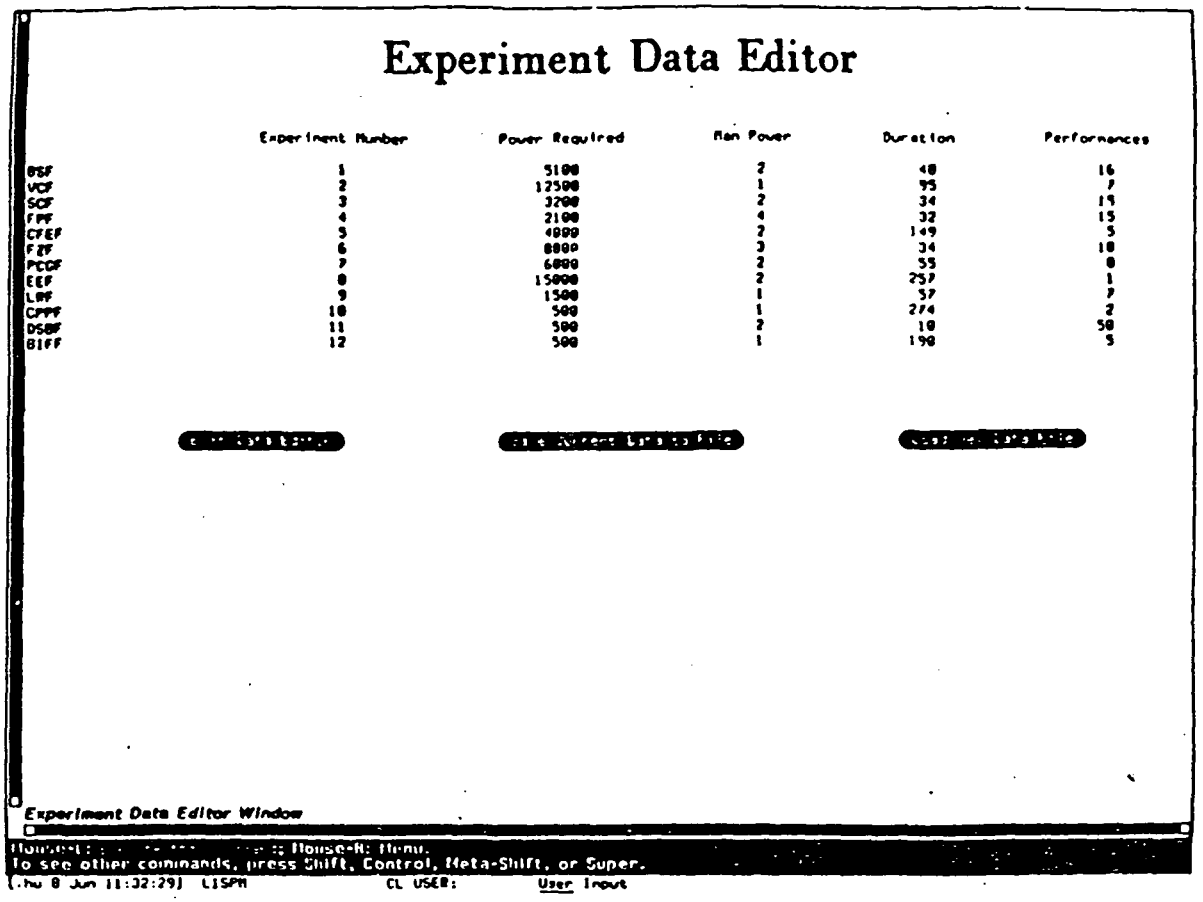
4.4 Task Results

The results of testing the Multiple Pass-Single Resource program for the Symbolics machine are presented in Table 1. The test began with a set of 18 experiments and the set was increased each time by six until 42 was reached. After this run, a set of 50 experiments was used. The execution times are expressed in seconds. Also, five replications were made for the set of 18 and 24 experiments only. The other sets had only two replications. This was due to the amount of time required for the larger sets. Finally, a graph showing the average execution time for each experiment set is included in Table 2.

The system developed on the Symbolics was tested extensively to ensure that the coded algorithm performed as intended. A sample session with the resource allocation program follows.



The Resource Allocation Program is initiated by typing the command (Allocate-Resources). At the start of the program, a menu will appear displaying the available data files for the program. The user may select the appropriate data file by simply placing the mouse on the file name and clicking. The menu will then disappear and the data will be displayed in the Experiment Data Editor as shown on the following page.

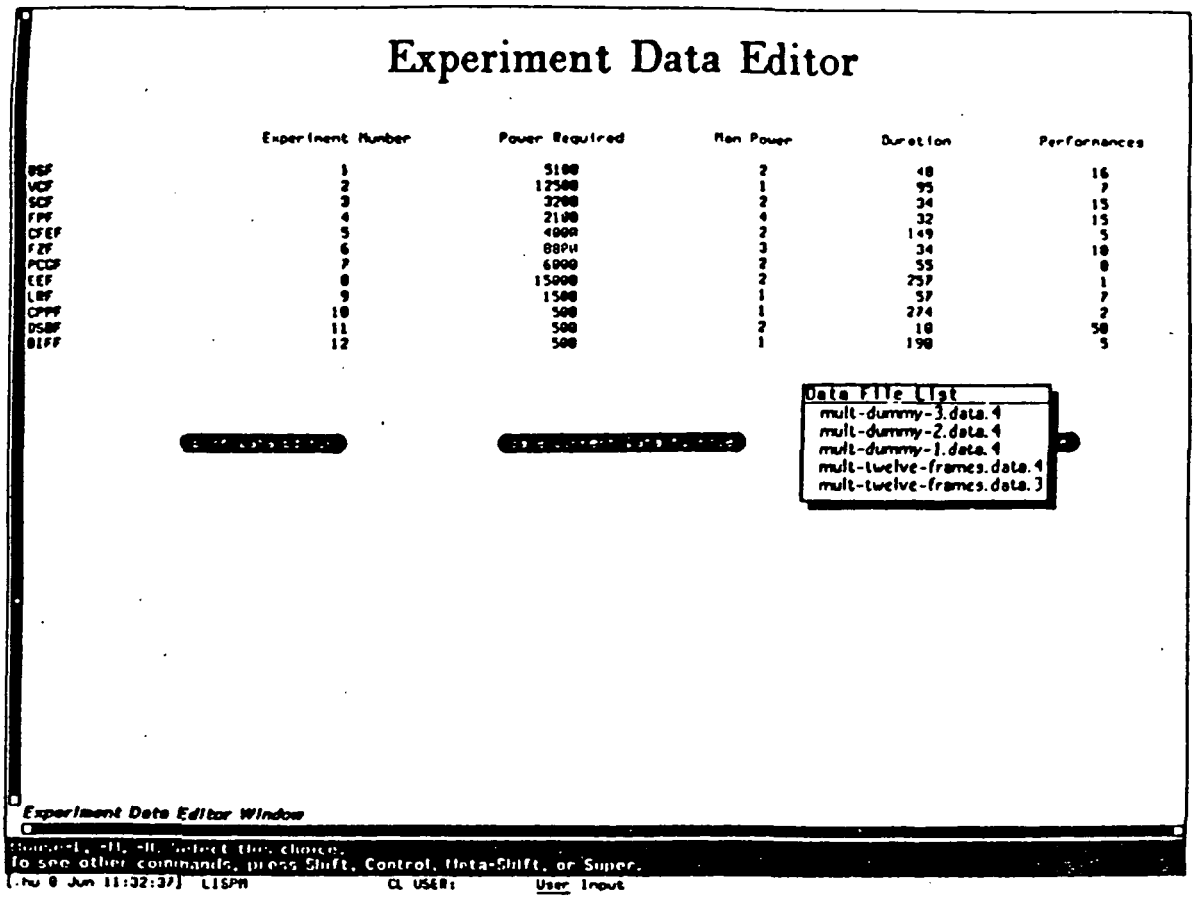


This is the experiment data editor window. Everything displayed on the screen except for the title, Experiment Data Editor, is mouse-sensitive. The columns represent resources, and the rows represent experiments. Some of the menu operations include: Load New Datafiles, Save Current Data to File and Exit Data Editor. Descriptions of the three mouse sensitive buttons are found on the next page.

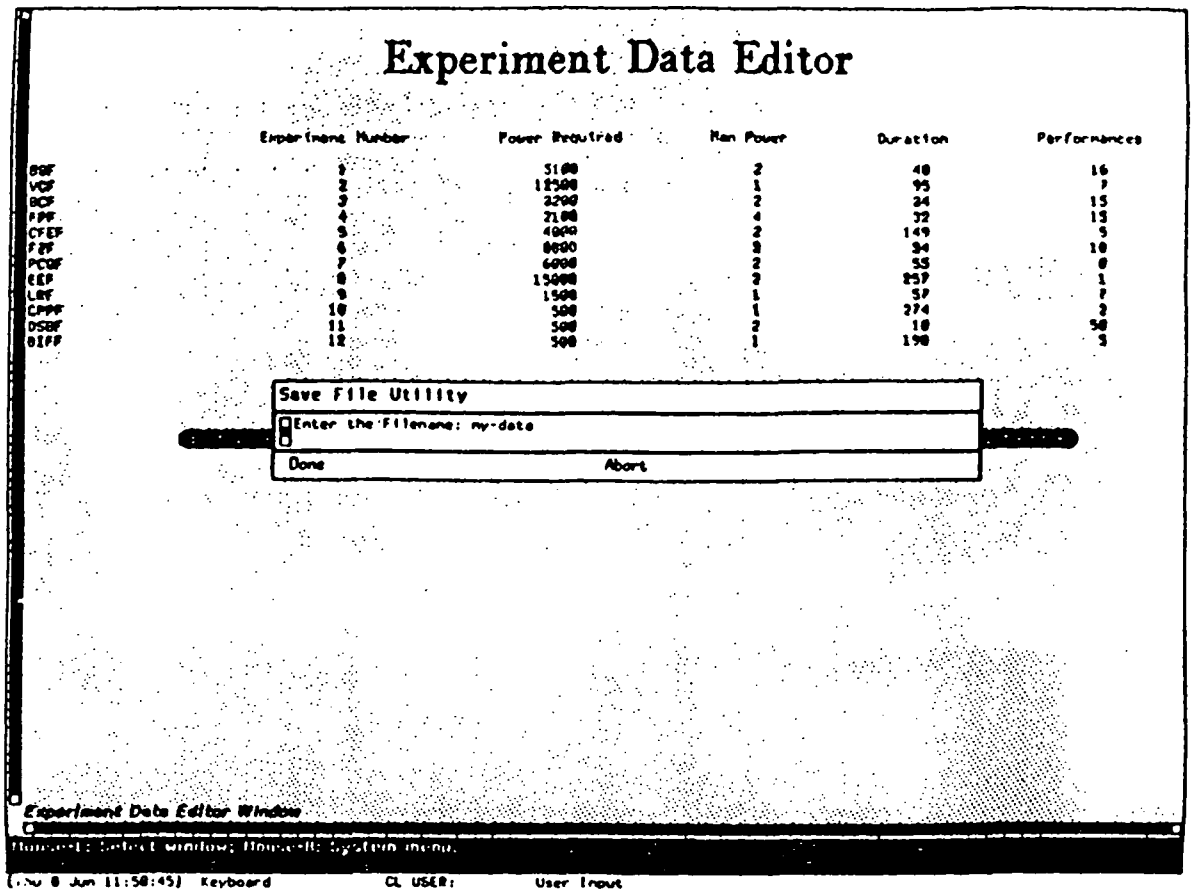
MENU OPERATIONS

- LOAD NEW DATAFILES
- SAVE CURRENT DATA TO FILE
- EXIT DATA EDITOR

The Load New Data File button enables you to load a new data file into the experiment data editor window, overwriting the datafile currently on display. The Save Current Data to File allows the user to save the data currently displayed in the window to disk. The Exit Data Editor leaves the data editor, and initiates the allocation process. The next page shows what happens when Load New Data File is clicked.



Clicking on the Load New Data File Button causes this screen to appear. A menu of the data files in the datafile directory is presented. A new data file to be edited can be selected by clicking on the file name. If we were to click on the Save Current Data to File button, the screen shown on the next page would appear.

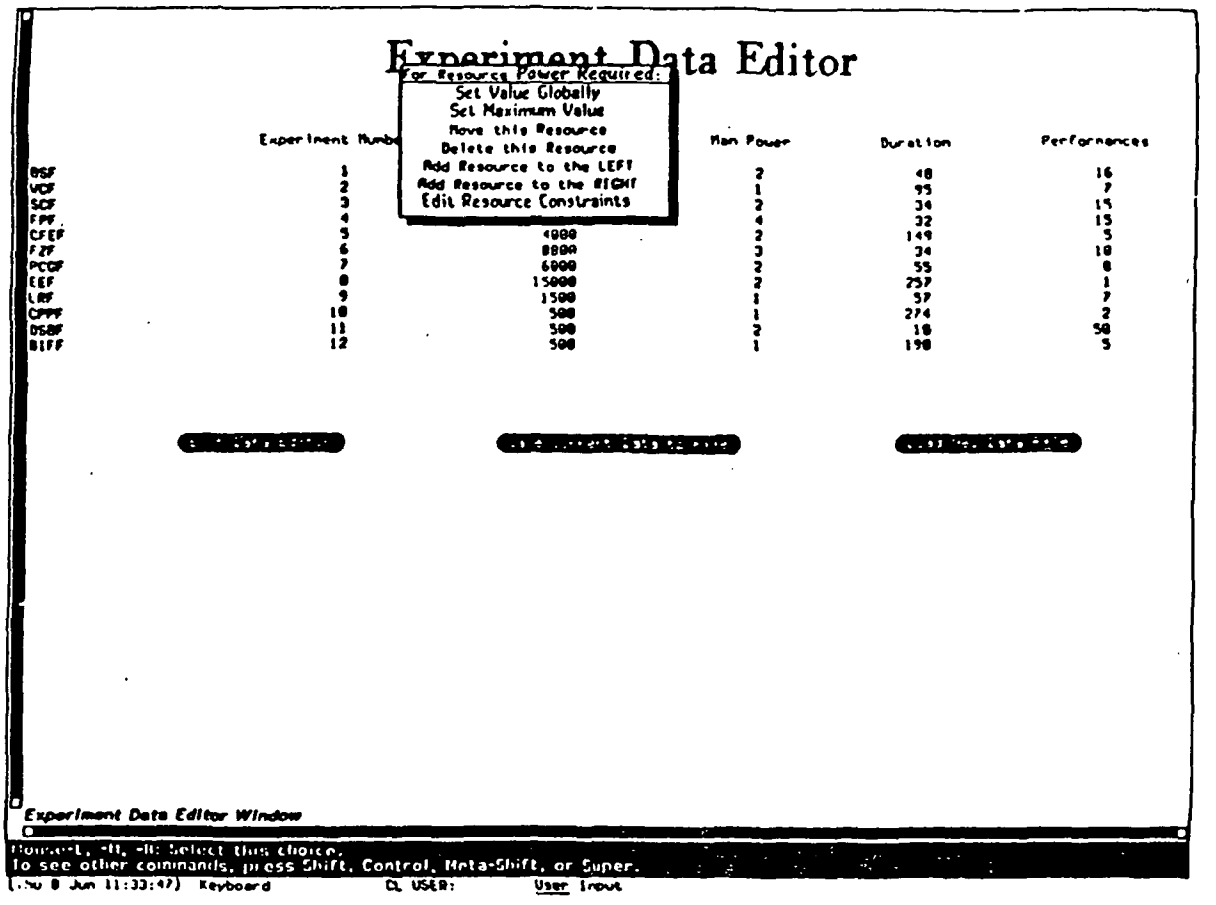


Clicking on the Save Current Data to File button, presents a window in which the filename the data to be saved on is entered. In order to save the file, type the filename, press return, and the click on Done. Clicking on abort will return program operation to the experiment data editor window without saving the file.

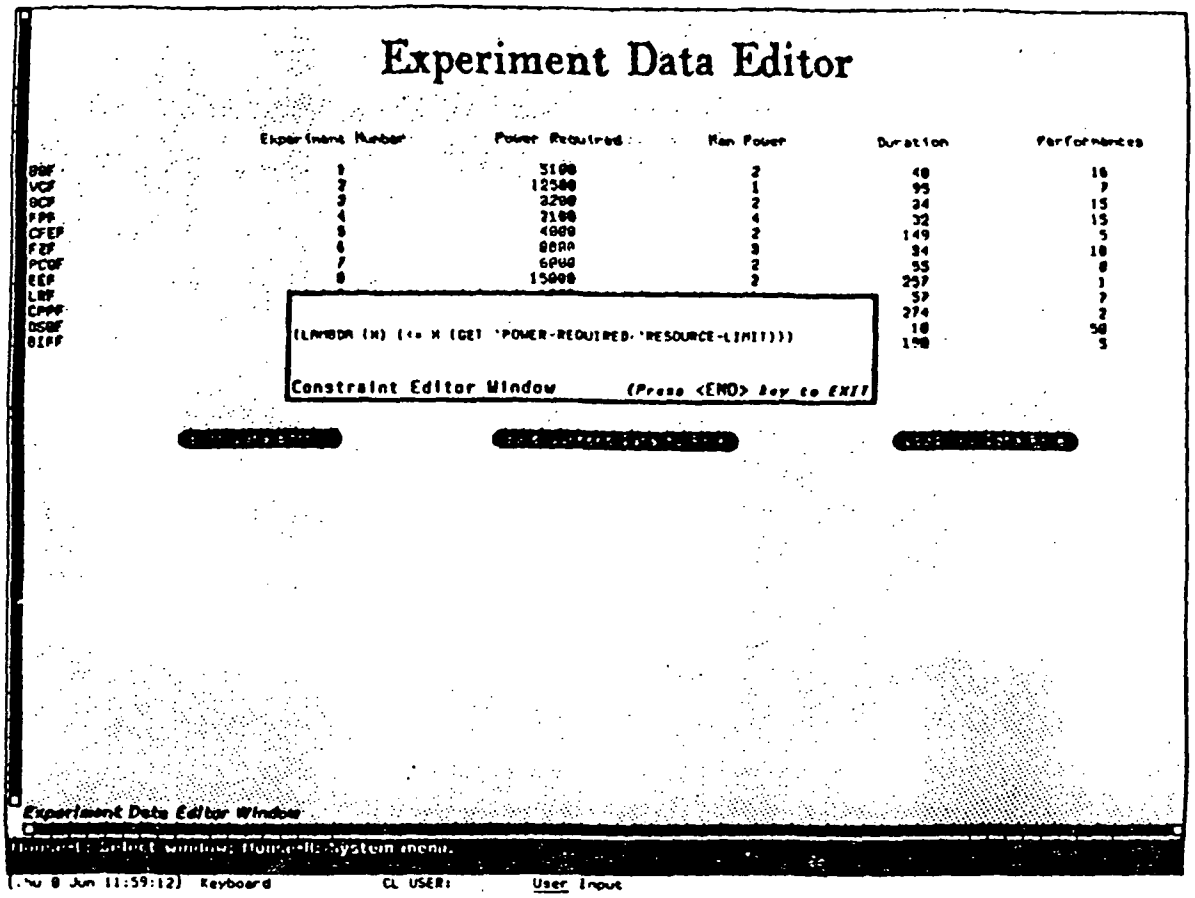
RESOURCE OPERATIONS

- **SET VALUE GLOBALLY**
- **SET MAXIMUM VALUE**
- **MOVE THIS RESOURCE**
- **DELETE THIS RESOURCE**
- **ADD RESOURCE**
- **EDIT RESOURCE CONSTRUCTS**

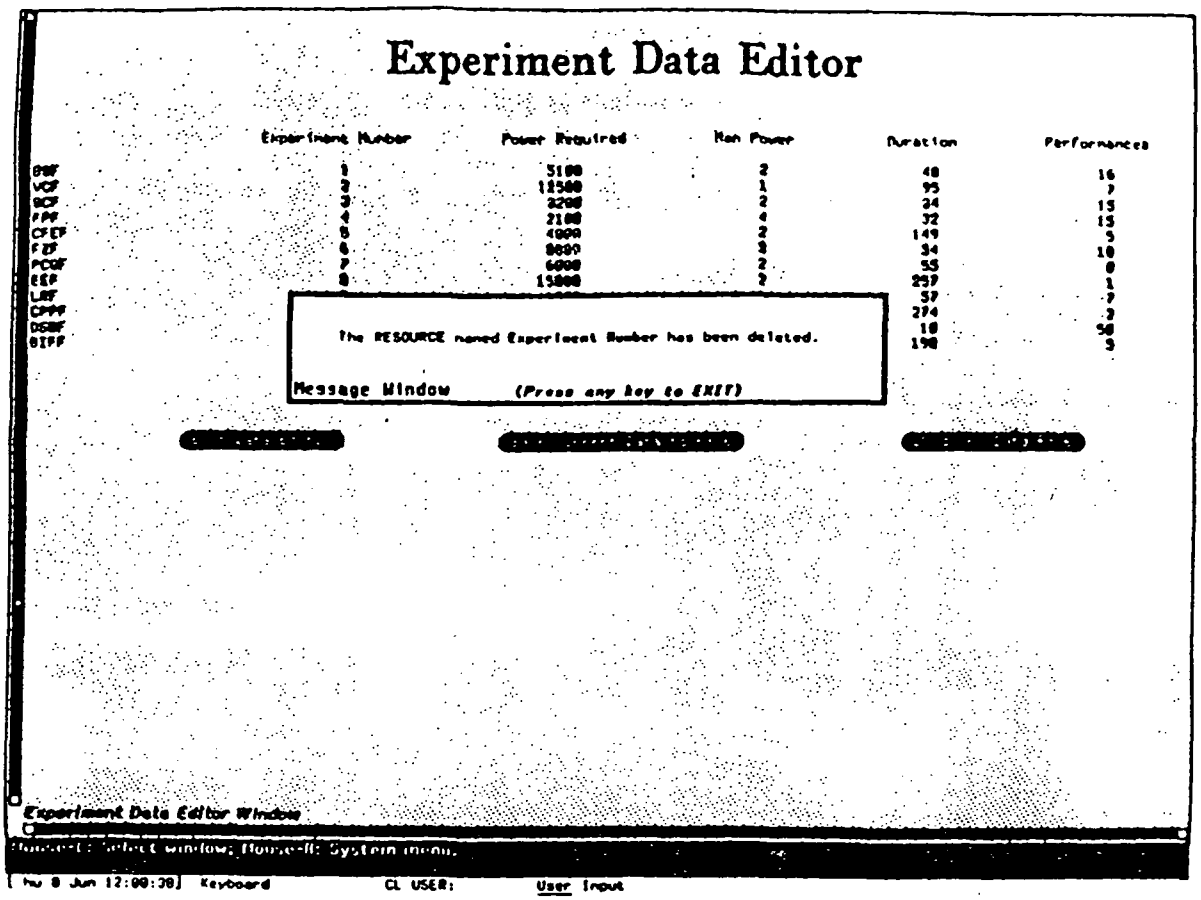
The columns of the experiment data editor window each represent a resource. Clicking on a column title will present the resource operations menu. There are six operations that can be performed on a resource. The first operation is Set Value Globally. This sets the selected resource to a global value in every experiment. The second operation is Set Maximum Value. This places an upper bound on the value a resource can take. Move This Resource allows the position of a column to be changed. Delete This Resource removes a resource from the experiment data editor window. Add a Resource can add a new resource to the data file, either to the right or to the left of a selected resource. Edit Resource Constraint edits the constraining function of a resource.



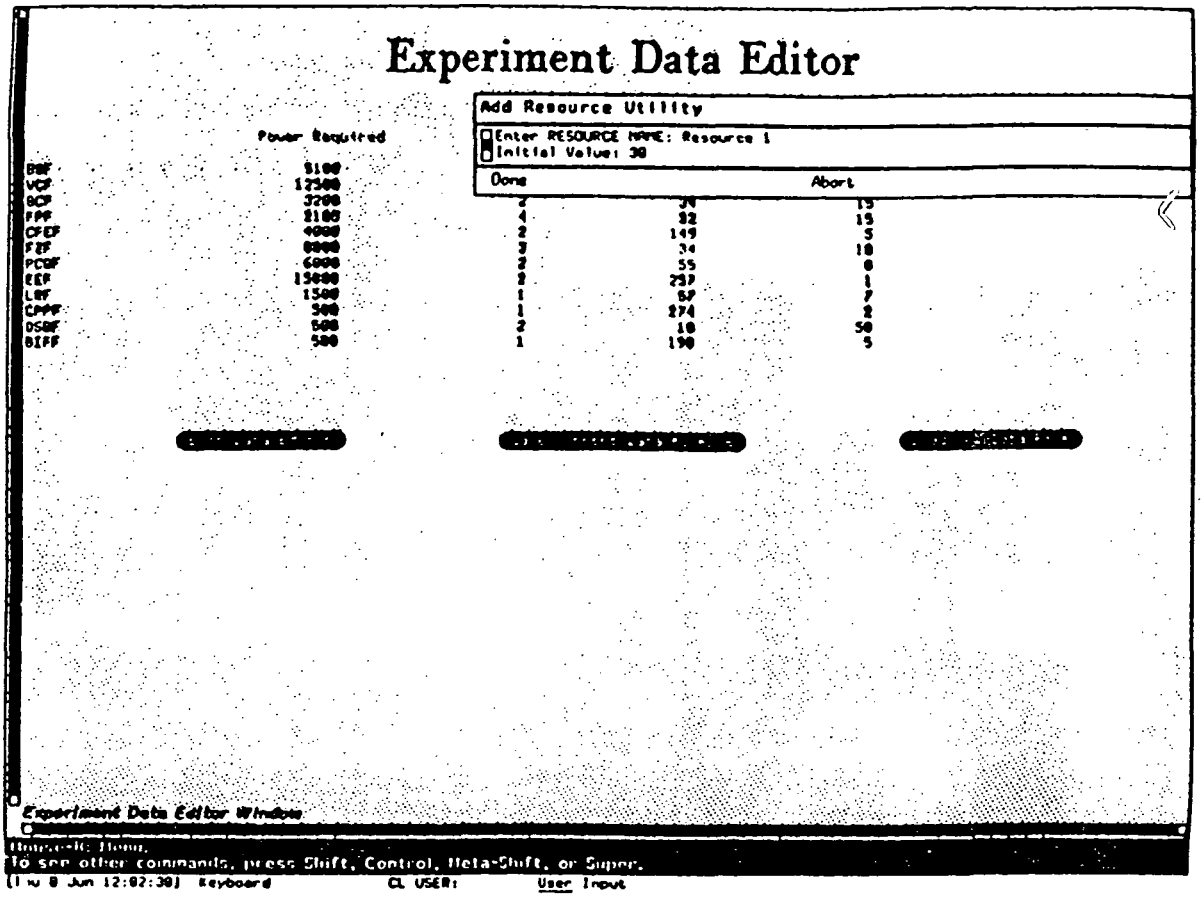
If the user were to click on the column title Power Required, a menu of operations that can be performed on this resource would appear. If the Edit Resource Constraint menu option had been selected, the screen on the following page would appear.



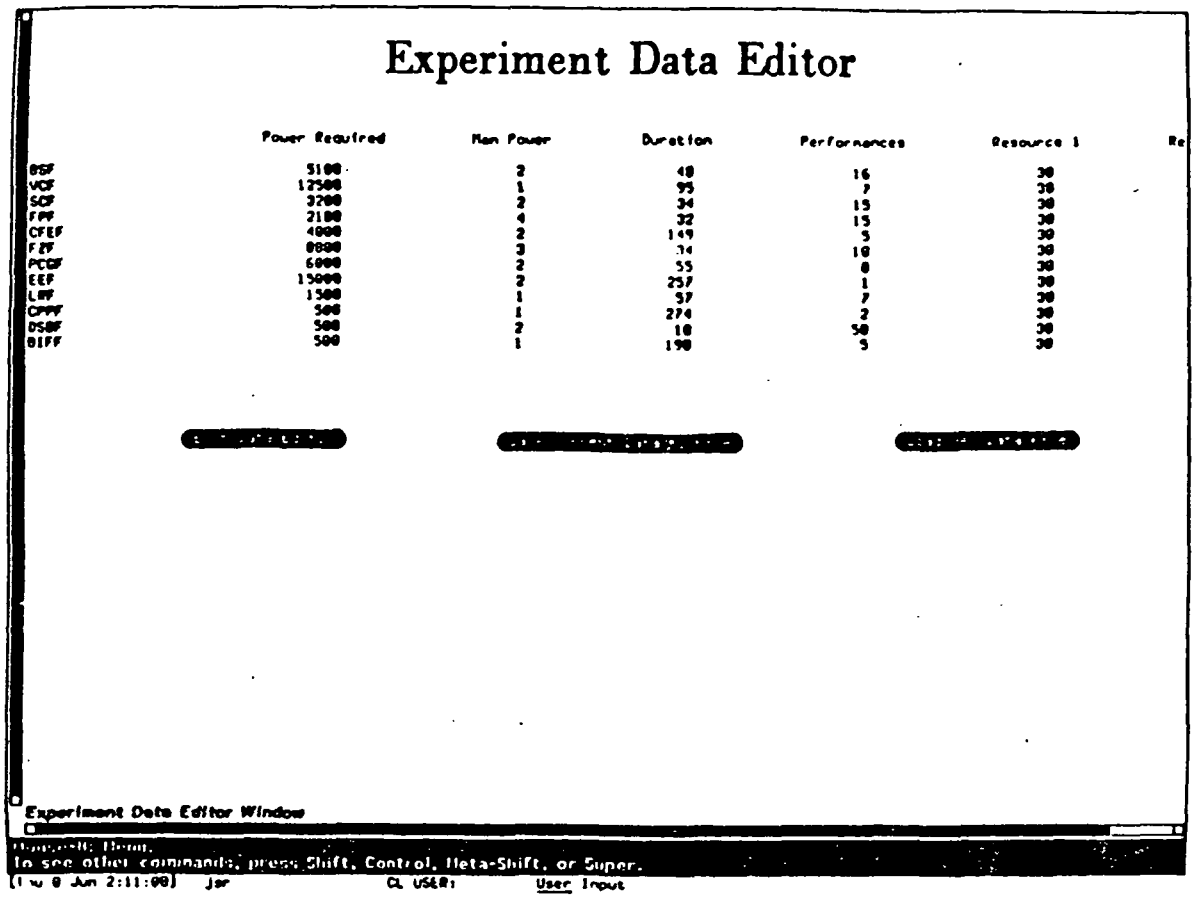
The Edit Resource Constraint menu option has been selected, presenting the Constraint Editor Window. The current resource constraint is displayed, and can be modified as desired. The constraint is expressed as a lambda expression, with X respresenting the sum of the resources used during one time slice. When the constraint is edited as much as desired, press the End key to return to the experiment data editor window.



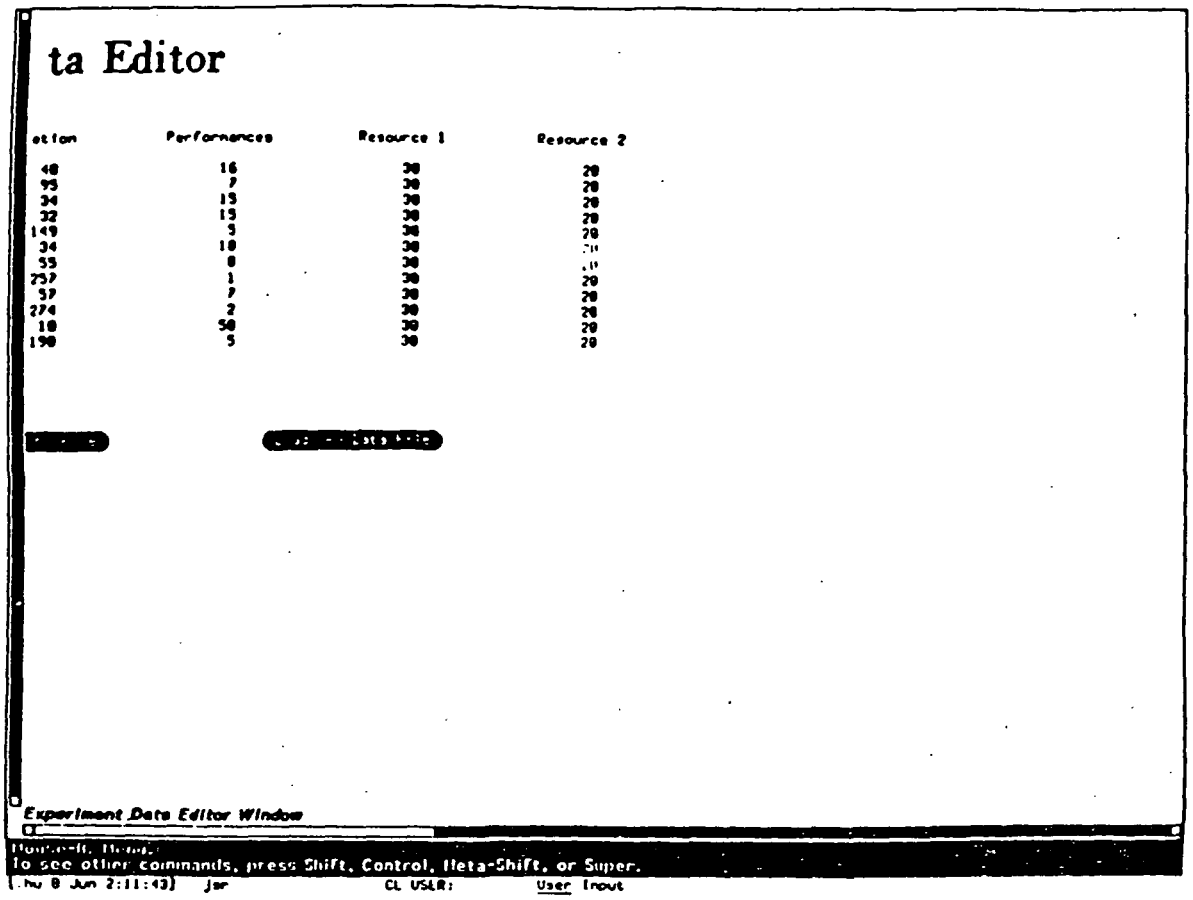
In this case the column title Experiment Number has been clicked on, and the Delete This Resource menu option has been selected. The Message Window confirms that the resource has been deleted.



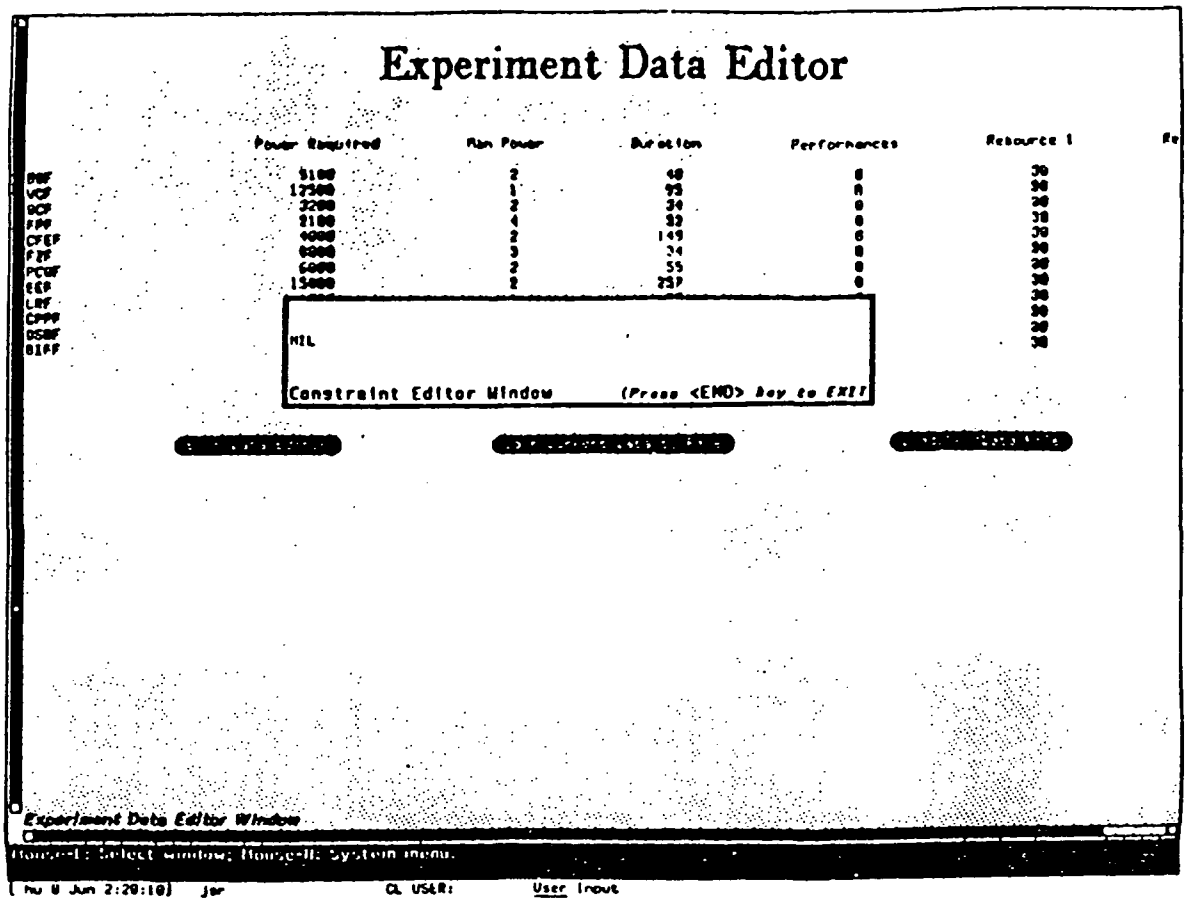
This screen depicts a situation in which the resource Performance has been clicked and the Add a Resource to the Right Menu option has been selected. The Add Resource Utility Window now appears. To add a resource first type the resource name, then click on the default Intitial value of 0, next type the new intitial value, press return, and choose Done.



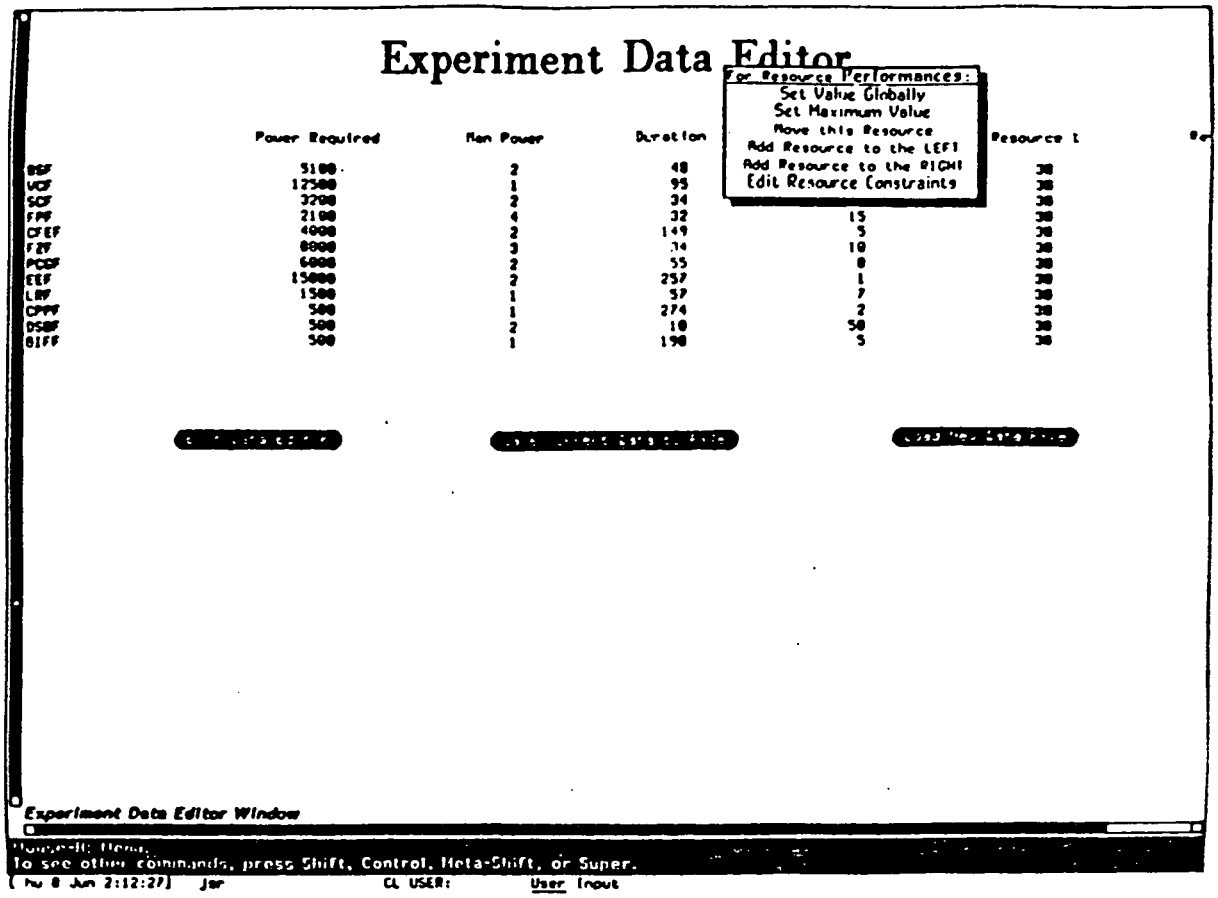
An interesting feature about the Experiment Data Editor window is that it is dynamic. This means that it allows resources and experiments to extend beyond the borders of the screen. Data beyond the borders can be seen by clicking on the scroll bars which are the arrows located in the bottom right hand corner of the screen. In this instance, the newly added resource, Resource 2 is partly visible on the right of the screen.



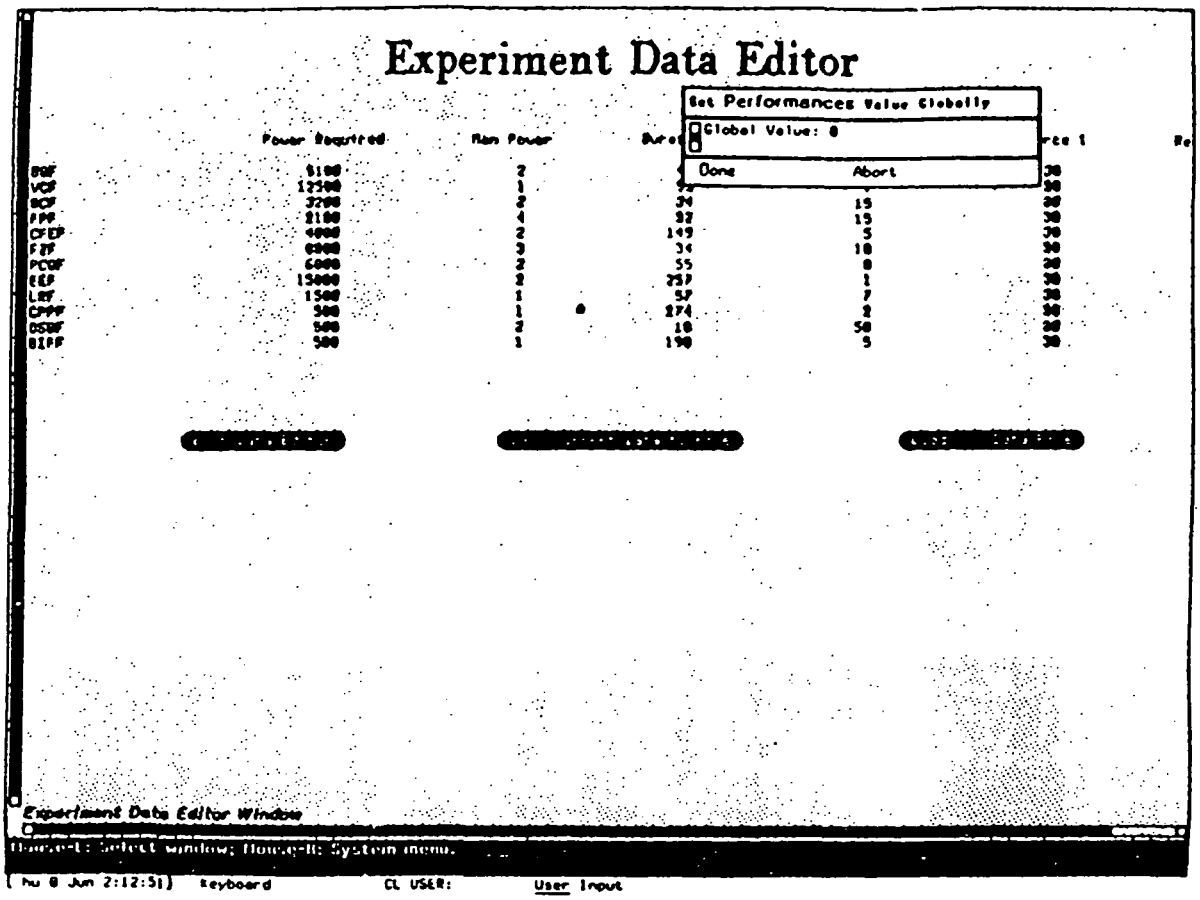
The Experiment Data Editor window has been scrolled to the right. This is done by moving the mouse to the right scrolling arrow and clicking. As a result, the Resource 2 column is fully revealed.



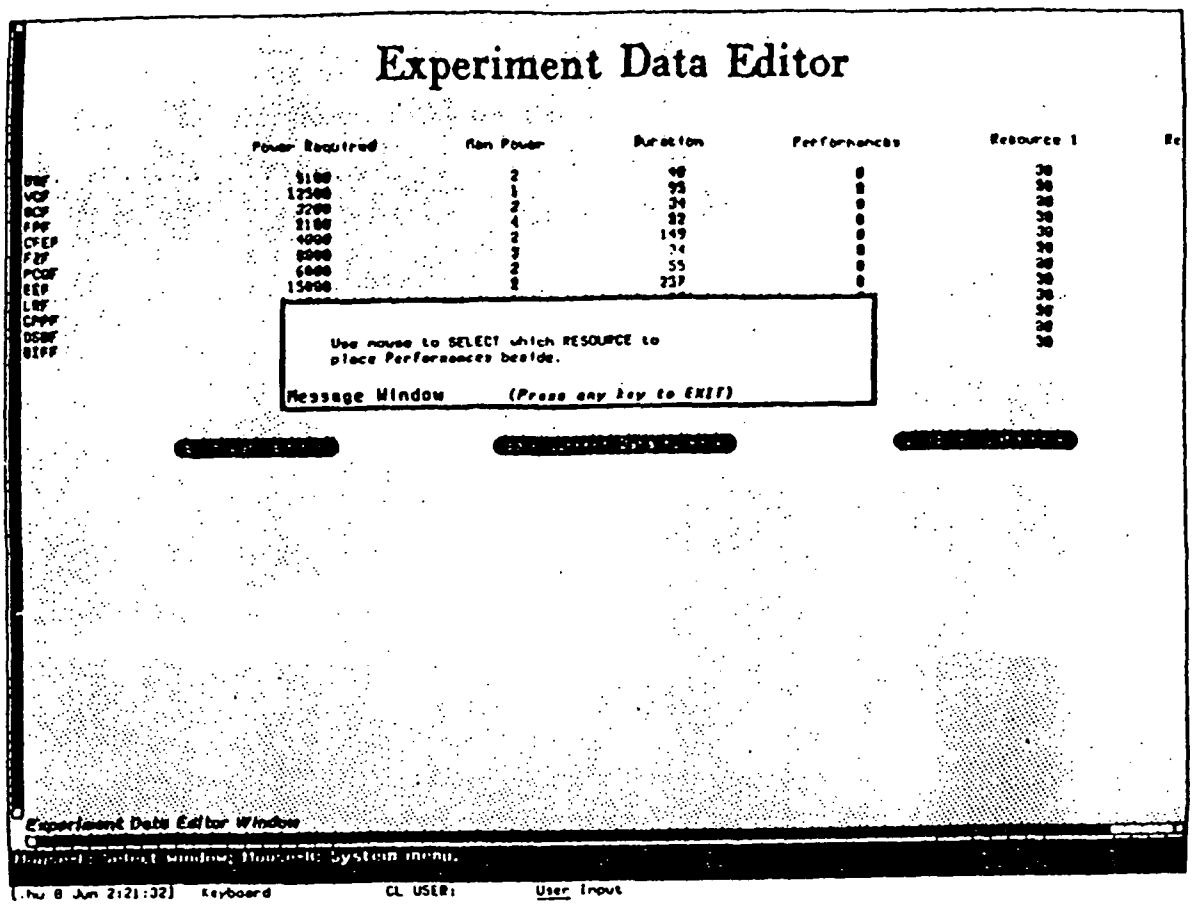
In this case the column title Resource 1 has been selected, and the Edit Resource Constraint menu option has been chosen. The Constraint Editor window now appears. Since Resource 1 was added using the Add a Resource option, its resource constraint is nil. A constraint for this resource can be added or it may be left nil. In order to exit this window, press the end key.



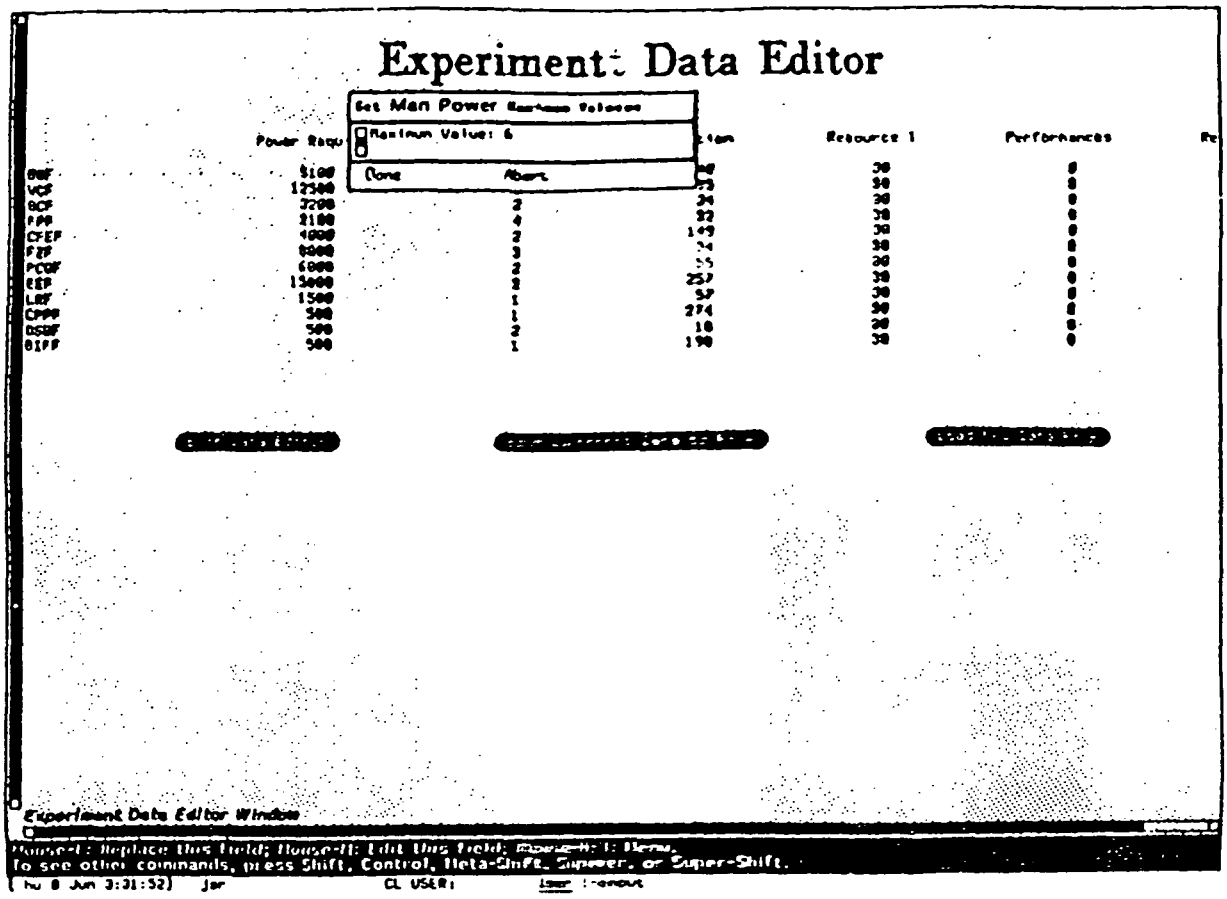
This is an instance in which the column title Performances has been selected. The Resource Options menu now appears. Any option can be selected by moving the mouse and clicking on it.



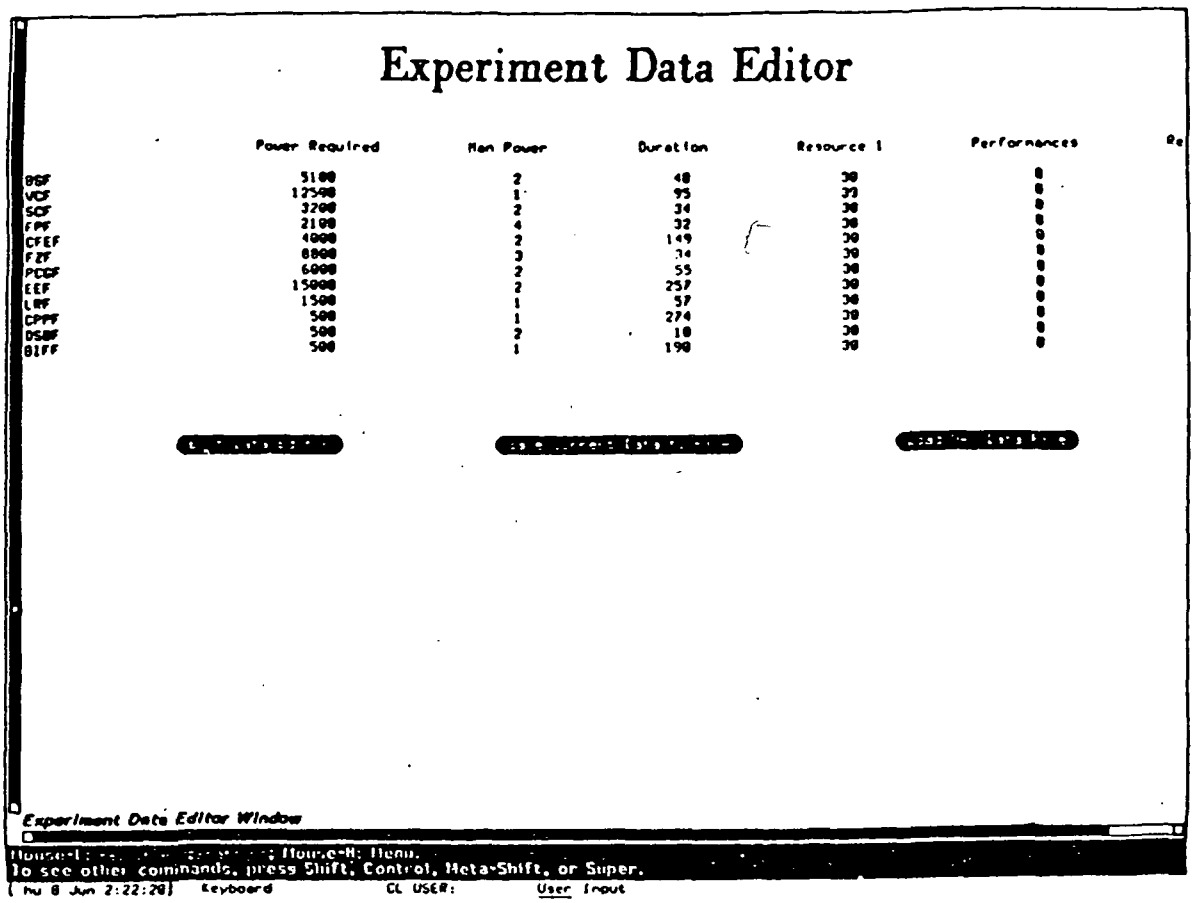
After having selected Performances, the Set Value Globally option has been chosen. The Set Value Globally window is presented. A global value for the Performance resources can be entered by typing the value, pressing return, and selecting Done, or the option can be aborted.



Once again the column title Performances is selected but in this case the Move This Resource option is chosen. Once this is done the message window appears. The message window describes the process for moving a resource. In order to close the message window, press any key. A resource is moved by clicking of the title of another resource. A menu will be presented with two options. The user can chose to add to the left of the selected resource or add to the right of the selected resource. Once the direction has been chosen,.the Experiment Data Editor window will be redrawn with the resource moved.



In this instance the column title Man Power has been selected, and the Set Maximum Value option has been chosen. In order to set a maximum value for the Man Power resource, simply type a new value, press return, and select Done. The user also has the choice to abort the option.

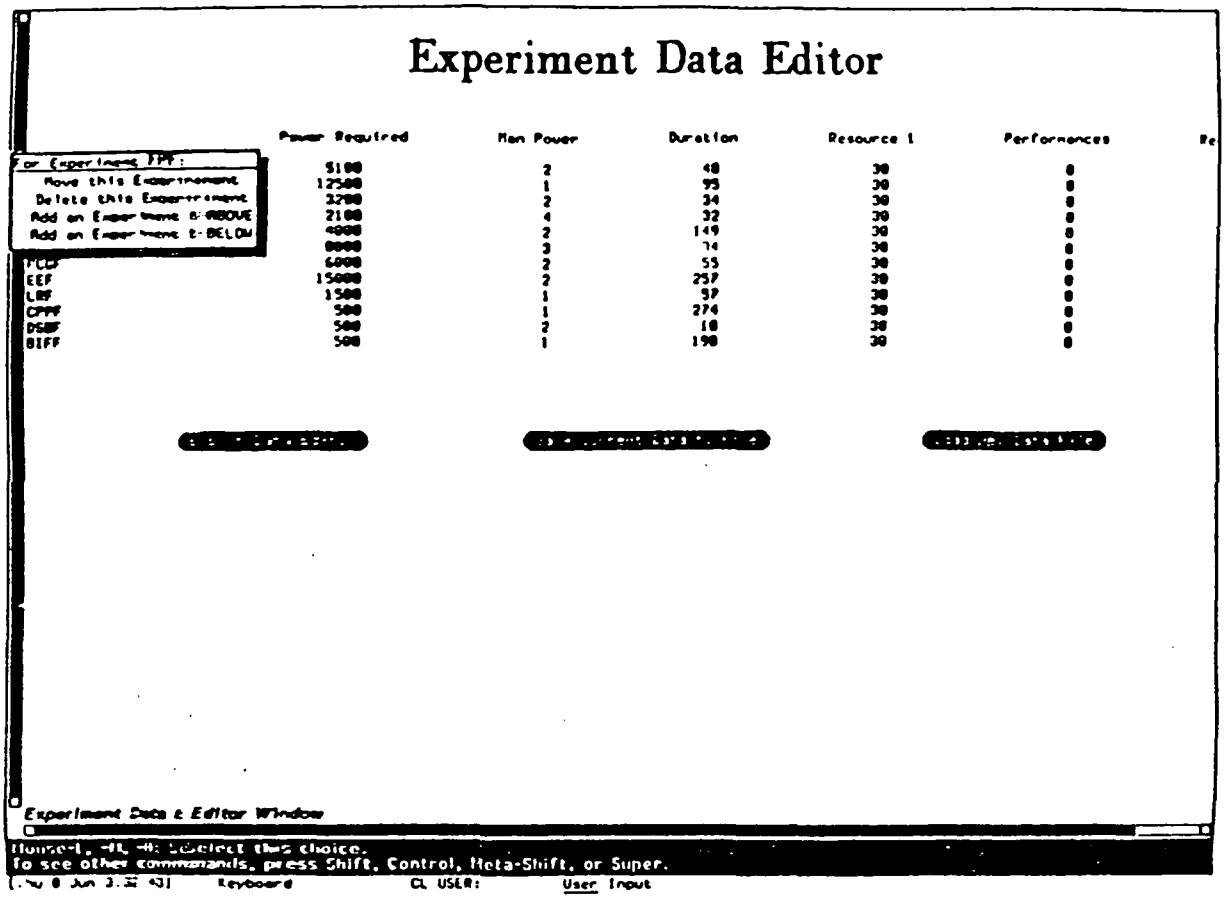


This screen shows what the revised Experiment Data window looks like with the changes made to this point. Thus far we have demonstrated what happens when we click on various columns which represent resources. We have also shown how some of the options operate. Now we will focus on the rows which represent different experiments.

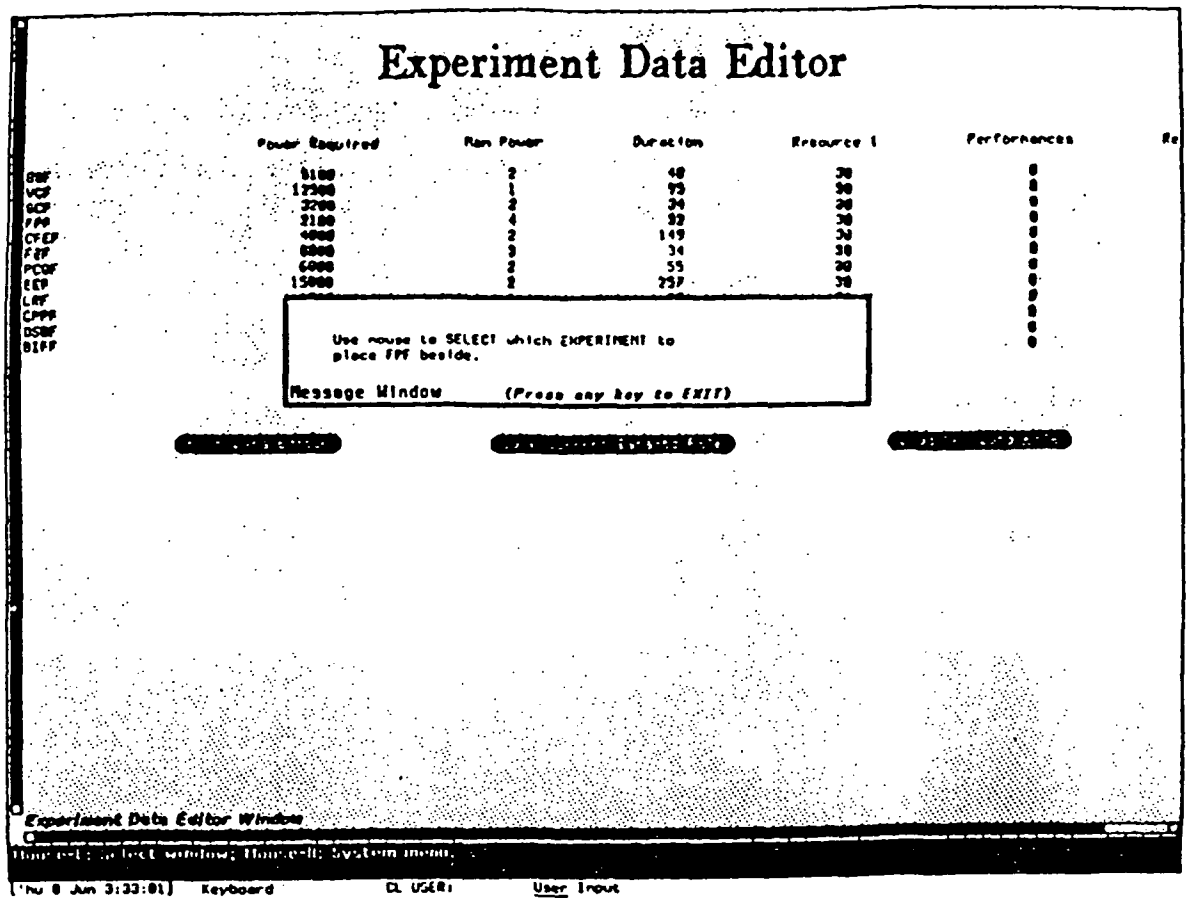
EXPERIMENT OPERATIONS

- **MOVE THIS EXPERIMENT**
- **DELETE THIS EXPERIMENT**
- **ADD AN EXPERIMENT**

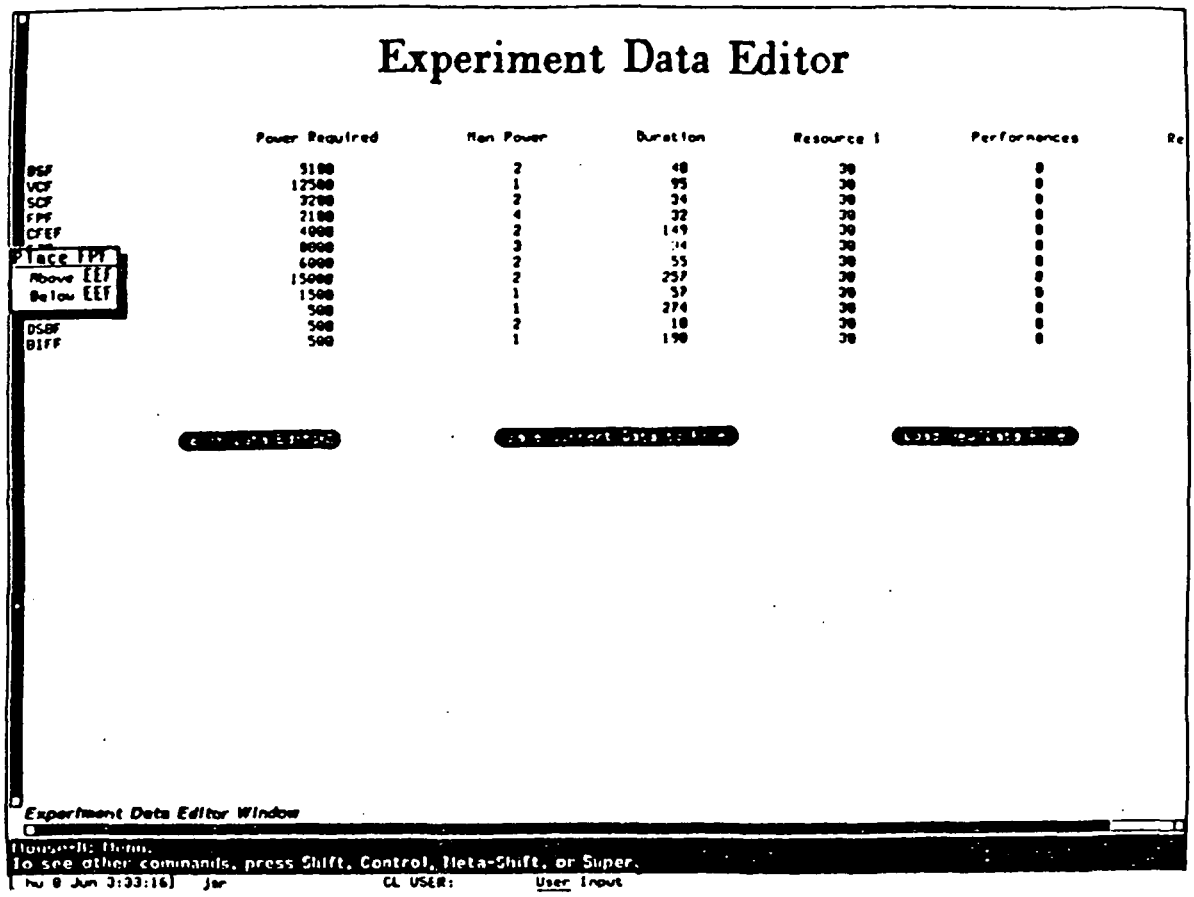
Each row in the Experiment Editor window represents an experiment. Clicking on an experiment name will present a menu of experiment operations. There are three operations that can be performed that can be performed on an experiment. The Move this Experiment option can change the position of a selected experiment. The Delete This Experiment option will delete a selected experiment. Finally, the Add an Experiment enables the user to add a new experiment above or below a selected experiment.



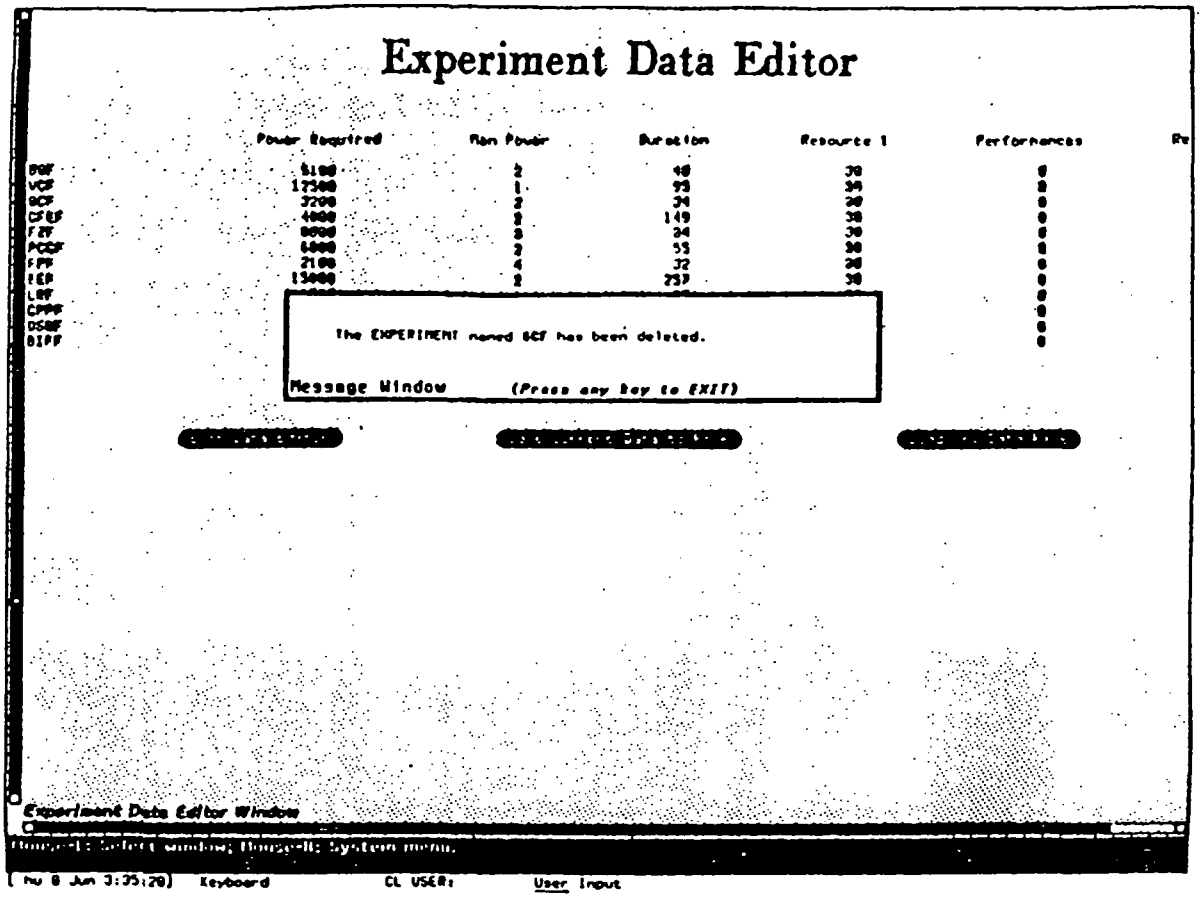
This is what the screen would look like if the Move This Experimennt options was selected.from the Experiment Options menu. TThe message window gives instructions for moving an experiment.. This process is described in detail on the next page. In order tco close the message window, simply press any key.



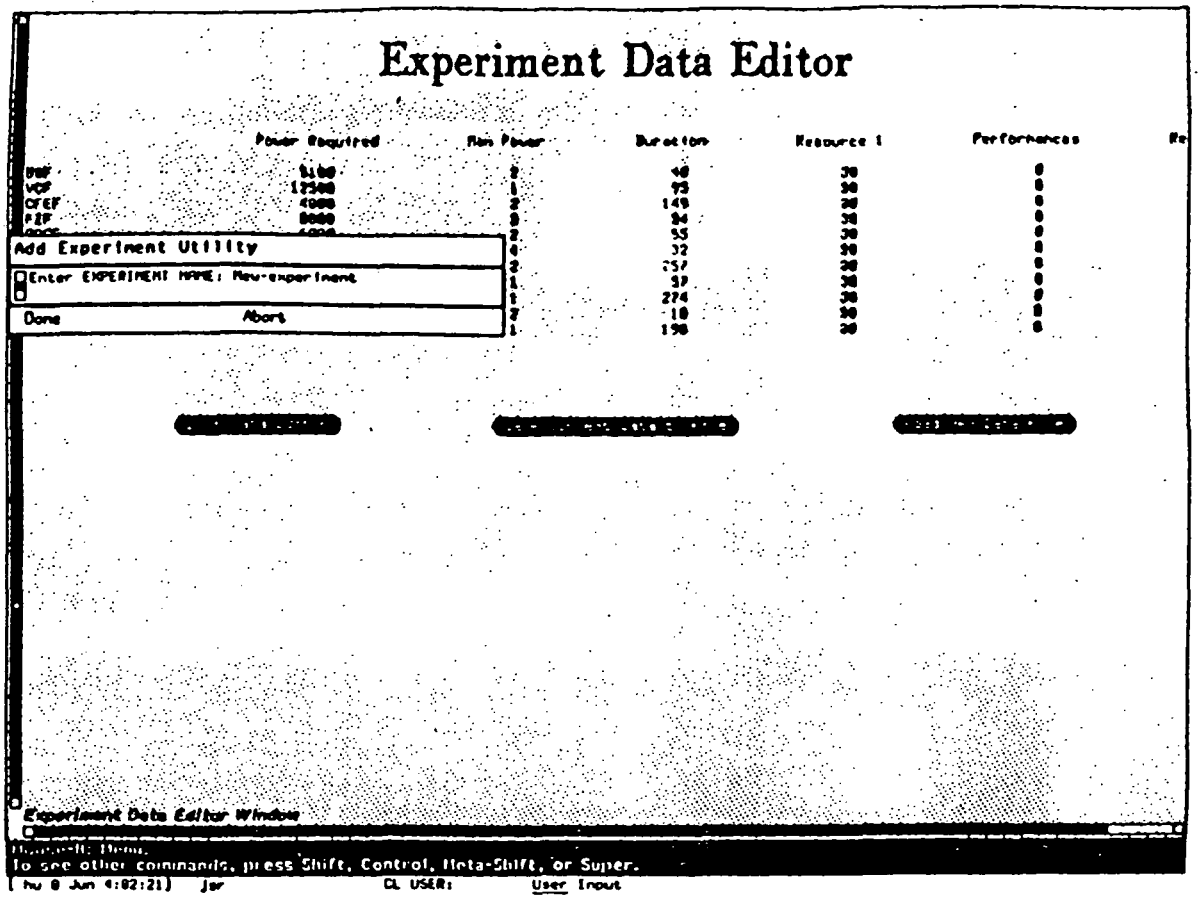
In this case the experiment name FPF has been selected. Once again, in order to select an experiment simply place the mouse on the desired experiment and click. After this is done, the experiment options menu is presented.



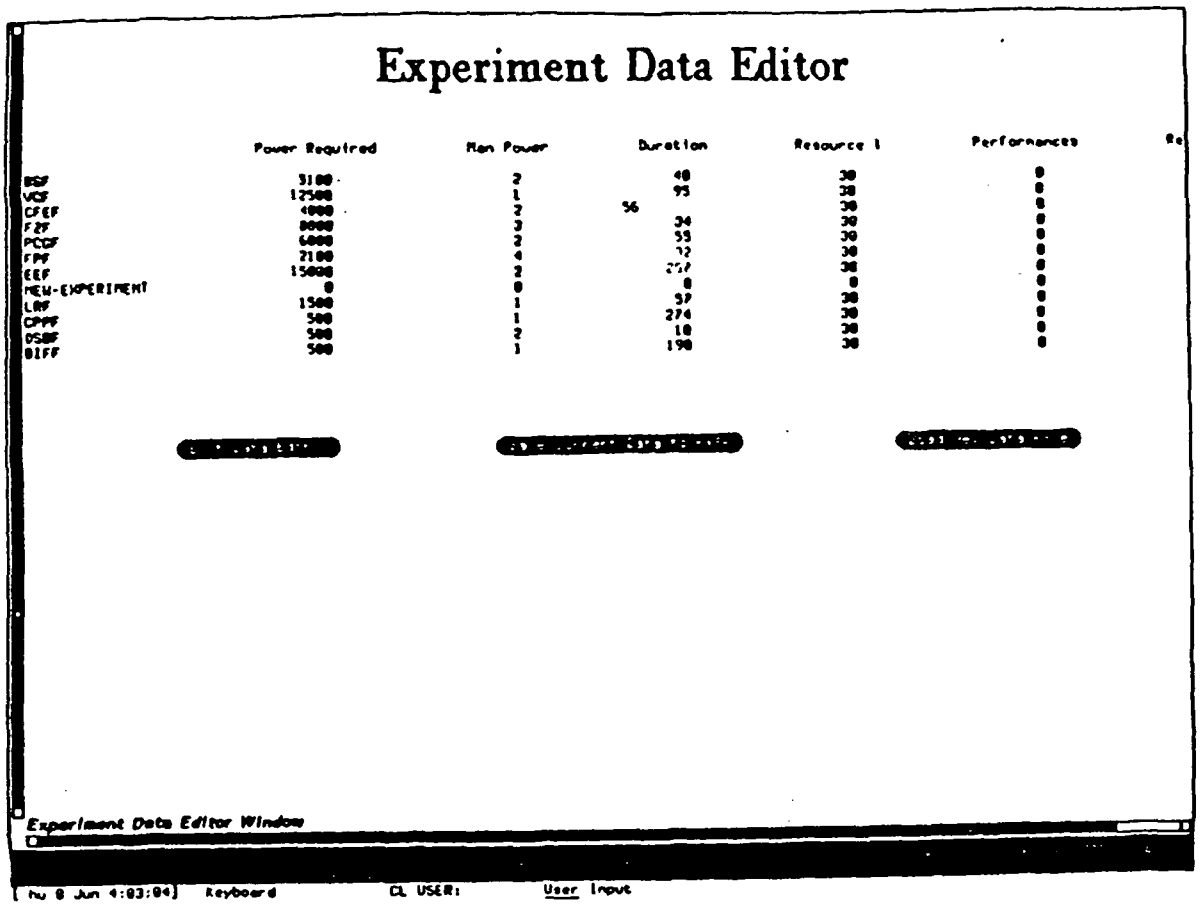
To move the FPF experiment, another experiment name must be selected. The user can do this by using the mouse to click on the desired experiment. In this instance, the EEF experiment has been selected as the experiment to place FPF. This is done by clicking either above EEF or below EEF.



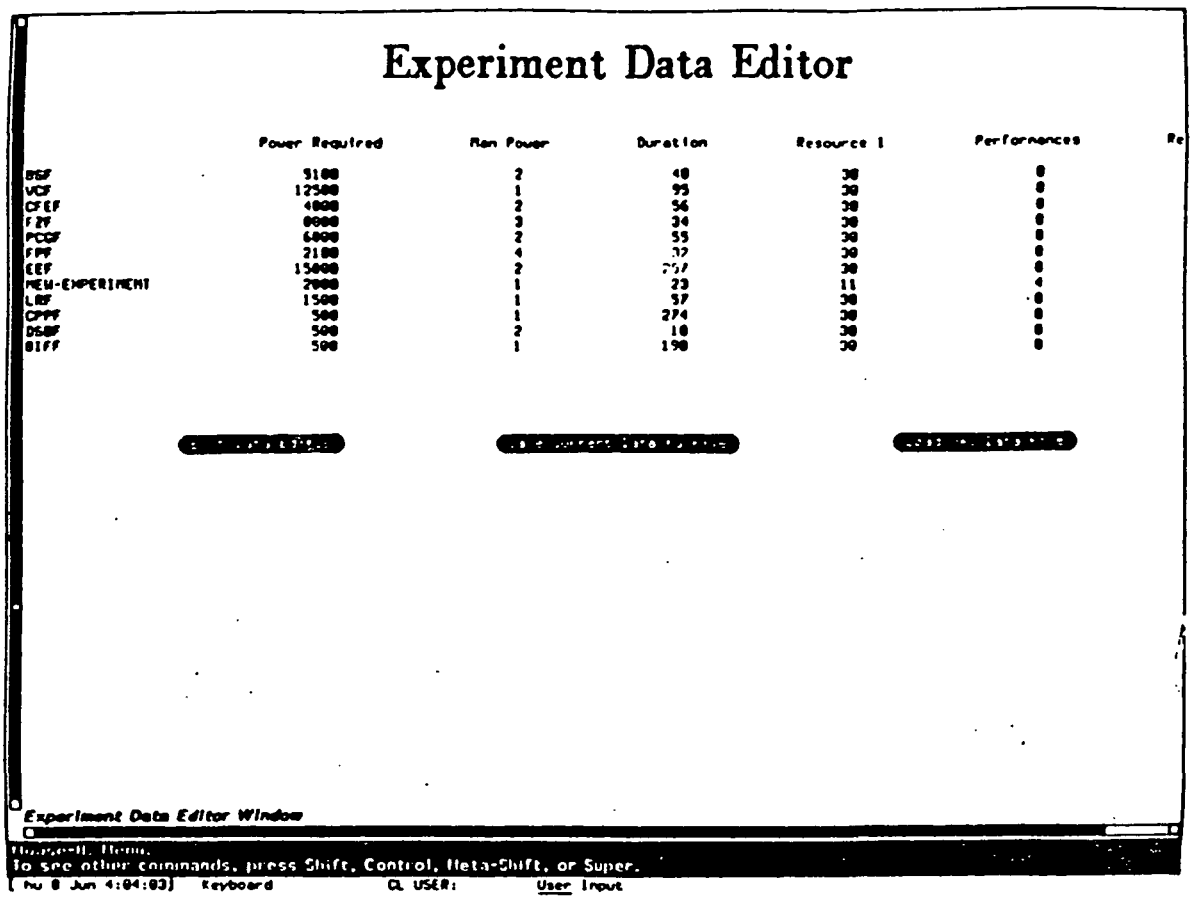
This is a situation in which the experiment name SCF has been selected, and the Delete This Experiment option has been chosen from the experiment operations menu. The message window confirms the deletion of SCF. In order to exit the message window, press any key.



In this case, the experiment name EEF has been selected, and the Add an Experiment Below option has been chosen from the experiment operations menu. The Add Experiment Utility Window is used to enter the new experiment name by typing the experiment name, pressing return and then clicking on Done. The user also has the option to abort the command.



This is what the screen looks like after the experiment New-Experiment has been added to the Experiment Data Editor Window. Notice that it has resource values of 0. Each resource value can be changed by clicking on the value, typing in a new value, and pressing return. In this case the Duration for experiment CFEF has been selected for editing.

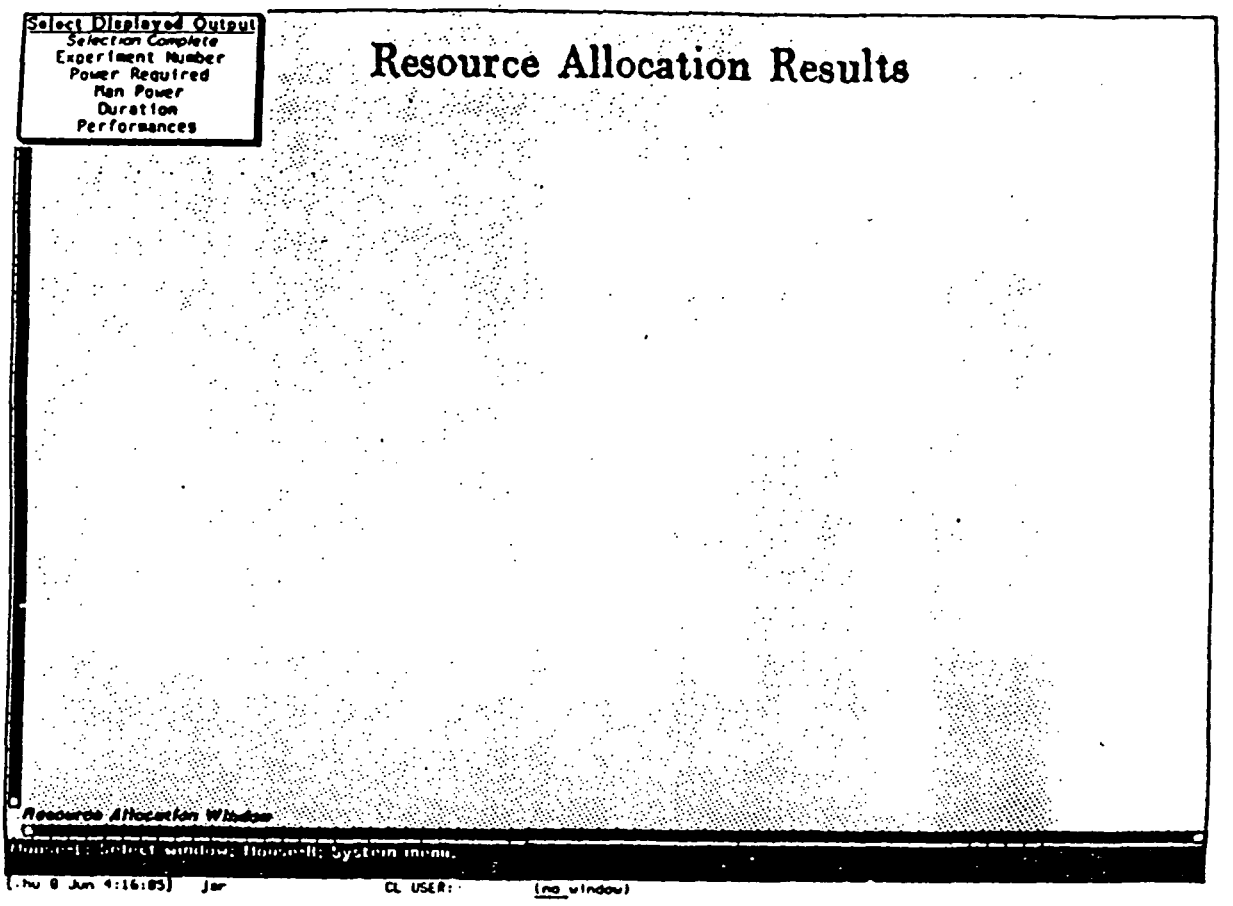


The user can easily change the resource values for the New-Experiment. This can be done by selecting each value individually and editing them. To select a value, simply place the mouse over the value you wish to edit and click. To edit, just type in the desired value.

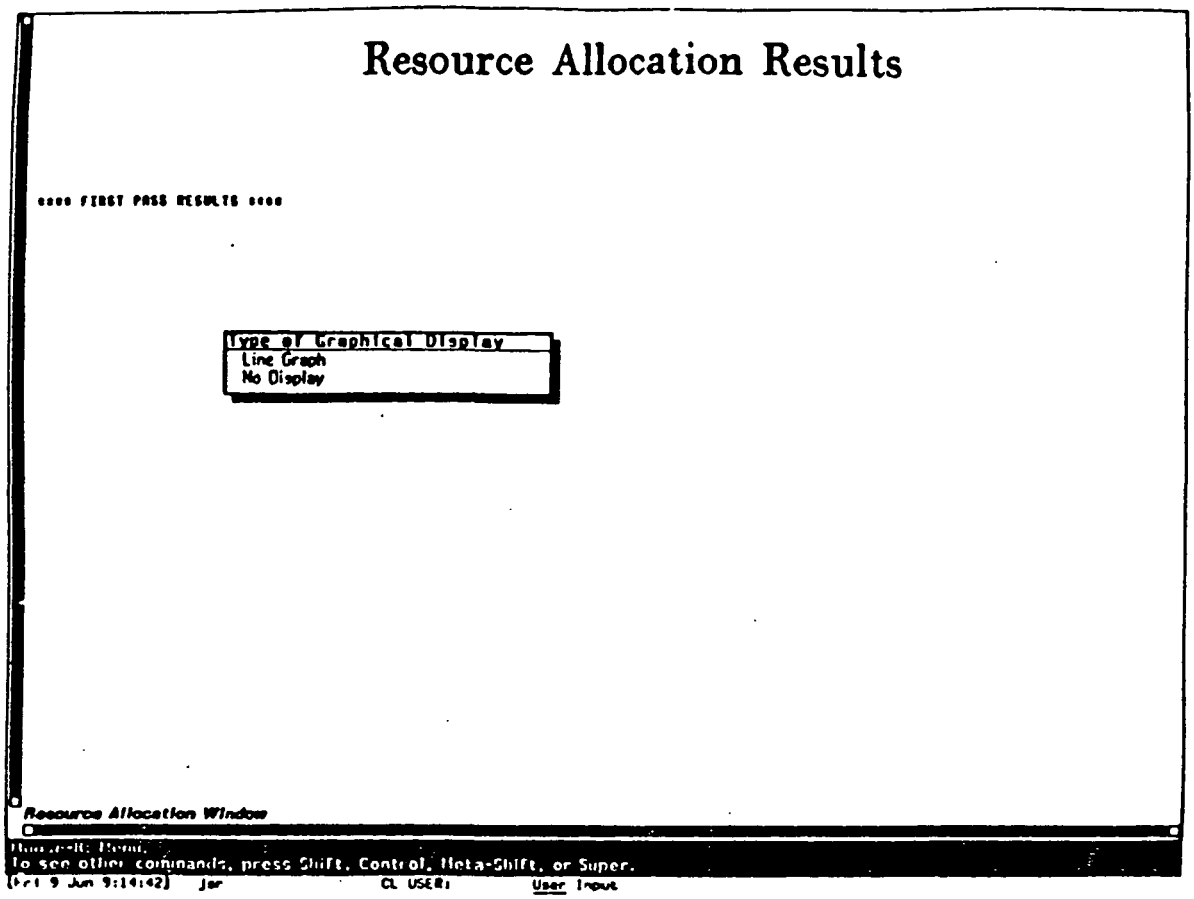
DISPLAY CHOICES

- SELECT DISPLAYED OUTPUT FROM RESOURCES
- TYPE OF GRAPHIC DISPLAY
  - NO GRAPH
  - LINE GRAPH
- SELECT GRAPHICS OUTPUT FROM THE DISPLAYED OUTPUT

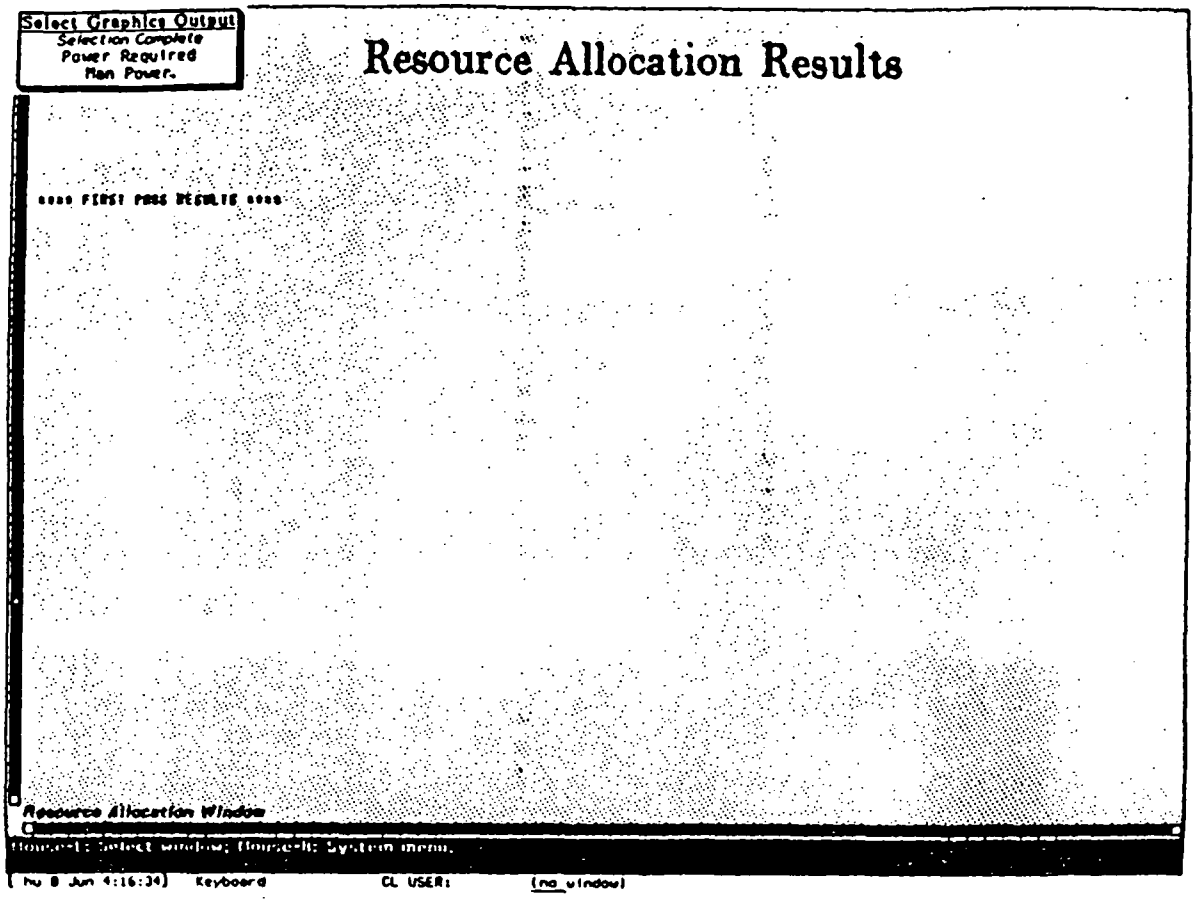
When the Exit Data Editor button is clicked, the data in the Experiment Data Editor window is passed to the allocator. Three menus will be presented. One menu is the Select Displayed Output menu. This is a menu from which the resources to be displayed during pass results are chosen. The second menu is the Type of Graphic Display menu. This menu allows the selection of a graph type on which to display resource data. The final menu is the Select Graphics Output menu. This menu provides the resources to be displayed on the graph.



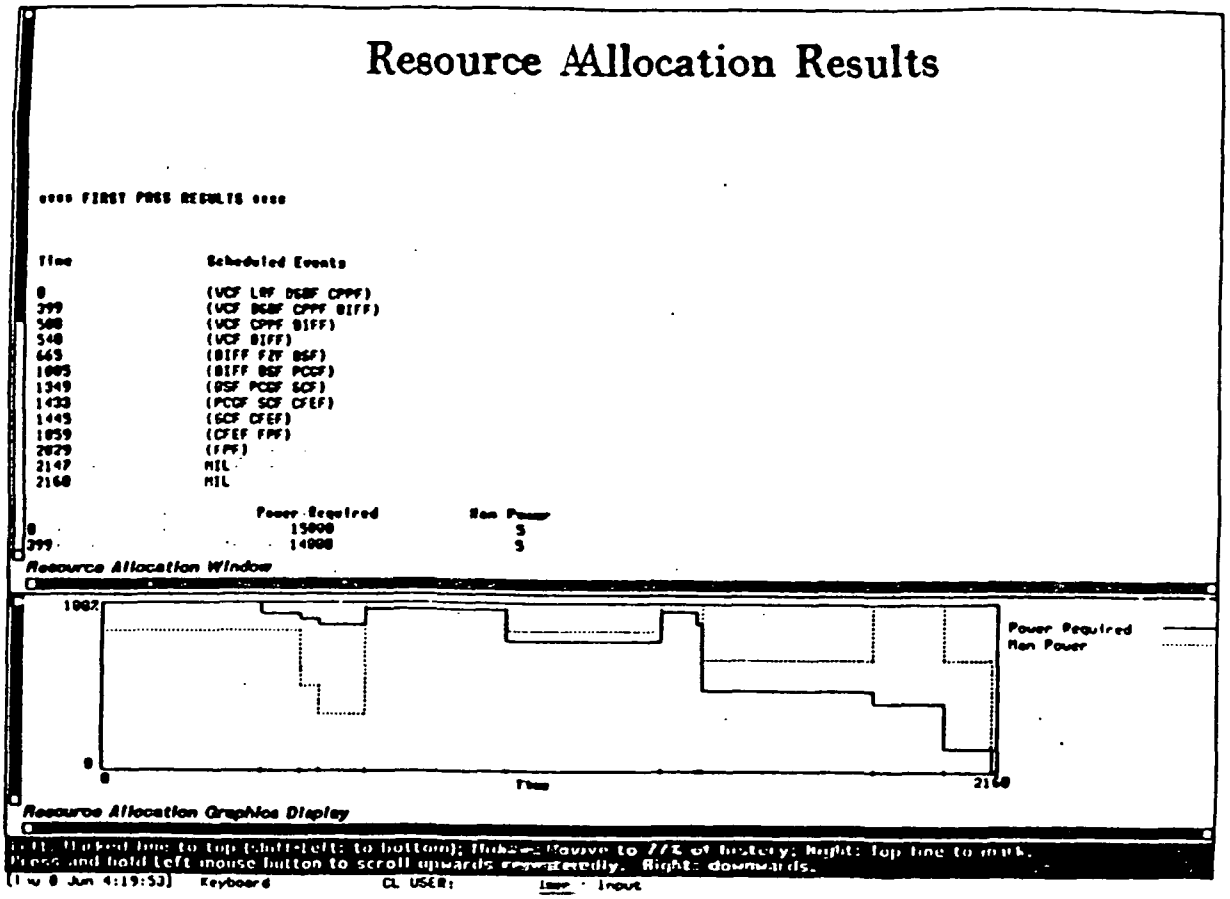
The Select Displayed Output menu is displayed. This menu allows the user to select the resources which will be displayed during the pass results. In order to choose a resource, simply place the mouse on the resource you wish to select. When this is done the resource will be highlighted. You may choose to pick one resource or all of them. Once you have highlighted the appropriate resource or resources, click on them. After you are done, click on Section Complete. The screen will disappear and the Type of Graphical Display menu will appear.



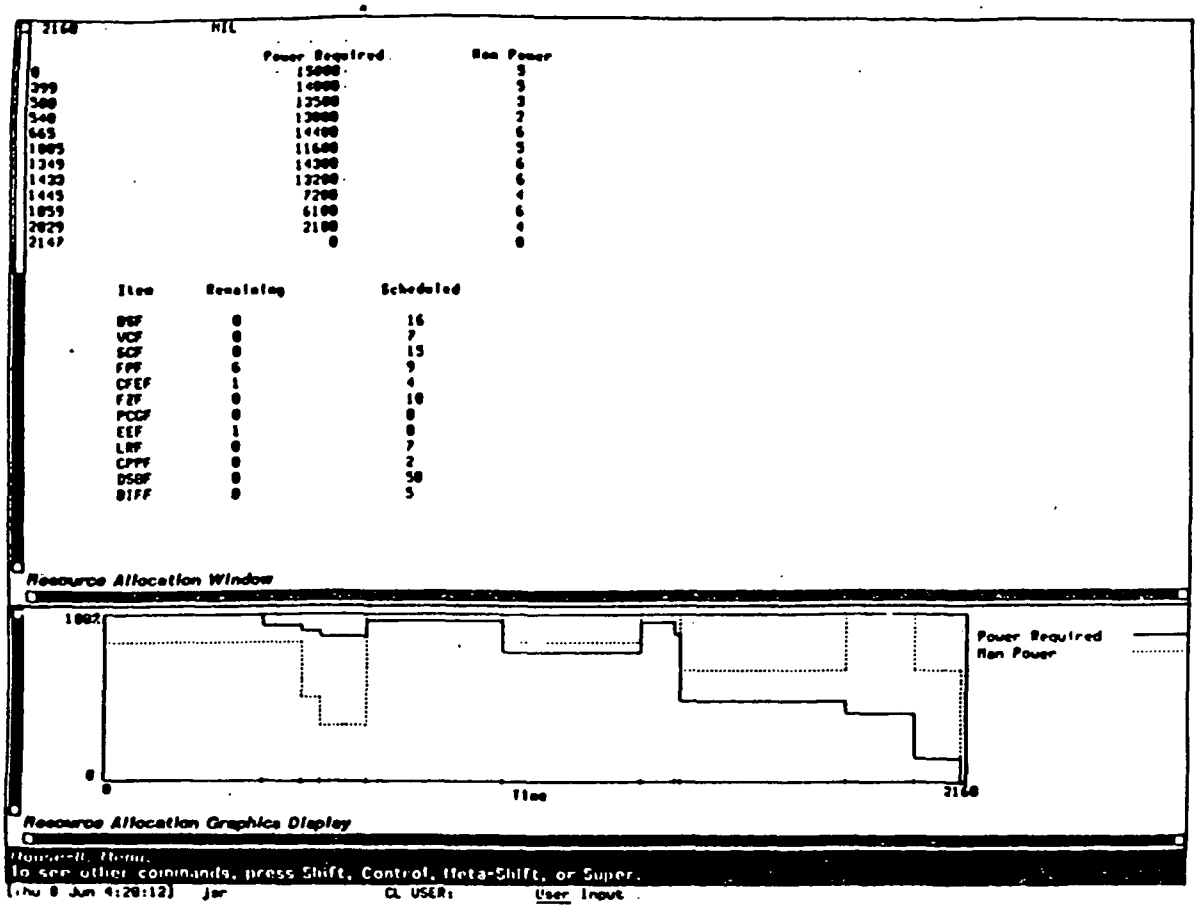
This is the Type of Graphical Display menu. The user can only generate graphs of resources selected from the Select Displayed Output menu. The user has the option to make a line graph of the available resources or to make no display. After the graph or no display option is chosen, the screen will disappear and the Select Graphics Output menu will appear.



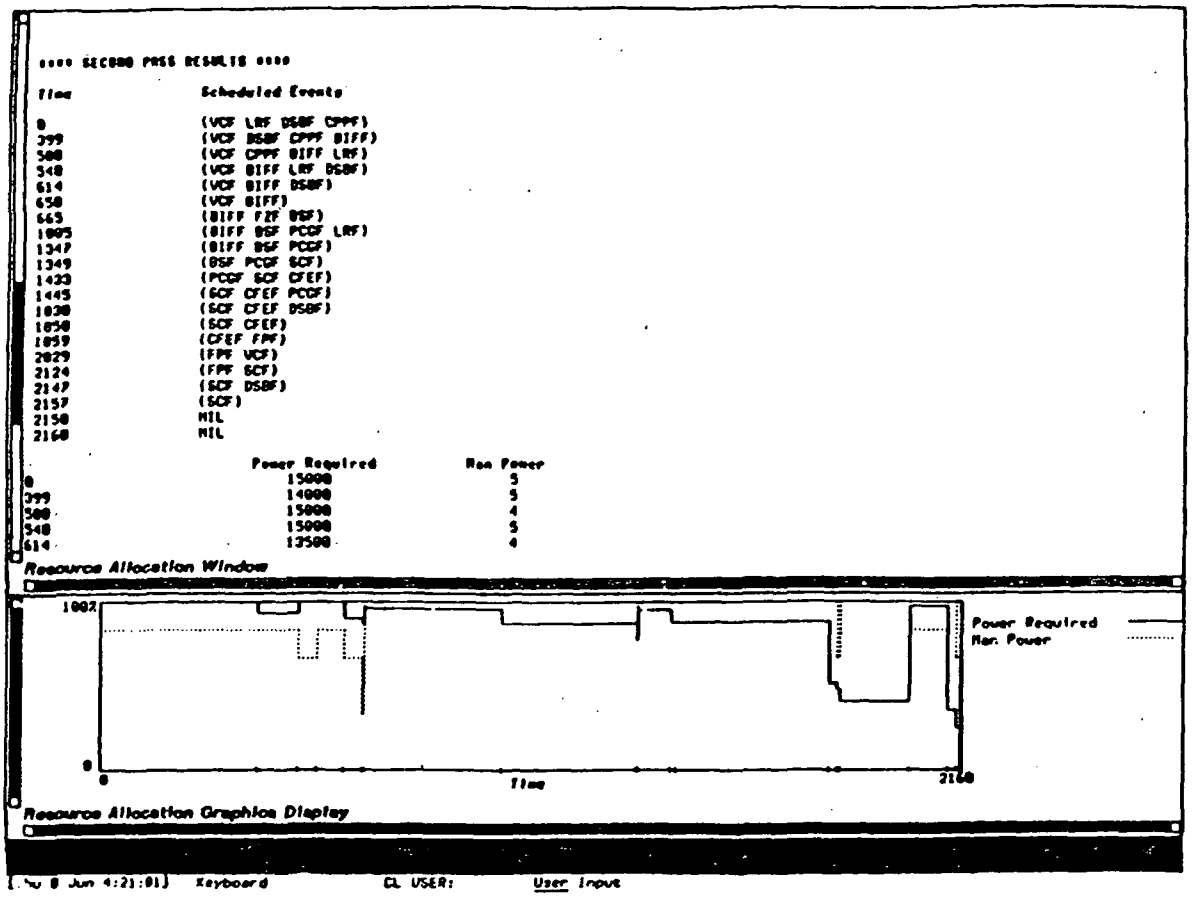
The Select Graphics Output menu is now presented. The resources that are to be included on the graphical display are selected from those listed on the menu. The user may decide to make graphs from all available resources or just a select few. Once this is done this screen will disappear and the results from the First Pass will appear.



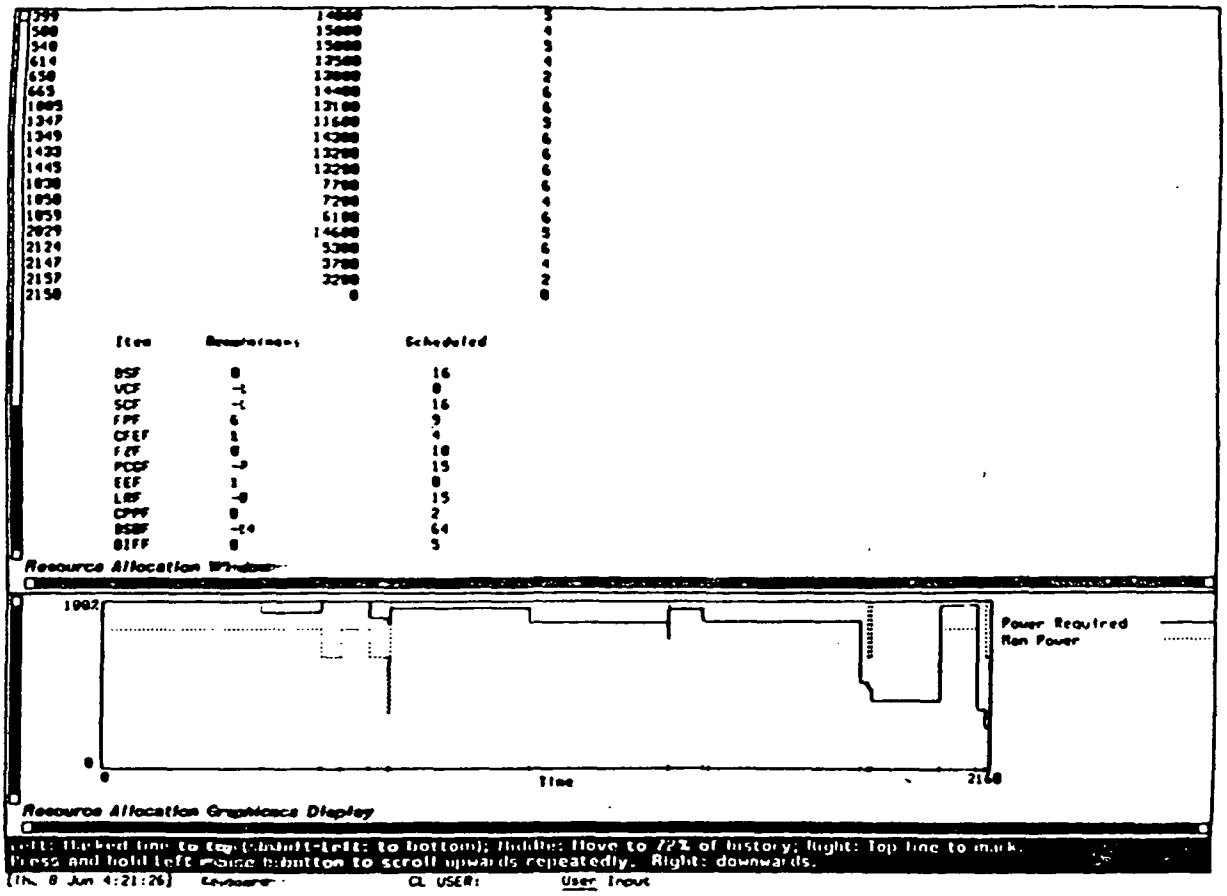
The screen is now divided into two different windows. Each window can be scrolled independently. The top half of the screen is the Resource Allocation Window. In this window the First Pass Results are displayed. The bottom half of the screen is the Resource Allocation Graphics Display Window. In this instance, Power Required and Man Power were the two resources selected from the Select Displayed Output menu. Line Graph was selected from the Type of Graphical Display Window. Power Required and Man Power were also selected from the Select Graphics Output Window. It is important to remember that the First Pass Results only satisfy minimum requirements. This accounnts for the gaps in the graphs.



The Resource Allocation Window is longer than one screen. Thus the results from the First Pass exceed what is visible. In order to display the rest of the pass results the screen can be scrolled down. To do this just place the mouse on the scroll down arrow and click.



After the First Pass Results are presented, the program will continue and the Second Pass Results can be scrolled up. Notice that the Second Pass attempts to fill in the gaps left by the previous pass. The line graph is now much more complete than before. The Second Pass is similar to the First Pass in that the Resource Allocation Window is longer than one screen.

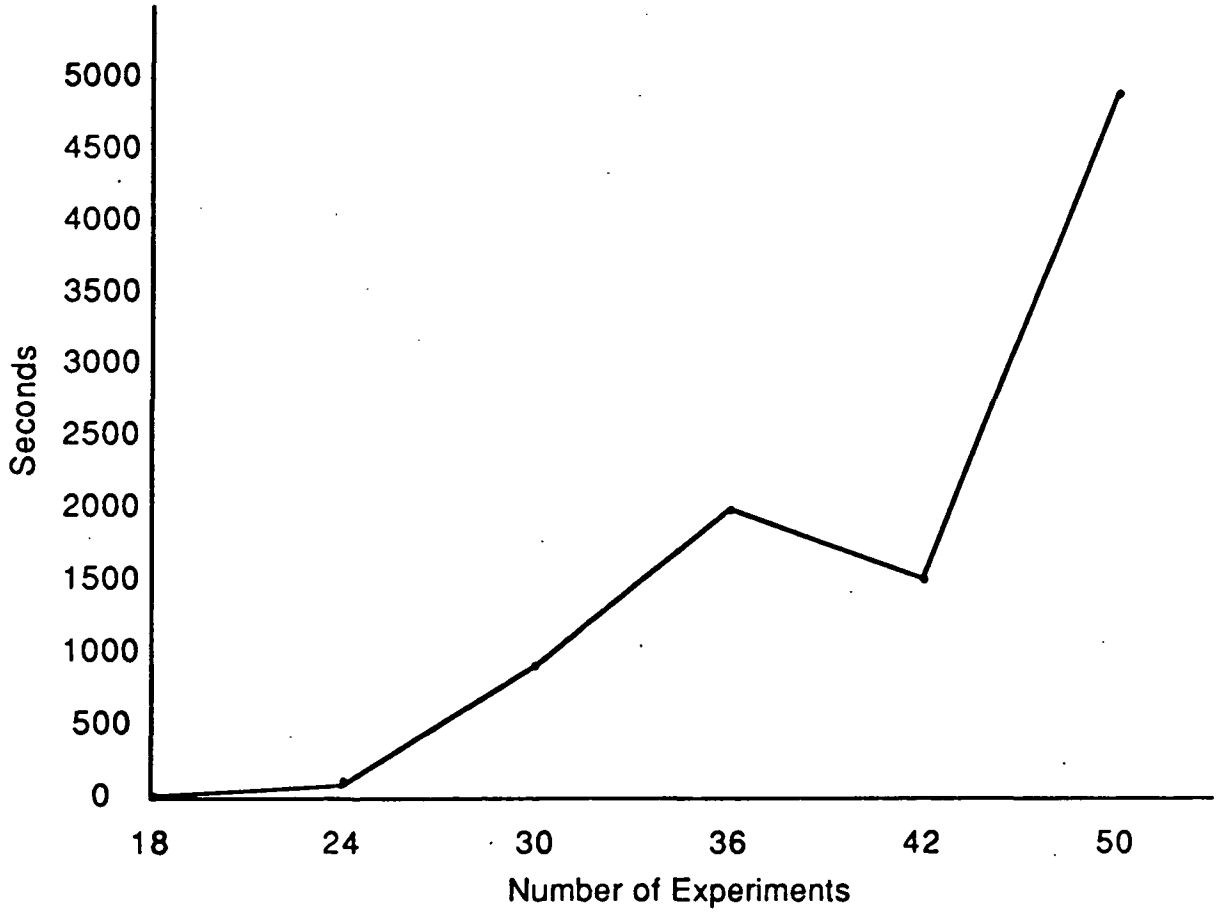


The rest of the Second Pass Results can be seen by scrolling down. In order to do this, follow the same process of moving the mouse to the scroll down arrow and clicking.

Table 1. Multiple Pass Algorithm Timing Test Results

Test Number	18	24	30	36	42	50
Run #1	29.09	110.9	888.72	2005.57	1536.94	4804.59
Run #2	23.96	113.2	895.8	1977.82	1542.82	4821.25
Run #3	27.22	111.3				
Run #4	24.16	114.34				
Run #5	26.84	120.99				
Average	26.25	114.14	892.27	1991.69	1539.88	4812.92

Table 2. The Timing Tests Results for the Multiple Pass Algorithm



In conclusion, the Multiple Pass algorithm performed in a satisfactory manner; however, more work needs to be done to refine the algorithm to reduce the total execution time so that larger sets of data can be tested. More tests need to be performed to ensure the algorithm is suitable for being considered for future work. The Free Expansion algorithm needs to further refined so that an exhaustive search is avoided, yet meaningful results are obtained.

5.0 Connecting A Symbolics to A VAX

5.1 Task Statement

The purpose of this task was to provide a cursory look at two ways of connecting a Symbolics lisp machine to a VAX minicomputer.

5.2 Task Conditions

The conditions of this task was that the machines to be considered were already in place at the NASA MSFC facilities. The Symbolics machine is a 3670 and the VAX is a 785 machine. They are currently located in separate buildings at MSFC that are some distance apart. At present, there is an existing network that could be used as a medium for connecting the machines, if necessary and possible. The desired result is to have the two machines be able to share memory during execution and not just to pass files between them.

5.3 Task Approach

There are two basic ways of connecting the Symbolics Lisp machine to a VAX. These are software and hardware. The least expensive from an implementation stand point is usually the software approach. This approach consists of cables and protocol systems. The cost of this approach is situation dependent; however, the EtherNet cable can be purchased for approximately one dollar per linear foot.

The primary consideration in any situation is the location of the machines to be connected. The distance between them determines the amount and cost of the cable needed. The other expenses include the connector boxes for each machine and the software to facilitate the communications.

Symbolics supports all the traditional communication protocols, such as TCPIP, DECNET, etc. These are available from Symbolics, Inc. along with the price list. However, the cabling should be purchased from another source (Inmac) to reduce cost.

The other approach, hardware, is a more expensive proposition. A company in Amherst, NH, provides a hardware product, Bus-link, for connecting a Symbolics machine to a VAX. Basically, this device connects the machines at the bus level and allows the Symbolics to map and address the memory of the VAX, as if it resided in the Symbolics. This allows existing programs on the VAX to operate and write their information so the Symbolics can directly address it. Thus, a direct coupling of knowledge-base and conventional systems can occur. The cost of this device with the associated peripherals is between $30,000 and $40,000. A more detailed discussion of this product is provide in the company information provided to the Mission Planning personnel.

5.4 Task Results

It is recommended that the software approach be used to connect the Symbolics and the VAX machines. This is the lowest cost approach and will come closer to accomplishing the objectives of MSFC Mission Planning personnel. The main consideration here is that the Mission Planning personnel would like to have the programs that already exist on the VAX to be able to communicate with some programs on the Symbolics. Thus, the direction of communication is important; thus, the Bus-link device is not the preferred approach to solving this potential problem. If the choice of direction changes then the Bus-link may be the most acceptable alternative.

6.0 FORTRAN from Lisp

6.1 Task Statement

This task involves finding ways to call Lisp functions from inside FORTRAN other than just spawning a process. The intent here is for an application in FORTRAN to be able to call Lisp functions during execution and to be able to pass data and information back and forth.

6.2 Task Conditions

The conditions of this task are (1) Lisp must be called from inside a FORTRAN application, (2) data and/or information must be passed, (3) the two languages are resident on the same computer, and (4) the computer should be a VAX.

The first two conditions are taken from the task statement, the third condition is very important. This condition must be used or the complexity of the problem is to great to make accomplishment possible. Trying to go across any connection between machines makes this task virtually impossible because of the variability of the different connection methods, hardware, etc. The fourth condition was specified by the Mission Planning personnel; however, strict adherence to this was not given.

6.3 Task Approach

The first thing done under this task was to check the most familiar environment to UAH. This is the Symbolics Lisp machine. While this was not in compliance with the fourth condition, it was deemed necessary to acquire an understanding of the task. Also, a fundamental question as to being able to do this at all still existed in our minds.

The ability to call FORTRAN from Lisp and vice versa on a Symbolics is provided. This is easily accomplished, when compared to other processors, because the operating system of the machine is Lisp. Thus, a call from Lisp to FORTRAN is an operating system function and from FORTRAN to Lisp is an operating system call. Therefore, the interaction between these two languages are relatively easy. Certain restrictions do apply. These mainly have to do with how arrays are handled and some cautions on value referencing. A detailed explanation can be found in the Symbolics FORTRAN manual.

6.4 Task Results

At present, it is not possible to call Lisp from inside FORTRAN on a VAX except when spawning a process. Also, it is not advisable to use FORTRAN on a Symbolics because of the reduced execution speed and increased compilation speed. The only remaining possibility is to have the FORTRAN program and the process that is spawned to use some shared memory for message passing. This is not an easy solution, thus, it is not a preferred method. Before this problem can obtain an easy solution, some technological advances need to be made and incorporated on the VAX. The main thing that needs to occur is for the operating system needs to allow programs that run simultaneously to communicate with each other.

7.0 Trees and Forest Task

7.1 Task Statement

The purpose of this task was to review the software product Trees and Forest as to their suitability as a programming language for the Mission Planning personnel to use in developing a scheduling system.

7.2 Task Conditions

The conditions of this task were that a review of the software would be conducted using the documentation provided by MSFC Mission Planning personnel. There would be no need for developing a prototype system in the language. Just a review of the capabilities and limitations would be conducted.

7.3 Task Approach

In 1973, under funding from the National Aeronautics and Space Administration, an advanced programming language was developed. This language was called PLANS and its objective was to reduce the cost of developing and maintaining software to support scheduling and resource allocation tasks. PLANS was ideally, but not uniquely, suited to writing scheduling programs. Another product was developed to support PLANS, it was called PLUS. This product was a library of utility programs written in PLANS and which represented logic that is common to a broad range of operations planning and analysis software.

Avyx took PLANS and PLUS, revised them and re-implemented them to make them PC compatible. The resulting products are called TREES and FOREST. TREES corresponds to PLANS and FOREST to PLUS.

TREES resulted from the known deficiencies in existing languages used for scheduling and resource allocation. These deficiencies are:

(1) the language level did not correspond to the level of the functions typically found in the algorithms, and

(2) the data structures of the languages (usually only arrays) did not correspond to those typical of the application problems, thus contributing greatly to software development time.

According to the developers of TREES, it was designed to achieve theese goals:

(1) to allow designers of experimental or constantly changing scheduuling and resource management algorithms to translate algorithm design to working code directly from their basic functional descriptions.

(2) to allow designers to do this without performing intermediate and detailed program design steps, without possessing highly specialized programming expertise, and with only a minimum of span time and manpower costs.

These two goals are directly related to overcoming the deficiencies previously stated.

Also, the developers believe that scheduling and resource management problems often involve information structures which are logically hierarchical. That is, a component-subcomponent relationship exist among the items composing the information structures. Thus, the structures are made up of different levels of nodes. This is best conceptualized as a tree. Not only are the results of the scheduling process hierarchical in nature, but so are the inputs.

TREES was designed around this type of structure and it allows for the manipulation of these structures, as well as content, at execution time. While this feature distinguishes TREES from conventional languages like FORTRAN,

COBOL, PL/1, ALGOL, and ADA; it does not separate it from LISP. However, TREES claims to be easier to use and understand by the user than LISP.

Because it is intended to be used by domain experts rather than programmers, the language has been designed to minimize functionally nonessential details, such as data type declarations, entry declarations, etc. These features are more appropriate in languages which are intended to handle quantitative problems. TREES does possess quantitative capabilities, but emphasizes more the manipulation of the data structures.

TREES possesses the following capabilities:

- variables
- logical operators
- keywords
- trees data structure
- functions
- statements
- input/output
- iteration and recession.

In addition to the above data structures of variables and trees, arrays are supported.

7.4 Task Results

• TREES is an interpretive language. It does have a pseudo-compiler, but I'm not sure how much performance increase it gives.

• TREES requires the programmer to conceptualize the scheduling and/or resource allocation differently than used, as far as programming data structures in concerned.

• The tree data structure is very well suited for the scheduling and resource allocation problems.

• The language is PC based which gives it a broader range of applications and use.

• TREES possesses many FORTRAN similarities. For the scientific community this will make it easier to develop the basic skills of the language. However, it may eliminate the advantage of using the tree structure, because the user will tend to use the programming techniques that he/she already knows. In most cases, FORTRAN programmers use arrays.

• You can accomplish the same results using LISP or other unstrucured list languages, as far as programming is concerned.

• TREES syntax is not as friendly or transparent as the developers lead you to believe. Sophisticated techniques would require a great deal of programming ability.

• It is recommended that TREES not be used for the development of a scheduling system. This is based on a demonstration of the software and conversation with Avyx personnel. It is believed that the number of nodes that can be generated with the current version of TREES is a serious limitation. To give you an example, TREES would not be able to solve the 18 experiment problem because of the node limitation.

• It is recommended that TREES be used for conceptualizing scheduling and resource allocation problems. Ideas that individual Mission Planning personnel may have about scheduing and/or resourced allocation problems could be tested using TREES to better understand the issues involved. This is based on the fact that the data structures in TREES are very well suited to these types of problems and on the similarities to FORTRAN. This similarity will allow most user to learn the language a little easier. However, there is one caveat. All users should

be required to conceptualize and develop their applications utilizing the tree structure of TREES and not arrays that are typically used in FORTRAN.

8.0 Software Data Structure Conversion

8.1 Task Statement

The purpose of this research was to continue to examine the advantages and disadvantages of using object oriented programming techniques to assist in solving the scheduling/resource allocation problem that is particular to MSFC NASA Mission Planning. This is further targeted to the future problems associated with activity planning for the Space Station.

In the first Interim Report (UAH Research Report JRC 90-07) a detail description was given on a prototype software system called the Two Pass - Multiple Resource Allocation Program. Although this system was developed in Common Lisp on a Symbolics Lisp Machine, the full power of object oriented programming techniques had not been utilized. It was decided that this software should be modified in such a manner that the data could be represented in object form.

8.2 Task Conditions

The conditions of this task are that the prototype was developed on a Symbolics Lisp Machine and that the object-oriented paradigm (Flavors) that is presently supported by this platform was appropriate. As with the original prototype design, the system focused on time and resource constraints and excluded consideration of inter-experiment dependencies.

Although the object-oriented programming (OOP) paradigm has been discussed as with all personnel involved in this current research effort, a general review of these principals may be beneficial. OOP has been steadily gaining acceptance as an alternative software design methodology, especially for large, distributed systems. OOP techniques have proven most useful in applications that can be visualized as a collection of objects of distinct classes, each with their own data and processing requirements, that must collaborate for the system as a whole to

function properly. As an analogy, consider a team of engineers working together to design a new car. Those responsible for the interior may be interested in ergonomic data for their work, whereas those designing the engine may be using fuel efficiency data, EPA requirements, and so on. But both groups must work together to decide, for instance, whether the engine will be in the front or the back. For this type of problem, then, each individual can operate with a large degree of autonomy, as long as they collaborate when necessary. Now imagine trying to specify an "algorithm" for designing a car -- step by step instructions explaining exactly what needs to be done and when. That sounds pretty difficult, but suppose we concentrate on the car first and think about *its* organization rather than that of the design process. We can easily break the car down into a hierarchy of subsystems (like maybe the fuel system, and below that the fuel injection and fuel storage subsystems, and so on), until the leaves of our hierarchical tree are individual parts, whose design we *can* specify. Now we have a tree containing not only structural information about the car, but also procedural information about designing it. We will have been given some design parameters describing, probably in general terms, what kind of car we should design, so now we need only fill those values in and filter them down through the tree, until a concrete design begins to take shape. So, in this case, it would seem easier to concentrate on the *object* first, rather than the *process*.

In contrast to this problem, however, consider the task of building the car once it has been designed. The assembly line approach has proven to be the best solution here, since each process is so tightly bound to the output of the previous process and the input of the next process. In this analogy to conventional programming, the car being built is like a large data structure being passed to one processing unit after another, in sequence, until it is finished. It's not difficult to write down an "algorithm" for making a car, so it would probably be better to concentrate on the *process* rather than the *object*. Unfortunately, most real-world problems, including the resource allocation problem, are not as well defined as an automobile assembly line. For these more interesting problems,

it has become clear that we need a new, more natural, way to think about writing programs.

These examples explain why OOP makes it easier to conceptualize the automated resource allocation system, but there are many other advantages as well. Consider the problem of information presentation. We have said that it may be beneficial to present procedural information differently, depending on the user's cognitive presentation biases. Remember that in OOP we construct a hierarchical tree containing not only structural information, but procedural information (ie., code) as well. So when we want to present a step in a procedure, for example, we simply activate the little piece of code, attached to that step, that tells us how it should be presented, given the current user's preferences. This organization becomes particularly efficient when we consider that we may ask for a presentation of that step in hundreds of locations throughout the system.

8.3 Task Approach

The approach taken in this task was to create flavor objects that would represent the resource allocation data and modify the actual software system itself to access and utilize this new data structure. The data representation of both the resources and the activities (experiments) were converted from its original list structure to this object format. The resource object structure is shown in figure 1 and the activity object structure is shown in figure 2. Appendix A contains the actual Lisp computer code (or Flavor definitions) for each of the object structures.

As a consequence of the data structure change many of the data accessing functions had to be changed. In Lisp a list is similar to an ordered set in that each item (or atom) contained in that list occupies a particular position with in the list. However, accessing information from the list is very dependent on each piece of data being precisely in a specific position in the list. To retrieve the fifth data item, the software would be required to pass over the first four items until it arrived at the desired location. This is

obviously not the desired mechanism for data retrieval. It limits the ability of the system programmer to modify the data structure or the procedures the access the specific pieces of information.

As stated earlier, using resource and activity objects allows for data abstraction and encapsulation. This means that the system designer can now freely modify procedures and specific data items. In the original prototype, in an attempt to improve on a ordinary list structure, a property list was utilized. This allowed the user to more freely access the information by providing some degree of abstraction. However, internally the system still was storing the information in list form. The conversion in the second prototype from this property list to flavor objects allowed complete encapsulation and departure from from the internal list structure.

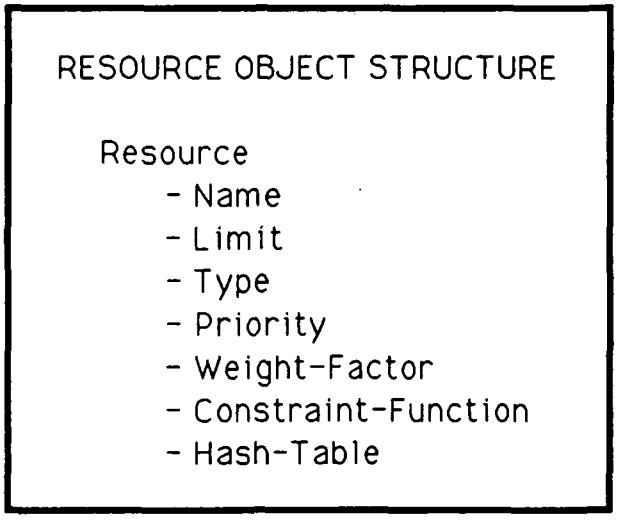


Figure 4

The resource objects are instances of the flavor resource which is the generalized description of a generic resource. The flavor structure provides slots called instance variables that can contain information about the flavor instances. Each individual resource is an individual flavor instance whose slots contain information that uniquely describes its properties and behavior. The instance variables for the resource objects are the resource name, limit, type, priority, weight-factor, constraint-function, and hash-table. A description of each of these instance variables is provide below.

Name - The actual name of the resource (ie. Man-Power).

Limit - The maximum available quantity of this resource at an instance of time.

Type - Is the resource non-depletable, depletable, or replenishable.

Priority - Used in the current maximization algorithm to order resources (ie. primary, secondary, etc...)

Weight-Factor - Will be used in future implementation to arrive at better overall resource utilization.

Constrain-Function - mathematical expression that describes the constraining factors for the resource.

Hash-Table - contains a historical hash table that shows resource utilization as a function of time.

Currently, the software system allocates the resources Power and Man-Power. However, there is no limitation on the number of resources that can be allocated.

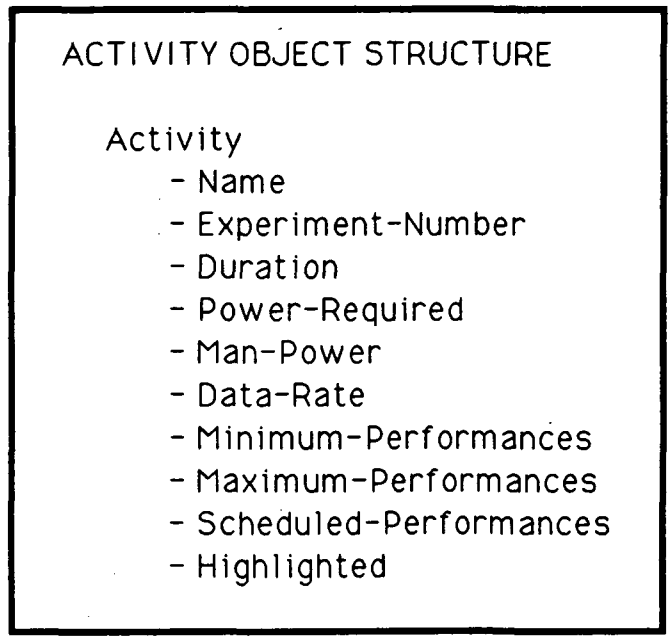


Figure 5

Activity objects, similar to the resource objects, are individual flavor instances of the flavor activity. They have their object definitions contained in instance variables. The activity object's instance variables are the activity name, experiment-number, duration, power-required, man-power, data-rate, minimum-performances, maximum-performances, scheduled-performances, and highlighted. A description of each of these instance variables is provide below.

Name - the name of the activity.

Experiment-Number - An activity identification number (if specified)

Duration - the time required to complete the activity.

Power-Required - the instantaneous power requirements of the activity.

Man-Power - the instantaneous personnel requirements of the activity.

Data-Rate - the instantaneous data production rate of the activity.

Minimum-Performances - the requested minimum number of activity performances.

Maximum-Performances - the requested upper limit of number of performances.

Scheduled-Performances - the actual number of performances of the activity that have been scheduled.

Highlighted - the current state of the the menu item, showing if this activity is currently selected.

8.4 Task Results

The data structure changes described in the preceding sections were performed on the prototype resource allocation software system. Additional testing is needed to determine the extent of any performance gains. Also, software procedural changes need to be implemented in the form of flavor methods instead of traditional function calls. This additional change will allow the flavor instance variables to be directly accessed by the procedural code used in the software system.

The use of hash-tables as a means of storing the time history of the resource allocation process, as well as individual resource utilization, has proven to be an effective and easily manipulative means of storing this information. The graphics functions in the software simply traverses the time line and remove specific values from the tables. Therefore tabular and graphical representations of the results are made easier to obtain.

9.0 Software Functionality Modifications and Enhancements

9.1 Task Statement

The purpose of this research project was to continue the development of the resource allocation system prototype. After a performance review at the end of the first interim term, it was decided that it would be desirable to add additional capabilities to the prototype software. First, the general algorithm that was in use should be modified from a multiple performance allocation to a single step performances approach. Secondly, since the allocation results are distributed across a time line, it would be desirable to construct a mechanism that would allow the operator to interject at a specific point in time and make a change to the allocation. The system should then perform a re-allocation of the resources starting at that point on the time line.

9.2 Task Conditions

The prototype software resides on a Symbolics Lisp Machine. Any modifications to the software were designed solely for the use on this platform and may not easily be ported to other platforms. Also, the data structures of the software were pre-existing and were not modified in the modification process.

9.3 Task Approach

Although a general description of the resource allocation software system's allocation algorithm is described in detail in the previous Interim Report (UAH Research Report JRC 90-07), it may be beneficial to include a brief description of the original resource allocation algorithm. The original algorithm employed by the prototype system would scan the multitude of combinations of activities selecting a single combination that best utilized a primary resource. The system then immediately allocated the entire

number of minimum requested performances (if possible) for each activity that was included in the selected combination of activity performances for that time slice. This therefore treated the minimum requested number of performances as one singular and continuous performance. The allocated activities were then removed from consideration in future allocation combinations during pass one of the system. This approach, although simple, demonstrated many short comings and was deemed too coarse.

The modified approach reduced the allocation step size by only allocating a single performance of each of the activities in the selected combination instead of the original entire minimum number. Each of the activities minimum requested number of performances was then reduced by one. Unlike the original prototype, the activity remained in the pass one allocation process until it had exhausted its requested minimum number of performances instead of immediately being removed.

In a similar manner pass two operations were changed. Although it may be less obvious, pass two attempted to allocate multiple performances of different activities when ever possible. Now single performances of each selected activity were performed.

The backtracking capability was created to allow the operator to effect changes to the allocation process. As the system allocated the resources to the activities a rough schedule is produced. Often as the grouping of activities process is being performed, multiple groups of activities are found that have near equal overall resource utilization. Since the choice of a single group from a list of similar groupings is completely arbitrary, the computer would simply take the first member in the list. This selection was then placed on the agenda for allocation. Although in the immediate time frame the selection method seems just as valid as any other method for choosing a candidate from the group of possible candidates, the selection can cause major changes in future allocation groupings. Therefore it was deemed desirable to construct a mechanism that would allow some user control over the candidate selection process.

The backtracking functions required access and control of three data histories. First, a running history of the actual groups of

possible alternative allocation selections had to be constructed in order for the software system to be able to show possible backtracking choices. Secondly, the resource utilization history for each of the resources needed resetting for future reallocation. And finally, the activity schedule had to cleared of future scheduled items. All of these data histories were in the form of hash-tables.

The data structures were reset for downstream reallocation. Although each of the data structures were hash-tables that use the allocation time as their key words; the downstream resetting requirements were not the same for each table. For instance, it became necessary to swap the newly selected group for the previous group first. Then, the correct resource utilization and new time history could be calculated. All the downstream activities were then removed and their corresponding number of scheduled events reduced. The time history that was used as the key words to the hash-tables was deleted from the point in time of the backtracking. A new resource allocation process is then started from the point of backtracking.

The backtracking process is initiated by selecting a mouse sensitive item from the display. This display shows the allocation time and the current items allocated at that time. It is the time item that is mouse sensitive. Selecting a time for backtracking causes a menu of group selections from which the user must select an alternative. The reallocation process then begins and the display is refreshed. The system is cyclic in that the user may backtrack as many times as is desired. However, the system is a two pass system. Once the results from pass one have been accepted, the user can only backtrack through pass two allocations.

9.4 Task Results

The software system was modified from a multiple allocation to a single allocation step process. The modified Lisp code is provided in Appendix F. The system, at least under limited evaluation, performs a better overall resource allocation based on resource utilization than the previous approach. However, this comes with a

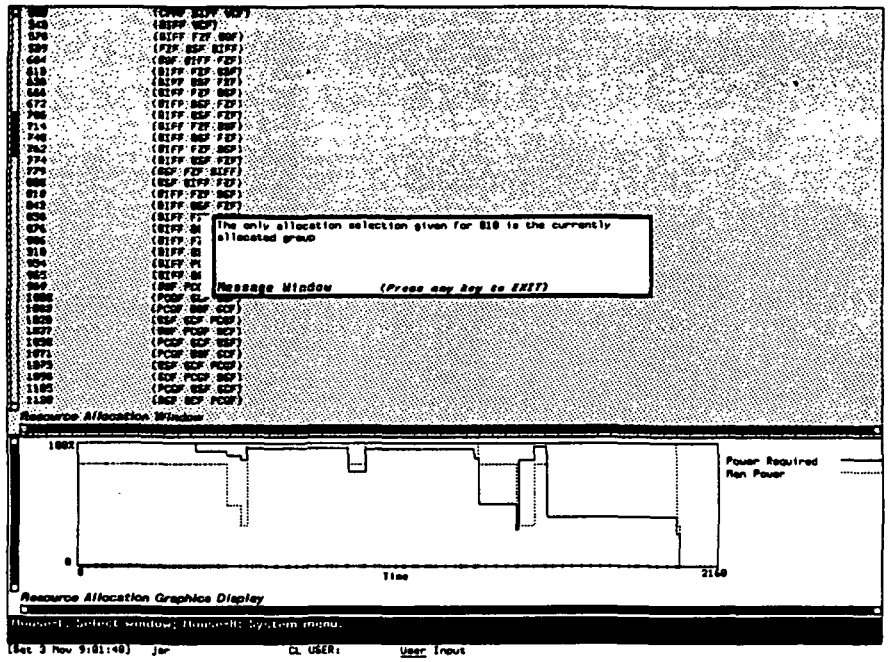


Figure 6

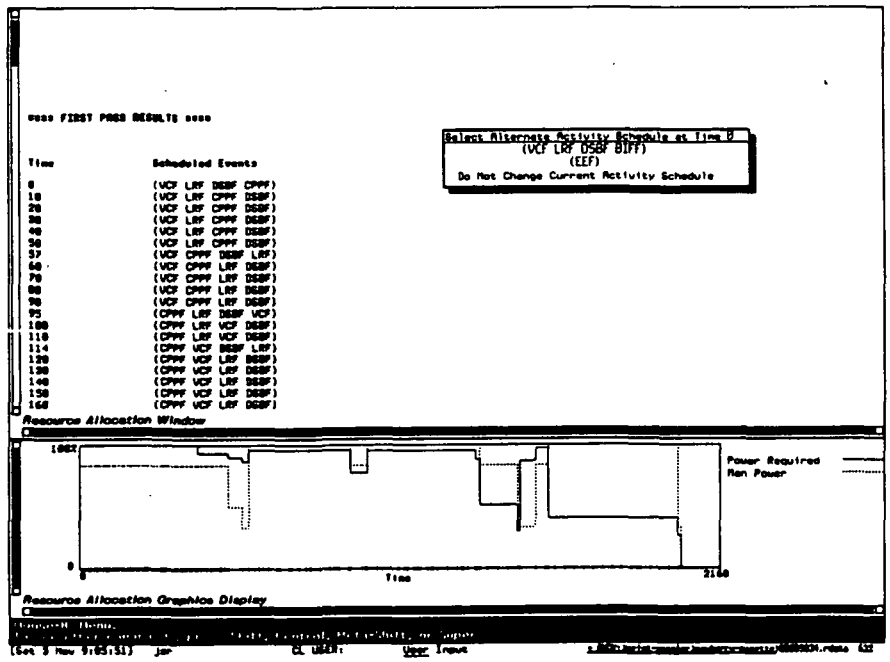


Figure 7

price. The system which was already under criticism for the time requirements necessary for non-trival problems was slowed even more. The exact amount of this reduced allocation speed has not yet been quantified. This will magnify the necessity for evaluating new group selection techniques.

The backtracking capabilities have been implemented in the system with good success. The user can modify the activity schedule and effect changes on the resulting overall resource allocation. Remember the software system is currently designed as a two pass system. As mentioned earlier each of the two passes are considered as being independent of the other for backtracking. Thus the effects of backtracking are confined to the current pass of the system

Since the resetting process is relatively small when compared to the overall problem of resource allocation, the incremental time used in backtracking is not significant. However, in a dynamic environment such as Lisp, the released data or garbage as it is sometimes called can cause the system itself to slow. This effect can be seen if repeated backtracking is performed. If excessive amounts of backtracking and reallocation cycles have been performed the system's performance is substantially affected.

10.0 Portability of Resource Allocation To A TI MicroExplorer

10.1 Task Statement

The purpose of this research was to investigate the performance of the resource allocation software on the TI MicroExplorer platform. At the interim review of the software prototype. It was determined that portability and varying platforms for the system should be investigated. The system was easily ported to a MacIvory system and performed comparable to the Symbolics Lisp Machines. Since the Mission Planning Group at MSFC had a TI MicroExplorer, it was decided that the software system would be ported to this platform and a performance evaluation performed.

10.2 Task Conditions

The development language of the TI MicroExplorer is Common Lisp. The ported software system therefor was limited to the domain of functionality of this platform.

10.3 Task Approach

Since the Symbolics Lisp machine was the original development platform for the Resource Allocation Software System, any functions that were utilized within the system that were specific to this platform had to be modified or replaced by functions that were compatible with the TI MicroExplorer. Although the TI MicroExplorer uses a Flavors System similar to that of the Symbolics, it is currently several generations behind in its development. This in most cases did not pose a tremendous problem. However, the windowing system employs a different type of flavor. There is no predefined, so called "dynamic", window that allows scrolling, graphics, etc... Therefore, a composite flavor that would

cause the TI MicroExplorer windows to behave similarly to those on the Symbolics Lisp machines had to be constructed.

Mouse sensitivity is another facility that the TI MicroExplorer does not easily provide. This causes problems in the Activity and Resource Editing Module of the software system since it relies so heavily on complicated procedures that are initiated via mouse gestures and selections. Since this is a non-essential portion of the software system this module was omitted from the initial implementation of the software on the TI platform. Also the backtracking capabilities while included in the software were inhibited from operation due to similar mouse sensitivity problems. Both of these modules of the software system will be added for this platform.

10.4 Task Results

The software has been ported to the TI MicroExplorer. Additions and modifications were produced that allow the system to function on this platform. The analysis of the performance of the overall Resource Allocation Software system remains incomplete at this time. Mouse sensitive parts of the system that were omitted in the initial implementation of the software system will be added. A complete transfer of all data files is needed and an evaluation of the systems performance on this platform conducted. These activities are proposed as part of a continuing research effort.

11.0 Frontier of Feasibility Software System

11.1 Task Statement

Experimentation in space is rapidly becoming one of the most exciting areas in science. Experiments from such widely diverse areas as medicine and metallurgy are performed side-by-side onboard space-based experimentation platforms. The Space Shuttle is currently the workhorse of this effort, but NASA's Space Station Freedom will assume much of this task when it is constructed.

Each experiment or activity to be performed onboard a platform has certain resource and time requirements. Since the platform has only a limited supply of resources available, these activities are in competition with one another. Determining which activities can be performed is a complex problem that due to its nature has multiple solutions.

It is likely that multiple performances of a single experiment are desirable, therefore, each such experiment must be performed multiple times during the mission duration. One method for simplifying the solution set of this problem is to generate a number of possible solutions based solely on resource and time constraints for use with a scheduling program. It is therefore the purpose of this research to examine the techniques for arriving at theses possible solutions.

11.2 Task Conditions

The prototype software resides on a Symbolics Lisp Machine. Any modifications to the software were designed solely for the use on this platform and may not easily be ported to other platforms. The prospective of the system is to view the possible starting points of a scheduler without taking into consideration any intra-activity or temporal constraints.

11.3 Task Approach

The Frontier of Feasibility System is designed to generate "good" starting points for a scheduling program. This system is not a scheduler, but is instead a resource allocation program which operates at a very course level of granularity. A scheduling program is concerned with placing activities on a time line, while ensuring that no constraints are violated. The main thrust of a scheduling package is the ordering of the activities on the time line. The Frontier of Feasibility System does not attempt to establish a time line schedule, but instead, only attempts to generate starting points for a scheduling program by allocating the available resources. The Symbolics Lisp code listing is provided in Appendix G.

Activities

Experimentation is not the only consumer of resources onboard a platform. Life support, instrumentation, and other onboard systems are also in competition for the available resources. For this reason, in this paper competitors for resources will be referred to as activities. Each activity is defined by its consumption of various resources, duration, and performance criteria.

Activities are given an abbreviated name and an experiment number. Duration is perhaps one of the most important facts given in the activity description. It is assumed that two or more performances of a single activity cannot occur simultaneously. However, it is possible for several different activities to be operating at the same time, resources allowing. Therefore, by taking the mission duration and dividing it by the duration of a single performance of an activity, it is possible to arrive at a hard constraint on the maximum number of performances possible for an activity.

The activity description also includes resource usage information. This lists the amount of each resource that will be required to perform that activity one time. It is assumed in the

Frontier of Feasibility System that this resource usage is continuous throughout the duration of the activity. This is not an accurate representation of reality, but the purpose of this system is to provide a good starting point for a scheduler, not a finished answer.

The user also enters a minimum requested and maximum desired number of performances for each activity into the description. This provides the system with a minimum number of performances of each activity that must be scheduled to meet the user's bottom line. Any remaining resources are then allocated among the activities. The maximum desired number of performances places an upper limit on the number of performances of an activity that will be scheduled. This prevents the system from allocating resources to useless activity repetition. The upper limit established by the user is verified by the system to ensure that it is feasible.

```
( VCF  (experiment-number  (2))
       (power-required  (10))
       (duration  (1))
       (performances  (1))
       (max-performances  (4))
       (scheduled-performances  (0)))
```

Figure 8. A representation of an activity as a Lisp list.

Resources

The resources available aboard the platform are each given an abbreviated name and an amount available. Resources can be classified into several different categories. Non-consumable resources are not depleted by use, and are available in a constant quantity for the duration of the mission. Consumable resources have an initial level which is depleted as activities are performed.

Replenishable resources are those that can be temporarily depleted, but which through processes onboard the platform, may be replenished during the mission.

The current version of the Frontier of Feasibility System uses one resource during its search process. Versions currently in development examine the problem using multiple resources.

Graphical Representation of Search Space

The Frontier of Feasibility System is based around the idea of representing the resource allocation problem's possible solutions as a tree graph. The process of creating a feasible combination of activity performances can be easily demonstrated using a tree graph. A manager's decisions about which activity to perform more times can be followed down a path on the tree.

For instance, if the manager decided to add one performance to the right-most activity, the node created would be one further down the right-hand-side branch. From this new node, the manager will make another decision regarding which activity to increase next. This process is repeated until the manager is satisfied with the results. Therefore, we adopted this structure as a good reference frame when seeking ways to calculate a solution set more quickly.

Tree Structure

Each node on the tree graph represents one possible combination of activity performances. An example root node would be (1 1 1), representing one performance of three different activities. The children of this node would be (1 1 2), (1 2 1), and (2 1 1). Each child represents its parent with an additional performance of one activity. Only certain activities can be modified on each branch. The first, left-most, branch allows the modification of all activities. On the other branches, only the activities to the right of the activity corresponding to the branch number can be modified. For instance, in a twelve activity problem, if you are looking at the fifth branch, only the fifth through twelfth activities can be

modified. The first four activities remain at their minimum requested.

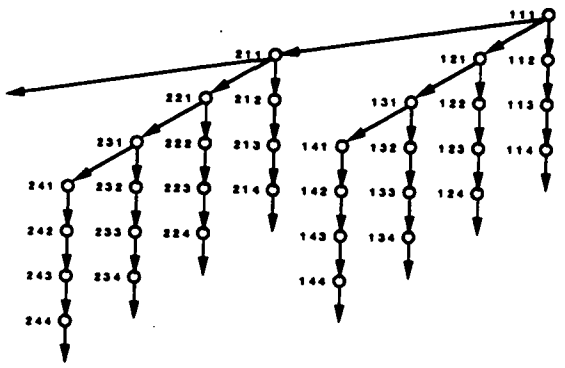


Figure 9. A three activity tree graph.

When dealing with a large number of activities, each of which can be performed multiple times, the size of the tree becomes quite large. It is therefore necessary to devise methods for reducing the size of the search space. One of the simplest is to make the root node values equal to the minimum number of requested performances of each activity. This action can greatly reduce the size of the space that must be searched. Since each activity also has a maximum number of performances requested, it is possible to restrict the depth of the tree.

A human manager makes decisions, in terms of the tree graph, by starting at the root node and moving down the tree from parent to child, until he can go no further due to constraints. A node to which no more performances of any activity can be added without violating a constraint is said to be a Frontier Node, commonly referred to as a

leaf node. The Frontier Nodes fall along a barrier which we call the Frontier of Feasibility. It is the nodes that fall along the Frontier that offer the best starting points for a scheduling program.

Sorting the Activities

It is important to realize that the ordering of the activities within the nodes affects the shape of the tree. Each activity has a range of possible performances from its minimum requested to its maximum desired. Typically, the activities with a large range use a small amount of resources, while those with a very narrow range use large quantities of resources. If the activities are sorted so that the largest range is on the left, and the smallest on the right, then the tree will be very wide. This is because each new performance of the first activity represents a new branch. If the activities are sorted in reverse order, from smallest to largest range, then the tree will be deeper and narrower. In this case, there will only be a few branches to the left, thereby restricting the width of the tree.

Which sorting method is best is still being decided. Each method has its advantages and disadvantages. The second method narrows the width of the tree, and thereby the number of Frontier Nodes. But this method makes the calculations for trading between activities more cumbersome. Method one, although it has a larger Frontier, has an easily demonstrated process for handling trades. So, for the purposes of this paper, we will be discussing the problem in terms of the first method, largest to smallest range.

State Space Search Methods

There are many different search methods available which could be used to find the possible solutions to this problem. These are methods which have been developed over time to handle problems similar to the Space Station resource allocation problem. However, most of these methods were developed to seek an optimal solution, or a single answer. Since the purpose of the Frontier of Feasibility

System is to generate several "good" starting points for a scheduler, many of these methods were ruled out.

Modified Breadth Search

It was decided that none of the other regular search methods would complete the search in an acceptable length of time. The structure of the tree suggested a new search method. The Frontier Node of the right-most branch is easily calculated, since only the number of performances of the right-most activity can be changed. Simply, divide the resources remaining after all activities have been performed their minimum requested number of times, by the amount of resources necessary for the right-most activity. This calculation yields the number of performances which can be added to the minimum requested. By adding this number to the right-most minimum and combining this new total with the rest of the root node, we have calculated the right-most Frontier Node.

Using this Frontier Node as a starting point, it is possible to cross the tree along the Frontier of Feasibility, thereby eliminating the need to search the tree in depth. As discussed earlier, the order in which the activities are sorted can greatly affect the search process. We have chosen to discuss the largest to smallest range sort method because it can be more clearly demonstrated in the context of this paper. Using this method, the first frontier node that we have just calculated has maximized the number of performances of the largest resource using activity.

The Frontier search method is composed of six main steps:

1. Examine the number of performances of each activity in the node, from left to right, for one which is performed more than the minimum required number of performances. This step begins its examination at the second node from the left, because of the way Step 5 operates.

2. Reduce the current number of performances of that activity by one.

3. Reset all activities to the left of the activity found in Step 1, to their minimum required number of performances.

4. Recalculate the available resources.

5. Starting just left of the activity found in Step 1 and continuing to the left, increase the number of performances of each activity as much as possible with the available resources. Each new performance reduces the amount of resources available.

6. When no more performances can be added, store the new Frontier Node and repeat the process.

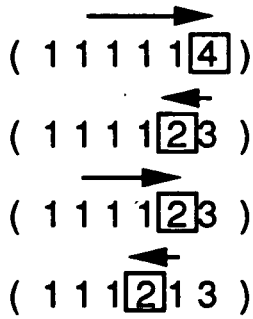


Figure 10. Example of the six stage process.

The benefit of using the largest to smallest range sort method is that removing one performance of an activity in Step 3, guarantees

at least one performance of another activity when executing Step 5. This method sorts the activities from smallest to largest resource users and thereby ensures that enough resources are freed up to add one performance to the left.

11.4 Task Results

The six stage process describe above produces several hundred thousand solutions in a small problem. Almost all of these Frontier Nodes utilize from 95% to 100% of the available resources. There are several possible mechanisms under consideration to select only a small subset of these solutions. One of the most promising of these, reduces the size of the solution set by selecting a starting node further to the left in the tree. This eliminates all branches right of the start node from consideration. Random sampling is another method which could be used. The system would randomly, or at set intervals, store the node currently under consideration. This method would provide a smaller solution set, which still represented most of the branches.

While the system can calculate new nodes fairly rapidly, storage of the growing solution set slows the systems performance to an unacceptable level. This problem can be bypassed in several ways, for instance, by only storing those solutions that use 100% of the available resources or only the first 10,000 solutions which are generated.

From the generated solution set, the user must choose a node that represents a "good" starting point. We are currently working on an interface which will allow the user to review the solution set and examine a node in detail. The user would be able to modify the number of performances of any activity, in order to improve the "goodness" of the node. The combination of these two systems will provide the user with a powerful tool for generating rough solutions to the resource allocation problem.

12.0 Conclusions

1. The object-oriented programming techniques would be too cumbersome for handling complete mission data set. This is based on the manner in which the problem was modeled in the prototype developed. In this prototype, everything was treated as an object and the mission timeline was divided into seconds. If the timeline is handled in a different manner; that is not as an object, then the object-oriented approach may be very feasible. The object-oriented approach should not be eliminated without further study.
2 KEE is not suited for the scheduling nor resource allocation problem. This is because of the extensive amount of code that needs to be developed to handle efficienctly the bookkeepping procedures. While it is possible to write these functions in KEE, a significant increase in execution time will be experienced. This may not be satisfactory for the decision makers.
3. Ethernet is the most feasible way of connecting Lisp machines and VAX for MSFC Mission Planning personnel at present.
5 It is not possible to call Lisp from inside FORTRAN and vice versa on a VAX.
6. Resource allocation algorithms show much potential. More heuristics for increasing the efficiency of the search process need to be developed and studied before ruling this approach out completely.

Appendix A
Code Listing, for Object-Oriented Programming Task

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

;*****************************************************************************
;;; top level function to call others
(defmethod (correct-representations-and-build-linkages-after-data-load mission) ()
  ;;;operations on experiment, performances, steps
  (maphash #'(lambda (exp-name exp)
               exp-name
               (correct-time-representation exp))
           experiment-template-table)
  (get-possible-crew-combinations-for-all-steps self)
  (write-crew-lockin-to-step-level self)
  (replace-names-with-objects self)
  (transfer-experiments-from-template-table self nil)
    ;;;operations on resources
  ;(connect-resource-availability-start-and-end-times init-obj)
  (transfer-shift-times-to-crew-members init-obj)
  ;;;operations on time slices
  (initialize-time self))

(defmethod (write-crew-lockin-to-step-level mission) ()
  (maphash #'(lambda (exp instance)
               exp
               (write-crew-lockin-to-step-level instance))
           experiment-template-table))

(defmethod (write-crew-lockin-to-step-level experiment) ()
  (loop for (lockin-start lockin-end) in crew-lockin
        for crew-combo = (crew-combinations (find-step-numbered self lockin-start))
        do
    (loop for step-number from lockin-start to lockin-end
          for step = (find-step-numbered self step-number)
          do
      (when (and (null (crew-monitor step))
                 (equal crew-combo (crew-combinations step)))
        (setf (crew-lockin step) lockin-start)))))

(defmethod (initialize-time mission) ()
  (build-initial-time self)
  (load-targets-into-time-steps init-obj)
  (load-attitudes-into-time-steps init-obj)
  )

(defmethod (restore-data-to-start mission) ()
  (setf experiment-table (make-hash-table))
  (transfer-experiments-from-template-table self nil)
  (initialize-time self))

(defmethod (replace-names-with-objects mission) ()
  (maphash #'(lambda (exp instance)
               exp
               (replace-names instance))
           experiment-template-table))

(defmethod (replace-names experiment) ()
  (loop for slot in '(startup-steps shutdown-steps prototype-step-list )
        do
    (mapc #'replace-names (symbol-value-in-instance self slot) )))

(defmethod (replace-names step) ()
  (loop for slot in
            '(consumable-resource-list durable-resource-list)
        for keyword in '(:consumable :durable)
        do
    (setf (symbol-value-in-instance self slot)
        (loop for (resource-name quant) in (symbol-value-in-instance self slot)
              collect (list (get-object-named (init-obj *mission*) keyword resource-name)
                            quant))))
  (setf non-depletable-resource-list
        (loop for (resource-name quant tolerance) in non-depletable-resource-list
              collect (list (get-object-named (init-obj *mission*)
                                              :non-depletable resource-name)
                            quant tolerance)))
```

```
  (setf crew-combinations
        (loop for combination in crew-combinations
              collect
              (loop for crew in combination
                    collect (get-object-named (init-obj *mission*) :crew crew))))
  (setf target-list
        (loop for target in target-list
                    collect (get-object-named (init-obj *mission*) :target target)))
  (setf attitude-list
        (loop for attitude in attitude-list
                    collect (get-object-named (init-obj *mission*) :attitude attitude))))

;*****************************************************************************
;;;functions to build linkages
(defmethod (transfer-experiments-from-template-table mission) (&optional (query t))
  (let ((experiment-list nil) (instance-list nil))
    (when query
      (maphash #'(lambda (key instance)
                   (push key experiment-list)
                   (push instance instance-list))
               experiment-template-table)
      (setf query nil)
      (loop until (setf query
                        (dw:menu-choose
                          '(("Use All Experiments" :all)
                            ("Use None of These Experiments" :none)
                            ("Use Some of These Experiments - Present Menu" :some))
                          :prompt (format nil " ~A " experiment-list)))))
    (cond ((or (null query) (eql query :all))
           (maphash #'(lambda (key instance)
                        (setf (gethash key experiment-table) (copy-self instance))
                        )
                    experiment-template-table))
          ((eql query :none) nil)
          ((eql query :some)
           (format t "this is a stub in transfer-experiments-from-template-table")))))
#||
(defmethod (connect-resource-availability-start-and-end-times nasa-init-obj) ()
  (loop for slot in '(consumable-resource-list non-depletable-resource-list)
        do
    (loop for resource in (symbol-value-in-instance self slot)
          do
      (connect-resource-availability resource)
           (when (and (eql (length (quantity-availability-list resource)) 1)
                      (null (qty (first (quantity-availability-list resource))))
                      (maximum-available resource))
        (setf (qty (first (quantity-availability-list resource)))
              (maximum-available resource))))))

(defmethod (connect-resource-availability non-durable-resource) ()
    (cond ((and (null maximum-available)(null quantity-availability-list)) nil)
          ((null quantity-availability-list)
           (setf quantity-availability-list
                 (ncons (make-instance
                          'quantity-availability
                          :name (name self)
                          :owner-obj self
                          :available-times-list
                          (ncons (make-instance
                                   'available-time
                                   :begin 0
                                   :end (max-time (init-obj *mission*)))))))
           (setf (owner-obj (first (available-times-list (first quantity-availability-list))))
                 (first quantity-availability-list)))
          (t (let ((time-list nil)(time-length nil)(max-quant 0))
               (loop for quantity-availability-obj in quantity-availability-list
                     do
                 (when (> (qty quantity-availability-obj) max-quant)
                     (setf max-quant (qty quantity-availability-obj)))
                 (loop for object in (available-times-list quantity-availability-obj)
                       do
                   (push (begin object) time-list)))
               (setf time-list (sort time-list #'<))
```


C-2

```
              (setf time-length (1- (length time-list)))
              (setf maximum-available max-quant)
              (loop for quantity-availability-obj in quantity-availability-list
                    do
                (loop for object in (available-times-list quantity-availability-obj)
                      for time-position = (position (begin object) time-list)
                      do
                  (if (eql time-position time-length)
                      (setf (end object) (max-time (init-obj *mission*)))
                      (setf (end object) (1- (nth (1+ time-position) time-list))))))))))
|||#

(defmethod (transfer-shift-times-to-crew-members nasa-init-obj) ()
  (loop for crew in crew-list
        do
    (setf (available-times-list crew)
          (copy-available-times-list self (work-shift crew)))
    (loop for available-time-obj in (available-times-list crew)
          do
      (setf (owner-obj available-time-obj) crew))))

(defmethod (copy-available-times-list nasa-init-obj) (shift-number)
  (loop for available-time-obj in (nth (1- shift-number) shift-availability-objs)
        collect (make-instance 'available-time :begin (begin available-time-obj)
                               :end (end available-time-obj))))

(defmethod (build-initial-time mission) ()
  (setf time-slice-holder
        (make-instance 'time-slice :start-time 0 :end-time (max-time init-obj))))

(defmethod (load-targets-into-time-steps nasa-init-obj) ()
  (loop for target-obj in target-list
        do
    (loop for available-time-obj in (available-times-list target-obj)
          do
      (schedule-event
        *mission* target-obj 'target-list (begin available-time-obj)
        (end available-time-obj)))))

(defmethod (load-attitudes-into-time-steps nasa-init-obj) ()
  (loop for attitude-object in attitude-list
        do
    (loop for available-time-obj in (available-times-list attitude-object)
          do
      (schedule-event
        *mission* attitude-object 'attitude-list (begin available-time-obj)
        (end available-time-obj)))))

;*****************************************************************************
;;; this section is used to convert various time representations to one standard
(defmethod (correct-time-representation experiment) ()
  (setf min-performance-delay-time
        (translate-seconds-to-time-periods
          (translate-time-list-to-seconds min-performance-delay-time))
        max-performance-delay-time
        (translate-seconds-to-time-periods
          (translate-time-list-to-seconds max-performance-delay-time))
        performance-time-window
        (translate-seconds-to-time-periods
          (translate-time-list-to-seconds performance-time-window)))
  (setf performance-windows
        (loop for (begin end performances) in performance-windows
              collect (list (translate-seconds-to-time-periods
                              (translate-time-list-to-seconds begin))
                            (translate-seconds-to-time-periods
                              (translate-time-list-to-seconds end))
                            performances)))
  (loop for slot in '(startup-steps shutdown-steps prototype-step-list)
        do
    (loop for step in (symbol-value-in-instance self slot)
          do
      (correct-time-representation step))))
```

```
(defmethod (correct-time-representation step) ()
  (setf max-duration (translate-seconds-to-time-periods max-duration))
  (setf min-duration (translate-seconds-to-time-periods min-duration))
  (setf step-delay-max (translate-seconds-to-time-periods step-delay-max))
  (setf step-delay-min (translate-seconds-to-time-periods step-delay-min)))

;*****************************************************************************
;;; theses methods and functions are used to setup the possible combinations of crew
;;; members that satisfy the crew requirements specifications of each step

(defmethod (get-possible-crew-combinations-for-all-steps mission) ()
  (maphash #'(lambda (key instance)
               key
               (loop for slot in '(startup-steps shutdown-steps prototype-step-list )
                     do
                 (loop for step in (symbol-value-in-instance instance slot)
                       do
                   (setf (crew-combinations step)
                         (get-possible-combinations-of-crew self (crew-requirements step))))))
           experiment-template-table))

(defmethod (get-possible-combinations-of-crew mission) (crew-requirements)
  (if (gethash crew-requirements crew-combinations-table)
      (gethash crew-requirements crew-combinations-table)
      (setf (gethash crew-requirements crew-combinations-table)
            (generate-possible-combinations-of-crew self crew-requirements))))

(defmethod (generate-possible-combinations-of-crew mission) (crew-requirements)
  (when crew-requirements
    (merge-candidate-sets
      self (generate-candidate-sets self crew-requirements) crew-requirements)))

(defmethod (generate-candidate-sets mission) (crew-requirements)
  (let ((candidate-sets nil))
    (loop for (description-list quant) in crew-requirements
          for description-set = nil
          do
      (loop for (type tag) in description-list
            for possible-set = nil
            do
        (if (eql type 'duty-position)
            (loop for crew-obj in (crew-list init-obj)
                  do
              (when (eql (duty-position crew-obj) tag)
                (push (name crew-obj) possible-set)))
            (loop for crew-obj in (crew-list init-obj)
                  do
              (when (eql (name crew-obj) tag)
                (push (name crew-obj) possible-set))))
        (setf description-set (concatenate 'list possible-set description-set)))
      (push (list description-set quant) candidate-sets))
    candidate-sets))

(defmethod (merge-candidate-sets mission) (candidate-sets crew-requirements)
  (let ((final-combinations nil) (all-combinations nil))
    (cond ((null candidate-sets)
           (error "generate-possible-combinations-of-crew was unable to generate a candidate
                   set with requirements ~S" crew-requirements))
          ((= (Length candidate-sets) 1)
           (setf all-combinations (generate-combinatorics self (first candidate-sets))))
          (t (setf all-combinations
                   (generate-possible-combinations-of-crew-aux
                     self (generate-combinatorics self (first candidate-sets))
                     (cdr candidate-sets)))))
    (loop for combination in all-combinations
          do
      (unless (combination-contains-duplicates-p self combination)
        (push combination final-combinations)))
    final-combinations))

(defmethod (generate-possible-combinations-of-crew-aux mission)
           (existing-combinatoric candidate-sets)
  (if (null candidate-sets)
```

```
      existing-combinatoric
      (generate-possible-combinations-of-crew-aux
        self
        (merge-combinatorics
          self existing-combinatoric (generate-combinatorics self (first candidate-sets)))
        (cdr candidate-sets))))

(defmethod (merge-combinatorics mission) (first-set second-set)
  (loop for grouping-one in first-set
        with result = nil
        do
    (loop for grouping-two in second-set
          do
      (push (concatenate 'list (copy-list grouping-one) (copy-list grouping-two)) result))
    finally (return result)))

(defmethod (generate-combinatorics mission)(candidate-set-and-quant)
  (let ((candidate-set (first candidate-set-and-quant))
        (quant (second candidate-set-and-quant))
        (solution-list nil))
    (loop for i from 1 to quant
          for next-solution = nil
          do
      (if (= i 1)
         (loop for crew in candidate-set
               do
           (push (list crew) solution-list))
         (loop for solution in solution-list
               do
           (loop for crew in candidate-set
                 for combo = (if (member crew solution)
                                 nil
                                 (concatenate 'list (list crew) (copy-list solution)))
                 do
             (when (and combo (new-entry-p combo next-solution))
               (push combo next-solution)))
           finally (setf solution-list next-solution))))
    solution-list))


(defun new-entry-p (combo next-solution)
  (let ((result t))
  (cond ((null next-solution) t)
        (t (loop for set in next-solution
                 until (null result)
                 do
             (when (every #'(lambda (x) (member x combo)) set)
               (setf result nil)))))
  result))

(defmethod (combination-contains-duplicates-p mission) (combination)
  (let ((combination-copy (copy-alist combination))
        (flag nil))
    (loop for crew-obj in combination
          until flag
          do
      (setf combination-copy (cdr combination-copy))
      (when (member crew-obj combination-copy)
        (setf flag t)))
    flag))

;;;end of crew combination generation
;*****************************************************************************
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

;; presentation types associated with nasa-init-obj editing
(define-presentation-type single-valued-nasa-init-obj-edit-display (())
   :history t
   :printer ((obj stream)
      (with-character-style ('(:fix :bold-italic :normal) stream :bind-line-height t)
             (format stream "~%MISSION DURATION ~S ~%MISSION TIME INCREMENT ~S"
                    (max-time obj)
                    (time-inc obj))))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'nasa-init-obj) input
                  (signal 'dw:input-not-of-required-type
                          :type 'nasa-init-obj
                          :string input)))))

(define-presentation-type  misc-obj-edit-display (())
   :history t
   :printer ((obj stream)
      (with-character-style ('(:fix :roman :small) stream :bind-line-height t)
             (format stream "~%~A~%~A~%"(first (display-string obj))
                    (second (display-string obj))))
      )
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'query-obj) input
                  (signal 'dw:input-not-of-required-type
                          :type 'query-obj
                          :string input)))))

(define-presentation-type consumable-name-for-edit-display (())
   :history t
   :printer ((obj stream)
             (with-character-style ('(:fix :italic :normal) stream )
               (format stream "~%   NAME ~S ~%"
                       (name obj) )))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'consumable-resource)
                  input
                  (signal 'dw:input-not-of-required-type
                          :type 'consumable-resource
                          :string input)))))



(define-presentation-type name-for-edit-display (())
   :history t
   :printer ((obj stream)
      (with-character-style ('(:fix :roman :normal) stream :bind-line-height t)
             (format stream "~%NAME ~S~%" (name obj))))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'availabilty)
                  input
                  (signal 'dw:input-not-of-required-type
                          :type 'availabilty
                          :string input)))))

(define-presentation-type quantity-availability-edit-display (())
   :history t
   :printer ((obj stream)
      (with-character-style ('(:fix :bold :small) stream :bind-line-height t)
             (format stream "          QUANTITY = ~S~%" (qty obj))))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'quantity-availability ) input
                  (signal 'dw:input-not-of-required-type
                          :type 'quantity-availability
                          :string input)))))

(define-presentation-type durable-resource-edit-display (())
```

```
   :history t
   :printer ((obj stream)
             (if (send stream :operation-handled-p ':format-cell)
                 (progn
                   (formatting-cell (stream :align :center) (format stream (name obj)))
                   (formatting-cell (stream :align :center)
                     (format stream "~S" (available-quantity obj))))
                 (format stream "#<DURABLE RESOURCE EDIT DISPLAY ~S ~S>"
                         (name obj) (available-quantity obj))))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'durable-resource) input
                  (signal 'dw:input-not-of-required-type
                          :type 'durable-resource
                          :string input)))))

(define-presentation-type available-time-edit-display(())
   :history t
   :printer ((obj stream)
             (if (send stream :operation-handled-p :format-cell)
                 (progn
                   (formatting-cell (stream :align :center) (format stream "~A" (begin obj)))
                   (formatting-cell (stream :align :center) (format stream "~A" (end obj))))
                 (format stream "#<AVAILABLE-TIME-EDIT-DISPLAY ~A ~A>" (begin obj) (end obj))))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'available-time) input
                  (signal 'dw:input-not-of-required-type
                          :type 'available-time
                          :string input)))))
;*****************************************************************************
;;; presentation types associated with editing experiment templates
(define-presentation-type experiment-template-edit-display (())
   :history t
   :printer ((obj stream)
             (format stream "~%MIN-PERFORMANCES ~A MAX-PERFORMANCES ~A DESIRED-PERFORMANCES ~A MAX
-PERFORMANCE-DELAY-TIME ~A MIN-PERFORMANCE-DELAY-TIME ~A" (name obj) (min-performances obj) (max-p
erformances obj) (desired-performances obj) (max-performance-delay-time obj) (min-performance-dela
y-time obj)))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'experiment-template) input
                  (signal 'dw:input-not-of-required-type
                          :type 'experiment-template
                          :string input)))))

(define-presentation-type experiment-template-name-edit-display (())
   :history t :printer ((obj stream)
                        (format stream "~%EXPERIMENT NAME: ~A" (name obj)))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'experiment-template) input
                  (signal 'dw:input-not-of-required-type
                          :type 'experiment-template
                          :string input)))))

(define-presentation-type  step-template-for-editing (())
   :history t
   :printer ((obj stream)
             (present-step obj stream))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'step) input
                  (signal 'dw:input-not-of-required-type
                          :type 'step
                          :string input)))))


(define-presentation-type shutdown-step-template-for-editing (())
   :history t
   :printer ((obj stream)
             (present-step obj stream))
```

```
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'step) input
                  (signal 'dw:input-not-of-required-type
                          :type 'step
                          :string input)))))

(define-presentation-type  prototype-step-template-for-editing (())
   :history t
   :printer ((obj stream)
             (present-step obj stream))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'step) input
                  (signal 'dw:input-not-of-required-type
                          :type 'step
                          :string input)))))


;*****************************************************************************
;;;presentation type associated with editing nasa-screen-manager

(define-presentation-type nasa-screen-manager-edit-display (())
   :history t
   :printer ((obj stream)
             (format stream "~% SCREEN MANAGER")
             (FORMAT STREAM "~% CURRENT RESOURCE NAME: ~A"(current-resource obj))
             (FORMAT STREAM "~% RESOURCE DISPLAY DIMENSIONS")
             (FORMAT STREAM "~% LEFT COORDINATE: ~A, RIGHT COORDINATE: ~A, UPPER COORDINATE: ~A, B
OTTOM COORDINATE: ~A" (left-x obj) (right-x obj) (upper-y obj) (lower-y obj))
             (FORMAT STREAM "~% MINIMUM WIDTH (pixels) EACH TIME PERIOD: ~A; WIDTH EACH TIME PERIO
D: ~A" (min-x-delta obj) (x-delta obj))
             (FORMAT STREAM "~% TIME UNITS BETWEEN HORIZONTAL SCALE MARKERS: ~A"
                     (h-scale-inc obj))
             (FORMAT STREAM "~% UNITS BETWEEN VERTICAL SCALE MARKERS FOR CURRENT RESOURCE: ~A"
                     (v-scale-inc obj))
             (FORMAT STREAM "~% LENGTH OF TICK MARKS ON SCALES: ~A" (scale-length obj)))
   :parser ((stream)
            (let ((input (read-from-string (dw:read-standard-token stream))))
              (if (eql (type-of input) 'nasa-screen-manager) input
                  (signal 'dw:input-not-of-required-type
                          :type 'nasa-screen-manager
                          :string input)))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-CLEAR-INIT-EDIT-HISTORY
                                       :MENU-ACCELERATOR "Clear All Histories"
                                       :MENU-LEVEL :INIT-EDIT-MENU)
    ()
    (clear-all-histories (screen-manager *mission*) 'init-edit))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-FROM-INIT-EDIT-TO-EDIT
                                       :MENU-ACCELERATOR "Return To Obj Edit"
                                       :MENU-LEVEL :INIT-EDIT-MENU)
    ()
    (select-configuration *mission* 'edit))


(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-REDISPLAY-INIT-OBJ
                                       :MENU-ACCELERATOR "Redisplay Init Obj"
                                       :MENU-LEVEL :INIT-EDIT-MENU)
    ()
   (clear-all-histories (screen-manager *mission*) 'init-edit)
     (edit-obj *mission* 'init-obj))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-REDISPLAY-INIT-EDIT-OBJ
                                       :MENU-ACCELERATOR "Redisplay"
                                       :MENU-LEVEL :INIT-OBJ-EDIT-MENU)
    ()
   (clear-history (screen-manager *mission*) 'init-obj-edit )
     (edit-init-sub-obj *mission* 'init-obj-edit))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-REDISPLAY-DURABLE-RESOURCE
                                       :MENU-ACCELERATOR "Redisplay"
                                       :MENU-LEVEL :DURABLE-RESOURCE-MENU)
    ()
   (clear-history (screen-manager *mission*) 'durable-resource-edit)
     (edit-init-sub-obj *mission* 'durable))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-REDISPLAY-CONSUMABLE-RESOURCE
                                       :MENU-ACCELERATOR "Redisplay"
                                       :MENU-LEVEL :CONSUMABLE-RESOURCE-MENU)
    ()
   (clear-history (screen-manager *mission*) 'consumable-resource-edit)
     (edit-init-sub-obj *mission* 'consumable))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-REDISPLAY-CREW-RESOURCE
                                       :MENU-ACCELERATOR "Redisplay"
                                       :MENU-LEVEL :CREW-RESOURCE-MENU)
    ()
   (clear-history (screen-manager *mission*) 'crew-resource-edit)
     (edit-init-sub-obj *mission* 'crew))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-REDISPLAY-TARGET-RESOURCE
                                       :MENU-ACCELERATOR "Redisplay"
                                       :MENU-LEVEL :TARGET-RESOURCE-MENU)
    ()
   (clear-history (screen-manager *mission*) 'target-resource-edit)
     (edit-init-sub-obj *mission* 'target))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-REDISPLAY-ATTITUDE-RESOURCE
                                       :MENU-ACCELERATOR "Redisplay"
                                       :MENU-LEVEL :ATTITUDE-RESOURCE-MENU)
    ()
   (clear-history (screen-manager *mission*) 'attitude-resource-edit)
     (edit-init-sub-obj *mission* 'attitude))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (create-new-obj experiment-template) ()
  (query-user-for-new-values self)
  (add-exp-temp-to-table *mission* self name)
  )

(defmethod (create-new-obj experiment) ()
  (query-user-for-new-values self)
  (add-exp-to-table *mission* self name))

(defmethod (query-user-for-new-values experiment) ()
  (let ((choice nil) (choice-list '(yes no))) choice choice-list
  (dw:accepting-values
      (*standard-output*
        :own-window t :label
        (format nil
                "Input Values For New Experiment"))
      (setf name
            (accept 'symbol :default 'none :query-identifier 'name
                    :stream *standard-output*
                    :prompt (format nil "enter name of experiment"))
            min-performances
            (accept 'number :default 0 :query-identifier 'min-performances
                    :stream *standard-output* :prompt
                    (format nil "enter minimum number of performances   "))
            max-performances
            (accept 'number :default 0 :query-identifier 'max-performances
                    :stream *standard-output* :prompt
                    (format nil "enter maximum number of performances"))
            desired-performances
            (accept 'number :default 0 :query-identifier 'desired-performances
                    :stream *standard-output*
                    :prompt (format nil "enter desired number of performances"))
            min-performance-delay-time
            (accept 'number :default 0 :query-identifier 'min-performance-delay-time
                    :stream *standard-output*
                    :prompt (format nil "enter min performance delay time "))
            max-performance-delay-time
            (accept 'number :default 0 :query-identifier 'max-performance-delay-time
                    :stream *standard-output*
                    :prompt (format nil "enter max performance delay time "))))
  (query-user-for-new-values-aux self)))

(defmethod (query-user-for-new-values-aux experiment) ()
  (let ((choice nil) (choice-list '(yes no)))
  (loop until (setf choice (dw:menu-choose choice-list :prompt "do you want to create any startup
steps?")))
  (when (eql choice 'yes)
    (create-new-obj (make-instance 'startup-step-template) self))
  (setf choice nil)
  (loop until (eql choice 'no)
        do
    (loop until (setf choice (dw:menu-choose choice-list :prompt "create another startup step?")))
    (when (eql choice 'yes)
          (create-new-obj (make-instance 'startup-step-template) self)
          (setf choice nil)))
  (setf choice nil)
  (loop until (setf choice (dw:menu-choose choice-list :prompt "do you want to create any shutdown
 steps?")))
  (when (eql choice 'yes)
    (create-new-obj (make-instance 'shutdown-step-template) self))
  (setf choice nil)
  (loop until (eql choice 'no)
        do
    (loop until (setf choice (dw:menu-choose choice-list :prompt "create another shutdown step?"))
)
    (when (eql choice 'yes)
          (create-new-obj (make-instance 'shutdown-step-template) self)
          (setf choice nil)))
  (loop until (setf choice (dw:menu-choose choice-list :prompt "do you want to create any regular
steps?")))
  (when (eql choice 'yes)
```

```
    (create-new-obj (make-instance 'step-template) self))
  (setf choice nil)
  (loop until (eql choice 'no)
        do
    (loop until (setf choice (dw:menu-choose choice-list :prompt "create another step?")))
    (when (eql choice 'yes)
          (create-new-obj (make-instance 'step-template) self)
          (setf choice nil)))))


(defmethod (create-new-step experiment-template) ()
  (let ((choice nil)
        (choice-list '((NONE none)
                       ("Startup Step" startup-step-template )
                       ("Shutdown Step" shutdown-step-template)
                       ("Step" step-template))))
    (loop until (setf choice (dw:menu-choose choice-list :prompt "Indicate type of step to be crea
ted, or none")))
    (unless (eql choice 'none)
      (create-new-obj (make-instance choice) self))))

(defmethod (copy-self experiment) (&rest ignore)
  (make-instance 'experiment
                 :name name
                 :non-depletable-tolerance-list non-depletable-tolerance-list
                 :min-performances min-performances
                 :max-performances max-performances
                 :desired-performances desired-performances
                 :latest-start-time latest-start-time
                 :performance-time-window performance-time-window
                 :performance-windows performance-windows
                 :crew-lockin crew-lockin
                 :strategy strategy
                 :experiment-time-window experiment-time-window
                 :max-performance-delay-time max-performance-delay-time
                 :min-performance-delay-time  min-performance-delay-time
                 :schedule-shutdown-with-performance schedule-shutdown-with-performance
                 :startup-steps startup-steps
                 :shutdown-steps shutdown-steps
                 :prototype-step-list prototype-step-list
                 :desired-monitor-steps desired-monitor-steps
                 ))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-SELECT-EDITOR-CONFIG
                                       :MENU-ACCELERATOR "Select Obj Editor"
                                       :MENU-LEVEL :NASA-TOP-MENU)
    ()
   (unless (program-framework (screen-manager *mission*))
     (setup-streams (screen-manager *mission*) dw:*program-frame*))
   (select-configuration  *mission* 'edit))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-EDIT-INIT-OBJ
                                       :MENU-ACCELERATOR "Edit Mission Resources"
                                       :MENU-LEVEL :TABLES-MENU)
    ()
     (edit-obj *mission* 'init-obj))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-EDIT-experiment-templates
                                       :MENU-ACCELERATOR "Edit Experiment Descriptions"
                                       :MENU-LEVEL :TABLES-MENU)
    ()
   (edit-experiment-templates *mission*))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-EDIT-SCREEN-MANAGER
                                       :MENU-ACCELERATOR "Edit Screen Manager"
                                       :MENU-LEVEL :TABLES-MENU)
    ()
     (edit-obj *mission* 'screen-manager))


(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-CLEAR-TABLES-HISTORY
                                       :MENU-ACCELERATOR "Clear History"
                                       :MENU-LEVEL :TABLES-MENU)
    ()
   (clear-history (screen-manager *mission*) 'edit))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-FROM-EDIT-TO-MAIN
                                       :MENU-ACCELERATOR "Return To Main Screen"
                                       :MENU-LEVEL :TABLES-MENU)
    ()
     (select-configuration *mission* 'experiment))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-FROM-EDIT-2-TO-MAIN
                                       :MENU-ACCELERATOR "Return To Main Screen"
                                       :MENU-LEVEL :TABLES-MENU-2)
    ()
     (select-configuration *mission* 'experiment))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND (COM-PERFORMANCE-SCHEDULER-CLEAR-TABLES-2-HISTORY
                                       :MENU-ACCELERATOR "Clear History"
                                       :MENU-LEVEL :TABLES-MENU-2)
    ()
   (clear-history (screen-manager *mission*) 'tables-2))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10; Default-character-style: (:FIX :
ROMAN :NORMAL) -*-
(defvar *standard-margin-components* '((DW:MARGIN-BORDERS)
                                       (DW:MARGIN-WHITE-BORDERS :THICKNESS 2)
                                       (DW:MARGIN-SCROLL-BAR :MARGIN :LEFT)
                                       (DW:MARGIN-SCROLL-BAR :MARGIN :BOTTOM)
                                       (DW:MARGIN-WHITESPACE :MARGIN :LEFT :THICKNESS 10)))

(DW:DEFINE-PROGRAM-FRAMEWORK PERFORMANCE-SCHEDULER
  :COMMAND-DEFINER T
  :SELECT-KEY #\a
  :selected-pane NASA-LISP-LISTENER
  :terminal-io-pane NASA-LISP-LISTENER
  :COMMAND-TABLE
  (:INHERIT-FROM '("colon full command" "standard arguments" "standard scrolling")
   :KBD-ACCELERATOR-P t)
  :STATE-VARIABLES ()
  :PANES
  ((NASA-EXP-AND-PER-ASSISTANT-TITLE
     :TITLE :REDISPLAY-STRING "NASA Experiment Performance Scheduler Assistant"
     :HEIGHT-IN-LINES 1 :REDISPLAY-AFTER-COMMANDS NIL)
   (NASA-EXP-AND-PER-ASSISTANT-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :NASA-TOP-MENU)
   (ERROR-TITLE
     :TITLE :REDISPLAY-STRING "NASA Exp Perf Scheduler Asst Error Report"
     :HEIGHT-IN-LINES 1 :REDISPLAY-AFTER-COMMANDS NIL)
   (ERROR-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :ERROR-MENU)
   (ERROR-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                  :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                  :more-p nil
                  :MARGIN-COMPONENTS
                  *standard-margin-components*)
   (GENERAL-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :GENERAL-MENU)
   (GENERAL-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                    :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                    :more-p nil
                    :MARGIN-COMPONENTS
                    *standard-margin-components*)
   (PERFORMANCES-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :PERFORMANCES-MENU)
   (EXPERIMENT-DESCRIBER :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                         :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                         :more-p nil
                         :MARGIN-COMPONENTS
                         *standard-margin-components*)
   (CURRENT-OP-MODE-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                            :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                            :more-p nil
                            :MARGIN-COMPONENTS
                            '((DW:MARGIN-BORDERS)
                              (DW:MARGIN-WHITE-BORDERS :THICKNESS 2)
                              (DW:MARGIN-WHITESPACE :MARGIN :LEFT :THICKNESS 10)))
   (PERFORMANCES-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                         :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                         :more-p nil
                         :MARGIN-COMPONENTS
                         *standard-margin-components*)
   (EXPERIMENTS-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :EXPERIMENTS-MENU)
   (EXPERIMENTS-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                        :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                        :more-p nil
                        :MARGIN-COMPONENTS
                        *standard-margin-components*)
   (RESOURCES-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :RESOURCES-MENU)
   (RESOURCES-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                      :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                      :more-p nil
                      :MARGIN-COMPONENTS
                      *standard-margin-components*)
   (TABLES-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :TABLES-MENU)
   (TABLES-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                   :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                   :more-p nil
                   ;;; :redisplay-function 'display-experiments-table-summary-aux
                   ;;;:incremental-redisplay t
```

```
                :MARGIN-COMPONENTS
                *standard-margin-components*)
  (init-edit-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :init-edit-MENU)
  (init-obj-edit-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :init-obj-edit-MENU)
  (target-resource-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :target-resource-MENU)
  (crew-resource-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :crew-resource-MENU)
  (attitude-resource-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :attitude-resource-MENU)
  (consumable-resource-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :consumable-resource-MENU)
  (durable-resource-COMMAND :COMMAND-MENU :ROWS 1 :MENU-LEVEL :durable-resource-MENU)
  (target-resource-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                           :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                           :more-p nil
                           :MARGIN-COMPONENTS
                           *standard-margin-components*)
  (init-obj-display :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                    :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                    :more-p nil
                    :MARGIN-COMPONENTS
                    *standard-margin-components*)
  (attitude-resource-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                             :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                             :more-p nil
                             :MARGIN-COMPONENTS
                             *standard-margin-components*)
  (crew-resource-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                         :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                         :more-p nil
                         :MARGIN-COMPONENTS
                         *standard-margin-components*)
  (consumable-resource-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                               :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                               :more-p nil
                               :MARGIN-COMPONENTS
                               *standard-margin-components*)
  (durable-resource-DISPLAY :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                            :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                            :more-p nil
                            :MARGIN-COMPONENTS
                            *standard-margin-components*)
  (TABLES-COMMAND-2 :COMMAND-MENU :ROWS 1 :MENU-LEVEL :TABLES-MENU-2)
  (TABLES-DISPLAY-2 :DISPLAY :END-OF-PAGE-MODE :SCROLL :SCROLL-FACTOR 1
                    :DEFAULT-CHARACTER-STYLE '(:FIX :ROMAN :SMALL)
                    :more-p nil
                    :MARGIN-COMPONENTS
                    *standard-margin-components*)
  (NASA-LISP-LISTENER :LISTENER :HEIGHT-IN-LINES 3 :MORE-P NIL
                      :MARGIN-COMPONENTS
                      *standard-margin-components*))
:CONFIGURATIONS
'((DW::NASA-PERFORMANCE-SCHEDULER
    :LAYOUT
      (DW::NASA-PERFORMANCE-SCHEDULER
        :COLUMN NASA-EXP-AND-PER-ASSISTANT-TITLE NASA-EXP-AND-PER-ASSISTANT-COMMAND
        SUB-AREAS-1 NASA-LISP-LISTENER)
      (SUB-AREAS-1 :ROW EXPERIMENT-WINDOW RESOURCES-WINDOW)
      (EXPERIMENT-WINDOW
        :COLUMN EXPERIMENTS-COMMAND EXPERIMENTS-DISPLAY)
      (RESOURCES-WINDOW :COLUMN RESOURCES-COMMAND RESOURCES-DISPLAY))
    :SIZES
      (DW::NASA-PERFORMANCE-SCHEDULER
        (NASA-EXP-AND-PER-ASSISTANT-TITLE 1 :LINES)
        (NASA-EXP-AND-PER-ASSISTANT-COMMAND
          :ASK-WINDOW SELF :SIZE-FOR-PANE NASA-EXP-AND-PER-ASSISTANT-COMMAND)
        (NASA-LISP-LISTENER 3 :LINES) :THEN (SUB-AREAS-1 :EVEN))
      (SUB-AREAS-1 (EXPERIMENT-WINDOW .35) (RESOURCES-WINDOW .65))
      (RESOURCES-WINDOW
        (RESOURCES-COMMAND :ASK-WINDOW SELF :SIZE-FOR-PANE RESOURCES-COMMAND )
        :THEN (RESOURCES-DISPLAY :EVEN))
      (EXPERIMENT-WINDOW
        (EXPERIMENTS-COMMAND :ASK-WINDOW SELF :SIZE-FOR-PANE EXPERIMENTS-COMMAND )
        :THEN (EXPERIMENTS-DISPLAY :EVEN))))
 (dw::edit-init-config
   :layout
```

```
      (dw::edit-init-config :COLUMN init-edit-command init-edit-displays NASA-LISP-LISTENER)
      (init-edit-displays :row other-init-edit-display&cmd durable-resource-display&cmd
                          consumable-resource-display&cmd)
      (other-init-edit-display&cmd
        :COLUMN init-obj-edit-command init-obj-display crew-resource-command
        crew-resource-display  target-resource-command target-resource-display
        attitude-resource-command attitude-resource-display)
      (durable-resource-display&cmd
        :column durable-resource-command durable-resource-display)
      (consumable-resource-display&cmd
        :column consumable-resource-command consumable-resource-display))
    (:SIZES
      (dw::edit-init-config
        (init-edit-command :ASK-WINDOW SELF :SIZE-FOR-PANE TABLES-COMMAND)
        (NASA-LISP-LISTENER 3 :LINES) :THEN (init-edit-displays :EVEN))
      (init-edit-displays (other-init-edit-display&cmd .33)
                          (durable-resource-display&cmd .33)
                          (consumable-resource-display&cmd .34))
      (other-init-edit-display&cmd
        (init-obj-edit-command :ASK-WIND  SELF :SIZE-FOR-PANE init-obj-edit-command)
        (crew-resource-command :ASK-WINDOW SELF :SIZE-FOR-PANE crew-resource-command)
        (target-resource-command :ASK-WINDOW SELF :SIZE-FOR-PANE target-resource-command)
        (attitude-resource-command :ASK-WINDOW SELF :SIZE-FOR-PANE attitude-resource-command)
        (crew-resource-display .25) (target-resource-display .25)
        (attitude-resource-display .25) :then (init-obj-display :even))
      (durable-resource-display&cmd
        (durable-resource-command :ASK-WINDOW SELF :SIZE-FOR-PANE durable-resource-command)
        :then (durable-resource-display :even))
      (consumable-resource-display&cmd
        (consumable-resource-command :ASK-WINDOW SELF
                                     :SIZE-FOR-PANE consumable-resource-command)
        :then (consumable-resource-display :even))))
  (DW::GENERAL-INFO-CONFIG
    (:LAYOUT
      (DW::GENERAL-INFO-CONFIG
        :COLUMN NASA-EXP-AND-PER-ASSISTANT-TITLE NASA-EXP-AND-PER-ASSISTANT-COMMAND
        SUB-AREAS NASA-LISP-LISTENER)
      (SUB-AREAS :ROW GENERAL-WINDOW RESOURCES-WINDOW)
      (GENERAL-WINDOW
        :COLUMN GENERAL-COMMAND GENERAL-DISPLAY)
      (RESOURCES-WINDOW :COLUMN RESOURCES-COMMAND RESOURCES-DISPLAY))
    (:SIZES
      (DW::GENERAL-INFO-CONFIG
        (NASA-EXP-AND-PER-ASSISTANT-TITLE 1 :LINES)
        (NASA-EXP-AND-PER-ASSISTANT-COMMAND
          :ASK-WINDOW SELF :SIZE-FOR-PANE NASA-EXP-AND-PER-ASSISTANT-COMMAND)
        (NASA-LISP-LISTENER 3 :LINES) :THEN (SUB-AREAS :EVEN))
      (SUB-AREAS (GENERAL-WINDOW .35) (RESOURCES-WINDOW .65))
      (RESOURCES-WINDOW
        (RESOURCES-COMMAND :ASK-WINDOW SELF :SIZE-FOR-PANE RESOURCES-COMMAND )
        :THEN (RESOURCES-DISPLAY :EVEN))
      (GENERAL-WINDOW
        (GENERAL-COMMAND :ASK-WINDOW SELF :SIZE-FOR-PANE GENERAL-COMMAND)
        :THEN (GENERAL-DISPLAY :EVEN))))
  (DW::NASA-CONFIG-2
    (:LAYOUT
      (DW::NASA-CONFIG-2
        :COLUMN NASA-EXP-AND-PER-ASSISTANT-TITLE NASA-EXP-AND-PER-ASSISTANT-COMMAND
        SUB-AREAS NASA-LISP-LISTENER)
      (SUB-AREAS :ROW PERFORMANCE-WINDOW RESOURCES-WINDOW)
      (PERFORMANCE-WINDOW
        :COLUMN EXPERIMENT-DESCRIBER CURRENT-OP-MODE-DISPLAY PERFORMANCES-COMMAND
        PERFORMANCES-DISPLAY)
      (RESOURCES-WINDOW :COLUMN RESOURCES-COMMAND RESOURCES-DISPLAY))
    (:SIZES
      (DW::NASA-CONFIG-2
        (NASA-EXP-AND-PER-ASSISTANT-TITLE 1 :LINES)
        (NASA-EXP-AND-PER-ASSISTANT-COMMAND
          :ASK-WINDOW SELF :SIZE-FOR-PANE NASA-EXP-AND-PER-ASSISTANT-COMMAND)
        (NASA-LISP-LISTENER 3 :LINES) :THEN (SUB-AREAS :EVEN))
      (SUB-AREAS (PERFORMANCE-WINDOW .35) (RESOURCES-WINDOW .65))
      (RESOURCES-WINDOW
        (RESOURCES-COMMAND :ASK-WINDOW SELF :SIZE-FOR-PANE RESOURCES-COMMAND )
```

```
      :THEN (RESOURCES-DISPLAY :EVEN))
    (PERFORMANCE-WINDOW
      (EXPERIMENT-DESCRIBER 6 :LINES) (CURRENT-OP-MODE-DISPLAY 3 :LINES)
      (PERFORMANCES-COMMAND :ASK-WINDOW SELF :SIZE-FOR-PANE PERFORMANCES-COMMAND )
      :THEN (PERFORMANCES-DISPLAY :EVEN))))
(DW::ERROR-REPORTING
  (:LAYOUT
    (DW::ERROR-REPORTING
      :COLUMN ERROR-TITLE ERROR-COMMAND ERROR-DISPLAY NASA-LISP-LISTENER))
  (:SIZES
    (DW::ERROR-REPORTING
      (ERROR-TITLE 1 :LINES)
      (ERROR-COMMAND :ASK-WINDOW SELF :SIZE-FOR-PANE ERROR-COMMAND)
      (NASA-LISP-LISTENER 3 :LINES) :THEN (ERROR-DISPLAY :EVEN))))
(DW::TABLES-REPORTING
  (:LAYOUT
    (DW::TABLES-REPORTING
      :COLUMN TABLES-COMMAND TABLES-DISPLAY NASA-LISP-LISTENER))
  (:SIZES
    (DW::TABLES-REPORTING
      (TABLES-COMMAND :ASK-WINDOW SELF :SIZE-FOR-PANE TABLES-COMMAND)
      (NASA-LISP-LISTENER 3 :LINES) :THEN (TABLES-DISPLAY :EVEN))))
(DW::TABLES-REPORTING-2
  (:LAYOUT
    (DW::TABLES-REPORTING-2
      :COLUMN TABLES-COMMAND-2 TABLES-DISPLAY-2 NASA-LISP-LISTENER))
  (:SIZES
    (DW::TABLES-REPORTING-2
      (TABLES-COMMAND-2 :ASK-WINDOW SELF :SIZE-FOR-PANE TABLES-COMMAND)
      (NASA-LISP-LISTENER 3 :LINES) :THEN (TABLES-DISPLAY-2 :EVEN))))
))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(setf *suppress-glyph* t)

(defvar *mission-table* (make-hash-table))
;*****************************************************************************
;;;resource mixins
(defflavor available-time ((begin nil)
                           (end nil)
                           (owner-obj nil))
        ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor availability ((name nil)
                         (available-times-list nil)) ;list of instance of available-time
        ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor quantity-availability ((qty nil)
                                  (owner-obj nil))
           (availability)
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

;*****************************************************************************
;;; resources come in six types
;;; crew members are self-explanatory
;;; targets are locations on the earth
;;; attitudes refer to the orientation of the satalite with respect to ?
;;; durable resources are things that are not consumed, but are available in some
;;; fixed quantity, such as video recorders, or manned maneuver units
;;; consumable resources are things which are consumed, such as food rations, most
;;; chemicals, etc.
;;; finally, non-depletable-resource [nasa term, not mine] is an item which is
;;; consumed, but is also re-generated at some rate, such as wattage from fuel cells,
;;; oxygen thru an activated charcoal filter, water thru waste re-cycling, etc.
(defflavor durable-resource ((name nil)
                             (available-quantity nil))
        ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor non-durable-resource ((name nil)
                                 (quantity-availability-list nil))
           ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor consumable-resource ()
           (non-durable-resource)
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor non-depletable-resource ()
           (non-durable-resource)
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)
```

```
(defflavor crew-member
        ((duty-position nil)
         (work-shift nil))
        (availability)
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor target ()
           (availability)
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor attitude ()
           (availability)
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

;****************************************
;;; the query obj is used to provide generic capability to a context sensitive environment
(defflavor query-obj (type (display-string nil))
        ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

;*****************************************************************************
;;; flavors devoted to the depiction of time and capturing scheduled events
(defflavor time-slice-axis ((end-one-x 0)
                            (end-one-y 0)
                            (end-two-x 0)
                            (end-two-y 0)
                            (spike-coord-list nil)
                            (orientation nil))
          ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

;*****************************************************************************
;;; the screen manager attempts to orchestrate the user interface [at least, that was
;;; the programmers initial concept]
(defflavor nasa-screen-manager
        ((program-framework nil)
         (stream-table (make-hash-table))
         (left-x 50)
         (right-x 1050)
         (lower-y 475)
         (upper-y 25)
         (x-delta nil)
         (h-scale-inc 20) ;;; the number of time slices between scale markers
         (v-scale-table (make-hash-table))
         (current-resource nil)
         (v-scale-inc 10)
         (scale-length 5) ;length of spikes on scales
         (min-x-delta 4)
         (last-config nil)
         (y-axis-table (make-hash-table))
         ;(make-instance 'time-slice-axis :orientation 'vertical)
         (x-axis (make-instance 'time-slice-axis :orientation 'horizontal))
         (y-axis nil)
         (owner-obj nil))
        ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
```

```
  :initable-instance-variables)

(defflavor nasa-init-obj
        ((mission-id nil)
         (mission-launch-date nil)              ;list of day month year
         (mission-launch-time nil)              ;list of hour minute second
         (universal-start-time nil)
         (mission-duration nil)                 ; list of days hours minutes seconds
         (mission-end-date nil)
         (mission-end-time nil)
         (universal-end-time nil)
         (seconds-until-start-of-day nil) ;list of seconds and a flag indicating
                                                ;whether a new day
         (seconds-per-week 604800)
         (seconds-per-day 86400)
         (seconds-per-hour 3600)
         (seconds-per-shift nil)
         (first-sunday-start-time nil)
         (number-of-crew-shifts nil)
         (shift-start-times '((1 (0 8 0 0)) (2 (0 -4 0 0))))
         (max-time nil)
         (time-inc 60) ;;; seconds per time period
         (durable-resource-list nil)
         (non-depletable-resource-list nil)
         (consumable-resource-list nil)
         (crew-list nil);;; a-list (name (list of lists of (begin-avail-time end-avail-time)))
         (target-list nil);;;a-list (name (available times))
         (attitude-list nil);;;a-list (name (available times))
         (owner-obj nil)
         (shift-availability-objs nil)
         (misc-objs '((durable-resource ,(make-instance 'query-obj :type 'durable-resource))
                      (crew-member ,(make-instance 'query-obj :type 'crew-member))
                      (consumable-resource
                        ,(make-instance 'query-obj :type 'consumable-resource))
                      (non-depletable-resource
                        ,(make-instance 'query-obj :type 'non-depletable-resource))
                      (target ,(make-instance 'query-obj :type 'target))
                      (attitude ,(make-instance 'query-obj :type 'attitude))
                      (experiment ,(make-instance 'query-obj :type 'experiment))
                      (performance ,(make-instance 'query-obj :type 'performance))
                      (step ,(make-instance 'query-obj :type 'step))
                     )))
        ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor mission ((experiment-template-table (make-hash-table))
                    (experiment-table (make-hash-table))
                    (time-slice-holder nil)
                    (screen-manager (make-instance 'nasa-screen-manager))
                    (init-obj (make-instance 'nasa-init-obj))
                    (selected-time-slice nil)
                    (selected-performance nil)
                    (operation nil)
                    (crew-combinations-table (make-hash-table :test #'equal))
                    (time-table (make-hash-table))
                    (power-table (make-hash-table))
                    (sorted-power-keys nil)
                    (sorted-time-keys nil)
                    (title nil)
                    (sorted-instance-list nil)
                    (multiple-scheduling  nil))
           ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(declare (special temp-list))

(defmethod (make-instance mission :after) (&rest ignore)
```

```
  (setf (owner-obj screen-manager ) self
        (owner-obj init-obj ) self))

(defvar *mission* (make-instance 'mission))


(defflavor experiment ((name nil)
                       (min-performances 0)
                       (max-performances 0)
                       (desired-performances nil)
                       (performance-list nil)
                       (latest-start-time nil)
                       (performance-time-window nil) ;;;aka max perf duration - code
                       ;;;was developed before I realized that I was dealing with one
                       ;;;value instead of two
                       (performance-windows nil)
                       (crew-lockin nil) ;;; nil or a list of lists of first and last
                       ;;; steps requiring lockin ex ((1 5) (7 9))
                       (non-depletable-tolerance-list nil)
                       (strategy nil) ;;;see esp users manual section on scenarios --
                       ;;;when used, strategy will consist of keyword :cascade or
                       ;;;:max-weigth, and list of scenarios and weights.  example
                       ;;;(:cascade ((((:consecutive 1 15)) 90)
                       ;;;           (((:consecutive 2 14)) 45)
                       ;;;           (((:consecutive 2 14) (:sequential (14))) 70)
                       ;;;           (((:sequential (2 1 3 5 6 1))) 80)))
                       (experiment-time-window nil) ;;;max time between start first
                       ;;;step first performance, and end last step, last performance
                       (max-performance-delay-time nil) ;;;max time between end of last
                       ;;;step of one performance and start of first step, next
                       ;;;performance
                       (min-performance-delay-time 0) ;;;min time between end of last
                       ;;;step of one performance and start of first step, next
                       ;;;performance
                       (schedule-shutdown-with-performance t);if I need this one,
                       ;;;why don't i need one for start-up? I need this for use
                       ;;;during automatic scheduling, to prevent scheduling and
                       ;;;unscheduling  of shutdown steps after each performance.
                       ;;;Note that automatic scheduling must insure shutdown
                       ;;;scheduled with last auto performance, and this flag is "on"
                       ;;; afterwards
                       (startup-steps nil)
                       (shutdown-steps nil)
                       (prototype-step-list nil)
                       (desired-monitor-steps nil)
                       (min-performances-displayed-p nil))
          ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor performance ((number 0)
                        (scheduled-start-time nil)
                        (scheduled-end-time nil)
                        (performance-time-window nil)
                        (scheduled-p nil)
                        (required-p nil)
                        (step-list nil)
                        (execute-start-up-steps-p nil)
                        (execute-shutdown-steps-p nil)
                        (last-time-slice nil)
                        (owning-experiment nil))
          ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defflavor step ((id nil)
                 (number nil)
```

```
                (scheduled-start-time nil)
                (scheduled-end-time nil) ;;;needed because of variable duration
                (max-duration nil)
                (min-duration nil)
                (step-delay-min nil)
                (step-delay-max nil)
                (next-step nil)
                (previous-step nil)
                (last-time-slice nil)
                (cumulative-consumable-list nil)
                (resource-carry-thru nil)
                (consumable-resource-list nil) ;;a-list (resource-name qty)
                (durable-resource-list nil) ;;a-list (resource-name qty))
                (non-depletable-resource-list nil)  ;;a-list (resource-name qty tolerance)
                ;;;no tolerance entry or nil entry is equivalent to zero
                (crew-requirements nil);;;list of lists:  inner list is list of list
                ;;;of how identified, crew-members and qty to be used. expample
                ;;;(((duty-position pilot nav asst-pilot) 2)
                ;;; ((duty-position senior-mission-scientist mission-scientist)1)
                ;;; ((name smith jones) 1))
                (crew-combinations nil) ;;;list of lists -- each inner list
                ;;;represents a combination of crew members, by object, which
                ;;;satisfy the crew requirements
                (failed-crew-combinations nil)
                (crew-lockin nil) ;;; nil or the number of the step holding the
                ;;; lockin crew list -- note that even if specified as a lockin
                ;;; step, flag will be nil unless crew lockin requirements are the
                ;;; same and monitoring is not required
                (crew-monitor nil)
                (crew-duration nil)
                (crew-cycle nil)
                (crew-early-shift nil)
                (crew-late-shift nil)
                (concurrent-with nil) ;;; (exp step)
                (target-list nil) ;;; A LIST OF LISTS; INNER LIST CONSIST OF
                ;;; ONE OF THE KEY WORDS :intersect :select :avoid AS THE FIRST
                ;;; ELEMENT, AND A LIST OF TARGETS AS THE SECOND ELEMENT; KEY WORDS
                ;;; CANNOT BE REPEATED
                (attitude-list)  ;;;(avoid-or-required attitude-list)
                (scheduled-crew-list nil)  ;;;list of list of (crew-id lockin)
                (crew-monitoring-time 1.0) ;;;fraction of step length crew members
                ;;;required to monitor this step
                (owning-object nil))
            ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defmethod (:print-self step) (stream ignore ignored)
  (cond ((null owning-object)
         (format stream "#<STEP ~A ~A ~A>" id number nil))
        ((typep owning-object 'experiment)
         (format stream "#<STEP ~A ~A ~A>" id number (name owning-object)))
        ((typep owning-object 'performance)
         (format stream "#<STEP ~A ~A Perf # ~A of ~a>"
                 id number (number owning-object)(name (owning-experiment owning-object))))
        (t (format stream "#<STEP ~A ~A ~A>" id number owning-object))))

(defflavor startup-step () (step) (:conc-name nil) :writable-instance-variables
           :readable-instance-variables :initable-instance-variables)

(defflavor shutdown-step () (step) (:conc-name nil) :writable-instance-variables
           :readable-instance-variables :initable-instance-variables)

(defflavor experiment-template () (experiment) (:conc-name nil) :writable-instance-variables :read
able-instance-variables :initable-instance-variables)

(defflavor step-template () (step) (:conc-name nil) :writable-instance-variables
           :readable-instance-variables :initable-instance-variables)

(defflavor shutdown-step-template () (shutdown-step) (:conc-name nil)
           :writable-instance-variables
```

```
          :readable-instance-variables :initable-instance-variables)

(defflavor startup-step-template () (startup-step) (:conc-name nil)
          :writable-instance-variables
          :readable-instance-variables :initable-instance-variables)


(defflavor time-slice ((start-time nil)
                       (end-time nil)
                       (performance-step-table (make-hash-table :test #'equal))
                       ;;;key is list (exp perf step)
                       (crew-list nil) ;;; until the mechanism for implementing
                       ;;; monitoring is devised, simply a list of
                       ;;;(crew-member committed  who-info)
                       (consumable-resource-list nil) ;list of (resource committed who-info)
                       (cumulative-consumable-table (make-hash-table))
                       (non-depletable-resource-list nil)
                       ;;;list of (resource committed tolerance who-info)
                       (durable-resource-list nil) ;list of (resource committed who-info)
                       (target-list nil) ;;;targets available in this time-sliced
                       (attitude-list nil) ;;;attitude during this time-slice
                       (next-slice nil)
                       (prev-slice nil)
                       (start-x nil)
                       (top-y nil))
          ()
  (:conc-name nil)
  :writable-instance-variables
  :readable-instance-variables
  :initable-instance-variables)

(defun clear-listener ()
  (send (gethash 'listener (stream-table (screen-manager *mission*))) :clear-history))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-


(defvar *help-window* nil)

(defmethod (help mission) ()
  (cond (*help-window*
         (send *help-window* :expose)
         )
        (t
         (setf *help-window* (tv:make-window 'tv:window
                                             :edges '(100 100 1000 600)
                                             :expose-p t
                                             :activate-p t
                                             :blinker-p nil
                                             :default-character-style
                                             '(:fix :roman :normal)
                                             :save-bits t
                                             :label "Mission Help Window"))
         (format *help-window* "~%TURN THE DYNAMIC GARBAGE COLLECTOR ON !!!~%~%To load the data necessary
to run the model, execute the method (load-mission-data *mission*). ~%~%To cause the model to run, execut
e the function (test-scheduler *mission* [list of experiment names] ~%[number of replications each]). The
 last argument is a single number. ~%To get a list of experiment names, execute the method (get-list-of-l
oaded-experiment-names *mission*) ~%After the model has been run, if you wish to run it again, execute th
e function ~%(restore-data-to-start *mission*), and the test-scheduler again.  ~%~%To get printed output
of the results, execute the function (output-mission-data *mission* ~%[OPTIONAL list-of-time-slice-instan
ces]).  This will cause files in the directory NASA-EXP-SCH-2:OUTPUT-DATA; to be deleted and expunged, an
d new files created for the time line and each experiment that has been scheduled.  When the optional lis
t-of-time-slice-instances is supplied, only those time slices will be written out.  ~%~%To get a list of
time slices covering a time period, execute the function (get-time-instance-list *mission* ~%start-time e
nd-time [OPTIONAL time-slice-instance]).  The start-time and end-time are in terms of mission time ~%peri
ods; that is, the number of seconds since launch divided by the time increment (currently 60).  See the f
ile NASA-EXP-SCH-2:NASA=-EXP-SCH-2;TIME-TRANSLATORS.LISP for functions that can assist in obtaining the c
orrect ~%values. The optional time-slice-instance is used when you have a handle on an instance which is
closer to the ~%desired instances than the first instance.~%~%Data can also be written out in a binary fo
rm by executing the method (dump-mission-to-file *mission* ~%[OPTIONAL (FILENAME NASA-EXP-SCH-2:BIN-FILES
;MISSION-FASD-FILE.BIN]).  The method name comes from the use of the sys:dump-forms-to-file function, and
 the file name from the use of FASD [FASt Dump] forms for every object.  If you haven't used these before
, be advised that they cannot handle recursive structures; you must modify the ~%saved instance to remove
 backpoints to objects, and restore the backpointers upon reload.  ~%~%To reload a saved mission, simple
execute (load [filename]).  ~%~%To view this message again, execute (help *mission*)")))

(help *mission*)
```

```
;;; -*- Package: USER; Base: 10; Mode: LISP; Syntax: Common-lisp; -*-

(defmethod (load-mission-data mission) ()
  (load-mission-data init-obj)
  (load-resource-data self)
  (load-all-experiment-data self)
  (setup-crew-member-duty-shifts init-obj)
  (correct-representations-and-build-linkages-after-data-load self))

(defmethod (load-lockin-test mission) ()
  (load-mission-data init-obj)
  (load-resource-data self)
  (let ((experiment (make-instance 'experiment :name 'lockin-test)))
    (load-experiment-data experiment 'lockin-test)
    (setf (gethash 'lockin-test experiment-template-table) experiment))
  (setup-crew-member-duty-shifts init-obj)
  (correct-representations-and-build-linkages-after-data-load self))

(defmethod (load-mission-data nasa-init-obj) ()
  (load "nasa-exp-sch-2:data;mission-data" :verbose nil)
  (loop for (slot value) in temp-list
        do
    (setf (symbol-value-in-instance self slot) value))
  (determine-initial-universal-times self)
  (determine-end-times self)
  (setf max-time (floor (- universal-end-time universal-start-time) time-inc)))

(defmethod (load-resource-data mission) ()
  (load-consumable-resource-data self)
  (load-non-depletable-resource-data self)
  (load-durable-resource-data self)
  (load-crew-resource-data self)
  (load-target-resource-data self)
  (load-attitude-resource-data self))

(defmethod (load-consumable-resource-data mission) ()
  (load "nasa-exp-sch-2:data;consumable-resources" :verbose nil)
  (when temp-list
  (setf (consumable-resource-list init-obj)
        (loop for (symbol value) in temp-list
              for resource =
                  (make-instance
                    'consumable-resource
                    :name symbol
                    :quantity-availability-list
                    (ncons (make-instance
                             'quantity-availability
                             :name symbol
                             :qty value
                             :available-times-list
                             (ncons (make-instance 'available-time
                                                   :begin 0
                                                   :end (max-time (init-obj *mission*)))))))
              collect resource))
  (loop for resource in (consumable-resource-list init-obj)
        do
    (loop for qty-avail-obj in (quantity-availability-list resource)
          do
      (setf (owner-obj qty-avail-obj) resource)
      (loop for avail-time-obj in (available-times-list qty-avail-obj)
            do
        (setf (owner-obj avail-time-obj) qty-avail-obj))))))

(defmethod (load-non-depletable-resource-data mission) ()
  (load "nasa-exp-sch-2:data;non-depletable-resources" :verbose nil)
  (when temp-list
  (setf (non-depletable-resource-list init-obj)
        (loop for (symbol qty-av-list) in temp-list
              collect
                (make-instance
                  'non-depletable-resource
                  :name symbol
                  :quantity-availability-list
```

```
                (loop for (day hour minute second quant) in qty-av-list
                      collect
                        (make-instance
                          'quantity-availability
                          :available-times-list
                          (ncons
                            (make-instance
                              'available-time
                              :begin
                              (translate-seconds-to-time-periods
                                (translate-time-list-to-seconds
                                (list day hour minute second)))))
                          :qty quant)))
              ))
  (loop for resource in (non-depletable-resource-list init-obj)
        do
    (loop for qty-avail-obj in (quantity-availability-list resource)
          do
      (setf (owner-obj qty-avail-obj) resource)
      (loop for avail-time-obj in (available-times-list qty-avail-obj)
            do
        (setf (owner-obj avail-time-obj) qty-avail-obj))))))

(defmethod (load-durable-resource-data mission) ()
  (load "nasa-exp-sch-2:data;durable-resources" :verbose nil)
  (when temp-list
  (setf (durable-resource-list init-obj)
        (loop for (nname aavailable-quantity) in temp-list
              collect (make-instance 'durable-resource
                                     :name nname
                                     :available-quantity aavailable-quantity)))))

(defmethod (load-crew-resource-data mission) ()
  (load "nasa-exp-sch-2:data;crew-resources" :verbose nil)
  (when temp-list
    (setf (crew-list init-obj)
          (loop for (crew-name crew-position crew-shift) in temp-list
                collect (make-instance 'crew-member
                                       :name crew-name
                                       :duty-position crew-position
                                       :work-shift crew-shift)))
    ))

(defmethod (load-target-resource-data mission) ()
  (load "nasa-exp-sch-2:data;target-resources" :verbose nil)
  (when temp-list
    ()
    ))

(defmethod (load-attitude-resource-data mission) ()
  (load "nasa-exp-sch-2:data;attitude-resources" :verbose nil)
  (when temp-list
    ()
    ))

(defmethod (load-all-experiment-data mission) ()
  (load "nasa-exp-sch-2:data;experiment-list" :verbose nil)
  (loop for experiment-name in temp-list
        for experiment = (make-instance 'experiment)
        do
    (load-experiment-data experiment experiment-name)
    (setf (gethash experiment-name experiment-template-table) experiment)))

(defmethod (load-experiment-data experiment) (experiment-name)
  (load (format nil "nasa-exp-sch-2:exp-data;~S" experiment-name) :verbose nil)
  (unless (eql (first temp-list) experiment-name)
    (error "~% Experiment Name in Experiment List, ~S, Does Not Match Name in File, ~S"
           experiment-name (first temp-list)))
  (setf name (first temp-list))
  (load-experiment-data-aux self (cdr temp-list))
  (when strategy
    (setf strategy
          (list
```

```
            (first strategy)
            (sort (copy-alist (second strategy)) #'> :key #'second)))))


(defmethod (load-experiment-data-aux experiment) (data-list)
  (cond ((null data-list) nil)
        ((member (first (first data-list))
                 '(prototype-step-list startup-step-list shutdown-step-list))
         (build-steps self (first data-list))
         (load-experiment-data-aux self (cdr data-list)))
        (t (setf (symbol-value-in-instance self (first (first data-list)))
                 (second (first data-list)))
           (load-experiment-data-aux  self (cdr data-list)))))

(defmethod (build-steps experiment) (data-list)
  (setf (symbol-value-in-instance self (first data-list))
        (loop for step-data in (second data-list)
              for step = (make-instance 'step :owning-object self)
              collect step
              do
          (build-step step step-data non-depletable-tolerance-list))))

(defmethod (build-step step) (step-data non-depletable-tolerance-list)
  (let ((result nil))
  (loop for (slot value) in step-data
        do
    (setf (symbol-value-in-instance self slot) value))
  (loop for (resource quant) in non-depletable-resource-list
        do
    (if (member resource non-depletable-tolerance-list :key #'first)
        (push (list resource quant
                    (second
                      (first (member resource non-depletable-tolerance-list :key #'first))))
              result)
        (push (list resource quant 0) result)))
  (setf non-depletable-resource-list result)
  (setf min-duration (translate-time-list-to-seconds min-duration)
        max-duration (translate-time-list-to-seconds max-duration)
        step-delay-min (translate-time-list-to-seconds step-delay-min)
        step-delay-max (translate-time-list-to-seconds step-delay-max))
  (when crew-monitor
    (setf crew-duration (translate-seconds-to-time-periods
                          (translate-time-list-to-seconds crew-duration))
          crew-cycle (translate-seconds-to-time-periods
                       (translate-time-list-to-seconds crew-cycle))
          crew-early-shift (translate-seconds-to-time-periods
                             (translate-time-list-to-seconds crew-early-shift))
          crew-late-shift (translate-seconds-to-time-periods
                            (translate-time-list-to-seconds crew-late-shift))))))
```

```
;;; -*- Package: USER; Base: 10; Mode: LISP; Syntax: Common-lisp; -*-

(defsystem nasa-exp-sch-2
    (:default-pathname "nasa-exp-sch-2: nasa-exp-sch-2;"
     :pretty-name "NASA Experiment and Performance Tool"
     :default-package 'cl-user
     :patchable nil
     :initial-status :experimental
     :bug-reports ("bug-nasa-exp&perf-scheduler"
                   "Report problems with NASA Experiment and Performance Tool code")
     :advertised-in (:herald :finger :disk-label)
     :maintaining-sites (:mayberry)
     :source-category (:basic)
     :distribute-sources t
     :distribute-binaries t)
  (:module globals ("globals" "framework" ))
  (:module graphics-defs ("edit-presentation-types" )
           (:uses-definitions-from globals))
  (:module methods ("nasa-init-obj-methods" "new-mission-methods""screen-manager-methods"
                    "resource-methods" "step-methods" "experiment-methods" "time-translators"
                    "time-slice-methods" "performance-methods")
           (:uses-definitions-from graphics-defs))
  (:module loader ("load-methods" "after-data-load-methods")
           (:uses-definitions-from globals))
  (:module output ("output-to-file" "output-methods")
           (:uses-definitions-from globals))
  (:module scheduler ("scheduler-feasibility-methods-performance-level"
                      "scheduler-feasibility-methods-step-level" "scheduler-methods"
                      "scheduler-feasibility-methods-crew-steps"
                      "scheduler-feasibility-methods-other-steps"
                      "scheduler-feasibility-pre-and-post-step"
                      "scheduler-feasibility-methods-resource"
                      "scheduler-feasibility-methods-targets"
                      "scheduler-feasibility-methods-non-depletable"
                      "scheduler-feasibility-methods-durable-resource")
           (:uses-definitions-from globals))
  (:module unscheduler ("unschedule-methods")
            (:uses-definitions-from globals))
  (:module commands ("framework-commands" "presentation-commands" "editor-framework-commands")
           (:uses-definitions-from graphics-defs))
  (:module help ("help-methods")
           ;;;i lie - it doesn't use any definitions other that these in globals; but
           ;;;this will insure it is loaded last!
           (:uses-definitions-from commands unscheduler scheduler output loader methods)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-


(defmethod (delete-resource nasa-init-obj) (type)
  (delete-resource-aux self (case type
                              (durable-resource 'durable-resource-list)
                              (consumable-resource 'consumable-resource-list)
                              (crew-member 'crew-list)
                              (attitude 'attitude-list)
                              (target 'target-list))))

(defmethod (delete-resource-aux nasa-init-obj) (type)
  (let ((choice nil) (the-list (cons '(Quit quit)
                                     (mapcar #'(lambda (obj) (list (name obj) obj))
                                             (symbol-value-in-instance self type)))))
    (loop until
          (setf choice
                (dw:menu-choose the-list
                                :prompt "Select Name of Resource to be Deleted or Quit")))
    (unless (eql choice 'quit)
      (setf (symbol-value-in-instance self type)
            (delete choice (symbol-value-in-instance self type))))))

(defmethod (add-resource nasa-init-obj) (obj slot)
  (push obj (symbol-value-in-instance self slot))
  ;;; add code for any other actions to be done when adding a resource
  )

(defmethod (edit-sub-obj nasa-init-obj) (tag)
  (case tag
    (init-obj-edit (present self 'single-valued-nasa-init-obj-edit-display
             :stream (select-stream *mission* 'init-obj-edit)))
    (durable (display-durable-resource-for-editing
      self (select-stream *mission* 'durable-resource-edit )))
    (consumable (display-consumables-for-editing
                 self (select-stream *mission* 'consumable-resource-edit)))
    (target (display-targets-for-editing self (select-stream *mission* 'target-resource-edit)))
    (attitude
      (display-attitudes-for-editing self (select-stream *mission* 'attitude-resource-edit)))
    (crew (display-crew-for-editing self (select-stream *mission* 'crew-resource-edit)))))

(defmethod (edit-self nasa-init-obj) ()
  (select-configuration *mission* 'init-obj-edit)
    (setup-query-string self)
    (present self 'single-valued-nasa-init-obj-edit-display
             :stream (select-stream *mission* 'init-obj-edit))
    (display-durable-resource-for-editing
      self (select-stream *mission* 'durable-resource-edit ))
    (display-consumables-for-editing self (select-stream *mission* 'consumable-resource-edit))
    (display-crew-for-editing self (select-stream *mission* 'crew-resource-edit))
    (display-targets-for-editing self (select-stream *mission* 'target-resource-edit))
    (display-attitudes-for-editing self (select-stream *mission* 'attitude-resource-edit)))

(defmethod (setup-query-string nasa-init-obj) ()
  (unless (display-string (second (assoc 'durable-resource misc-objs)))
  (setf
    (display-string (second (assoc 'durable-resource misc-objs)))
    '("MOUSE LEFT HERE TO CREATE A NEW DURABLE RESOURCE" "MOUSE CENTER TO DELETE A DURABLE RESOURC
E")
    (display-string (second (assoc 'crew-member misc-objs)))
    '("MOUSE LEFT HERE TO CREATE A NEW CREW MEMBER" "MOUSE CENTER TO DELETE A CREW MEMBER")
    (display-string (second (assoc 'consumable-resource misc-objs)))
    '("MOUSE LEFT HERE TO CREATE A NEW CONSUMABLE RESOURCE"
      "MOUSE CENTER TO DELETE A CONSUMABLE RESOURCE")
    (display-string (second (assoc 'target misc-objs)))
    '("MOUSE LEFT HERE TO CREATE A NEW TARGET" "MOUSE CENTER TO DELETE A TARGET")
    (display-string (second (assoc 'attitude misc-objs)))
    '("MOUSE LEFT HERE TO CREATE A NEW ATTITUDE" "MOUSE CENTER TO DELETE AN ATTITUDE")
    )))

(defmethod (display-available-times-for-editing availability) (stream)
  (formatting-table (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
```

```
        (formatting-cell (stream :align :right) "  BEGIN AVAILABLE TIME  ")
        (formatting-cell (stream :align :right) "END AVAILABLE TIME"))
    (loop for available-time in available-times-list
          do
      (formatting-row (stream)
        (present available-time 'available-time-edit-display :stream stream)))))

(defmethod (display-durable-resource-for-editing nasa-init-obj) (stream)
  (with-character-style ('(:fix :bold :normal) stream :bind-line-height t)
    (format stream "~% DURABLE RESOURCES FOR MISSION~%"))
  (when durable-resource-list
    (formatting-table (stream :equalize-multiple-column-widths t)
      (formatting-column-headings (stream :underline-p nil)
          (formatting-cell (stream :align :left) "     RESOURCE NAME     ")
          (formatting-cell (stream :align :left) "AVAILABLE QUANTITY"))
      (loop for resource in durable-resource-list
            do
        (formatting-row (stream)
          (present resource 'durable-resource-edit-display :stream stream)))))
  (present (second (assoc 'durable-resource misc-objs)) 'misc-obj-edit-display :stream stream))

(defmethod (display-consumables-for-editing nasa-init-obj) (stream)
  (with-character-style ('(:fix :bold :normal) stream :bind-line-height t)
    (format stream "~% CONSUMABLE RESOURCES FOR MISSION"))
  (loop for resource in consumable-resource-list
        do
    (present resource 'consumable-name-for-edit-display :stream stream )
    (loop for quantity-availability in (quantity-availability-list resource)
          do
      (present quantity-availability 'quantity-availability-edit-display :stream stream )
      (display-available-times-for-editing quantity-availability stream)))
  (present (second (assoc 'consumable-resource misc-objs))
           'misc-obj-edit-display :stream stream))

(defmethod (display-crew-for-editing nasa-init-obj) (stream)
  (with-character-style ('(:fix :bold :normal) stream :bind-line-height t)
    (format stream "~% CREW MEMBERS FOR MISSION"))
  (loop for crew-member in crew-list
        do
    (present crew-member 'name-for-edit-display :stream stream)
    (display-available-times-for-editing crew-member stream))
  (present (second (assoc 'crew-member misc-objs)) 'misc-obj-edit-display :stream stream))


(defmethod (display-targets-for-editing nasa-init-obj) (stream)
  (with-character-style ('(:fix :bold :normal) stream :bind-line-height t)
    (format stream "~% TARGETS FOR MISSION"))
  (loop for target in target-list
        do
    (present target 'name-for-edit-display :stream stream)
    (display-available-times-for-editing target stream))
  (present (second (assoc 'target misc-objs)) 'misc-obj-edit-display :stream stream))

(defmethod (display-attitudes-for-editing nasa-init-obj) (stream)
  (with-character-style ('(:fix :bold :normal) stream :bind-line-height t)
    (format stream "~% ATTITUDES FOR MISSION"))
  (loop for attitude in attitude-list
        do
    (present attitude 'name-for-edit-display :stream stream)
    (display-available-times-for-editing attitude stream))
  (present (second (assoc 'attitude misc-objs)) 'misc-obj-edit-display :stream stream))



(defmethod (get-resource-list nasa-init-obj) ()
    (mapcar #'(lambda (x) (list (name x) x))
            (append consumable-resource-list durable-resource-list)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (get-list-of-loaded-experiment-names mission) ()
  (let ((result nil))
    (maphash #'(lambda (exp ignore)
                 (push exp result))
             experiment-table)
    (sort result #'alphalessp)))

(defmethod (get-resource-list mission) ()
  (get-resource-list init-obj))

(defmethod (add-resource mission) (obj slot)
  (add-resource init-obj obj slot)
  ;;;add code for any other function that must be done when adding a new resource
  )

(defmethod (delete-resource mission) (type)
  (case type
    ((target attitude crew-member consumable-resource durable-resource)
     (delete-resource init-obj type))
    (experiment-template (delete-exp-template self))
    (experiment (delete-exp self)))
  ;;; add code to clear up any other pointer, including displays
  )

(defmethod (delete-exp-template mission) ()
  (format tv:initial-lisp-listener "this is a stub (delete-exp-template mission)"))

(defmethod (delete-exp mission) ()
  (format tv:initial-lisp-listener "this is a stub (delete-exp mission) "))

(defmethod (edit-init-sub-obj mission) (tag)
  (edit-sub-obj init-obj tag))

(defmethod (edit-obj mission) (obj-tag)
  (edit-self (symbol-value-in-instance self obj-tag)))

(defmethod (report-error mission) (error-msg)
  (format tv:initial-lisp-listener "~%~A"error-msg))

(defmethod (select-configuration mission) (key)
  (select-configuration screen-manager key))

(defmethod (select-stream mission) (key)
  (select-stream screen-manager key))

(defmethod (clear-history mission) (key)
  (clear-history screen-manager key))

(defmethod (select-configuration-and-clear-history mission) (key)
  (select-configuration screen-manager key)
  (clear-history screen-manager key))

(defmethod (edit-experiment-templates mission) ()
  (let ((stream (select-stream self 'tables-2)))
    (unless (display-string (second (assoc 'experiment (misc-objs init-obj))))
    (setf
      (display-string (second (assoc 'experiment (misc-objs init-obj))))
      '("MOUSE LEFT HERE TO CREATE A NEW EXPERIMENT" "MOUSE CENTER TO DELETE AN EXPERIMENT")
      (display-string (second (assoc 'performance (misc-objs init-obj))))
      '("MOUSE LEFT HERE TO CREATE A NEW PERFORMANCE" "MOUSE CENTER TO DELETE AN PERFORMANCE")
      (display-string (second (assoc 'step (misc-objs init-obj))))
      '("MOUSE LEFT HERE TO CREATE A NEW STEP" "MOUSE CENTER TO DELETE AN STEP")))
    (select-configuration-and-clear-history self 'tables-2)
    (maphash #'(lambda (key experiment-template)
                 key
                 (display-experiment-template-for-editing experiment-template stream))
             experiment-template-table)
    (present (second (assoc 'experiment (misc-objs init-obj))) 'misc-obj-edit-display :stream stre
am)))

(defmethod (display-experiment-template-for-editing experiment) (stream)
```

```
  (present self 'experiment-template-name-edit-display :stream stream)
  (present self 'experiment-template-edit-display :stream stream)
  (loop for slot in '(startup-steps shutdown-steps prototype-step-list)
        do
    (format stream "~A"slot)
    (mapc #'(lambda (step)
              (present step 'step :stream stream))
          (symbol-value-in-instance self slot))
    (present (second (assoc 'step (misc-objs (init-obj *mission*))))
             'misc-obj-edit-display :stream stream)))

(defmethod (add-exp-temp-to-table mission) (experiment-template name)
  (setf (gethash name experiment-template-table ) experiment-template))

(defmethod (add-exp-to-table mission) (experiment name)
  (setf (gethash name experiment-table ) experiment))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-


(defmethod (:fasd-form available-time) ()
  `(make-instance 'available-time
                  :begin ',begin
                  :end ',end))

(defmethod (:fasd-form availability) ()
  `(make-instance 'availability
                  :name ',name
                  :available-times-list ',available-times-list))

(defmethod (:fasd-form quantity-availability) ()
  `(make-instance 'quantity-availability
                  :name ',name
                  :available-times-list ',available-times-list
                  :qty ',qty
                  :owner-obj ',(name owner-obj)))

(defmethod (:fasd-form durable-resource) ()
  `(make-instance 'durable-resource
                  :name ',name
                  :available-quantity ',available-quantity))

(defmethod (:fasd-form non-durable-resource) ()
  `(make-instance 'non-durable-resource
                  :name ',name
                  :quantity-availability-list ',quantity-availability-list))

(defmethod (:fasd-form consumable-resource) ()
  `(make-instance 'consumable-resource
                  :name ',name
                  :quantity-availability-list ',quantity-availability-list))

(defmethod (:fasd-form non-depletable-resource) ()
  `(make-instance 'non-depletable-resource
                  :name ',name
                  :quantity-availability-list ',quantity-availability-list))

(defmethod (:fasd-form  crew-member) ()
  `(make-instance 'crew-member
                  :duty-position ',duty-position
                  :work-shift ',work-shift
                  :name ',name
                  :available-times-list ',available-times-list))

(defmethod (:fasd-form target) ()
  `(make-instance 'target
                  :name ',name
                  :available-times-list ',available-times-list))

(defmethod (:fasd-form attitude) ()
  `(make-instance 'attitude
                  :name ',name
                  :available-times-list ',available-times-list))


(defmethod (:fasd-form nasa-init-obj)()
  `(make-instance 'nasa-init-obj
                  :mission-id ',mission-id
                  :mission-launch-date ',mission-launch-date
                  :mission-launch-time ',mission-launch-time
                  :universal-start-time ',universal-start-time
                  :mission-duration ',mission-duration
                  :mission-end-date ',mission-end-date
                  :mission-end-time ',mission-end-time
                  :universal-end-time ',universal-end-time
                  :seconds-until-start-of-day ',seconds-until-start-of-day
                  :seconds-per-week ',seconds-per-week
                  :seconds-per-day ',seconds-per-day
                  :seconds-per-shift ',seconds-per-shift
```

```
                    :first-sunday-start-time ',first-sunday-start-time
                    :number-of-crew-shifts ',number-of-crew-shifts
                    :shift-start-times ',shift-start-times
                    :max-time ',max-time
                    :time-inc ',time-inc
                    :durable-resource-list ',durable-resource-list
                    :non-depletable-resource-list ',non-depletable-resource-list
                    :consumable-resource-list ',consumable-resource-list
                    :crew-list ',crew-list
                    :target-list ',target-list
                    :attitude-list ',attitude-list
                    :shift-availability-objs ',shift-availability-objs))

(defmethod (:fasd-form nasa-screen-manager) ()
  `(make-instance 'nasa-screen-manager
                    :program-framework ',program-framework
                    :stream-table ',stream-table
                    :left-x ',left-x
                    :right-x ',right-x
                    :lower-y ',lower-y
                    :upper-y ',upper-y
                    :x-delta ',x-delta
                    :h-scale-inc ',h-scale-inc
                    :v-scale-table ',v-scale-table
                    :current-resource ',current-resource
                    :v-scale-inc ',v-scale-inc
                    :scale-length ',scale-length
                    :min-x-delta ',min-x-delta
                    :last-config ',last-config
                    :y-axis-table ',y-axis-table
                    :x-axis ',x-axis
                    :y-axis ',y-axis))

(defmethod (:fasd-form mission) ()
  `(make-instance 'mission
                    :experiment-template-table ',experiment-template-table
                    :experiment-table ',experiment-table
                    :time-slice-holder ',time-slice-holder
                    :screen-manager ',screen-manager
                    :init-obj ',init-obj
                    :selected-time-slice ',selected-time-slice
                    :selected-performance ',selected-performance
                    :operation ',operation
                    :crew-combinations-table ',crew-combinations-table
                    :time-table ',time-table
                    :power-table ',power-table
                    :sorted-power-keys ',sorted-power-keys
                    :sorted-time-keys ',sorted-time-keys
                    :title ',title
                    :sorted-instance-list ',sorted-instance-list
                    :multiple-scheduling ',multiple-scheduling))

(defmethod (:fasd-form experiment) ()
  `(make-instance 'experiment
                    :name ',name
                    :min-performances ',min-performances
                    :max-performances ',max-performances
                    :desired-performances ',desired-performances
                    :performance-list ',performance-list
                    :latest-start-time ',latest-start-time
                    :performance-time-window ',performance-time-window
                    :performance-windows ',performance-windows
                    :crew-lockin ',crew-lockin
                    :non-depletable-tolerance-list ',non-depletable-tolerance-list
                    :strategy ',strategy
                    :max-performance-delay-time ',max-performance-delay-time
                    :min-performance-delay-time ',min-performance-delay-time
                    :schedule-shutdown-with-performance ',schedule-shutdown-with-performance
                    :startup-steps ',startup-steps
                    :shutdown-steps ',shutdown-steps
                    :prototype-step-list ',prototype-step-list
                    :desired-monitor-steps ',desired-monitor-steps
                    :min-performances-displayed-p ',min-performances-displayed-p))
```

```
(defmethod (:fasd-form performance) ()
  `(make-instance 'performance
                  :number ',number
                  :scheduled-start-time ',scheduled-start-time
                  :scheduled-end-time ',scheduled-end-time
                  :performance-time-window ',performance-time-window
                  :scheduled-p ',scheduled-p
                  :required-p ',required-p
                  :step-list ',step-list
                  :execute-start-up-steps-p ',execute-start-up-steps-p
                  :execute-shutdown-steps-p ',execute-shutdown-steps-p
                  :last-time-slice ',(if last-time-slice (start-time last-time-slice) nil)
                  ))

(defmethod (:fasd-form step) ()
  `(make-instance 'step
                  :id ',id
                  :number ',number
                  :scheduled-start-time ',scheduled-start-time
                  :scheduled-end-time  ',scheduled-end-time
                  :max-duration ',max-duration
                  :min-duration ',min-duration
                  :step-delay-min ',step-delay-min
                  :step-delay-max ',step-delay-max
                  :next-step ',next-step
                  :previous-step nil
                  :last-time-slice ',(if last-time-slice (start-time last-time-slice) nil)
                  :cumulative-consumable-list ',cumulative-consumable-list
                  :resource-carry-thru ',resource-carry-thru
                  :consumable-resource-list ',consumable-resource-list
                  :durable-resource-list ',durable-resource-list
                  :non-depletable-resource-list ',non-depletable-resource-list
                  :crew-requirements ',crew-requirements
                  :crew-combinations ',crew-combinations
                  :failed-crew-combinations ',failed-crew-combinations
                  :crew-lockin ',crew-lockin
                  :crew-monitor ',crew-monitor
                  :crew-duration ',crew-duration
                  :crew-cycle ',crew-cycle
                  :crew-early-shift ',crew-early-shift
                  :crew-late-shift ',crew-late-shift
                  :concurrent-with ',concurrent-with
                  :target-list ',target-list
                  :attitude-list ',attitude-list
                  :scheduled-crew-list ',scheduled-crew-list
                  :crew-monitoring-time ',crew-monitoring-time
                  :owning-object nil))

(defmethod (:fasd-form time-slice) ()
  `(make-instance 'time-slice
                  :start-time ',start-time
                  :end-time ',end-time
                  :performance-step-table ',performance-step-table
                  :crew-list ',crew-list
                  :consumable-resource-list ',consumable-resource-list
                  :cumulative-consumable-table ',cumulative-consumable-table
                  :non-depletable-resource-list ',non-depletable-resource-list
                  :durable-resource-list ',durable-resource-list
                  :target-list ',target-list
                  :attitude-list ',attitude-list
                  :next-slice ',(if next-slice next-slice nil)
                  :prev-slice ',(if prev-slice (start-time prev-slice) nil)
                  :start-x ',start-x
                  :top-y ',top-y))

(defmethod (:fasd-form time-slice-axis) ()
  `(make-instance 'time-slice-axis
                  :end-one-x ',end-one-x
                  :end-one-y ',end-one-y
                  :end-two-x ',end-two-x
                  :end-two-y ',end-two-y
                  :spike-coord-list ',spike-coord-list
                  :orientation ',orientation))
```

```
(defmethod (dump-mission-to-file mission)
           (&optional (filename "nasa-exp-sch-2:bin-files;mission-fasd-file.bin"))
  (sys:dump-forms-to-file filename
                          `((setf *mission* ',self)
                            (restore-object-linkages *mission*))
                          `(:package 'user)))

(defmethod (restore-object-linkages mission) (&rest ignore)
  (restore-object-linkages time-slice-holder)
  (loop for table in '(experiment-template-table experiment-table)
        do
    (maphash #'(lambda (exp instance)
                 exp
                 (loop for slot in '(startup-steps prototype-step-list shutdown-steps)
                       for prev-step = nil
                       do
                   (loop for step in (symbol-value-in-instance instance slot)
                         do
                     (restore-object-linkages step instance prev-step)
                     (setf prev-step step))
                 (loop for performance in (performance-list instance)
                       do
                   (restore-object-linkages performance instance))))
             (symbol-value-in-instance self table)))
  (restore-object-linkages init-obj self))

(defmethod (restore-object-linkages step) (owner prev-step)
  (setf owning-object owner)
  (if (typep owner 'experiment)
      (when prev-step (setf previous-step (id prev-step)
                            (next-step prev-step) id))
      (when prev-step (setf previous-step prev-step
                            (next-step prev-step) self))))

(defmethod (restore-object-linkages performance) (owner &rest ignore)
  (let ((last-slice nil))
  (setf owning-experiment owner)
  (loop for step in step-list
        with prev-step = nil
        do
    (restore-object-linkages step self prev-step)
    (when (scheduled-start-time step)
      (setf last-slice (get-time-instance *mission* (scheduled-start-time step) last-slice))
      (setf (last-time-slice step) last-slice))
    (setf prev-step step))
  (setf last-time-slice last-slice)))

(defmethod (restore-object-linkages time-slice) (&optional previous-slice &rest ignore)
  (when previous-slice
    (setf prev-slice previous-slice))
  (when next-slice
    (restore-object-linkages next-slice self)))

(defmethod (restore-object-linkages nasa-init-obj) (&rest ignore)
  (loop for slot in '(attitude-list target-list crew-list consumable-resource-list non-depletable-
resource-list)
        do
    (loop for resource in (symbol-value-in-instance    f slot)
          do
      (restore-object-linkages resource resource))))

(defmethod (restore-object-linkages availability) (owner &rest ignore)
  (loop for avail-obj in available-times-list
        do
    (setf (owner-obj avail-obj) owner)))

(defmethod (restore-object-linkages non-durable-resource) (owner &rest ignore)
  (loop for quant-avail-obj in quantity-availability-list
        do
    (setf (owner-obj quant-avail-obj) owner)
    (restore-object-linkages quant-avail-obj quant-avail-obj)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (output-shift-available-times mission) ()
  (loop for shift in (shift-availability-objs INIT-OBJ)
        for count from 1
        do
    (with-open-file (stream (format nil "nasa-exp-sch-2:output-data;shift-available--S"count)
                            :direction :output)
      (format stream " SHIFT AVAILABILITY TIMES FOR SHIFT ~S " count)
      (FORMAT STREAM "~%     START                                END")
      (LOOP FOR OBJ IN SHIFT
            DO
        (FORMAT STREAM "~%")
        (output-time-date-to-stream init-obj STREAM (BEGIN OBJ))
        (format stream "     ")
        (output-time-date-to-stream init-obj STREAM (END OBJ))))))

(defmethod (output-mission-data mission) (&optional (time-line-list nil))
  (fs:wildcard-map "nasa-exp-sch-2:output-data;*.*.*" #'delete-file)
  (fs:expunge-directory "nasa-exp-sch-2:output-data;")
  (if time-line-list
      (output-time-line-list self time-line-list)
      (output-time-line self nil))
  (output-scheduled-experiments sel: ))

(defmethod (output-time-line-list mission) (time-line-list)
  (loop for (time-slice exp-name) in time-line-list
        do
    (output-time-line
      self time-slice exp-name
      (format nil "nasa-exp-sch-2:output-data;time-line-data-for--S" exp-name))))

(defmethod (output-time-line mission) (&optional time-slice title filename)
  (with-open-file (stream (if filename filename "nasa-exp-sch-2:output-data;time-line-data")
                          :direction :output)
    (cond (time-slice
           (setf time-slice (find-first-slice time-slice)))
          (t (setf time-slice time-slice-holder)))
    (when time-slice
      (output-time-slice time-slice stream title))))

(defun find-first-slice (time-slice)
  (cond ((null (prev-slice time-slice)) time-slice)
        (t (find-first-slice (prev-slice time-slice)))))

(defmethod (output-time-slice time-slice) (stream title)
  (format stream "~%************************************************")
  (when title
    (format stream "~% TIMELINE ~S" title))
  (FORMAT STREAM "~%  START TIME = ")
  (output-time-date-to-stream (init-obj *MISSION*) STREAM start-time)
  (format stream "     END TIME =  ")
  (output-time-date-to-stream (init-obj *MISSION*) STREAM end-time)
  (when crew-list
  (format stream "~%   CREW MEMBER SCHEDULED DURING THIS PERIOD~%")
  (formatting-table (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
      (formatting-cell (stream :align :left) (format stream "CREW MEMBER"))
      (output-step-headings stream))
    (loop for (crew step) in crew-list
          do
      (formatting-row (stream)
        (formatting-cell (stream :align :left) (format stream "~s"(name crew)))
        (output-step-values step stream)))))
  (when consumable-resource-list
  (format stream "~%   CONSUMABLE RESOURCES SCHEDULED THIS PERIOD~%")
  (formatting-table (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
      (formatting-cell (stream :align :left) (format stream "RESOURCE"))
      (formatting-cell (stream :align :right) (format stream "QUANTITY"))
      (output-step-headings stream))
    (loop for (consumable quant step) in consumable-resource-list
          do
```

```
      (formatting-row (stream)
        (formatting-cell (stream :align :left) (format stream "~s"(name consumable)))
        (formatting-cell (stream :align :right) (format stream "~s" quant))
        (output-step-values step stream)))))
  (unless (zerop (send cumulative-consumable-table :filled-elements))
  (format stream "~%   CUMULATIVE CONSUMABLE RESOURCE USAGE~%")
  (let ((cum-list nil))
    (maphash #'(lambda (resource quant)
                 (push (list resource quant ) cum-list ))
             cumulative-consumable-table)
    (setf cum-list (sort cum-list #'alphalessp :key #'first))
    (formatting-table (stream :equalize-multiple-column-widths t )
      (formatting-column-headings (stream :underline-p nil)
        (formatting-cell (stream :align :left) (format stream "RESOURCE"))
        (formatting-cell (stream :align :right) (format stream "QUANTITY")))
      (loop for (resource quant) in cum-list
            do
        (formatting-row (stream)
          (formatting-cell (stream :align :left) (format stream "~s"(name resource)))
          (formatting-cell (stream :align :right) (format stream "~s" quant)))))))
  (when non-depletable-resource-list
  (format stream "~%   NON-DEPLETABLE RESOURCES SCHEDULED THIS PERIOD~%")
  (formatting-table (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
      (formatting-cell (stream :align :left) (format stream "RESOURCE"))
      (formatting-cell (stream :align :right) (format stream "QUANTITY"))
      (formatting-cell (stream :align :right) (format stream "TOLERANCE"))
      (output-step-headings stream))
    (loop for (non-depletable quant tolerance step) in non-depletable-resource-list
          do
      (formatting-row (stream)
        (formatting-cell (stream :align :left) (format stream "~s"(name non-depletable)))
        (formatting-cell (stream :align :right) (format stream "~s" quant))
        (formatting-cell (stream :align :right) (format stream "~s" tolerance))
        (output-step-values step stream)))))
  (when durable-resource-list
  (format stream "~%   DURABLE RESOURCES SCHEDULED THIS PERIOD~%")
  (formatting-table (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
      (formatting-cell (stream :align :left) (format stream "RESOURCE"))
      (formatting-cell (stream :align :right) (format stream "QUANTITY"))
      (output-step-headings stream))
    (loop for (durable quant step) in durable-resource-list
          do
      (formatting-row (stream)
        (formatting-cell (stream :align :left) (format stream "~s"(name durable)))
        (formatting-cell (stream :align :right) (format stream "~s" quant))
        (output-step-values step stream)))))
  (when next-slice
    (output-time-slice next-slice stream title)))

(defun output-step-headings (stream)
      (formatting-cell (stream :align :CENTER) (format stream "STEP ID"))
      (formatting-cell (stream :align :CENTER) (format stream "STEP NUMBER"))
      (formatting-cell (stream :align :CENTER) (format stream "PERFORMANCE NUMBER"))
      (formatting-cell (stream :align :CENTER) (format stream "EXPERIMENT NAME")))

(defmethod (output-step-values step) (stream)
  (formatting-cell (stream :align :CENTER) (format stream "~s" id))
  (formatting-cell (stream :align :CENTER) (format stream "~s" number))
  (formatting-cell (stream :align :CENTER) (format stream "~s" (number owning-object )))
  (formatting-cell (stream :align :CENTER)
    (format stream "~s" (name (owning-experiment owning-object )))))

(defmethod (output-scheduled-experiments mission) ()
  (maphash #'(lambda (exp instance)
               exp
               (when (performance-list instance)
                 (unless (every #'(lambda (perf)
                                    (null (scheduled-p perf))) (performance-list instance))
                   (output-performances instance))))
           experiment-table))
```

```
(defmethod (output-performances experiment) ()
  (let (days hours mins secs)
  (with-open-file (stream (format nil "nasa-exp-sch-2:output-data;exp-~S"name)
                          :direction :output)
    (format stream "~% EXPERIMENT    ~S" NAME)
    (FORMAT stream "~% MIN PERFORMANCES   ~S MAX PERFORMANCES  ~S"
            min-performances max-performances)
    (multiple-value-setq (days hours mins secs)
      (translate-mission-period-to-mission-time (init-obj *mission*)
                                                min-performance-delay-time))
    (format stream "~% MIN PERFORMANCE DELAY TIME ~S ~S ~S ~S" days hours mins secs)
    (multiple-value-setq (days hours mins secs)
      (translate-mission-period-to-mission-time (init-obj *mission*)
                                                max-performance-delay-time))
    (format stream "~% MAX PERFORMANCE DELAY TIME ~S ~S ~S ~S" days hours mins secs)
    (multiple-value-setq (days hours mins secs)
      (translate-mission-period-to-mission-time (init-obj *mission*)
                                                performance-time-window))
    (format stream "~% PERFORMANCE DURATION ~S ~S ~S ~S" days hours mins secs)
    (FORMAT STREAM "~% PERFORMANCE WINDOWS~%")
    (formatting-table (stream :equalize-multiple-column-widths t :dont-snapshot-variables t)
      (formatting-column-headings (stream :underline-p nil)
        (formatting-cell (stream :align :right) (format stream "START"))
        (formatting-cell (stream :align :right) (format stream " "))
        (formatting-cell (stream :align :right) (format stream " "))
        (formatting-cell (stream :align :right) (format stream " "))
        (formatting-cell (stream :align :right) (format stream "END"))
        (formatting-cell (stream :align :right) (format stream " "))
        (formatting-cell (stream :align :right) (format stream " "))
        (formatting-cell (stream :align :right) (format stream " "))
        (formatting-cell (stream :align :right) (format stream "NUMBER OF PERFORMANCES")))
      (LOOP FOR (START END PERFORMANCES) IN performance-windows
            DO
        (formatting-row (stream :dont-snapshot-variables t)
            (multiple-value-setq (days hours mins secs)
              (translate-mission-period-to-mission-time (init-obj *mission*) START))
            (formatting-cell (stream :align :right) (format stream "~S" days))
            (formatting-cell (stream :align :right) (format stream "~S" hours))
            (formatting-cell (stream :align :right) (format stream "~S" mins))
            (formatting-cell (stream :align :right) (format stream "~S" secs))
            (multiple-value-setq (days hours mins secs)
              (translate-mission-period-to-mission-time (init-obj *mission*) END))
            (formatting-cell (stream :align :right) (format stream "~S" days))
            (formatting-cell (stream :align :right) (format stream "~S" hours))
            (formatting-cell (stream :align :right) (format stream "~S" mins))
            (formatting-cell (stream :align :right) (format stream "~S" secs))
            (formatting-cell (stream :align :right) (format stream "~S" PERFORMANCES)))))
    (when strategy
      (format stream "~%STRATEGY")
      (LOOP FOR (strat-list weight) in strategy
            do
        (format stream "~% WEIGHT ~S STEPS  " WEIGHT )
        (LOOP FOR ELEMENT IN STRAT-LIST
              DO
          (COND ((EQL (FIRST ELEMENT) :CONSECUTIVE)
                 (FORMAT STREAM ", ~S THRU ~S" (SECOND ELEMENT) (THIRD ELEMENT)))
                ((EQL (FIRST ELEMENT) :SEQUENTIAL)
                 (LOOP FOR STEP-NUMBER IN (SECOND ELEMENT)
                       DO
                   (FORMAT STREAM ",~S "STEP-NUMBER)))))))
    (loop for performance in
              (setf performance-list (sort performance-list #'< :key #'number))
          do
      (when (scheduled-p performance)
        (output-performance performance stream))))))

(defmethod (output-performance performance) (stream)
  (LET (days hours mins secs)
    (format stream "~%**************************************************")
  (format stream "~%~%     PERFORMANCE  ~S" NUMBER)
  (multiple-value-setq (days hours mins secs)
    (translate-mission-period-to-mission-time (init-obj *mission*) SCHEDULED-START-TIME))
  (format stream "~% SCHEDULED START TIME ~S ~S ~S ~S" days hours mins secs)
```

```
  (multiple-value-setq (days hours mins secs)
    (translate-mission-period-to-mission-time (init-obj *mission*) SCHEDULED-END-TIME))
  (format stream "~% SCHEDULED END TIME ~S ~S ~S ~S" days hours mins secs))
  (LOOP FOR STEP IN STEP-LIST
        DO
    (OUTPUT-STEP STEP STREAM)))

(defmethod (output-prototype-experiments mission) ()
  (maphash #'(lambda (key value)
               key
               (output-prototype-experiment value))
           experiment-template-table ))

(DEFMETHOD (OUTPUT-BAD-EXPERIMENTS MISSION) ()
  (LOOP FOR EXPERIMENT-NAME IN '(ALLOY-S BRIDGMAN CONTFLOW HW-MAINT VAP-CRYS WM-MAINT)
        FOR EXPERIMENT = (GETHASH EXPERIMENT-NAME EXPERIMENT-TEMPLATE-TABLE)
        DO
    (output-prototype-experiment EXPERIMENT)))

(defmethod (output-prototype-experiment experiment) ()
  (with-open-file (stream (format nil "nasa-exp-sch-2:output-data;prototype-exp--S" name)
                          :direction :output
    (let (days hours mins secs)
      (format stream "~% EXPERIMENT    ~S" NAME)
      (FORMAT stream "~% MIN PERFORMANCES   ~S MAX PERFORMANCES  ~S"
              min-performances max-performances)
      (multiple-value-setq (days hours mins secs)
        (translate-mission-period-to-mission-time (init-obj *mission*)
                                                  min-performance-delay-time))
      (format stream "~% MIN PERFORMANCE DELAY TIME ~S ~S ~S ~S" days hours mins secs)
      (multiple-value-setq (days hours mins secs)
        (translate-mission-period-to-mission-time (init-obj *mission*)
                                                  max-performance-delay-time))
      (format stream "~% MAX PERFORMANCE DELAY TIME ~S ~S ~S ~S" days hours mins secs)
      (multiple-value-setq (days hours mins secs)
        (translate-mission-period-to-mission-time (init-obj *mission*)
                                                  performance-time-window))
      (format stream "~% PERFORMANCE DURATION ~S ~S ~S ~S" days hours mins secs)
      (FORMAT STREAM "~% PERFORMANCE WINDOWS~%")
      (formatting-table (stream :equalize-multiple-column-widths t :dont-snapshot-variables t)
        (formatting-column-headings (stream :underline-p nil)
          (formatting-cell (stream :align :right) (format stream "START"))
          (formatting-cell (stream :align :right) (format stream " "))
          (formatting-cell (stream :align :right) (format stream " "))
          (formatting-cell (stream :align :right) (format stream " "))
          (formatting-cell (stream :align :right) (format stream "END"))
          (formatting-cell (stream :align :right) (format stream " "))
          (formatting-cell (stream :align :right) (format stream " "))
          (formatting-cell (stream :align :right) (format stream " "))
          (formatting-cell (stream :align :right) (format stream "NUMBER OF PERFORMANCES")))
        (LOOP FOR (START END PERFORMANCES) IN performance-windows
              DO
          (formatting-row (stream :dont-snapshot-variables t)
            (multiple-value-setq (days hours mins secs)
              (translate-mission-period-to-mission-time (init-obj *mission*) START))
            (formatting-cell (stream :align :right) (format stream "~S" days))
            (formatting-cell (stream :align :right) (format stream "~S" hours))
            (formatting-cell (stream :align :right) (format stream "~S" mins))
            (formatting-cell (stream :align :right) (format stream "~S" secs))
            (multiple-value-setq (days hours mins secs)
              (translate-mission-period-to-mission-time (init-obj *mission*) END))
            (formatting-cell (stream :align :right) (format stream "~S" days))
            (formatting-cell (stream :align :right) (format stream "~S" hours))
            (formatting-cell (stream :align :right) (format stream "~S" mins))
            (formatting-cell (stream :align :right) (format stream "~S" secs))
            (formatting-cell (stream :align :right) (format stream "~S" PERFORMANCES)))))))
      (when strategy
        (format stream "~%STRATEGY")
        (LOOP FOR (strat-list weight) in strategy
              do
          (format stream "~% WEIGHT ~S STEPS  ' WEIGHT )
          (LOOP FOR ELEMENT IN STRAT-LIST
                DO
```

```
          (COND ((EQL (FIRST ELEMENT) :CONSECUTIVE)
                 (FORMAT STREAM ", ~S THRU ~S" (SECOND ELEMENT) (THIRD ELEMENT)))
                ((EQL (FIRST ELEMENT) :SEQUENTIAL)
                 (LOOP FOR STEP-NUMBER IN (SECOND ELEMENT)
                       DO
                   (FORMAT STREAM ",~S "STEP-NUMBER)))))))
      (when non-depletable-tolerance-list
        (format stream "~% NON-DEPLETABLE RESOURCE TOLERANCES~%")
        (FORMATTING-TABLE
          (stream :equalize-multiple-column-widths t :dont-snapshot-variables t)
          (formatting-column-headings (stream :underline-p nil)
            (formatting-cell (stream :align :left) (format stream "RESOURCE"))
            (formatting-cell (stream :align :right) (format stream "TOLERANCE")))
          (LOOP FOR (RESOURCE TOLERANCE) IN non-depletable-tolerance-list
                DO
            (formatting-cell (stream :align :left) (format stream "~A" RESOURCE))
            (formatting-cell (stream :align :right) (format stream "~A" TOLERANCE)))))
      (WHEN crew-lockin
        (FORMAT STREAM "~% CREW LOCKIN REQUIREMENTS~%")
        (FORMATTING-TABLE
          (stream :equalize-multiple-column-widths t :dont-snapshot-variables t)
          (formatting-column-headings (stream :underline-p nil)
            (formatting-cell (stream :align :CENTER) (format stream "FROM STEP"))
            (formatting-cell (stream :align :CENTER) (format stream "THRU STEP")))
          (LOOP FOR (START-STEP END-STEP) IN crew-lockin
                DO
            (formatting-cell (stream :align :CENTER) (format stream "~A" START-STEP))
            (formatting-cell (stream :align :CENTER) (format stream "~A" END-STEP)))))
      (COND (STRATEGY
             (FORMAT STREAM "~%~% STEPS")
             (loop for step in prototype-step-list
                   do
               (output-step step stream )))
            (T
             (format stream "~% START UP STEPS")
             (loop for step in startup-steps
                   do
               (output-step step stream ))
             (format stream "~% CORE STEPS")
             (loop for step in prototype-step-list
                   do
               (output-step step stream ))
             (format stream "~% SHUTDOWN STEPS")
             (loop for step in shutdown-steps
                   do
               (output-step step stream )))))))

(DEFMETHOD (OUTPUT-STEP STEP) (STREAM)
  (format stream "~%**************************************************")
  (LET (DAYS HOURS MINS SECS)
    (FORMAT STREAM "~%~%          STEP      ~S   NUMBER    ~S~%" ID NUMBER)
    (formatting-table (stream :equalize-multiple-column-widths t :dont-snapshot-variables t)
      (formatting-column-headings (stream :underline-p nil)
        (formatting-cell (stream :align :left) (format stream " "))
      (formatting-cell (stream :align :right) (format stream "DAYS"))
      (formatting-cell (stream :align :right) (format stream "HOURS "))
      (formatting-cell (stream :align :right) (format stream "MINUTES "))
      (formatting-cell (stream :align :right) (format stream "SECONDS ")))
      (LOOP FOR SLOT IN '(SCHEDULED-START-TIME SCHEDULED-END-TIME max-duration min-duration
                                              step-delay-min step-delay-max)
            FOR LABEL IN '("SCHEDULED START TIME" "SCHEDULED END TIME" "MAX DURATION"
                           "MIN DURATION" "MIN DELAY" "MAX DELAY")

            DO
        (FORMATTING-ROW (STREAM :dont-snapshot-variables t)
          (multiple-value-setq (days hours mins secs)
            (translate-mission-period-to-mission-time (init-obj *mission*)
                                                      (symbol-value-in-instance self SLOT )))
          (formatting-cell (stream :align :left) (format stream "~A" label))
          (formatting-cell (stream :align :right) (format stream "~S" days))
          (formatting-cell (stream :align :right) (format stream "~S" hours))
          (formatting-cell (stream :align :right) (format stream "~S" mins))
          (formatting-cell (stream :align :right) (format stream "~S" secs)))))))
```

```
(when durable-resource-list
  (format stream "~%DURABLE RESOURCES~%")
  (FORMATTING-TABLE (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
    (formatting-cell (stream :align :left) (format stream "RESOURCE"))
    (formatting-cell (stream :align :right) (format stream "QUANTITY")))
    (LOOP FOR (RESOURCE QUANT) IN DURABLE-RESOURCE-LIST
          DO
      (FORMATTING-ROW (STREAM)
        (FORMATTING-CELL (stream :align :left) (FORMAT STREAM "~S"(NAME RESOURCE)))
        (FORMATTING-CELL (stream :align :right) (FORMAT STREAM "~S" QUANT))))))
(when NON-DEPLETABLE-resource-list
  (format stream "~%NON-DEPLETABLE RESOURCES~%")
  (FORMATTING-TABLE (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
    (formatting-cell (stream :align :left) (format stream "RESOURCE"))
    (formatting-cell (stream :align :right) (format stream "QUANTITY")))
    (LOOP FOR (RESOURCE QUANT) IN NON-DEPLETABLE-RESOURCE-LIST
          DO
      (FORMATTING-ROW (STREAM)
        (FORMATTING-CELL (stream :align :left) (FORMAT STREAM "~S"(NAME RESOURCE)))
        (FORMATTING-CELL (stream :align :right) (FORMAT STREAM "~S" QUANT))))))
(when CONSUMABLE-resource-list
  (format stream "~%CONSUMABLE RESOURCES~%")
  (FORMATTING-TABLE (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
    (formatting-cell (stream :align :left) (format stream "RESOURCE"))
    (formatting-cell (stream :align :right) (format stream "QUANTITY")))
    (LOOP FOR (RESOURCE QUANT) IN CONSUMABLE-RESOURCE-LIST
          DO
      (FORMATTING-ROW (STREAM)
        (FORMATTING-CELL (stream :align :left) (FORMAT STREAM "~S"(NAME RESOURCE)))
        (FORMATTING-CELL (stream :align :right) (FORMAT STREAM "~S" QUANT))))))
(WHEN cumulative-consumable-list
  (FORMAT STREAM "~%CUMULATIVE CONSUMABLES~%")
  (FORMATTING-TABLE (stream :equalize-multiple-column-widths t )
    (formatting-column-headings (stream :underline-p nil)
    (formatting-cell (stream :align :left) (format stream "RESOURCE"))
    (formatting-cell (stream :align :right) (format stream "QUANTITY")))
    (LOOP FOR (RESOURCE QUANT) IN cumulative-consumable-list
          DO
      (FORMATTING-ROW (STREAM)
        (FORMATTING-CELL (stream :align :left) (FORMAT STREAM "~S"(NAME RESOURCE)))
        (FORMATTING-CELL (stream :align :right) (FORMAT STREAM "~S" QUANT))))))
(when crew-requirements
  (format stream "~%CREW REQUIREMENTS")
  (loop for (crew-list quant) in crew-requirements
        do
    (format stream "~% NUMBER REQUIRED ~S FROM THE FOLLOWING:" quant)
    (loop for (specification tag) in crew-list
          do
      (format stream "~%IDENTIFIER   ~S  IDENTITY  ~S" specification tag)))
  (FORMAT STREAM "~%POSSIBLE CREW COMBINATIONS")
  (LOOP FOR CREW-LIST IN crew-combinations
        DO
    (FORMAT STREAM "~% COMBINATION:  ")
    (LOOP FOR CREW IN CREW-LIST
          DO
      (FORMAT STREAM "~S   "(NAME CREW))))
  (cond (crew-monitor
         (format stream "~% CREW MONITOR: ~S ~%" CREW-MONITOR)
         (FORMATTING-TABLE
           (stream :equalize-multiple-column-widths t :DONT-SNAPSHOT-VARIABLES T)
           (formatting-column-headings (stream :underline-p nil)
             (formatting-cell (stream :align :left) (format stream " "))
             (formatting-cell (stream :align :RIGHT) (format stream "DAYS"))
             (formatting-cell (stream :align :RIGHT) (format stream "HOURS"))
             (formatting-cell (stream :align :RIGHT) (format stream "MINUTES"))
             (formatting-cell (stream :align :RIGHT) (format stream "SECONDS")))
           (LOOP FOR SLOT IN '(CREW-CYCLE CREW-DURATION CREW-EARLY-SHIFT CREW-LATE-SHIFT)
                 FOR LABEL IN '("MONITOR CYCLE:" "DURATION OF MONITOR:"
                                "MAX MONITOR EARLY SHIFT:" "MAX MONITOR LATE SHIFT:")
                 DO
```

```
          (multiple-value-setq (days hours mins secs)
            (translate-mission-period-to-mission-time (init-obj *mission*)
                                                      (SYMBOL-VALUE-IN-INSTANCE SELF SLOT)))
          (FORMATTING-ROW (STREAM :DONT-SNAPSHOT-VARIABLES T)
            (formatting-cell (stream :align :left) (format stream "~A"LABEL))
            (formatting-cell (stream :align :RIGHT) (format stream "~S" DAYS))
            (formatting-cell (stream :align :RIGHT) (format stream "~S" hours))
            (formatting-cell (stream :align :RIGHT) (format stream "~S" mins))
            (formatting-cell (stream :align :RIGHT) (format stream "~S" secs)))))
          (WHEN scheduled-crew-list
            (FORMAT STREAM "~%SCHEDULED CREW LIST:   ~%")
            (formatting-table
               (stream :equalize-multiple-column-widths t :DONT-SNAPSHOT-VARIABLES T)
              (formatting-column-headings (stream :underline-p nil)
              (formatting-cell (stream :align :left) (format stream "FROM"))
              (formatting-cell (stream :align :RIGHT) (format stream "DAYS"))
              (formatting-cell (stream :align :RIGHT) (format stream "HOURS"))
              (formatting-cell (stream :align :RIGHT) (format stream "MINUTES"))
              (formatting-cell (stream :align :RIGHT) (format stream "SECONDS"))
              (formatting-cell (stream :align :left) (format stream "TO"))
              (formatting-cell (stream :align :RIGHT) (format stream "DAYS"))
              (formatting-cell (stream :align :RIGHT) (format stream "HOURS"))
              (formatting-cell (stream :align :RIGHT) (format stream "MINUTES"))
              (formatting-cell (stream :align :RIGHT) (format stream "SECONDS"))
              (formatting-cell (stream :align :RIGHT) (format stream "USING"))
              (LOOP FOR i FROM 2 TO (LENGTH (FIRST (FIRST SCHEDULED-CREW-LIST)))
                    DO
                (formatting-cell (stream :align :RIGHT) (format stream " "))))
            (LOOP FOR (CREW-LIST START END) IN SCHEDULED-CREW-LIST
                  DO
              (FORMATTING-ROW (STREAM :DONT-SNAPSHOT-VARIABLES T)
                (formatting-cell (stream :align :left) (format stream " "))
              (multiple-value-setq (days hours mins secs)
                (translate-mission-period-to-mission-time (init-obj *mission*) START))
              (formatting-cell (stream :align :RIGHT) (format stream "~S" DAYS))
              (formatting-cell (stream :align :RIGHT) (format stream "~S" hours))
              (formatting-cell (stream :align :RIGHT) (format stream "~S" mins))
              (formatting-cell (stream :align :RIGHT) (format stream "~S" secs))
              (formatting-cell (stream :align :left) (format stream " "))
              (multiple-value-setq (days hours mins secs)
                (translate-mission-period-to-mission-time (init-obj *mission*) END))

              (formatting-cell (stream :align :RIGHT) (format stream "~S" DAYS))
              (formatting-cell (stream :align :RIGHT) (format stream "~S" hours))
              (formatting-cell (stream :align :RIGHT) (format stream "~S" mins))
              (formatting-cell (stream :align :RIGHT) (format stream "~S" secs))
              (LOOP FOR CREW IN CREW-LIST
                    DO
                (formatting-cell (stream :align :RIGHT) (format stream "~S" (NAME CREW))))
             )))))
        (T
         (FORMAT STREAM "~%SCHEDULED CREW LIST:   ")
         (LOOP FOR CREW IN scheduled-crew-list
               DO
           (FORMAT STREAM "~S   "(NAME CREW))))))
(WHEN TARGET-LIST
  (FORMAT STREAM "~% TARGET INFORMATION")
  (LOOP FOR (DESIGNATOR SUBLIST) IN TARGET-LIST
        DO
    (CASE DESIGNaTOR
      (:AVOID (FORMAT STREAM "~% TARGETS TO BE AVOIDED~%"))
      (:INTERSECT (FORMAT STREAM "~% TARGETS WHOSE PRESENCE MUST INTERSECT~%"))
      (:SELECT (FORMAT STREAM "~%TARGETS OF WHICH AT LEAST ONE MUST BE PRESENT~%")))
    (LOOP FOR TARGET IN SUBLIST
          DO
      (FORMAT STREAM "~S   "(NAME TARGET)))))
(WHEN attitude-list
  (FORMAT STREAM "~%ATTITUDE INFORMATION~%")
  (LOOP FOR (DESIGNATOR SUBLIST) IN ATTITUDE-LIST
        DO
    (CASE DESIGNATOR
      (:AVOID (FORMAT STREAM "~%ATTITUDES TO BE AVOIDED"))
      (:INTERSECT (FORMAT STREAM "~%ATTITUDES WHOSE PRESENCE MUST INTERSECT~%"))
```

```
          (:SELECT (FORMAT STREAM "~%ATTITUDES OF WHICH AT LEAST ONE MUST BE PRESENT~%")))
        (LOOP FOR ATTITUDE IN SUBLIST
              DO
          (FORMAT STREAM "~S    "(NAME ATTITUDE)))))
    (WHEN PREVIOUS-STEP
      (FORMAT STREAM "~% PREVIOUS STEP: ~S" (IF (SYMBOLP previous-step ) previous-step
                                                (id previous-step))))
    (WHEN NEXT-STEP
      (FORMAT STREAM "~% NEXT STEP: ~S" (IF (SYMBOLP NEXT-step ) NEXT-step
                                            (id NEXT-step))))))

(defmethod (output-durable-resource durable-resource) (stream)
  (format stream "~%~% DURABLE RESOURCE ~S ~S" name available-quantity ))

(defmethod (output-non-depletable-resource non-depletable-resource) (stream)
  (format stream "~%~% NON DEPLETABLE RESOURCE ~S" name)
  (output-non-durable-resource self stream))

(defmethod (output-consumable-resource consumable-resource) (stream)
  (format stream "~%~% CONSUMABLE RESOURCE ~S" name)
  (output-non-durable-resource self stream))

(defmethod (output-non-durable-resource non-durable-resource) (stream)
  (loop for qty-avail in quantity-availability-list
        do
    (format stream "~% Quantity ~S Available in Time Periods: ~%     BEGIN
END"
            (qty qty-avail))
    (loop for avail-obj in (available-times-list qty-avail)
          do
        (FORMAT STREAM "~%")
        (output-time-date-to-stream (init-obj *mission*) STREAM (begin avail-obj))
        (format stream "     ")
        (output-time-date-to-stream (init-obj *mission*) STREAM (end avail-obj)))))

(defmethod (output-durable-resources nasa-init-obj) ()
  (with-open-file (stream "nasa-exp-sch-2:output-data;durable-resources" :direction :output)
    (loop for durable-resource in durable-resource-list
          do
      (output-durable-resource durable-resource stream))))

(defmethod (output-non-depletable-resources nasa-init-obj) ()
  (with-open-file (stream "nasa-exp-sch-2:output-data;non-depletable-resources"
                          :direction :output)
    (loop for non-depletable-resource in non-depletable-resource-list
          do
      (output-non-depletable-resource non-depletable-resource stream))))

(defmethod (output-consumable-resources nasa-init-obj) ()
  (with-open-file (stream "nasa-exp-sch-2:output-data;consumable-resources" :direction :output)
    (loop for consumable-resource in consumable-resource-list
          do
      (output-consumable-resource consumable-resource stream))))

(defmethod (output-resources nasa-init-obj) ()
  (output-durable-resources self)
  (output-non-depletable-resources self)
  (output-consumable-resources self))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (compute-and-store-cumulative-consumption performance) (&rest ignore)
  (setf (cumulative-consumable-list (first step-list))
        (consumable-resource-list (first step-list)))
  (when (second step-list)
    (compute-and-store-cumulative-consumption
      (second step-list) (cumulative-consumable-list (first step-list)))))

(defmethod (compute-and-store-cumulative-consumption step) (prev-consum-list)
  (loop for (resource quant) in prev-consum-list
        for same-resource = (member resource consumable-resource-list :key #'first)
        do
    (if same-resource
        (push (list resource (+ quant (second (first same-resource))))
              cumulative-consumable-list)
        (push (list resource quant) cumulative-consumable-list)))
  (loop for (resource quant) in consumable-resource-list
        for already-included-p = (member resource cumulative-consumable-list :key #'first)
        do
    (unless already-included-p
      (push (list resource quant) cumulative-consumable-list)))
  (when next-step
    (compute-and-store-cumulative-consumption next-step cumulative-consumable-list)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

;*****************************************************************************
;;;object presented for init-obj edit
(DEFINE-PRESENTATION-TO-COMMAND-TRANSLATOR
  PERFORMANCE-SCHEDULER-CREATE-NEW-resource
   (MISC-OBJ-EDIT-DISPLAY
     :GESTURE :LEFT
     :DOCUMENTATION "Create A New Resource Object"
     )
  (owner-object)
  (cp:build-command 'com-performance-scheduler-create-new-resource
                    owner-object))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND
  (COM-PERFORMANCE-SCHEDULER-CREATE-NEW-RESOURCE)
    ((owner-object 'misc-obj-edit-display))
   (create-new-obj owner-object)
   )

(DEFINE-PRESENTATION-TO-COMMAND-TRANSLATOR
  PERFORMANCE-SCHEDULER-DELETE-RESOURCE
   (MISC-OBJ-EDIT-DISPLAY
     :GESTURE :MIDDLE
     :DOCUMENTATION "Delete A Resource Object"
     )
  (owner-object)
  (cp:build-command 'com-performance-scheduler-delete-resource
                    owner-object))

 (DEFINE-PERFORMANCE-SCHEDULER-COMMAND
  (COM-PERFORMANCE-SCHEDULER-DELETE-RESOURCE)
    ((owner-object 'misc-obj-edit-display))
   (delete-resource owner-object)
   )


(DEFINE-PRESENTATION-TO-COMMAND-TRANSLATOR
  PERFORMANCE-SCHEDULER-ADD-AVAILABLE-TIME
   (NAME-FOR-EDIT-DISPLAY
     :GESTURE :LEFT
     :DOCUMENTATION "Add Additional Times This Resource Available"
     )
  (owner-object)
  (cp:build-command 'com-performance-scheduler-add-available-time
                    owner-object))

 (DEFINE-PERFORMANCE-SCHEDULER-COMMAND
  (COM-PERFORMANCE-SCHEDULER-ADD-AVAILABLE-TIME)
    ((owner-object 'name-for-edit-display))
   (add-available-time owner-object))

(DEFINE-PRESENTATION-TO-COMMAND-TRANSLATOR
  PERFORMANCE-SCHEDULER-delete-AVAILABLE-TIME
   (NAME-FOR-EDIT-DISPLAY
     :GESTURE :middle
     :DOCUMENTATION "Delete Time Period This Resource Available"
     )
  (owner-object)
  (cp:build-command 'com-performance-scheduler-delete-available-time
                    owner-object))

 (DEFINE-PERFORMANCE-SCHEDULER-COMMAND
  (COM-PERFORMANCE-SCHEDULER-DELETE-AVAILABLE-TIME)
    ((owner-object 'name-for-edit-display))
   (delete-available-time owner-object))

(DEFINE-PRESENTATION-TO-COMMAND-TRANSLATOR
  PERFORMANCE-SCHEDULER-ADD-AVAILABLE-TIME-FOR-QUANTITY
   (QUANTITY-AVAILABILITY-EDIT-DISPLAY
     :GESTURE :LEFT
     :DOCUMENTATION "Add Additional Times This Quantity Available"
     )
```

```
  (owner-object)
  (cp:build-command 'com-performance-scheduler-add-available-time-for-quantity
                    owner-object))

 (DEFINE-PERFORMANCE-SCHEDULER-COMMAND
  (COM-PERFORMANCE-SCHEDULER-ADD-AVAILABLE-TIME-FOR-QUANTITY)
    ((owner-object 'quantity-availability-edit-display))
   (add-available-time owner-object))

(DEFINE-PRESENTATION-TO-COMMAND-TRANSLATOR
  PERFORMANCE-SCHEDULER-ADD-QUANTITY-AND-AVAILABILITY
   (CONSUMABLE-NAME-FOR-EDIT-DISPLAY
     :GESTURE :LEFT
     :DOCUMENTATION "Add Additional Quantity And Times This Resource Available"
     )
  (owner-object)
  (cp:build-command 'com-performance-scheduler-add-quantity-and-availability
                    owner-object))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND
  (COM-PERFORMANCE-SCHEDULER-ADD-QUANTITY-AND-AVAILABILITY)
    ((owner-object 'consumable-name-for-edit-display))
   (add-quantity-availability owner-object))

;*****************************************************************************
;;; objects presented for experiment template edit
(DEFINE-PRESENTATION-TO-COMMAND-TRANSLATOR
  PERFORMANCE-SCHEDULER-CREATE-NEW-step
   (experiment-template-name-edit-display
     :GESTURE :LEFT
     :DOCUMENTATION "Create A New Step"
     )
  (owner-object)
  (cp:build-command 'com-performance-scheduler-create-step
                    owner-object))

(DEFINE-PERFORMANCE-SCHEDULER-COMMAND
  (com-performance-scheduler-create-step)
    ((owner-object 'experiment-template-name-edit-display))
   (create-new-step owner-object)
   )
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (delete-resource query-obj) ()
  (delete-resource *mission* type))

(defmethod (create-new-obj query-obj) ()
  (create-new-obj (make-instance type)))

(defmethod (create-new-obj durable-resource) ()
  (let ((new-name 'unnamed)(new-available-quantity 0))
    (dw:accepting-values
      (*standard-output* :own-window t
                         :label
                         (with-character-style ('(:fix :bold :very-large )
                                                nil :bind-line-height t)
                           "Describe          New          Resource                "))
      (setf new-name
            (accept 'symbol :default new-name :query-identifier 'new-name
                    :stream *standard-output*
                    :prompt (format nil "Enter Name of Durable Resource"))
            new-available-quantity
            (accept 'number :default new-available-quantity
                    :query-identifier 'new-available-quantity
                    :stream *standard-output* :prompt
                    (format nil "Enter Quantity of Durable Resource Available"))))
    (setf name new-name available-quantity new-available-quantity))
  (add-resource *mission* self 'durable-resource-list))

(defmethod (create-new-obj consumable-resource) ()
  (let ((new-name 'unnamed))
    (dw:accepting-values
      (*standard-output* :own-window t :label
                         (with-character-style ('(:fix :bold :very-large )
                                                nil :bind-line-height t)
                           "Describe          New          Resource                "))
      (setf new-name
            (accept 'symbol :default new-name :query-identifier 'new-name
                    :stream *standard-output*
                    :prompt (format nil "Enter Name of Consumable Resource"))))
    (setf name new-name))
  (add-quantity-availability self)
  (add-resource *mission* self 'consumable-resource-list))


(defmethod  (quantity-already-exists-p consumable-resource) (new-quantity)
  (loop for quantity-availability in quantity-availability-list
        do
    (when (= new-quantity (qty quantity-availability))
      (report-error *mission* (format nil "~%An object already exists for consumable resource ~S o
f quantity ~S.  New availability times must be added to the existing object" name new-quantity))
      (return t))))

(defmethod (add-quantity-availability consumable-resource) ()
  (let ((qty-avail-obj nil) (choice nil) (new-quantity 0) )
    (loop until (and choice (eql choice 'no))
          do
      (loop until
            (setf choice
                  (dw:menu-choose
                    '((yes yes) (no no))
                    :prompt (format nil "Describe Another Quantity For ~S?"name))))
      (unless (eql choice 'no)
        (setf qty-avail-obj (make-instance 'quantity-availability :owner-obj self :name name))
        (dw:accepting-values
          (*standard-output* :own-window t :label
                             (with-character-style ('(:fix :bold :very-large )
                                                    nil :bind-line-height t)
                               "Describe          New          Resource                "))
          (setf new-quantity
                (accept 'number :default new-quantity :query-identifier 'new-quantity
                        :stream *standard-output*
                        :prompt (format nil "Enter Quantity Available          "))))
        (unless (quantity-already-exists-p self new-quantity)
```

```
          (setf (qty qty-avail-obj) new-quantity)
          (get-available-times
            qty-avail-obj
            (format nil "Specify An Available Time Period for Quantity ~S of ~S?"
                    new-quantity name))
          (push qty-avail-obj quantity-availability-list))))
    ))

(defmethod (add-available-time availability) ()
  (get-available-times self (format nil "Specify An Available Time Period for ~S?" name)))

(defmethod (delete-available-time availability) ()
  (let ((choice-list (loop for avail-obj in available-times-list
                           collect (list (format nil "~A thru ~A" (begin avail-obj)
                                                 (end avail-obj)) avail-obj))))
        (choice nil))
    (loop until (setf choice (dw:menu-choose (push '(NONE NONE) choice-list)
                                             :prompt "Choose time period to delete or NONE")))
    (unless (eql choice 'none)
      (setf available-times-list (delete choice available-times-list)))))

(defmethod (get-available-times availability) (query-string)
  ;;;get-available-times elicites the times that a resource is to be available and
  ;;;checks whether the new times are logical (begin before end) and ensures they
  ;;;don't overlap other times.  Additionally, if the object is a
  ;;;quantity-availability (implicitly, belonging to a consumable resource, checks
  ;;;not only the current quantity but other quantities as well.
  (let ((avail-obj nil) (choice nil) (new-begin 0) (new-end 0))
    (loop until (and choice (eql choice 'no))
          do
      (loop until
            (setf choice
                  (dw:menu-choose
                    '((yes yes) (no no))
                    :prompt query-string
                    )))
      (unless (eql choice 'no)
        (setf avail-obj (make-instance 'available-time :owner-obj self))
        (dw:accepting-values
          (*standard-output* :own-window t :label
                             (with-character-style ('(:fix :bold :very-large )
                                                    nil :bind-line-height t)
                               "Describe      Available      Times                    "))
          (setf
          new-begin
          (accept 'number :default new-begin
                  :query-identifier 'new-begin
                  :stream *standard-output* :prompt
                  (format nil "Enter Time Resource Becomes Available     "))
          new-end
          (accept 'number :default new-end
                  :query-identifier 'new-end
                  :stream *standard-output* :prompt
                  (format nil "Enter Last Time Resource is Available     "))))
        (setf (begin avail-obj) new-begin)
        (setf (end avail-obj) new-end)
        (unless (improper-times-p self new-begin new-end)
              (push avail-obj available-times-list))))))

(defmethod (improper-times-p availability) (new-begin new-end)
  (cond ((< new-end new-begin)
         (report-error
           *mission*
           (format
             nil
             "attempt to specify an end time earlier that the start time for ~S of type ~S"
             (name self) (type-of self)))
         t)
        ((= new-begin new-end)
         (report-error
           *mission*
           (format nil
                   "attempt to specify an end time equal to the start time for ~S of type ~S"
```

```
                    (name self) (type-of self)))
          t)
         ((overlapping-times-p self new-begin new-end) t)
         (t nil)))

(defmethod (overlapping-times-p availability) (new-begin new-end)
  (overlapping-times-p-aux self new-begin new-end))

(defmethod (overlapping-times-p quantity-availability) (new-begin new-end)
  (overlapping-times-p (owner-obj self) new-begin new-end (qty self)))

(defmethod (overlapping-times-p consumable-resource) (new-begin new-end quant)
  (loop for quantity-availability in quantity-availability-list
        do
    (when (overlapping-times-p-aux quantity-availability new-begin new-end quant)
      (return t))))

(defmethod (overlapping-times-p-aux availability) (new-begin new-end &optional quant)
  (loop for available-time in available-times-list
        do
    (unless
      (or (and (< new-begin (begin available-time))
               (< new-end (begin available-time)))
          (and (> new-begin (end available-time))
               (> new-end (end available-time))))
      (report-error
        *mission*
        (if (typep self 'quantity-availability)
            (format nil
                    "the new beginning ~S and ending time ~S for quantity ~S overlap an existing a
vailable time frame.  You must modify the exiting one first, whose beginning time is ~S and ending
 time is ~S for ~S , quantity = ~S, of type ~S" new-begin new-end quant (begin available-time) (en
d available-time) (name self) (qty self) (type-of self))
          (format nil
                  "the new beginning ~S and ending time ~S overlap an existing available time frame.
  You must modify the exiting one first, whose beginning time is ~S and ending time is ~S for ~S o
f type ~S"
                  new-begin new-end (begin available-time) (end available-time) (name self)
                  (type-of self))))
      (return t))))

(defmethod (create-new-obj crew-member) ()
  (get-name-and-available-times self)
  (add-resource *mission* self 'crew-list))

(defmethod (get-name-and-available-times availability) ()
  (let ((new-name 'unknown))
    (dw:accepting-values (*standard-output* :own-window t :label "Enter Name of New Resource")
      (setf new-name (accept 'symbol :default new-name :query-identifier 'new-name
                             :stream *standard-output* :prompt
                             "Enter Name  ")))
    (setf name new-name)
    (get-available-times
      self (format nil "Specify An Available Time Period for ~S?" new-name))))

(defmethod (create-new-obj attitude) ()
  (get-name-and-available-times self)
  (add-resource *mission* self 'attitude-list))

(defmethod (create-new-obj target) ()
  (get-name-and-available-times self)
  (add-resource *mission* self 'target-list))

;;;*****************************************************************************
;;; methods to program crew member shifts

(defmethod (setup-crew-member-duty-shifts nasa-init-obj) ()
  (setf seconds-per-shift (/ seconds-per-day 2))
  (correct-shift-start-time-representation self)
  (setf shift-availability-objs
        (list (setup-crew-member-duty-shifts-aux
                self
                1 )
```

```
                (setup-crew-member-duty-shifts-aux
                  self 2)))
  nil

(defmethod (create-first-available-time-period nasa-init-obj)
           (shift-number)
   (let ((start-time (second (assoc shift-number shift-start-times))))
     (values
       (make-instance
                 'available-time
                 :begin (if (< start-time universal-start-time)
                            0
                            (translate-universal-time-to-time-period start-time))
                 :end (1- (translate-universal-time-to-time-period
                            (+ start-time seconds-per-shift)))))
       start-time seconds-per-day))))

(defmethod (correct-shift-start-time-representation nasa-init-obj) ()
   (setf shift-start-times
         (loop for (shift-num start-time-list) in shift-start-times
               collect
                 (list shift-num
                         (+ universal-start-time
                            (translate-time-list-to-seconds start-time-list))))))

(defmethod (setup-crew-member-duty-shifts-aux nasa-init-obj)
           (shift-number)
   (let ((shift-available-objs nil)
         (second-shift-start-time nil))
     (multiple-value-setq (shift-available-objs second-shift-start-time)
       (create-first-available-time-period self shift-number))
     (loop with done = nil
           until done
           for count from 1
           for shift-start-time from second-shift-start-time
               by seconds-per-day
           for shift-end-time = (+ seconds-per-shift shift-start-time)
           do
       (cond ((shift-time-falls-on-a-sunday-p *mission* shift-start-time)
              (setf second-shift-start-time shift-start-time  done t))
             (t (push (make-instance
                        'available-time
                        :begin (translate-universal-time-to-time-period
                                 shift-start-time)
                        :end (1- (translate-universal-time-to-time-period shift-end-time)))
                      shift-available-objs))))
     (loop for shift-start-time from (+ second-shift-start-time seconds-per-day)
                                     by seconds-per-day
                                     below (- universal-end-time seconds-per-shift)
           for counter from 0 by 1
           do
       (unless (zerop (mod counter 7))
         (push (make-instance
                 'available-time
                 :begin (translate-universal-time-to-time-period shift-start-time)
                 :end (1- (translate-universal-time-to-time-period
                            (+ shift-start-time seconds-per-shift))))
               shift-available-objs)))
     (nreverse shift-available-objs)))

(defmethod (shift-time-falls-on-a-sunday-p mission) (shift-start-time)
   (< (first-sunday-start-time init-obj)
     shift-start-time
     (+ (first-sunday-start-time init-obj) (seconds-per-day init-obj))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

#||
'(find-time-crew-available-after crew-available-in-time-periods-aux-2 crew-available-in-time-perio
ds-aux crew-available-in-time-periods-p crew-not-present-in-time-periods-p crew-not-present-in-tim
e-periods-aux find-earliest-time-crew-combination-available crew-combination-available-in-periods-
aux crew-combination-available-in-periods-p step-schedulable-crew-viewpoint-aux step-schedulable-c
rew-viewpoint-p )
||#

(defmethod (:print-self consumable-resource) (stream &rest ignore)
  (format stream "#<CONSUMABLE-RESOURCE ~A>" NAME))

(defmethod (:print-self non-depletable-resource) (stream &rest ignore)
  (format stream "#<NON-DEPLETABLE-RESOURCE ~A>" NAME))

(defmethod (:print-self crew-member) (stream &rest ignore)
  (format stream "#<CREW-MEMBER ~A>" NAME))

(defmethod (:print-self available-time) (stream &rest ignore)
  (format stream "#<AVAILABLE-TIME ~A ~A>" BEGIN END))

(defmethod (:print-self time-slice) (stream &rest ignore)
  (format stream "#<TIME-SLICE ~A ~A>" start-time END-time))

(defmethod (:print-self durable-resource) (stream &rest ignore)
  (format stream "#<DURABLE-RESOURCE ~A>" name))

(defmethod (:print-self experiment) (stream &rest ignore)
  (format stream "#<EXPERIMENT ~A>" name))

(defmethod (:print-self performance) (stream &rest ignore)
  (format stream "#<PERFORMANCE ~S EXP ~S>"number
          (if owning-experiment (name owning-experiment) nil)))

(defmethod (step-schedulable-crew-viewpoint-p step)
           (scheduled-period-list start-time &key (dont-use-current-crew nil))
  (let ((result :all-combinations-failed) (combination-result nil)
        (new-start-time nil) (new-time-list nil)(combination-list nil))
    (cond ((or (null crew-requirements) crew-monitor) (setf result :success))
          ((and crew-lockin (not (= crew-lockin number)))
           (multiple-value-setq (result new-start-time)
             (crew-combination-available-in-periods-p self scheduled-period-list
             (scheduled-crew-list (find-step-numbered owning-object crew-lockin))
             start-time))
           (if (eql result :success)
               (setf start-time new-start-time
                     scheduled-crew-list
                     (scheduled-crew-list (find-step-numbered owning-object crew-lockin)))
               (setf result :lock-crew-failure)))
          ((null crew-combinations)
           (error "crew-combinations have not been set for step ~S" self))
          (t
           (when dont-use-current-crew
             (push scheduled-crew-list failed-crew-combinations)(setf scheduled-crew-list nil))
           (loop for crew-combination in crew-combinations until (eql result :success)
                 do
             (multiple-value-setq (combination-result new-start-time)
        ;;;crew-combination-available-in-periods-p returns :success and start-time if
        ;;;sucessful, and returns nil and the time (if any) the combination is
        ;;;available
               (crew-combination-available-in-periods-p
                 self scheduled-period-list crew-combination start-time))
             (cond  ((eql combination-result :success)
                     (setf scheduled-crew-list crew-combination)
                     (setf result :success))
                    (t
                     (when new-start-time
                       (push crew-combination combination-list)
                       (push new-start-time new-time-list)))))
           (setf new-start-time nil)
           (cond ((eql result :success) nil)
                 ((null new-time-list)
```

```
                (setf start-time nil failed-crew-combinations nil))
               (t (loop for time in new-time-list
                        for crew-combo in combination-list
                        do
                    (unless (member crew-combo failed-crew-combinations :test #'equal)
                      (cond ((null new-start-time)
                             (setf new-start-time time scheduled-crew-list crew-combo))
                            ((< time new-start-time)
                             (setf new-start-time time scheduled-crew-list crew-combo))
                            (t nil))))
                  (setf start-time new-start-time)
                  (when (null new-start-time)
                    (setf result  :all-combinations-failed start-time (1+ start-time)))))))
    (values result start-time)))

(defmethod (find-step-numbered performance) (step-number)
  (let ((result nil))
  (loop for step in step-list
        until result
        do
    (when (= (number step) step-number)
      (setf result step)))
  result))



(defmethod (crew-combination-available-in-periods-p step)
           (period-list crew-combination start-time)
  (let ((result :success))
    (loop for crew in crew-combination
          until (not (eql result :success))
          do
      (multiple-value-setq (result start-time)
        (crew-available-in-time-periods-p crew start-time max-duration))
      (cond ((and (not (eql result :success)) (null start-time))
           ;;;this crew member never available for a sufficiently long time
             nil)
            ((not (eql result :success))
             nil)
            (t (multiple-value-setq (result start-time)
                 (crew-not-present-in-time-periods-p self period-list crew start-time))
               nil)));;; we passed both checks

    (values result start-time)))



(defmethod (crew-combination-available-in-periods-aux step) (crew-combination start-time)
  (let ((result :crew-conbination-not-available))
    (loop until (or (eql result :success)
                    (null start-time)
                    (> (1- (+ start-time max-duration)) (max-time (init-obj *mission*))))
          do
      (multiple-value-setq (result start-time)
        (crew-combination-available-in-periods-p
          self
          (get-time-instance-list
            *mission* start-time (1- (+ max-duration start-time))
            (if last-time-slice
              last-time-slice
              (if previous-step
                  (last-time-slice previous-step)
                  nil)))
          crew-combination start-time)))
    (if (eql result :success) start-time nil)))

(defmethod (find-first-time-crew-scheduable-after step) (time)
  (let ((times nil))
    (loop for combination in crew-combinations
          for new-time = (find-earliest-time-crew-combination-available
                           self combination (1+ time))
          do
```

```
      (when new-time
        (push new-time times)))
    (if times (apply #'min times) nil)))

(defmethod (find-earliest-time-crew-combination-available step)
           (crew-combination start-time)
  (let ((result nil))
    (loop until (or (eql result :success)
                    (null start-time)
                    (> (1- (+ start-time max-duration)) (max-time (init-obj *mission*))))
          do
    (multiple-value-setq (result start-time)
      (crew-combination-available-in-periods-p
        self (get-time-instance-list
               *mission* start-time (1- (+ max-duration start-time))
               (if last-time-slice
                   last-time-slice
                   (if previous-step
                       (last-time-slice previous-step)
                       nil)))
        crew-combination start-time)))
    (if (eql result :success) start-time nil)))


(defmethod (crew-not-present-in-time-periods-p step) (periods-list crew start-time)
  (let ((result :success))
    (loop for period in periods-list
          ;;;until (not (eql result :success))
          do
      (when (resource-present-in-period period :crew crew)
        (setf result :crew-already-scheduled)
        (setf start-time (1+ (end-time period)))))
    ;;;(crew-not-present-in-time-periods-aux self crew (1+ (end-time period))))))
    (values result start-time)))

(defmethod (crew-not-present-in-time-periods-aux step) (crew start-time)
  (let ((result nil))
    (loop until (or (eql result :success)
                    (> (+ start-time max-duration) (max-time (init-obj *mission*))))
          do
      (multiple-value-setq (result start-time)
        (crew-not-present-in-time-periods-p
          self
          (get-time-instance-list
            *mission* start-time (1- (+ max-duration start-time))
            (if last-time-slice
              last-time-slice
              (if previous-step
                  (last-time-slice previous-step)
                  nil)))
          crew start-time)))
    (if (eql result :success) start-time nil)))

(defmethod (crew-available-in-time-periods-p crew-member) (start-time duration)
  (cond ((null start-time) (values nil nil))
        (t
         (let ((end-time (1- (+ duration start-time))) (result nil))
           (multiple-value-setq (result start-time)
             (crew-available-in-time-periods-aux self start-time end-time))
           (unless (eql result :success)
             (setf start-time (crew-available-in-time-periods-aux-2 self start-time duration)))
           (values result start-time)))))

(defmethod (crew-available-in-time-periods-aux availability) (time step-end-time)
  (let ((available-obj (available-at-time self time)))
    (cond ((null available-obj)
           nil)
          ;;; indicates some time period for which the crew member was unavailable
          ((> time step-end-time)
           ;;; for this to be true, we must have found an available object for each
           ;;; time period
           (setf available-obj :success))
```

```
          ((<= step-end-time (end available-obj))
           ;;; the time period of interest is completely covered by this
           ;;; available-time obj
           (setf available-obj :success))
          (t  ;;; the crew-member is available in the current time period, but we
           ;;; have not covered all times yet
           (setf available-obj
             (crew-available-in-time-periods-aux
               self (1+ (end available-obj)) step-end-time))))
    (values available-obj time)))


(defmethod (crew-available-in-time-periods-aux-2 crew-member) (start-time duration)
  (cond ((null start-time) (values :crew-not-available nil))
        (t
         (let ((result nil))
           (loop until (or (eql result :success)
                           (null start-time)
                           (> (1- (+ start-time duration))
                              (max-time (init-obj *mission*))))
                 do
             (setf start-time (find-time-crew-available-after self start-time))
             (multiple-value-setq (result start-time)
               (crew-available-in-time-periods-p self start-time duration)))
           (if (eql result :success) start-time nil)))))


(defmethod (find-time-crew-available-after crew-member) (start-time)
  (let ((result nil))
    (loop for available-obj in available-times-list
          until result
          do
      (when (> (begin available-obj) start-time)
        (setf result (begin available-obj))))
    result))
```

```
xf;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (step-schedulable-durable-viewpoint-p step)
           (period-list delay-list start-time)
  (let ((result :success)(new-time start-time))
    (loop for (resource quant) in durable-resource-list
          until (not (eql result :success))
          do
      (multiple-value-setq (result new-time)
        (sufficient-durable-resource-in-periods-p
          self period-list resource quant start-time))
      (cond ((not (eql result :success))
             (setf result :durable-resource-not-available)
             (when new-time
               (setf new-time (step-schedulable-durable-viewpoint-aux self new-time))))
            ((and resource-carry-thru
                  (not (zerop step-delay-min)))
             (multiple-value-setq (result new-time)
               (sufficient-durable-resource-in-periods-p
                 self delay-list resource quant
                 (+ max-duration start-time)))
             (cond ((not (eql result :success))
                    (setf result :durable-resource-not-available)
                    (when new-time
                      (setf new-time (step-schedulable-durable-viewpoint-aux self new-time))))
                   (t nil)))))
    (values result new-time)))

(defmethod (step-schedulable-durable-viewpoint-aux step) (start-time)
  (let ((result :success)(new-time start-time))
    (cond ((> (+ start-time min-duration) (max-time (init-obj *mission*)))
           (setf result :max-time-exceeded new-time nil))
          (t
           (multiple-value-setq (result new-time)
             (step-schedulable-durable-viewpoint-p
               self
               (get-time-instance-list
                 *mission* new-time (1- (+ max-duration new-time))
                 (if last-time-slice
                     last-time-slice
                     (if previous-step
                         (last-time-slice previous-step)
                         nil)))
               (if (or (null resource-carry-thru) (zerop step-delay-min))
                   nil
                   (get-time-instance-list
                     *mission* (+ max-duration new-time)
                     (1- (+ step-delay-min max-duration new-time)))
                   (if last-time-slice
                       last-time-slice
                       (if previous-step
                           (last-time-slice previous-step)
                           nil)))
               new-time))))
    (if (eql result :success) start-time new-time)))


(defmethod (sufficient-durable-resource-in-periods-p step)
           (period-list resource quant start-time)
  (let ((result :success) (new-time start-time))
    (loop for period in period-list
          until (not (eql result :success))
          do
      (multiple-value-setq (result new-time)
        (sufficient-durable-resource-in-period
          self period resource quant start-time)))
    (values result (if (eql result :success)
                       (+ max-duration start-time)
                       new-time))))

(defmethod (sufficient-durable-resource-in-period step)
           (period resource quant step-start-time)
  ;;;the start time of the period may be less that the start time of the step for the
```

```
  ;;;first period, and the end time may be greater than the end time of the step for
  ;;;the last period
  (let* ((result :success) (return-time step-start-time)
         (max-quant (available-quantity resource))
         (step-list nil) (commited-quant nil))
    (multiple-value-setq (commited-quant step-list)
      (find-quant-durable-resource-already-committed
                              period resource))
    (unless
      (and max-quant  (≥ max-quant
                         (+ quant
                            commited-quant)))
      (setf result :insufficient-durable-resource return-time
            (find-time-durable-resource-no-longer-held-by-steps self step-list resource)))
    (values result return-time)))

(defmethod (find-quant-durable-resource-already-committed time-slice) (resource)
  (let ((result 0)(step-list nil))
          (loop for (com-resource com-quant step) in durable-resource-list
                do
            (when (eql resource com-resource)
              (incf result com-quant)
              (push step step-list)))
    (values result (min step-list))))

(defmethod (find-time-durable-resource-no-longer-held-by-steps step) (step-list resource)
  (let ((result 0))
    (loop for step in step-list
          for last-time =
              (find-time-durable-resource-no-longer-held-by-steps-aux step resource)
          do
      (when (> last-time result)
        (setf result last-time)))
    result))

(defmethod (find-time-durable-resource-no-longer-held-by-steps-aux step) (resource)
  (cond ((and next-step (member resource durable-resource-list :key #'first ))
         (find-time-durable-resource-no-longer-held-by-steps-aux next-step resource))
        ((member resource durable-resource-list :key #'first )
         (1+ scheduled-end-time))
        (t scheduled-start-time)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (step-schedulable-non-depletable-viewpoint-p step)
           (period-list delay-list start-time)
  (let ((result :success)(new-time start-time))
    (loop for (resource quant tolerance) in non-depletable-resource-list
          until (not (eql result :success))
          do
      (multiple-value-setq (result new-time)
        (sufficient-non-depletable-in-periods-p
          self period-list resource quant tolerance start-time))
      (cond ((not (eql result :success))
             (setf result :non-depletable-not-available)
             (when new-time
              (setf new-time (step-schedulable-non-depletable-viewpoint-aux self new-time))))
            ((and resource-carry-thru (not (zerop step-delay-min)))
             (multiple-value-setq (result new-time)
               (sufficient-non-depletable-in-periods-p
                 self delay-list resource quant tolerance (+ start-time max-duration)))
             (cond ((not (eql result :success))
                    (setf result :non-depletable-not-available)
                    (when new-time
                      (setf new-time (step-schedulable-non-depletable-viewpoint-aux
                                       self new-time))))))))
    (values result (if (eql result :success) start-time new-time))))

(defmethod (step-schedulable-non-depletable-viewpoint-aux step) (start-time)
  (let ((result :success) (new-time nil))
    (cond ((> (1- (+ start-time max-duration)) (max-time (init-obj *mission*)))
           nil)
          (t
           (multiple-value-setq (result new-time)
             (step-schedulable-non-depletable-viewpoint-p
               self (get-time-instance-list
                      *mission* start-time (1- (+ max-duration start-time))
                      (if last-time-slice
                          last-time-slice
                          (if previous-step
                              (last-time-slice previous-step)
                              nil)))
               (if (or (null resource-carry-thru) (zerop step-delay-min))
                   nil
                   (get-time-instance-list
                     *mission* (+ max-duration new-time)
                     (1- (+ step-delay-min max-duration new-time)))
                   (if last-time-slice
                       last-time-slice
                       (if previous-step
                           (last-time-slice previous-step)
                           nil)))
               start-time))
           (cond ((eql result :success) start-time)
                 (t new-time))))))

(defmethod (sufficient-non-depletable-in-periods-p step)
           (period-list resource quant tolerance start-time)
  (let ((result :success) (new-time start-time) (return-time start-time)
        (return-result :success))
    (loop for period in period-list
          do
      (multiple-value-setq (result new-time)
        (sufficient-non-depletable-in-period
          self period resource quant tolerance start-time))
      (unless (eql result :success)
        (setf return-result result)
        (setf return-time new-time)))
    (values return-result (if (eql return-result :success)
                              (+ max-duration start-time)
                              return-time))))

(defmethod (sufficient-non-depletable-in-period step)
           (period resource quant tolerance start-time)
```

```
  ;;;the start time of the period may be less that the start time of the step for the
  ;;;first period, and the end time may be greater than the end time of the step for
  ;;;the last period
  (let ((result :success) (return-time start-time)
        (already-committed nil) (max-pos-tol nil) (max-neg-tol nil)
        (available-time-obj
          (resource-available-in-period resource (max start-time (start-time period)))))
    (multiple-value-setq (already-committed max-pos-tol max-neg-tol)
      (find-quant-non-depletable-already-committed period resource))
      (cond ((null available-time-obj) ;;; there is no availability object --
             ;;; implies 0 availability
             (setf result :non-depletable-not-available
                   return-time (start-time
                                 (find-earliest-available-time-after
                                   resource (1+ (start-time period))))))
            ((and (check-quantities
                    self already-committed max-pos-tol max-neg-tol quant tolerance
                    (qty (owner-obj available-time-obj)))
                  ;;; we have enough
                  (≥ (end available-time-obj) (end-time period))) ;;;we've looked at
             ;;;all times
             (setf return-time (1+ (end-time period))))
            ((check-quantities
               self already-committed max-pos-tol max-neg-tol quant tolerance
               (qty (owner-obj available-time-obj))) ;;; we have enough but
             ;;; haven't looked at all times
             (multiple-value-setq (result return-time)
               (sufficient-non-depletable-in-period
                 self period resource quant tolerance (1+ (end available-time-obj)))))
            (t ;;; there is some available, but not enough
             (setf result :non-depletable-not-available return-time
                   (min (1+ (end-time period)) (1+ (end available-time-obj))))))
      (values result return-time)))

(defmethod (check-quantities step)
           (already-committed max-pos-tol max-neg-tol quant tolerance avail-quant)
  (cond ((zerop tolerance) ;;; if there is no tolerance, consider the max amount of
         ;;; negative tolerance (reserve resource) which must be maintained
         (≤ (+ quant already-committed) (+ avail-quant max-neg-tol)))
        ((minusp tolerance) ;;; if the tolerance is negative, consider the largest
         ;;; required reserve
         (if (< tolerance max-neg-tol)
             (≤ (+ quant already-committed) (+ avail-quant tolerance))
             (≤ (+ quant already-committed) (+ avail-quant max-neg-tol))))
        (t
         (cond ((zerop max-neg-tol) ;;; we still must maintain sufficient reserve
                (≤ (+ quant already-committed) (+ avail-quant max-neg-tol)))
               (t
                (if (> tolerance max-pos-tol)
                    (≤ (+ quant already-committed ) (+ avail-quant tolerance))
                    (≤ (+ quant already-committed) (+ avail-quant max-pos-tol)))))))))

(defmethod (find-quant-non-depletable-already-committed time-slice) (resource)
  (let ((committed 0) (max-pos-tol 0) (neg-tol 0) )
    (loop for (com-resource com-quant tol-quant dummy) in non-depletable-resource-list
          do
            dummy
      (when (eql resource com-resource)
        (incf committed com-quant)
        (cond ((null tol-quant) nil)
              ((zerop tol-quant) nil)
              ((and (minusp tol-quant) (< tol-quant neg-tol))
               (setf neg-tol tol-quant))
              ((and (plusp tol-quant) (> tol-quant max-pos-tol))
               (setf max-pos-tol tol-quant)))))
    (values committed max-pos-tol  neg-tol)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

#||
'(BACKTRACK SCHEDULE-OTHER-STEPS )
||#

(defmethod (SCHEDULE-OTHER-STEPS performance)
           (current-step start-time &key (dont-use-current-crew nil))
  (cond ((null current-step) (values :success start-time))
        (t
         (let
           ((last-step (previous-step current-step))(new-time start-time) (result nil))
           (if last-step
               (multiple-value-setq (result new-time)
                 (step-schedulable-starting-between-inclusive-times-p
                   current-step
                   (if (numberp new-time) new-time
                       (calc-next-step-earliest-start-time last-step))
                   (calc-next-step-latest-start-time last-step)
                   :dont-use-current-crew dont-use-current-crew))
               (multiple-value-setq (result new-time)
                 (step-schedulable-starting-at-time-p
                   current-step start-time nil
                   :dont-use-current-crew dont-use-current-crew)))
           (cond ((eql result :success)
                  ;;; i have a start time within the window
                  (if last-step
                      (setf (scheduled-start-time current-step) new-time
                            (scheduled-end-time current-step)
                            (1- (+ new-time (max-duration current-step))))
                      (setf (scheduled-start-time current-step) start-time
                            (scheduled-end-time current-step)
                            (1- (+ start-time (max-duration current-step)))))
                  (multiple-value-setq (result new-time)
                    ;;; all others have a start time
                    (SCHEDULE-OTHER-STEPS self (next-step current-step)
                                          (calc-next-step-earliest-start-time current-step))))
                 ((and (listp result) (eql (first result) :lock-crew-failure))
                  (if (= (second result) (number current-step))
                      (schedule-other-steps-aux self current-step start-time)
                      nil))
                 ((null (previous-step current-step))
                  ;;; i am trying to schedule the first step, and it has failed -
                  ;;; return the values of result and new-time, and quit
                  nil)
                 (new-time ;;; i have a start time outside of the window
                  (when (and (crew-lockin current-step)
                             (= (crew-lockin current-step) (number current-step)))
                    (setf (failed-crew-combinations current-step) nil))
                  (multiple-value-setq (result new-time)
                    (BACKTRACK self (previous-step current-step) new-time)))
                 (t ;;; this step can never be scheduled
                  nil))
           (values result new-time)))))

(defmethod (schedule-other-steps-aux performance) (current-step start-time)
  (multiple-value-bind (result new-time)
      (schedule-other-steps self current-step start-time :dont-use-current-crew t)
    (values result new-time)))

(defmethod (BACKTRACK performance) (current-step earliest-start-time-of-next-step)
  (let ((prev (previous-step current-step)) result)
    (cond ((≤ (calc-next-step-earliest-start-time current-step)
              earliest-start-time-of-next-step
              (calc-next-step-latest-start-time current-step))
           ;;;the proposed new start time of the next step is within the delay limits
           ;;;of this step as currently scheduled.
           (multiple-value-setq (result earliest-start-time-of-next-step)
             (schedule-other-steps self current-step
                                   earliest-start-time-of-next-step))
           (values result earliest-start-time-of-next-step))
          ((null prev)
           ;;;if you get here, you are working on the first step, and the time it is
```

```
;;;currently scheduled in is not ok
(setf (scheduled-start-time current-step) nil)
(values :total-failure
        (- earliest-start-time-of-next-step (max-duration current-step))))
(t ;;;the proposed new start time of the next step is not within the delay
;;;limits of this step.  The earliest and latest start times for
;;;the this step are computed which would allow next step to be
;;;scheduled at the desired time [earliest-start-time-of-next-step]
(let ((earliest (calc-this-step-earliest-start-time
                  current-step earliest-start-time-of-next-step))
      (latest (calc-this-step-latest-start-time
                current-step earliest-start-time-of-next-step))
      (start-time nil))
  (multiple-value-setq (result start-time)
    (step-schedulable-starting-between-inclusive-times-p current-step earliest
                                                         latest))
  (cond (start-time ;(eql result :success)
          ;;;a start time for the current step has been found within the delay
          ;;;limits of this step which allows the next step to be scheduled at
          ;;;the desired time - now, we must check whether the new start time
          ;;;for the current step is compatable with the start time of its
          ;;;parent.
         (multiple-value-setq (result earliest-start-time-of-next-step)
           (backtrack self prev start-time))
         (values result earliest-start-time-of-next-step))
        (t
        ;;;a start time cannot be found which will permit this step
        ;;;to be scheduled within the delay limits imposed by scheduling
        ;;;the next step at earliest-start-time-of-next-step. Calculate
        ;;;the closing time of that window, and search forward from that
        ;;;time.
         (setf (scheduled-start-time (next-step current-step)) nil)
         (values result start-time)))))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

#||
'(find-end-time-without-shutdown-steps find-start-time-without-startup-steps find-earliest-schedul
able-time-after startup-or-shutdown-steps-required-p between-experiment-constaints get-time-instan
ce-list get-time-instance get-linked-object update-other-object link-steps copy-step find-step-num
bered remove-steps generate-required-steps copy-step-list calc-this-step-latest-start-time calc-th
is-step-earliest-start-time calc-next-step-latest-start-time calc-next-step-earliest-start-time bu
ild-list-from-linked-structure get-first-shutdown-step get-last-startup-step join-shutdown-steps j
oin-startup-steps performance-schedulable-at-starting-time-p-aux-2 performance-schedulable-at-star
ting-time-p-aux find-first-time-no-overlap find-new-performance-window performance-schedulable-at-
starting-time-p )
||#

;*****************************************************************************
;;; high level performance and step scheduling feasibility methods

(defmethod (performance-schedulable-at-starting-time-p performance)
           (starting-time &optional scenario-number last-performance)
  ;;;the purpose of this method is to check whether there is an up-front, above
  ;;;step level reason that the performance cannot be scheduled at the time
  ;;;designated
  (let (new-time ok)
    ;;;check within experiment begin time constraints
    (multiple-value-setq (ok new-time)
            (ok-to-schedule-performance-starting-at-starting-time-p
              owning-experiment starting-time last-performance))

      ;;;ok will be t if ok,  some other value otherwise
      ;;; new-time will be time to end, or nil if scheduling after last already
      ;;; scheduled performance; otherwise, will indicate earliest time to try
      ;;;check between experiment constraints
      ;;; have to check for directional and mutual dependencies, and for exclusions
      ;;; dependencies can be concurant start, during, and sequential
      ;;;check if startup or shutdown steps required
    (cond ((and (eql ok t) (null (strategy owning-experiment)))
           (multiple-value-setq (ok new-time)
             (performance-schedulable-at-starting-time-p-aux
               self starting-time scenario-number last-performance)))
          ((and (eql ok t) (strategy owning-experiment))
             (let ((table (make-hash-table)))
               (loop for i from 0 to
                         (if (eql (first (strategy owning-experiment))
                                  :max-weight)
                             0
                             (1- (length (second (strategy owning-experiment)))))
                     do
                  (setf (gethash i table ) starting-time))
            (multiple-value-setq (ok new-time scenario-number)
              (multiple-strategy-performance-scheduable-at-starting-time-p
                self table (if scenario-number scenario-number 0) starting-time
                last-performance))
              (values ok new-time scenario-number)))
          ((eql ok :start-time-not-within-performance-window)
           (setf new-time (find-new-performance-window owning-experiment starting-time)))
          ((eql ok :maximum-performances-violation)
           (setf new-time nil))
          ((eql ok :overlap)
           (setf new-time (find-first-time-no-overlap owning-experiment starting-time)))
          ((eql ok :performances-per-window-violation)
           (setf new-time (find-new-performance-window owning-experiment starting-time))))
    (values ok new-time)))

(defmethod (multiple-strategy-performance-scheduable-at-starting-time-p performance)
           (table scen-number time last-performance)
  (let ((result :multiple-scenario-failuare) (new-time nil)(new-scenario nil))
    (multiple-value-setq (result new-time)
      (performance-schedulable-at-starting-time-p-aux self time scen-number last-performance))
    (cond ((eql result :success)
           (values result new-time scen-number))
          (t (setf (gethash scen-number table) new-time)
             (setf new-time nil)
             (loop for new-scenario-number from 0 below (send table :filled-elements)
```

```
                    for new-scenario-start-time = (gethash new-scenario-number table)
                    do
                  (cond ((null new-scenario-start-time)
                         nil)
                        ((null new-time)
                         (setf new-time new-scenario-start-time
                               new-scenario new-scenario-number))
                        ((< new-scenario-start-time new-time)
                         (setf new-time new-scenario-start-time
                               new-scenario new-scenario-number))
                        (t nil)))
                (when new-time
                (multiple-value-setq (result new-time new-scenario)
                  (multiple-strategy-performance-scheduable-at-starting-time-p
                    self table new-scenario new-time last-performance))))
            (values result new-time new-scenario))))

(defmethod (find-new-performance-window experiment) (start-time)
  (loop for (start end performances) in performance-windows
        do
    (when (> start start-time)
      (return start))))


(defmethod (find-first-time-no-overlap experiment) (start-time)
  (let ((scheduled-times-list nil) (new-time nil))
    (loop for performance in performance-list
          do
      (when (scheduled-p performance)
        (push (list (if (execute-start-up-steps-p performance)
                        (find-start-time-without-startup-steps performance)
                        (scheduled-start-time performance))
                    (if (execute-shutdown-steps-p performance)
                        (+ (find-end-time-without-shutdown-steps performance)
                           min-performance-delay-time)
                        (+ (scheduled-end-time performance) min-performance-delay-time)))
              scheduled-times-list)))
    (loop for (start end) in
              (setf scheduled-times-list (sort scheduled-times-list #'< :key #'first))
          with done = nil until done
          do
      (cond ((< end start-time )
             ;;; this pair ends earlier than the time we are interested in
             nil)
            ((and new-time (< new-time start))
             ;;; we previously found a new time, and it is less than the start of the
             ;;; next performance -- we are done
             (setf done t))
             (new-time
               ;;; we previously found a new time, but it fails to be strictly less
               ;;; than the start time of the next already scheduled performance
               (setf new-time nil))
             ((> end start-time)
              ;;; this is the first end greater than the start time when new-time is
              ;;; still nil
              (setf new-time (1+ end)))))
    new-time))


(defmethod (performance-schedulable-at-starting-time-p-aux performance)
           (starting-time &optional (scenario-number nil) last-performance)
  ;;;the purpose of this method is to determine the scenario we are working on, setup
  ;;;the steps, and call aux-2 to do the real work
  (let (ok new-time shutdown-steps-p start-up-steps-p)
    (cond (scenario-number
           ;;;there is a strategy, and we are to examine a particular scenario
           (generate-required-steps owning-experiment self scenario-number)
           (compute-and-store-cumulative-consumption self)
           (multiple-value-setq (ok new-time)
             (performance-schedulable-at-starting-time-p-aux-2
               self starting-time  scenario-number last-performance))
           (values ok new-time))
```

```
          (t  ;;;default case of startup, core and shutdown steps
           (generate-required-steps owning-experiment self scenario-number)
           (multiple-value-setq (start-up-steps-p shutdown-steps-p)
             (startup-or-shutdown-steps-required-p owning-experiment starting-time))
           (when start-up-steps-p
             (join-startup-steps self (first step-list))
             (setf execute-start-up-steps-p t))
           (when shutdown-steps-p (join-shutdown-steps self)
                 (setf execute-shutdown-steps-p t))
           (compute-and-store-cumulative-consumption self)
           (multiple-value-setq (ok new-time)
             (performance-schedulable-at-starting-time-p-aux-2
               self starting-time nil last-performance))
           (values ok new-time)))))



(defmethod (performance-schedulable-at-starting-time-p-aux-2 performance)
           (starting-time &optional scenario-number last-performance)
  (let (result new-time (first-step (first step-list)))
    (multiple-value-setq (result new-time)
      (schedule-other-steps self first-step starting-time))
    (cond
      ((eql result :success)
       (setf scheduled-start-time (scheduled-start-time first-step)
             scheduled-end-time (scheduled-end-time (first (last step-list))))
       (multiple-value-setq (result new-time)
         (check-for-completion-within-performance-duration self result new-time))
       (when (eql result :success)
         (multiple-value-setq (result new-time)
           (check-for-min-delay-between-performance-violation self result new-time))
         (when (eql result :success)
           (multiple-value-setq (result new-time)
             (check-for-completion-within-performance-window self result new-time))))
       (setf new-time scheduled-start-time))
      ((and (not (eql result :success))
            new-time scenario-number
            (≤ new-time (max-time (init-obj *mission*))))
       (multiple-value-setq (result new-time)
         (performance-schedulable-at-starting-time-p-aux-2
           self new-time scenario-number))
       ))
    (values result new-time)))

;*****************************************************************************
;;;low level functions


(defmethod (join-startup-steps performance) (first-step)
  (let* ((startup-step-list (copy-step-list ( startup-steps owning-experiment )))
         (last-startup-step (first (last startup-step-list))))
    (setf (next-step last-startup-step) first-step
          (previous-step first-step) last-startup-step)
    (setf step-list (concatenate 'list startup-step-list step-list))))

(defmethod (join-shutdown-steps performance) ()
  (let* ((last-step (first (last step-list)))
         (shutdown-step-list (copy-step-list (shutdown-steps owning-experiment))))
    (setf (next-step last-step) (first shutdown-step-list)
          (previous-step (first shutdown-step-list)) last-step)
    (setf step-list (concatenate 'list step-list shutdown-step-list))))

(defmethod (get-last-startup-step experiment) ()
  (first (last startup-steps)))

(defmethod (get-first-shutdown-step experiment) ()
  (first shutdown-steps))

(defun build-list-from-linked-structure (top-of-structure accessor)
  (if (null top-of-structure)
      nil
```

```
      (cons top-of-structure (build-list-from-linked-structure
                              (funcall accessor top-of-structure) accessor))))

(defmethod (calc-next-step-earliest-start-time step) ()
  (+ scheduled-start-time
 step-delay-min max-duration))

(defmethod (calc-next-step-latest-start-time step) ()
  (+ scheduled-start-time step-delay-max max-duration))

(defmethod (calc-this-step-earliest-start-time step) (start-time)
  (- start-time (+ step-delay-max max-duration)))

(defmethod (calc-this-step-latest-start-time step) (start-time)
  (- start-time (+ step-delay-min max-duration)))

(defmethod (copy-step-list performance) (new-step-list)
  (loop for the-step in new-step-list
        for this-step = (copy-step the-step self)
        with prev-step = nil
        collect this-step
        do
    (setf (owning-object this-step) self)
    (when prev-step
      (link-steps prev-step this-step))
    (setf prev-step this-step)))

(defmethod (generate-required-steps experiment) (perf scenario-number)
  (remove-steps perf)
  (cond ((null scenario-number)
         ;;; default case of startup, prototype and shutdown steps
         (first (setf (step-list perf) (copy-step-list perf prototype-step-list))))
        ((and (eql (first strategy) :max-weight) (null (zerop scenario-number)))
         (error "~%generate-required-steps called with max-weight strategy,
                  and scenario-number not equal to zero for performance ~S of experiment ~S"
                perf self))
        (t  (loop for substrategy in (first (nth scenario-number (second strategy)))
                  with steps = nil
                  do
              (if (eql (first substrategy) :consecutive)
                  (setf steps
                        (concatenate
                          'list steps
                          (loop for i from (second substrategy) to (third substrategy)
                                collect (find-step-numbered self i))))
                  (setf steps
                        (concatenate
                          'list steps
                          (loop for i in (second substrategy)
                                collect (find-step-numbered self i)))))
                  finally (setf (step-list perf) steps))
            (first (setf (step-list perf)
                         (copy-step-list perf (step-list perf)))))))

(defmethod (remove-steps performance) ()
  (setf step-list nil))

(defmethod (find-step-numbered experiment) (desired-number)
  (let ((result nil))
    (cond ((and shutdown-steps (≥ desired-number (number (first shutdown-steps))))
           (loop for step in shutdown-steps
                 until (= desired-number (number step))
                 finally (setf result step)))
          ((and prototype-step-list (≥ desired-number (number (first prototype-step-list))))
           (loop for step in prototype-step-list
                 until (= desired-number (number step))
                 finally (setf result step)))
          (startup-steps
           (loop for step in startup-steps
                 until (= desired-number (number step))
                 finally (setf result step)))
          (t nil))
    result))
```

```
(defmethod (find-step-named performance) (desired-name)
  (loop for step in step-list
        until (= desired-name (name step))
        finally (return step)))

(defmethod (copy-step step) (&optional (owner nil) )
  (make-instance 'step
                 :id id
                 :number number
                 :max-duration max-duration
                 :min-duration min-duration
                 :step-delay-min step-delay-min
                 :step-delay-max step-delay-max
                 :cumulative-consumable-list nil
                 :consumable-resource-list consumable-resource-list
                 :durable-resource-list durable-resource-list
                 :non-depletable-resource-list non-depletable-resource-list
                 :crew-requirements crew-requirements
                 :crew-combinations crew-combinations
                 :crew-lockin crew-lockin
                 :crew-monitor crew-monitor
                 :crew-cycle crew-cycle
                 :crew-duration crew-duration
                 :crew-late-shift crew-late-shift
                 :crew-early-shift crew-early-shift
                 :concurrent-with concurrent-with
                 :target-list target-list
                 :attitude-list attitude-list
                 :scheduled-crew-list nil
                 :crew-monitoring-time crew-monitoring-time
                 :owning-object (if owner owner owning-object)))

(defun link-steps (prev-step n-step)
  (setf (next-step prev-step) n-step (previous-step n-step) prev-step))
;*****************************************************************************
;;;stubs

(defun update-other-object (arg) arg
  ;;;this stub is to be used to do actual scheduling of an object which is to be
  ;;;concurrently scheduled with the object currently being scheduled
  ;(format t "this is a stub [ defun update-other-object ] with 1 arg -A " arg)
  nil)

(defmethod (get-linked-object mission) (arg) arg
  ;;; this stub is to be used to retrieve the actual object to be scheduled
  ;;; concurrently with the currently being scheduled object.  the return is passed
  ;;; to update-other-object
  ;(format t "this is a stub: [get-linked-object mission] with 1 arg = ~A" arg)
  nil)

(defmethod (get-time-instance mission) (time-period &optional time-slice)
  (cond ((null time-slice)
         (get-time-instance time-slice-holder time-period))
        ((≤ (start-time time-slice) time-period (end-time time-slice))
         time-slice)
        (t (get-time-instance time-slice time-period))))

(defmethod (get-time-instance time-slice) (time-period)
  (cond ((≤ start-time time-period end-time)
         self)
        ((and (< end-time time-period) next-slice)
         (get-time-instance next-slice time-period))
        ((and (> start-time time-period ) prev-slice)
         (get-time-instance prev-slice time-period))))

(defmethod (get-time-instance-list mission) (start-time end-time &optional starting-instance)
  (when (< end-time start-time)
    (error "~%get-time-instance-list called with start-time ~S greater than end-time ~S"
           start-time end-time))
    (loop with done = nil until done
          with result = nil
          with next-instance = nil
          do
```

```
      (setf next-instance (get-time-instance
                           self start-time
                           (if next-instance next-instance starting-instance)))
      (cond ((> end-time (end-time next-instance))
             (push next-instance result)
             (setf start-time (1+ (end-time next-instance))))
            ((≤ (start-time next-instance) end-time (end-time next-instance))
             ;;; this is the last instance
             (push next-instance result)
             (setf done t)))
      finally (if result (return (reverse result)) result)))


(defmethod (between-experiment-constaints step) ()
  ;(format t "this is a stub [ between-experiment-constaints step] with no args")
  nil)


(defmethod (startup-or-shutdown-steps-required-p experiment) (time)
  (let ((startup-p t) (shutdown-p schedule-shutdown-with-performance))
    ;;;startup-p and shutdown-p initialized to t and
    ;;;schedule-shutdown-with-performance so that the proper values will be returned
    ;;;in the case where the first performance is being scheduled
    (unless startup-steps (setf startup-p nil))
    (unless shutdown-steps (setf shutdown-p nil))
  (when performance-list
      (loop for performance in performance-list
            with startup-flag = startup-p
            with shutdown-flag = shutdown-p
            ;;; if this flag is set, we should be scheduling a sequence of
            ;;; performances, each after the other, meaning that each will have to
            ;;; have shutdown steps scheduled and then un-scheduled unless we
            ;;; intervne
            until (and (null startup-flag) (null shutdown-flag))
            do
        (when (scheduled-p performance)
          (cond ((and startup-flag
                      (< (find-start-time-without-startup-steps performance) time))
                 ;;;this performance starts earlier than the new time, hence, startup
                 ;;;steps not needed
                 (setf startup-flag nil))
                ((< time (find-start-time-without-startup-steps performance))
                 ;;; this performance starts later than the new time, hence,
                 ;;; shutdown steps are not needed
                 (setf shutdown-flag nil))
                ((and shutdown-flag
                      (= (find-start-time-without-startup-steps performance) time))
                 ;;; this performance starts at the same time - save work by
                 ;;; returning immediately will nil nil, knowning another check will
                 ;;; reject this time
                 (setf startup-flag nil shutdown-flag nil))))
        finally (progn (setf startup-p startup-flag)
                       (setf shutdown-p shutdown-flag))))
  (values startup-p shutdown-p)))



(defmethod (find-start-time-without-startup-steps performance) ()
  (if execute-start-up-steps-p
      (loop for step in step-list
            with first-core-step = (first (prototype-step-list owning-experiment))
            do
        (when (and (eql (name first-core-step) (name step))
                   (= (id first-core-step) (id step)))
          (return (scheduled-start-time step))))
      (scheduled-start-time (first step-list))))

(defmethod (find-end-time-without-shutdown-steps performance) ()
  (if execute-shutdown-steps-p
      (loop for step in step-list
            with last-core-step = (first (last (prototype-step-list owning-experiment)))
```

```
      do
    (when (and (eql (name last-core-step) (name step))
               (= (id last-core-step) (id step)))
      (return (scheduled-end-time step))))
  (scheduled-end-time (first (last step-list)))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

#||
'(resource-available-in-period resource-available-in-periods get-object-named find-maximum-resourc
e-available find-quant-resource-already-committed sufficient-resource-in-period-aux sufficient-res
ource-in-period sufficient-resource-in-periods-p step-schedulable-durable-viewpoint-p find-earlies
t-step-schedulable-after-time step-schedulable-durable-viewpoint-aux step-schedulable-non-depletab
le-viewpoint-aux step-schedulable-non-depletable-viewpoint-p step-schedulable-consumable-viewpoint
-aux step-schedulable-consumable-viewpoint-p )
||#

(defmethod (step-schedulable-consumable-viewpoint-p step)
           (period-list start-time)
  (let ((result :success)(new-time start-time))
    (loop for (resource quant) in cumulative-consumable-list
          until (not (eql result :success))
          do
      (multiple-value-setq (result new-time)
        (sufficient-consumable-in-periods-p self period-list
                                            resource quant start-time))
      (when (eql result :success)
        (multiple-value-setq (result new-time)
          (sufficient-consumables-at-quant-availability-change-points
            self resource quant start-time)))
      (unless (eql result :success)
        (when new-time
          (setf new-time (step-schedulable-consumable-viewpoint-aux self new-time)))))
    (values result (if (eql result :success) start-time new-time))))

(defmethod (step-schedulable-consumable-viewpoint-aux step) (start-time)
  (let ((result :success))
    (multiple-value-setq (result start-time)
      (step-schedulable-consumable-viewpoint-p
        self (get-time-instance-list
               *mission* start-time (1- (+ max-duration start-time))
               (if last-time-slice
                   last-time-slice
                   (if previous-step
                       (last-time-slice previous-step)
                       nil)))
        start-time))
    (if (eql result :success) start-time nil)))

(defmethod (sufficient-consumables-at-quant-availability-change-points step)
           (resource quant start-time)
  (let ((result :success) (new-time start-time))
    (loop for period in (find-resource-availability-change-points resource start-time)
          while (eql result :success)
          do
      (multiple-value-setq (result new-time)
        (sufficient-consumable-in-period self period resource quant (end-time period))))
    (values result new-time)))

(defmethod (find-resource-availability-change-points consumable-resource) (time)
  (let ((result nil))
    (loop for quant-avail in quantity-availability-list
          with last-slice = nil
          do
      (loop for avail-obj in (available-times-list quant-avail)
            do
        (when (≤ time (end avail-obj))
          (setf last-slice (get-time-instance *mission* time last-slice))
          (push last-slice result))))
    (when result
      (setf result (sort result #'< :key #'end-time)))
    result))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; these methods check the availability of a resource with respect to a time-slice -
;;; namely the presence or absence of some resource in a time period, or the quantity
;;; in which the resource has already been committed.
(defmethod (sufficient-consumable-in-periods-p step)
           (period-list resource quant start-time)
```

```
  (let ((result :success) (new-time start-time))
    (loop for period in period-list
          do
      (multiple-value-setq (result new-time)
        (sufficient-consumable-in-period
          self period resource quant start-time)))
    (values result (if (eql result :success)
                       (+ max-duration start-time)
                       new-time))))

(defmethod (sufficient-consumable-in-period step)
           (period resource quant start-time)
  ;;;the start time of the period may be less that the start time of the step for the
  ;;;first period, and the end time may be greater than the end time of the step for
  ;;;the last period
    (let ((result :success) (return-time start-time)
          (already-committed
            (find-quant-consumable-already-committed period resource))
          (available-time-obj
            (resource-available-in-period resource (max start-time (start-time period)))))
      (cond ((null available-time-obj) ;;; there is no availability object --
             ;;; implies 0 availability
             (setf result :consumable-not-available
                   return-time (start-time
                                (find-earliest-available-time-after
                                  resource (1+ (start-time period))))))
            ((and (≥ (qty (owner-obj available-time-obj))
                     (+ quant already-committed)) ;;; we have enough
                  (≥ (end available-time-obj) (end-time period))) ;;;we've looked at
             ;;;all times
             (setf return-time (1+ (end-time period))))
            ((≥ (qty (owner-obj available-time-obj))
                (+ quant already-committed)) ;;; we have enough but
             ;;; haven't looked at all times
             (multiple-value-setq (result return-time)
               (sufficient-consumable-in-period
                 self period resource quant (1+ (end available-time-obj)))))
            (t ;;; there is some available, but not enough
             (setf result :consumable-not-available return-time
                   (min (1+ (end-time period)) (1+ (end available-time-obj))))))
      (values result return-time)))

(defmethod (find-quant-consumable-already-committed time-slice) (resource)
  (let ((result 0))
    (setf result (gethash resource cumulative-consumable-table))
    (unless result (setf result 0))
    result))


(defmethod (get-object-named nasa-init-obj) (resource-type resource)
  (unless (member resource-type '(:durable :consumable :non-depletable
                                  :crew :target :attitude))
    (error "get-object-named invoked on resource-type ~S" resource-type))
  (loop for obj in (case resource-type
                     (:durable durable-resource-list )
                     (:consumable consumable-resource-list)
                     (:non-depletable non-depletable-resource-list)
                     (:crew crew-list)
                     (:target target-list)
                     (:attitude attitude-list))
        do (when (eql (name obj) resource)
             (return obj))))

;;;*****************************************************************************
;;; methods for determining whether a resource is available from the resource
;;; availability data -- whether these are really need will be determined when i
;;; finally decide what information will be recorded in each time period.


(defmethod (resource-available-in-periods non-durable-resource) (period-list)
  (let ((result t))
    (loop for period in period-list
          do
```

```
      (unless (resource-available-in-period self (start-time period))
        (setf result nil)))
    result))

(defmethod (resource-available-in-period non-durable-resource) (time-period)
  ;;; returns an instance of available-time if sucessful
  (let ((result nil))
    (loop for quantity-availability-object in quantity-availability-list
          until result
          do
      (setf result (available-at-time quantity-availability-object time-period)))
    result))

(defmethod (find-earliest-available-time-after non-durable-resource) (time)
  (let ((after-list nil))
    (loop for quantity-availability-object in quantity-availability-list
          do
      (loop for available-time-obj in (available-times-list quantity-availability-object)
            do
        (when (> (begin available-time-obj) time)
          (push available-time-obj after-list))))
    (first (setf after-list (sort after-list #'< :key #'begin)))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

#||
'(step-schedulable-starting-between-inclusive-times-p available-at-time resource-present-in-period
 resource-present-in-periods-p-aux resource-present-in-periods-p resource-not-present-in-periods-p
 step-schedulable-attitude-viewpoint-aux step-schedulable-attitude-viewpoint-p step-schedulable-ta
rget-viewpoint-aux step-schedulable-target-viewpoint-p analyze-times-for-type-failure step-schedul
able-starting-at-time-aux step-schedulable-starting-at-time-p )
||#
;;;things that still need to be done


;;;time-slice storage should be changed from a list to linked objects, or the insert
;;;new mechanism must be redone


;*****************************************************************************
;;; this section deals with determining whether a step can be scheduled to begin at a
;;; specific time
  ;;;the proximity of this step to other steps in the same performance has already
  ;;;been checked by schedule-other-steps
  ;;;for now,
  ;;;ignore between-step and between experiment constraints
  ;;;ignore crew lockin
  ;;;ignore crew monitoring
  ;;;do check
  ;;;durable resource constraints
  ;;;non-depletable resource constraints
  ;;;consumable resource constraints
  ;;;target constraints
  ;;;attitude constraints
  ;;;crew availablity constraints (simplified)

(defmethod (step-schedulable-starting-at-time-p step)
           (start &optional last-slice &key (dont-use-current-crew nil))
  ;;;when successful, returns the ending plus one on the step ; otherwise, returns the
  ;;;first time after the starting time that the step can be scheduled at
  (let ((result nil))
    (cond
      ((> (+ start (1- min-duration)) (max-time (init-obj *mission*)))
       (setf result  :exceeds-mission-duration start nil))
      (t
       (let* ((sch-pers
                (get-time-instance-list
                  *mission* start (1- (+ max-duration start))
                  (if last-slice last-slice
                      (if previous-step (last-time-slice previous-step) nil))))
              (delay-pers
                (if (or (null resource-carry-thru )(zerop step-delay-min))
                    nil
                    (get-time-instance-list
                      *mission* (+ max-duration start)
                      (1- (+ max-duration start step-delay-min)))
                    (if sch-pers (first (last sch-pers)) nil)))
              consum-p non-dep-p dur-p tgt-p att-p crew-p tgt-time consum-time non-dep-time
              dur-time att-time crew-time (poss-lst nil))
         (multiple-value-setq (consum-p consum-time)
           (step-schedulable-consumable-viewpoint-p self sch-pers start))
         (multiple-value-setq (non-dep-p non-dep-time)
           (step-schedulable-non-depletable-viewpoint-p self sch-pers delay-pers start))
         (multiple-value-setq (dur-p dur-time)
           (step-schedulable-durable-viewpoint-p self sch-pers delay-pers start))
         (multiple-value-setq (tgt-p tgt-time)
           (step-schedulable-target-viewpoint-p self sch-pers start))
         (multiple-value-setq (att-p att-time)
           (step-schedulable-attitude-viewpoint-p self sch-pers start))
         (multiple-value-setq (crew-p crew-time)
           (step-schedulable-crew-viewpoint-p
             self sch-pers start :dont-use-current-crew dont-use-current-crew))
         (cond ((and (eql :success consum-p) (eql :success non-dep-p) (eql :success dur-p)
                     (eql :success tgt-p)(eql :success att-p) (eql :success crew-p))
                (setf scheduled-start-time start result :success
                      scheduled-end-time (1- (+ start max-duration)))
```

```
                (setf start (+ max-duration start step-delay-min)
                      last-slice
                      (if delay-pers (first (last delay-pers)) (first (last sch-pers)))))
               ((and (eql :success consum-p) (eql :success non-dep-p) (eql :success dur-p)
                     (eql :success tgt-p) (eql :success att-p) (eql :lock-crew-failure crew-p))
                (setf result (list :lock-crew-failure crew-lockin)))
               ((and (eql :success consum-p) (eql :success non-dep-p) (eql :success dur-p)
                     (eql :success tgt-p)(eql :success att-p)
                     (eql  :all-combinations-failed crew-p))
                (setf result :all-combinations-failed start
                      (find-first-time-crew-schedulable-after self start)))
               ((and consum-time non-dep-time dur-time tgt-time att-time crew-time)
                (unless (eql :success consum-p) (push consum-time poss-lst))
                (unless (eql :success non-dep-p)(push non-dep-time poss-lst))
                (unless (eql :success dur-p)(push dur-time poss-lst))
                (unless (eql :success tgt-p)(push tgt-time poss-lst))
                (unless (eql :success att-p)(push att-time poss-lst))
                (unless (eql :success crew-p)(push crew-time poss-lst))
                (multiple-value-setq (result start)
                  (step-schedulable-starting-at-time-aux self (apply #'max poss-lst))))
               (t (setf start nil result
                        (analyze-times-for-type-failure
                          self consum-time non-dep-time dur-time tgt-time att-time
                          crew-time)))))))
    (values result start)))

(defmethod (step-schedulable-starting-at-time-aux step) (start-time)
  (let ((result nil))
    (loop until (or (eql result :success)
                    (null start-time)
                    (> (1- (+ start-time max-duration)) (max-time (init-obj *mission*))))
          do
      (multiple-value-setq (result start-time)
        (step-schedulable-starting-at-time-p self start-time)))
    (values nil (if (eql result :success) scheduled-start-time nil))))

(defmethod (analyze-times-for-type-failure step)
           (consumable-time non-depletable-time durable-time target-time attitude-time
                            crew-time)
  (let ((result nil))
    (cond-every ((null consumable-time) (push :consumable-not-available result))
                ((null non-depletable-time) (push :non-depletable-not-available result))
                ((null durable-time) (push :durable-not-available result))
                ((null target-time) (push :target-not-available result))
                ((null attitude-time) (push :attitude-not-available result))
                ((null crew-time) (push :crew-not-available result)))
    result))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod (step-schedulable-attitude-viewpoint-p step) (period-list start-time)
  (let ((result :success))
  (loop for attitude in attitude-list
        until (not (eql result :success))
        do
    (multiple-value-setq (result start-time)
      (resource-present-in-periods-p self period-list :attitude attitude start-time))
    (unless (eql result :success)
      (when start-time
        (setf start-time
              (step-schedulable-attitude-viewpoint-aux
                self :attitude attitude start-time)))))
  (values result start-time)))

(defmethod (step-schedulable-attitude-viewpoint-aux step) (resource-type resource start-time)
  (let ((result nil))
    (loop until (or (eql result :success)
                    (> (1- (+ start-time max-duration)) (max-time (init-obj *mission*))))
          do
      (multiple-value-setq (result start-time)
        (step-schedulable-attitude-viewpoint-p
          self
          (get-time-instance-list
```

```
              :dont-use-current-crew dont-use-current-crew))
        (cond ((and (eql result :success)
                    (≤ first-start-time scheduled-start-time last-start-time))
               ;;; the step can be scheduled at the start time
               (setf new-time scheduled-start-time))
              ((eql result :success)
               ;;; this shouldn't happen
               (error "~% step-schedulable-starting-between-inclusive-times-p got a value of :s
uccess back, but the time was not within limits"))
              ((and (listp result) (eql (first result) :lock-crew-failure )) nil)
              ((null new-time)
               ;;; we can't find a time to schedule the step
               nil)
              ((≤ first-start-time new-time last-start-time)
               ;;; we can't schedule at the start time, but some other acceptable time
               ;;; was found
               (setf result :success)
               (setf new-time scheduled-start-time))
              (t ;;; we found a time, but it is not acceptable -- return nil result and
               ;;; new-time
               nil))))
  (values result new-time)))
```

```
;;; -*- Package: USER; Base: 10; Mode: LISP; Syntax: Common-lisp; -*-

(defmethod (step-schedulable-target-viewpoint-p step) (period-list start-time)
  (let ((result :success) (new-time start-time))
    (cond ((null target-list) nil)
          (t (multiple-value-setq (result new-time)
               (step-schedulable-target-intersect-p self period-list start-time))
             (unless (eql result :success)
               (multiple-value-setq (result new-time)
                 (step-schedulable-target-avoid-p self period-list start-time))
               (unless (eql result :success)
                 (multiple-value-setq (result new-time)
                   (step-schedulable-target-select-p self period-list start-time)))
               (when new-time
                 (setf new-time
                       (step-schedulable-target-viewpoint-aux self period-list new-time))))))
    (values result new-time)))

(defmethod (step-schedulable-target-viewpoint-aux step) (resource-type resource start-time)
  (let ((result nil))
    (loop until (or (eql result :success)
                    (> (1- (+ start-time max-duration)) (max-time (init-obj *mission*))))
          do
      (multiple-value-setq (result start-time)
        (step-schedulable-target-viewpoint-p
          self
          (get-time-instance-list
            *mission* start-time (1- (+ max-duration start-time))
            (if last-time-slice
              last-time-slice
              (if previous-step
                  (last-time-slice previous-step)
                  nil)))
          resource-type resource start-time
          )))
    (if (eql result :success) start-time nil)))

(defmethod (step-schedulable-target-intersect-p step) (period-list start-time)
  (let ((result :success) (new-time start-time))
  (loop for (designator target-sublist) in target-list
        until (not (eql result :success))
        do
    (cond ((eql designator :intersect)
           (loop for target in target-sublist
                 until (not (eql result :success))
                 do
             (multiple-value-setq (result new-time)
               (resource-present-in-periods-p self period-list :target target start-time))
             (unless (eql result :success)
               (setf result :intercept-target-failure))))
          (t nil)))
  (values result new-time)))

(defmethod (step-schedulable-target-avoid-p step) (period-list start-time)
  (let ((result :success) (new-time start-time))
  (loop for (designator target-sublist) in target-list
        until (not (eql result :success))
        do
    (cond ((eql designator :avoid)
           (loop for target in target-sublist
                 until (not (eql result :success))
                 do
             (multiple-value-setq (result new-time)
               (target-not-present-in-periods-p self period-list target start-time))
             (unless (eql result :success)
               (setf result :intercept-target-failure))))
          (t nil)))
  (values result new-time)))

(defmethod (target-not-present-in-periods-p step) (period-list target start-time)
  (let ((result :success))
    (loop for period in period-list
```

```
        do
      (cond ((resource-present-in-period period period :target target)
             (setf start-time (1+ (end-time period)) result :aviod-target-failure))
            (t nil)))
    (values result start-time)))

(defmethod (step-schedulable-target-select-p step) (period-list start-time)
  (let ((result :init-value) (new-time start-time))
    (loop for (designator target-sublist) in target-list
          until (member result '(:success :select-target-failure))
          do
      (cond ((eql designator :select)
             (setf result :select-target-failure)
             (loop for target in target-sublist
                   until (eql result :success)
                   do
               (multiple-value-setq (result new-time)
                 (resource-present-in-periods-p
                   self period-list :target target start-time))))
            (t nil)))
    (unless (eql result :success)
      (setf result :select-target-failure start-time new-time))
    (values result start-time)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

#||
'(check-for-completion-within-performance-window check-for-min-delay-between-performance-violation
 check-for-completion-within-performance-duration start-time-not-within-performance-window start-t
ime-violates-performances-per-window-restriction max-performances-violation-p start-time-is-within
-the-scheduled-time-of-some-other-performance-p ok-to-schedule-performance-starting-at-starting-ti
me-p )
||#
;*****************************************************************************
;;; pre step scheduling constraint checkers
(defmethod (ok-to-schedule-performance-starting-at-starting-time-p experiment) (start-time &rest i
gnore)
  (cond ((max-performances-violation-p self)
         (values :maximum-performances-violation nil))
        ((start-time-is-within-the-scheduled-time-of-some-other-performance-p
           self start-time)
         (values :overlap nil))
        ((start-time-not-within-performance-window self start-time)
         (values :start-time-not-within-performance-window nil))
        ((start-time-violates-performances-per-window-restriction self start-time)
         (values :performances-per-window-violation nil))
        (t (values t nil))))

(defmethod (start-time-is-within-the-scheduled-time-of-some-other-performance-p experiment)
           (starting-time)
  (when performance-list
    (loop for performance in performance-list
          for adjusted-end-time = (find-start-time-without-startup-steps performance)
          for adjusted-start-time = (find-end-time-without-shutdown-steps performance)
          do
      (cond ((null (scheduled-p performance))
             ;;; if the performance has not been scheduled, don't worry about it
             nil)
            ((< starting-time (- adjusted-start-time min-performance-delay-time))
             ;;;clearly, not a violation
             nil)
            ((≤ starting-time adjusted-start-time)
             ;;; the starting-time is before the core of the other steps, but not at
             ;;; least the minimum delay time before
               (return t))
            ((< (+ adjusted-end-time min-performance-del. -time) starting-time)
             ;;; clearly, not a violation
             nil)
            ((< adjusted-end-time starting-time)
             ;;; the starting-time is after the core of the other steps, but not at
             ;;; least the minimum delay time after
             (return t))
            ((≤ adjusted-start-time starting-time adjusted-end-time)
             ;;; the new performance is to start during the core steps of the other
             ;;; performance
             (return t)))))
  ;;; any violation causes an immediate return; hence, if we get here, there is not
  ;;; violation
  nil)


(defmethod (max-performances-violation-p experiment) ()
  (>
    (loop for performance in performance-list
          with count = 1
          do
      (when (scheduled-p performance)
        (incf count))
          finally (return count))
    max-performances))

(defmethod (start-time-violates-performances-per-window-restriction experiment) (starting-time)
    (loop for (start end allowed-performances) in performance-windows
          with count = 1 ;;;the performance we are tying to schedule
          for start-period = start
          for end-period = end
          do
```

```
      (when (≤ start-period starting-time end-period)
        (loop for performance in performance-list
              do
          (when (and (scheduled-p performance)
                     (≤ start-period (scheduled-start-time performance) end-period))
            (incf count)))
        (return (> count allowed-performances)))))

(defmethod (start-time-not-within-performance-window experiment) (starting-time)
  (let ((result nil))
    (loop for (start end performances) in performance-windows
          ;;; this loop finds if the performance is in a window - result must be
          ;;; "not-ed" before being returned
          until result
          do
      (when (≤ start starting-time end)
        (setf result t)))
    (not result)))

;*****************************************************************************
;;; post step feasibility constraint checks


(defmethod (check-for-completion-within-performance-duration performance) (ok new-time)
  (if (null new-time)
      (values "check-for-completion-within-performance-duration called with null new-time"
              new-time)
  (if (≤ (- scheduled-end-time scheduled-start-time)
         (performance-time-window owning-experiment))
      (values ok new-time)
      (values :not-completed-within-performance-duration nil))))

(defmethod (check-for-min-delay-between-performance-violation performance) (ok new-time)
  (if (null new-time)
      (values "check-for-min-delay-between-performance-violation called with null new-time"
              new-time)
  (loop for performance in (performance-list owning-experiment)
        with adjusted-start-time = nil
        do
    (when (and (scheduled-p performance)
               (< (scheduled-start-time performance)
                  (+ scheduled-end-time
                       (min-performance-delay-time owning-experiment))))
      (if (execute-start-up-steps-p performance)
          (progn
            ;;;if the performance has start-up steps, then these steps will have to
            ;;;be re-scheduled, and that must be taken into consideration when
            ;;;checking for the delay between performances
            (setf adjusted-start-time (find-start-time-without-startup-steps performance))
            (when (< adjusted-start-time
                     (+ scheduled-end-time
                          (min-performance-delay-time owning-experiment)))
              (return (values :min-between-performance-delay-violation performance))))
          (return (values :min-between-performance-delay-violation performance)))))
  (values ok new-time)))

(defmethod (check-for-completion-within-performance-window performance) (ok new-time)
  (when (null scheduled-end-time)
    (error "check-for-completion-within-performance-window called with null scheduled end time"
           ))
  (loop with done = nil until (eql done :done)
        for (start end performances) in (performance-windows owning-experiment)
        do
    (cond ((and (≤ start scheduled-start-time end)
                (< end scheduled-end-time))
           (setf done :almost-done new-time nil
                 ok (list :not-completed-within-performance-window (list start end))))
          ((eql done :almost-done)
           (setf new-time start done :done))
          (t nil)))
  (values ok new-time))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

#||
'(update-cumulative-consumables add-time-slice-to-list add-new-instance-to-time-slice-list schedul
e-event schedule-step-crew-members schedule-step-cumulative-consumables schedule-step-consumable-r
esources schedule-step-non-depletable-resources schedule-step-durable-resources schedule-step sche
dule-performance record-performance-and-step-times find-unscheduled-performance schedule-n-perform
ances-of-experiment-beginning test-scheduler

resource-available-in-period resource-available-in-periods get-object-named find-maximum-resource-
available find-quant-resource-already-committed sufficient-resource-in-period-aux sufficient-resou
rce-in-period sufficient-resource-in-periods-p step-schedulable-durable-viewpoint-p find-earliest-
step-schedulable-after-time step-schedulable-durable-viewpoint-aux step-schedulable-non-depletable
-viewpoint-aux step-schedulable-non-depletable-viewpoint-p step-schedulable-consumable-viewpoint-a
ux step-schedulable-consumable-viewpoint-p

check-for-completion-within-performance-window check-for-min-delay-between-performance-violation c
heck-for-completion-within-performance-duration start-time-not-within-performance-window start-tim
e-violates-performances-per-window-restriction max-performances-violation-p start-time-is-within-t
he-scheduled-time-of-some-other-performance-p ok-to-schedule-performance-starting-at-starting-time
-p

BACKTRACK SCHEDULE-OTHER-STEPS

find-time-crew-available-after crew-available-in-time-periods-aux-2 crew-available-in-time-periods
-aux crew-available-in-time-periods-p crew-not-present-in-time-periods-p crew-not-present-in-time-
periods-aux find-earliest-time-crew-combination-available crew-combination-available-in-periods-au
x crew-combination-available-in-periods-p step-schedulable-crew-viewpoint-aux step-schedulable-cre
w-viewpoint-p

step-schedulable-starting-between-inclusive-times-p available-at-time resource-present-in-period r
esource-present-in-periods-p-aux resource-present-in-periods-p resource-not-present-in-periods-p s
tep-schedulable-attitude-viewpoint-aux step-schedulable-attitude-viewpoint-p step-schedulable-targ
et-viewpoint-aux step-schedulable-target-viewpoint-p analyze-times-for-type-failure step-schedulab
le-starting-at-time-aux step-schedulable-starting-at-time-p

find-end-time-without-shutdown-steps find-start-time-without-startup-steps find-earliest-schedulab
le-time-after startup-or-shutdown-steps-required-p between-experiment-constaints get-time-instance
-list get-time-instance get-linked-object update-other-object link-steps copy-step find-step-numbe
red remove-steps generate-required-steps copy-step-list calc-this-step-latest-start-time calc-this
-step-earliest-start-time calc-next-step-latest-start-time calc-next-step-earliest-start-time buil
d-list-from-linked-structure get-first-shutdown-step get-last-startup-step join-shutdown-steps joi
n-startup-steps performance-schedulable-as-starting-time-p-aux-2 performance-schedulable-at-starti
ng-time-p-aux find-first-time-no-overlap find-new-performance-window performance-schedulable-at-st
arting-time-p )
||#

(defmethod (test-scheduler-all mission) ()
  (let ((the-list '(ACOUSTIC EPITAXY ALLOY-S BRIDGMAN HIGHTEMP MEMBRANE SOL-CRYS VAP-CRYS TRAIN-1)
)
    ;;;ACOUSTIC EPITAXY ALLOY-S BRIDGMAN HIGHTEMP MEMBRANE
                    ;;;SOL-CRYS VAP-CRYS TRAIN-1
        (result nil)) ;CONTFLOW HW-MAINT WM-MAINT
    (build-initial-time self)
    (push (loop for key in the-list
          for value = (gethash key experiment-table)
              collect
                (list value (list time-slice-holder key)
                      (schedule-n-performances-of-experiment-beginning
                        value (round (max-performances value) 4) 0)))
          result)
    (format t "~% result = ~S"result)
    (push (loop for key in the-list
          for value = (gethash key experiment-table)
              collect
                (list value (list time-slice-holder key)
                      (schedule-n-performances-of-experiment-beginning
                        value (- (max-performances value)
                                 (round (max-performances value) 4)) 0)))
          result)
    (schedule-desired-crew-monitoring self)
    result))
```

```
(defmethod (test-scheduler mission) (experiment-list num-of-perf-each)
  (let ((result nil))
   (loop for exp in experiment-list
         for instance = (gethash exp experiment-table)
         do
      (push (list instance (list time-slice-holder exp)
                   (schedule-n-performances-of-experiment-beginning
                     instance num-of-perf-each 0))
            result))
  result))

(defmethod (schedule-desired-crew-monitoring mission) ()
  (maphash #'(lambda (exp instance)
               exp
               (schedule-desired-crew-monitoring instance))
           experiment-table))

(defmethod (schedule-desired-crew-monitoring experiment) ()
  (when desired-monitor-steps
  (loop for performance in performance-list
        do
    (when (scheduled-p performance)
      (schedule-desired-crew-monitoring performance)))))

(defmethod (schedule-desired-crew-monitoring performance) ()
  (loop for step in (desired-monitor-steps owning-experiment)
        for performance-step = (find-step-named self (name step))
        do
    (schedule-feasible-crew-monitor performance-step)))

(defmethod (test-scheduler-2 mission) (the-list)
  (build-initial-time self)
    (loop for name in the-list
          for value = (gethash name experiment-table)
          for dummy = (setf (performance-list value) nil)
          for count from 1
          until (> count 3)
              collect
                (list value (list time-slice-holder name)
                      (schedule-n-performances-of-experiment-beginning value 1 0))
          do
      dummy
      (build-initial-time self)))

(defmethod (schedule-n-performances-of-experiment-beginning experiment)
           (number-of-perf beginning-time)
  (setf schedule-shutdown-with-performance nil)
  (let ((new-time nil) (result (list :success number-of-perf))
        (test nil) (scenario-number nil) (last-performance nil))
    (unless (eql name 'dummy-value)
      (loop for i from 1 to number-of-perf
            until (not (eql (first result) :success))
            for next-performance = (find-unscheduled-performance self)
            do
        (unless next-performance
          (setf next-performance (make-instance 'performance :owning-experiment self
                                                :number (1+ (length performance-list)))))
        (when (= i number-of-perf)
          (setf schedule-shutdown-with-performance t))
        (loop with done = nil until done
              do
          (multiple-value-setq (test new-time scenario-number)
            (performance-schedulable-at-starting-time-p
              next-performance beginning-time scenario-number
              (if last-performance last-performance
                  (find-performance-preceding self beginning-time))))
          (cond ((eql test :success)
                 (schedule-performance next-performance 'priority)
                 (setf (scheduled-p next-performance) t)
                 (setf beginning-time
                       (+ min-performance-delay-time (scheduled-end-time next-performance)))
                 (setf done t)
                 (push next-performance performance-list)
```

```
                (setf last-performance next-performance))
               (new-time (setf beginning-time new-time))
               ((null new-time)
                (setf done t result (list test i)))))))
    result))

(defmethod (find-performance-preceeding experiment) (time)
  (let ((result nil))
    (cond ((null performance-list) nil)
          (t (loop for performance in performance-list
                   do
                (when (scheduled-p performance)
                  (cond ((> (scheduled-start-time performance) time) nil)
                        ((null result)
                         (setf result performance))
                        ((> (scheduled-start-time performance) (scheduled-start-time result))
                         (setf result performance))
                        (t nil))))))
    result))

(defmethod (find-start-time-for-earliest-start-scenario experiment) (new-times-list)
  (let ((selected-time nil)(scenario-number nil))
    (if (every #'(lambda (x)
                   (null (second x)))
               new-times-list)
        (setf scenario-number new-times-list)
        (loop for (result new-time scenario-num) in new-times-list
              do
          (cond ((or (and new-time (null selected-time))
                     (and selected-time new-time (< new-time selected-time)))
                 (setf selected-time new-time)
                 (setf scenario-number scenario-num))
                ((and selected-time new-time (= new-time selected-time)
                      (< scenario-num scenario-number))
                 (setf scenario-number scenario-num))
                (t nil))))
    (values selected-time scenario-number)))

(defmethod (find-unscheduled-performance experiment) ()
  (loop for instance in performance-list
        do
    (unless (scheduled-p instance)
      (return instance))))

(defmethod (record-performance-and-step-times performance) ()
  (setf scheduled-p t
        scheduled-start-time (scheduled-start-time (first step-list)))
  (setf scheduled-end-time
        (scheduled-end-time (first (last step-list)))))

(defmethod (schedule-performance performance) (monitor-level)
  (let ((last-step
          (loop for step in step-list
                do
            (schedule-step step monitor-level)
                finally (return step))))
    (setf last-time-slice (last-time-slice last-step))
    (when (cumulative-consumable-list last-step)
      (update-cumulative-consumables
        (get-time-instance *mission* (1+ (scheduled-end-time last-step))
                           (last-time-slice last-step))
        (cumulative-consumable-list last-step)
        ))))
;;;*****************************************************************************
;;; high level step scheduling
(defmethod (schedule-step step) (monitor-level)
  (let ((time-slice nil))
    (setf last-time-slice
          (setf time-slice
                (schedule-step-durable-resources self)))
    (setf time-slice
          (schedule-step-non-depletable-resources self))
    (when (and time-slice (not (eql last-time-slice time-slice)))
```

```
      (setf last-time-slice time-slice))
    (setf time-slice
          (schedule-step-crew-members self monitor-level))
    (when (and time-slice (not (eql last-time-slice time-slice)))
      (setf last-time-slice time-slice))
    (setf time-slice
          (schedule-step-consumable-resources self))
    (when (and time-slice (not (eql last-time-slice time-slice)))
      (setf last-time-slice time-slice))
    (schedule-step-cumulative-consumables self)
  (when (between-experiment-constaints self)
    (update-other-object (get-linked-object *mission* self)))))

(defmethod (schedule-step-durable-resources step) ()
  (loop for (resource quant) in durable-resource-list
        with time-slice =
          (get-time-instance *mission* scheduled-start-time
                             (if previous-step (last-time-slice previous-step) nil))
        do
    (setf time-slice
          (schedule-event
            *mission*
            (list resource quant self)
            'durable-resource-list scheduled-start-time
            (if (and (not (zerop step-delay-min)) resource-carry-thru)
                (+ scheduled-end-time step-delay-min)
                scheduled-end-time)
            time-slice))
        finally (return time-slice)))

(defmethod (schedule-step-non-depletable-resources step) ()
  (loop for (resource quant tolerance) in non-depletable-resource-list
        with time-slice = (get-time-instance *mission* scheduled-start-time last-time-slice)
        do
    (setf time-slice (schedule-event
                       *mission*
                       (list resource quant tolerance self)
                       'non-depletable-resource-list scheduled-start-time
                       (if (and (not (zerop step-delay-min)) resource-carry-thru)
                           (+ scheduled-end-time step-delay-min)
                           scheduled-end-time) time-slice))
        finally (return time-slice)))

(defmethod (schedule-step-consumable-resources step) ()
  (loop for (resource quant) in consumable-resource-list
        with time-slice = (get-time-instance *mission* scheduled-start-time last-time-slice)
        do
    (setf time-slice
          (schedule-event
            *mission*
            (list resource quant self)
            'consumable-resource-list scheduled-start-time scheduled-end-time time-slice))
    finally (return time-slice)))

(defmethod (schedule-step-cumulative-consumables step) ()
  (let ((time-slice-list
          (get-time-instance-list
            *mission* scheduled-start-time scheduled-end-time
                                last-time-slice)))
    (loop for (resource quant) in cumulative-consumable-list
          do
      (loop for time-slice in time-slice-list
            for exisiting-quant = (gethash resource
                                           (cumulative-consumable-table time-slice))
            do

        (setf (gethash resource (cumulative-consumable-table time-slice))
              (if exisiting-quant
                  (+ exisiting-quant quant)
                  quant))))))

(defmethod (schedule-step-crew-members step) (monitor-level)
  (let ((result last-time-slice))
```

```
    (cond ((null crew-monitor)
           (loop for crew-member in scheduled-crew-list
                 with time-slice =
                   (get-time-instance *mission* scheduled-start-time last-time-slice)
                 do
             (setf time-slice (schedule-event
                               *mission*
                               (list crew-member self)
                               'crew-list scheduled-start-time scheduled-end-time
                               time-slice))
                 finally (setf result time-slice))
           )
          ((eql crew-monitor monitor-level)
           (setf result (schedule-feasible-crew-monitor self)))
          (t nil))
    result))

(defmethod (print-time-slices time-slice) ()
  (format t "~% ~S"self)
  (when next-slice
    (print-time-slices next-slice)))

(defmethod (schedule-event mission) (event slot begin end &optional desired-time-slice)
  ;;;cases which must be handled:
  ;;; the time slice starts and ends at the same time as the event
  ;;; the time slice starts at the same time as the event but ends after the event
  ;;; the time slice starts at the same time as the event but ends before the event
  ;;; the time slice starts before the event but ends at the same time as the event
  ;;; the time slice starts before the event starts and ends before the event ends
  ;;; the time slice starts before the event and ends after the event
  ;;; however; the time slice cannot start after the event, or get-time-instance has
  ;;; a bug
  (unless (and desired-time-slice
               (≤ (start-time desired-time-slice) begin (end-time desired-time-slice)))
    (setf desired-time-slice (get-time-instance self begin)))
  (let ((new-instance nil))
    (cond  ((and (= begin (start-time desired-time-slice))
                 (= end (end-time desired-time-slice)))
            (push event (symbol-value-in-instance desired-time-slice slot))
            desired-time-slice)
           ((and (= begin (start-time desired-time-slice))
                 (< end (end-time desired-time-slice)))
            ;;; time slice too long - create a new one after to old one
            (add-time-slice-after-this-one desired-time-slice end)
            (push event (symbol-value-in-instance desired-time-slice slot))
            desired-time-slice)
           ((and (= begin (start-time desired-time-slice))
                 (> end (end-time desired-time-slice)))
            ;;;time slice too short - add events to this one and the next one
            (push event (symbol-value-in-instance desired-time-slice slot))
            (schedule-event self event slot (1+ (end-time desired-time-slice)) end
                            (next-slice desired-time-slice))
            desired-time-slice)

           ((and (> begin (start-time desired-time-slice))
                 (= end (end-time desired-time-slice)))
            ;;;time slice begins too soon - add a new one as the previous
            (setf new-instance
                  (add-time-slice-before-this-one desired-time-slice begin))
            (push event
                  (symbol-value-in-instance desired-time-slice slot))
            new-instance)
           ((and (> begin (start-time desired-time-slice))
                 (< end (end-time desired-time-slice)))
            ;;;too long in both directions
            (add-time-slice-before-this-one desired-time-slice begin)
            (add-time-slice-after-this-one desired-time-slice end)
            (push event (symbol-value-in-instance desired-time-slice slot))
            desired-time-slice)
           ((and (> begin (start-time desired-time-slice))
                 (> end (end-time desired-time-slice)))
            (add-time-slice-before-this-one desired-time-slice begin)
            (push event (symbol-value-in-instance desired-time-slice slot))
```

```
            (schedule-event self event slot (1+ (end-time desired-time-slice)) end
                            (next-slice desired-time-slice)))
          )))


(defmethod (add-time-slice-before-this-one time-slice) (begin)
  (let ((new-slice (copy-self self)))
    (setf (end-time new-slice) (1- begin)
          start-time begin)
    (if prev-slice
        (setf (next-slice prev-slice) new-slice)
        (setf (time-slice-holder *mission*) new-slice))
    (setf (prev-slice new-slice) prev-slice)
    (setf (next-slice new-slice) self)
    (setf prev-slice new-slice)
    (setf (consumable-resource-list new-slice) consumable-resource-list)
    (maphash #'(lambda (key value)
                 (setf (gethash key (cumulative-consumable-table new-slice)) value))
             cumulative-consumable-table)
    (setf consumable-resource-list nil)
    self))

(defmethod (add-time-slice-after-this-one time-slice) (end)
  (let ((new-slice (copy-self self)))
    (setf (start-time new-slice) (1+ end)
          end-time end)
    (when next-slice
      (setf (prev-slice next-slice) new-slice))
    (setf (next-slice new-slice) next-slice)
    (setf (prev-slice new-slice) self)
    (setf next-slice new-slice)
    (setf (consumable-resource-list new-slice) nil)
    (maphash #'(lambda (key value)
                 (setf (gethash key (cumulative-consumable-table new-slice)) value))
             cumulative-consumable-table)
    self))

#||
;;; no longer used ?
(defmethod (add-new-instance-to-time-slice-list mission) (new-instance)
  (setf time-slice-list (add-time-slice-to-list self new-instance time-slice-list)))

(defmethod (add-time-slice-to-list mission) (new-instance slice-list)
  (cond ((null slice-list)
         ;;;last element
         (ncons new-instance))
        ((< (start-time new-instance) (start-time (first slice-list)))
         (cons new-instance slice-list))
        (t
         (cons (first slice-list)
               (add-time-slice-to-list self new-instance (cdr slice-list))))))
||#

(defmethod (update-cumulative-consumables time-slice) (cum-consum-list)
  (loop for (resource quant) in cum-consum-list
        do
    (setf (gethash resource cumulative-consumable-table)
          (if (gethash resource cumulative-consumable-table)
              (+ (gethash resource cumulative-consumable-table) quant)
              quant)))
  (unless (null next-slice)
    (update-cumulative-consumables next-slice cum-consum-list)))

;;;*****************************************************************************
;;; schedule crew monitor time
(defmethod (schedule-feasible-crew-monitor step) ()
  (let ((time-list (generate-list-of-monitor-times self))(result last-time-slice))
    (loop for (start end) in time-list
          for selected-combination = nil
          do
      (loop until selected-combination
            for combination in crew-combinations
            do
```

```
        (when (crew-combination-available-for-monitor self combination start end)
          (setf selected-combination combination)
          (setf result (schedule-crew-monitor self combination start end))))
      (unless selected-combination
      (loop until selected-combination
            for early-shift from 1 to crew-early-shift
            for shift-start = (- start early-shift)
            for shift-end = (- end early-shift)
            do
        (loop until selected-combination
              for combination in crew-combinations
              do
          (when (crew-combination-available-for-monitor
                  self combination shift-start shift-end)
            (setf selected-combination combination)
            (setf result (schedule-crew-monitor self combination shift-start shift-end))))))
      (unless selected-combination
      (loop until selected-combination
            for late-shift from 1 to crew-late-shift
            for shift-start = (+ start late-shift)
            for shift-end = (+ end late-shift)
            do
        (loop until selected-combination
              for combination in crew-combinations
              do
          (when (crew-combination-available-for-monitor
                  self combination shift-start shift-end)
            (setf selected-combination combination)
            (setf result (schedule-crew-monitor self combination shift-start shift-end)))))))
    result))

(defmethod (schedule-crew-monitor step) (combination shift-start shift-end)
  (let ((result nil))
    (push (list combination shift-start shift-end) scheduled-crew-list)
    (loop for crew-member in combination
          with time-slice =
            (get-time-instance *mission* scheduled-start-time last-time-slice)
          do
      (setf time-slice (schedule-event
                         *mission*
                         (list crew-member self)
                         'crew-list shift-start shift-end
                         time-slice))
          finally (setf result time-slice))
    result))

(defmethod (crew-combination-available-for-monitor step) (combination start end)
  (let ((result :success) (other-time nil))
    (loop while (eql result :success )
          with period-list = (get-time-instance-list
                               *mission* start end
                               (if last-time-slice last-time-slice
                                   (if previous-step (last-time-slice previous-step) nil)))
          for crew in combination
          do
      (multiple-value-setq (result other-time)
        (crew-available-in-time-periods-p crew start (1+ (- end start))))
      (when (eql result :success)
        (multiple-value-setq (result other-time)
          (crew-not-present-in-time-periods-p
            self period-list crew start))))
    (if (eql result :success ) result nil)))

(defmethod (generate-list-of-monitor-times step) ()
  (reverse
    (loop for time from (+ scheduled-start-time crew-cycle)
                  to scheduled-end-time by crew-cycle
          for monitor-start = (- time crew-duration)
          for monitor-end = (1- time)
          collect (list monitor-start monitor-end))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (setup-streams nasa-screen-manager) (dw:*program-frame*)
  (setf program-framework dw:*program-frame*)
  (setf (gethash 'error stream-table) (dw::get-program-pane 'error-DISPLAY)
        (gethash 'general stream-table) (dw::get-program-pane 'general-DISPLAY)
        (gethash 'exp-describer stream-table)(dw::get-program-pane 'experiment-describer)
        (gethash 'op-mode stream-table) (dw::get-program-pane 'CURRENT-OP-MODE-DISPLAY)
        (gethash 'performances stream-table) (dw::get-program-pane 'performances-DISPLAY)
        (gethash 'experiments stream-table) (dw::get-program-pane 'experiments-DISPLAY)
        (gethash 'resources stream-table) (dw::get-program-pane 'RESOURCES-DISPLAY)
        (gethash 'edit stream-table) (dw::get-program-pane 'TABLES-DISPLAY)
        (gethash 'init-obj-edit stream-table) (dw::get-program-pane 'init-cbj-display)
        (gethash 'durable-resource-edit stream-table)
        (dw::get-program-pane 'durable-resource-DISPLAY)
        (gethash 'consumable-resource-edit stream-table)
        (dw::get-program-pane 'consumable-resource-DISPLAY)
        (gethash 'crew-resource-edit stream-table)
        (dw::get-program-pane 'crew-resource-DISPLAY)
        (gethash 'target-resource-edit stream-table)
        (dw::get-program-pane 'target-resource-DISPLAY)
        (gethash 'attitude-resource-edit stream-table)
        (dw::get-program-pane 'attitude-resource-DISPLAY)
        (gethash 'listener stream-table )(dw::get-program-pane 'NASA-LISP-LISTENER )
        (gethash 'tables-2 stream-table) (dw::get-program-pane 'TABLES-DISPLAY-2)
        ))

(defmethod (clear-all-histories nasa-screen-manager) (master-key)
  (mapc #'(lambda (key) (clear-history self key))
        (case master-key
          (init-edit '(init-obj-edit durable-resource-edit consumable-resource-edit crew-resource-
edit target-resource-edit attitude-resource-edit)))))

(defmethod (clear-history nasa-screen-manager) (key)
  (let ((dw:*program-frame* program-framework))
    (send (gethash key stream-table) :clear-history)))

(defmethod (select-configuration  nasa-screen-manager) (key)
  (let ((dw:*program-frame* program-framework ))
    (case key
      (init-obj-edit (dw::set-program-frame-configuration 'dw::edit-init-config))
      (edit (dw::set-program-frame-configuration 'DW::TABLES-REPORTING))
      (error (dw::set-program-frame-configuration 'DW::ERROR-REPORTING))
      (performance (dw::set-program-frame-configuration 'DW::NASA-CONFIG-2))
      (general (dw::set-program-frame-configuration 'DW::GENERAL-INFO-CONFIG))
      (experiment (dw::set-program-frame-configuration 'DW::NASA-PERFORMANCE-SCHEDULER))
      (tables-2 (dw::set-program-frame-configuration 'DW::TABLES-REPORTING-2)))
    (gethash key stream-table)))

(defmethod (select-stream nasa-screen-manager) (key)
  (gethash key stream-table))

(defmethod (edit-self nasa-screen-manager) ()
  (apply #'update-self (cons self (get-new-values self)))
  (display-self self (select-configuration self 'edit)))

(defmethod (compute-resource-display-intoto nasa-screen-manager) () nil)

(defmethod (update-self nasa-screen-manager) (new-left-x new-right-x new-lower-y new-upper-y
                                                  new-x-delta new-h-scale-inc
                                                  new-v-scale-inc new-scale-length
                                                  new-min-x-delta new-resource-p)

    (unless (and (= left-x new-left-x)
                 (= right-x new-right-x)
                 (= lower-y new-lower-y)
                 (= upper-y new-upper-y)
                 (= x-delta new-x-delta)
                 (= h-scale-inc new-h-scale-inc)
                 (= v-scale-inc new-v-scale-inc)
                 (= scale-length new-scale-length)
                 (= min-x-delta new-min-x-delta)
                 (null new-resource-p))
```

```
      (setf left-x new-left-x
            right-x new-right-x
            lower-y new-lower-y
            upper-y new-upper-y
            x-delta new-x-delta
            h-scale-inc new-h-scale-inc
            v-scale-inc new-v-scale-inc
            scale-length new-scale-length
            min-x-delta new-min-x-delta)
      (compute-resource-display-intoto self)))

(defmethod (get-new-resource nasa-screen-manager) ()
  (let (choice
        (choice-list
          (nconc '(quit quit)
                 (delete (list (name current-resource) current-resource )
                         (get-resource-list owner-obj) :test #'equal))))
    (loop until
          setf choice
                (dw:menu-choose
                  choice-list
                  :prompt
                  (format nil
                          "The Current Resource is ~S; Select A Different Resource or Quit"
                          (name current-resource)))))
    (if (eql choice 'quit) nil choice)))

(defmethod (get-new-values nasa-screen-manager) ()
  (let (new-left-x new-right-x new-lower-y new-upper-y new-x-delta new-h-scale-inc
        new-v-scale-inc new-scale-length new-min-x-delta new-resource)
    (setf new-resource (get-new-resource self))
    (when new-resource
      (setf current-resource new-resource)
      (setf v-scale-inc (gethash (name current-resource) v-scale-table)
            y-axis (gethash (name current-resource) y-axis-table)))
    (dw:accepting-values
      (*standard-output*
        :own-window t :label
        (format nil
                "Indicate Modifications To Values For Display Control"))
      (setf new-left-x
            (accept 'number :default left-x :query-identifier 'new-left-x
                    :stream *standard-output*
                    :prompt (format nil "enter new left coordinate for resource display   "))
            new-right-x
            (accept 'number :default right-x :query-identifier 'new-right-x
                    :stream *standard-output* :prompt
                    (format nil "enter new right coordinate for resource display   "))
            new-lower-y
            (accept 'number :default lower-y :query-identifier 'new-lower-y
                    :stream *standard-output* :prompt
                    (format nil "enter new bottom coordinate for resource display   "))
            new-upper-y
            (accept 'number :default upper-y :query-identifier 'new-upper-y
                    :stream *standard-output*
                    :prompt (format nil "enter new top coordinate for resource display   "))
            new-min-x-delta
            (accept 'number :default min-x-delta :query-identifier 'new-min-x-delta
                    :stream *standard-output*
                    :prompt (format nil "enter new time increment minimum width   "))
            new-x-delta
            (accept 'number :default x-delta :query-identifier 'new-x-delta
                    :stream *standard-output*
                    :prompt (format nil "enter new time increment width   "))
            new-h-scale-inc
            (accept 'number :default h-scale-inc :query-identifier 'new-h-scale-inc
                    :stream *standard-output*
                    :prompt (format nil "enter new horizontal scale labeling increment    "))
            new-v-scale-inc
            (accept 'number :default v-scale-inc :query-identifier 'new-v-scale-inc
                    :stream *standard-output*
                    :prompt (format nil "enter new vertical scale labeling increment    "))
            new-scale-length
```

```
            (accept 'number :default scale-length :query-identifier 'new-scale-length
                    :stream *standard-output*
                    :prompt (format nil "enter new scale tick mark length    "))))
    (list new-left-x new-right-x new-lower-y new-upper-y new-x-delta new-h-scale-inc
          new-v-scale-inc new-scale-length new-min-x-delta new-resource)))

(defmethod (display-self nasa-screen-manager) (stream)
  (present self 'nasa-screen-manager-edit-display :stream stream))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (present-step step) (stream)
  ;;this is a first cut-- obviously, this needs to be broken up into several display
  ;;functions to handle the cases where the input is not a single value, to relieve
  ;;the user of the burden of knowing the syntax of each of the lists.
  (format stream "~%ID ~A MAX-DURATION ~A MIN-DURATION ~A STEP-DELAY-MIN ~A STEP-DELAY-MAX ~A CREW
-MONITORING-TIME ~A  CONCURRENT-WITH ~A" id  max-duration min-duration
          step-delay-min  step-delay-max  crew-monitoring-time concurrent-with )
  (format stream "~% CONSUMABLE-RESOURCE-LIST :")
  (if consumable-resource-list
      (mapc #'(lambda (resource-qty)
                (format stream "~%   ~A   ~A" (first resource-qty)(second resource-qty)))
            consumable-resource-list )
      (format stream "  NONE"))
  (format stream "~% DURABLE-RESOURCE-LIST:")
  (if durable-resource-list
      (mapc #'(lambda (resource-qty-releasable)
                (format stream "~%   ~A   ~A" (first resource-qty-releasable)
                        (second resource-qty-releasable)))
            durable-resource-list)
      (format stream "  NONE"))
  (format stream "~% CREW-REQUIREMENTS :")
  (if crew-requirements
      (mapc #'(lambda (crew-list-qty)
                (format stream "~%   ~A   ~A" (first crew-list-qty)(second crew-list-qty)))
            crew-requirements)
      (format stream "  NONE"))
  (format stream "~% TARGET-LIST:")
  (if target-list
      (mapc #'(lambda (target)(format stream "~%   ~A" target ))target-list )
      (format stream "  NONE"))
  (format stream "~% ATTITUDE-LIST:")
  (if attitude-list
      (mapc #'(lambda (attitude)
                (format stream "~%   ~A" attitude )) attitude-list )
      (format stream "  NONE")))

(defmethod (create-new-obj step-template) (owner)
  (setf owning-object owner)
  (push self (prototype-step-list owner)))

(defmethod (create-new-obj startup-step) (owner)
  (setf owning-object owner)
  (push self (startup-steps owner)))

(defmethod (create-new-obj shutdown-step) (owner)
  (setf owning-object owner)
  (push self (shutdown-steps owner)))

(defmethod (create-new-obj step) (owner)
  (setf owning-object owner)
  (push self (step-list owner)))

(defmethod (create-new-obj step :after) (&rest ignore)
  (format tv:initial-lisp-listener "this is a stub (create-new-obj step :after)"))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defmethod (copy-self time-slice) ()
  (let ((new-instance
          (make-instance 'time-slice :start-time start-time
                         :end-time end-time
                         :crew-list (copy-list crew-list)
                         :non-depletable-resource-list
                         (copy-alist non-depletable-resource-list)
                         :durable-resource-list (copy-alist durable-resource-list)
                         :target-list (copy-list target-list)
                         :attitude-list (copy-list attitude-list)
                         :start-x start-x
                         :top-y top-y)))

    (maphash #'(lambda (key value)
                 (setf (gethash key (performance-step-table new-instance)) value))
             performance-step-table)
    new-instance))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(defun translate-universal-time-to-time-period (univ-time)
  (floor (- univ-time (universal-start-time (init-obj *mission*)))
         (time-inc (init-obj *mission*))))

(defun translate-seconds-to-time-periods (seconds)
  (/ seconds (time-inc (init-obj *mission*))))

(defun translate-time-list-to-seconds (time-list)
  (+ (fourth time-list)
     (* 60 (+ (third time-list)
              (* 60 (+ (second time-list)
                       (* 24 (first time-list))))))))

(defmethod (translate-mission-period-to-universal-time nasa-init-obj) (mission-periods)
  (multiple-value-bind (secs mins hours day month year day-of-week)
      (decode-universal-time (+ universal-start-time (* time-inc mission-periods)))
    (values secs mins hours day (CASE month
                                  (1 'JAN)
                                  (2 'FEB)
                                  (3 'MAR)
                                  (4 'APR)
                                  (5 'MAY)
                                  (6 'JUN)
                                  (7 'JUL)
                                  (8 'AUG)
                                  (9 'SEP)
                                  (10 'OCT)
                                  (11 'NOV)
                                  (12 'DEC))
                                year (case day-of-week
                                           (0 'mon)
                                           (1 'tue)
                                           (2 'wed)
                                           (3 'thu)
                                           (4 'fri)
                                           (5 'sat)
                                           (6 'sun))))))

(DEFMETHOD (translate-mission-period-to-mission-time nasa-init-obj) (mission-period)
  (let ((days 0) (hours 0) (mins 0) (secs 0) (remainder 0))
    (multiple-value-setq (days remainder)
      (floor (* time-inc mission-period) seconds-per-day))
    (multiple-value-setq (hours remainder)
      (floor remainder seconds-per-hour))
    (multiple-value-setq (mins secs)
      (floor remainder 60))
    (values days hours mins secs)))

(defmethod (output-time-date-to-stream nasa-init-obj) (stream mission-periods)
  (multiple-value-bind (secs mins hours day month year day-of-week)
      (translate-mission-period-to-universal-time self mission-periods)
    (IF (< day 10)
        (format stream "~S,  ~S ~S ~S, ~S:~S:~S" day-of-week day month year hours mins secs)
    (format stream "~S, ~S ~S ~S, ~S:~S:~S" day-of-week day month year hours mins secs))))


;*****************************************************************************
;;; CALCULATIONS FOR INITIAL TIMES

(defmethod (determine-universal-start-time nasa-init-obj) ()
  (setf universal-start-time
        (encode-universal-time (third mission-launch-time)
                               (second mission-launch-time)
                               (first mission-launch-time)
                               (first mission-launch-date)
                               (second mission-launch-date)
                               (third mission-launch-date))))

(defmethod (determine-initial-universal-times nasa-init-obj) ()
  (determine-universal-start-time self)
  (multiple-value-bind
```

```
    (second minute hour day month year day-of-week)
      (decode-universal-time universal-start-time)
    year month day
    (determine-seconds-until-start-of-first-full-day self second minute hour)
    (determine-start-of-first-sunday self day-of-week)
    ))

(defmethod (determine-end-times nasa-init-obj) ()
  (setf universal-end-time
        (+ universal-start-time (translate-time-list-to-seconds mission-duration)))
  (multiple-value-bind
    (secs mins hrs day month year)
      (decode-universal-time universal-end-time)
    (setf mission-end-date (list day month year)
          mission-end-time (list hrs mins secs))))


(defmethod (determine-seconds-until-start-of-first-full-day nasa-init-obj)
           (second minute hour)
  (setf seconds-until-start-of-day
        (add-seconds-for-each-hour
          hour (add-seconds-for-each-minute minute (add-seconds-as-needed second)))))

(defmethod (determine-start-of-first-sunday nasa-init-obj) (day-of-week)
  (setf first-sunday-start-time
        (+ universal-start-time
           (add-seconds-for-each-day day-of-week seconds-until-start-of-day))))

(defun add-seconds-as-needed (second)
  ;;;return the number of seconds until the start of the next minute, and a flag to
  ;;;indicate whether we started on a partial minute
  (if (zerop second)
      (list 0 nil)
      (list (- 60 second) t)))


(defun add-seconds-for-each-minute (minute seconds-and-add-minute-flag)
  ;;;return the number of seconds until the start of the next hour, and a flag to
  ;;;indicate whether we started on a partial hour
  (when (second seconds-and-add-minute-flag)
    (incf minute))
  (cond ((zerop minute)
         ;;;we can have 0 minutes only if we had zero seconds -- hence we launched on
         ;;;the hour.
         (list 0 nil))
        (t (list (+ (first seconds-and-add-minute-flag)
                    (* (- 60 minute) 60))
                 t))))

(defun add-seconds-for-each-hour (hour seconds-and-add-hour-flag)
  (when (second seconds-and-add-hour-flag)
    (incf hour))
  (cond ((zerop hour)
         ;;;we can have 0 hours only if we has zero seconds and zero minutes -- hence
         ;;;we launched at midnight
         (list 0 nil))
        (t (list (+ (first seconds-and-add-hour-flag)
                    (* (- 24 hour) 60 60))
                 t))))

(defun add-seconds-for-each-day (day-of-the-week seconds-and-add-day-flag)
  (when (second seconds-and-add-day-flag)
    (incf day-of-the-week))
  (cond ((= day-of-the-week 7)
         ;;;to get here, we must have launched on sunday -- since the crew gets the
         ;;;rest of the launch day off, we need time on next sunday
         (+ (* 6 (seconds-per-day (init-obj *mission*)))
            (first seconds-and-add-day-flag)))
        ((and (zerop (first seconds-and-add-day-flag))
              (= day-of-the-week 6))
         ;;;once again, sunday, this time at midnight
         (* (* 7 (seconds-per-day (init-obj *mission*)))))
        (t ;;;if the day is 6 (sunday), then the mission launched after  midnight
```

```
;;; -*- Package: USER; Base: 10; Mode: LISP; Syntax: Common-lisp; -*-

(defmethod (unschedule-self performance) ()
  (mapc #'unschedule-self step-list)
  (setf scheduled-start-time nil scheduled-end-time nil scheduled-p nil step-list nil))

(defmethod (unschedule-steps-from performance) (first-step-number)
  (let ((unschedule-list (member first-step-number step-list :key #'number)))
    (setf scheduled-end-time (scheduled-end-time (previous-step (first unschedule-list))))
    (mapc #'unschedule-self unschedule-list)))

(defmethod (unschedule-shutdown-steps performance) ()
  (unschedule-steps-from
    self (find-step-numbered self (number (first (shutdown-steps owning-experiment))))))

(defmethod (unschedule-self step) ()
  (let ((period-list
          (get-time-instance-list
            *mission* scheduled-start-time scheduled-end-time last-time-slice)))
  (loop for period in period-list
        do
    (unschedule-crew self period)
    (unschedule-durables self period)
    (unschedule-non-depletables self period)
    (unschedule-consumables self period))
  (when (and resource-carry-thru (not (zerop step-delay-min)))
    (setf period-list (get-time-instance-list
                        *mission* (+ scheduled-start-time max-duration)
                        (1- (scheduled-start-time next-step))
                        (next-slice (first (last period-list)))))
    (loop for period in period-list
          do
      (unschedule-durables self period)
      (unschedule-non-depletables self period)))
  (setf period-list
        (get-time-instance-list
          *mission*
          (if next-step
              (1- (scheduled-start-time next-step))
              (+ scheduled-start-time max-duration))
          (max-time (init-obj *mission*))
          (next-slice (first (last period-list)))))
  (unless next-step
    (loop for period in period-list
          do
      (unschedule-cumulate-resources self period)))))

(defmethod (unschedule-crew step) (period)
  (loop for crew in scheduled-crew-list
        do
    (setf (crew-list period) (delete (list crew self) (crew-list period) :test #'equal))))

(defmethod (unschedule-durables step) (period)
  (loop for (resource quant) in durable-resource-list
        do
    (setf (durable-resource-list period)
          (delete (list resource quant self) (durable-resource-list period) :test #'equal))))

(defmethod (unschedule-non-depletables step) (period)
  (loop for (resource quant tolerance) in non-depletable-resource-list
        do
    (setf (non-depletable-resource-list period)
          (delete (list resource quant tolerance self) (non-depletable-resource-list period)
                  :test #'equal))))

(defmethod (unschedule-consumables step) (period)
  (loop for (resource quant) in consumable-resource-list
        do
    (setf (consumable-resource-list period)
          (delete (list resource quant self)
                  (consumable-resource-list period) :test #'equal)))
  (loop for (resource quant) in cumulative-consumable-list
        do
```

```
    (setf (gethash resource (cumulative-consumable-table period))
          (decf (gethash resource (cumulative-consumable-table period)) quant))))

(defmethod (unschedule-cumulate-resources step) (period)
  (loop for (resource quant) in cumulative-consumable-list
        do
    (setf (gethash resource (cumulative-consumable-table period))
          (decf (gethash resource (cumulative-consumable-table period)) quant))))
```

Appendix B
Symbolics Code Listing for the Multiple Pass Multiple Resource Allocation Program

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

;;;;;;;;;;;;;;Global Variables;;;;;;;;;;;;

(defflavor selection-menu ()
           (tv:drop-shadow-borders-mixin
            tv:multiple-menu))

(defflavor shadowed-tv-window ()
  (tv:drop-shadow-borders-mixin
    dw:dynamic-window))


(defvar *frames*)               ;;Loaded from data file.

(defvar *max-time*)

(defvar *time-list*)

(defvar *lambda-lists*)

(defvar *paths*)

(defvar *original-screen-size* nil)

(defvar *second-time* nil)

(defvar *current-file* "")

(defvar *Resource-File-Directory* "andy:>jsr>resource-allocation>multiple-data-files>")

(defvar *resources*)

(defvar *resource-variables*)

(defvar *resources-output* nil)

(defvar scheduled-items)

(defvar *maximizing-resource-list*)

(defvar *maximizing-resource-position*)

(defvar *graphical-output* nil)

(defvar *graphical-display* nil)

(defvar *resource-output-window* (tv:make-window 'dw:dynamic-window
                                                 :label "Resource Allocation Window"
                                                 :blinker-p nil))


(defvar *display-menu* (tv:make-window
                         'selection-menu
                         :label "Select Displayed Output"
                         :default-character-style '(:fix :roman :large)
                         :special-choices '(("Selection Complete" :funcall-with-self complete))))

(defvar *resource-menu-window* (tv:make-window 'dw:dynamic-window
                                               :label "Experiment Data Editor Window"
                                               :blinker-p t))

;(defvar *Data-choices-menu* (tv:make-window 'tv:momentary-menu
;                                           :borders 4
;                                           :label 'Alternate Data File List'))

(defvar *message-window* (tv:make-window 'dw:dynamic-window
                         ;       :blinker-p nil
                                 :edges-from '(300 300 850 400)
                                 :margin-components
                                 '((dw:margin-scroll-bar :visibility :if-needed)
                                   (dw:margin-ragged-borders :thickness 4)
```

```
                                (du:margin-label
                                  :margin :botton
                                  :string "Message Window       (Press any key to EXIT)"))))

(defvar *graphics-window* (tv:make-window 'du:dynamic-window
                                :blinker-p nil
                                :label "Resource Allocation Graphics Display"))

(defvar *Font* (si:backtranslate-font
               (fed:read-font-from-bfd-file 'sys:fonts;tv;40vr.bfd.newest' )))
```

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;Presentation types and actions for mouse sensitivity.;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;This defines the label presentation types.
(define-presentation-type label-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
          (format stream "the selection ~a" (car object))))

;;This is what is done when a column or row label is selected.
(define-presentation-action label-type
   (label-type t
    :gesture :left
    :context-independent t
    :documentation  "Resource Operations")
   (exit)
  (throw 'resource exit))

;;This defines the label presentation types.
(define-presentation-type exp-label-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
          (format stream "the selection ~a" (car object))))

;;This is what is done when a column or row label is selected.
(define-presentation-action exp-label-type
   (exp-label-type t
    :gesture :left
    :context-independent t
    :documentation  "Experiment Operations")
   (exit)
  (throw 'resource exit))



;;This defines the item presentation type and documentation line display
(define-presentation-type resource-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
          (format stream "the resource ~A" (car object))))

;;This is what is done when the item is selected
(define-presentation-action choose-type
   (resource-type t
    :gesture :left
    :context-independent t
    :documentation  "Change this value")
   (resource)
  (throw 'resource
    (list resource (get (caar resource)
     (read-from-string (format nil "~a-presentation" (cadar resource)))))))

;;This defines the item presentation type and documentation line display
(define-presentation-type control-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
          (format stream "the selection ~a" (car object))))

;;This is what is done when a command is selected
(define-presentation-action control-type
   (control-type t
    :gesture :left
    :context-independent t
    :documentation  "Execute this Command")
   (exit)
  (throw 'resource (read-from-string exit)))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  Program functions                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;This is the Driving Function for the Data Editor.
(defun examine-data ()
  (send *resource-menu-window* :select)
    (loop with again = t
       while again
       do
      (dw::with-output-truncation (*resource-menu-window* :horizontal t)
      (make-window-layout)
      (send *resource-menu-window* :set-cursor-visibility nil)
      (setq again
         (loop with finished = nil
       until finished
       as choice = (change-data-point)
       while choice
       do
           (cond ((atom choice)
          (case choice
            (load
      (open-input-file)
      (initialize-markers-and-variables)
      (return t))
            (save (save-new-file))
            (exit (return nil))))
         (t (case (car choice)
      (exp
        (take-experiment-action
          (cadr choice)
          (get-option-list (format nil "For Experiment ~'b⊂a~⊃"
           (cadr choice))
           '("Move this Experiment"
     "Delete this Experiment"
     "Add an Experiment ABOVE"
     "Add an Experiment BELOW")))
        (return t))
      (resource
        (take-resource-action
          (cadr choice)(caddr choice)
          (get-option-list (format nil "For Resource ~'b⊂a~⊃"
           (cadr choice))
           (cond ((member (cadr choice)
          '("Duration" "Performances")
          :test #'string-equal)
          '("Set Value Globally"
            "Set Maximum Value"
            "Move this Resource"
            "Add Resource to the LEFT"
            "Add Resource to the RIGHT"
            "Edit Resource Constraints"))
         (t
          '("Set Value Globally"
            "Set Maximum Value"
            "Move this Resource"
            "Delete this Resource"
            "Add Resource to the LEFT"
            "Add Resource to the RIGHT"
            "Edit Resource Constraints")))))
        (return t)))))))))
    (send *terminal-io* :select))

;;
(defun get-option-list (prompt options)
  (dw:menu-choose options
       :prompt prompt
       :center-p t
       :row-wise nil))

;;
(defun take-resource-action (resource pos action)
  (cond ((string-equal action "Set Value Globally")
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
      (let ((value (get-stream '((number :prompt "Global Value"
         :default 0
         :query-identifier jsr))
        (format nil "Set ~'b∈A~⊃Value Globally     " resource))))
        (if value
            (initialize-experiment-resource-value
      (make-variable-from-string resource ) value))))
     ((string-equal action "Set Maximum Value")
      (let ((resource-var (make-variable-from-string resource)))
        (zl:putprop resource-var
            (get-stream '((number :prompt "Maximum Value"
          :default ,(get resource-var 'resource-limit)
          :query-identifier jsr))
        (format nil "Set ~'b∈A~⊃Maximum Value     "
        resource))
          'resource-limit)))
     ((string-equal action "Edit Resource Constraints")
      (modify-resource-constraint-equations (make-variable-from-string resource)))
     ((string-equal action "Move this Resource")
      (send-message-to-user (format nil "~2%     Use mouse to SELECT which RESOURCE to~
                    ~%     place ~'b∈A~⊃beside." resource))
      (remove-resource resource nil)
      (let ((position (find-position 'label-type resource)))
        (setq *resources* (insert-item-in-list *resources* resource position)
      *resource-variables* (insert-item-in-list *resource-variables*
     (make-variable-from-string resource) position))))
     ((string-equal action "Delete this Resource")
      (remove-resource resource))
     ((string-equal action "Add Resource to the LEFT")
      (add-resource pos))
     ((string-equal action "Add Resource to the RIGHT")
      (add-resource (+ 1 pos)))))

(defun modify-resource-constraint-equations (resource)
  (send *message-window* :set-margin-components
     '((dw:margin-scroll-bar :visibility :if-needed)
       (dw:margin-ragged-borders :thickness 4)
       (dw:margin-label
         :margin :bottom
         :string "Constraint Editor Window       (Press <END> key to EXIT)")))
  (send *message-window* :clear-history)
  (send *message-window* :select)
  (format *message-window* "~2%")
  (send *message-window* :set-cursor-visibility :blink)
  (edit-constraint-equation resource)
  (send *message-window* :deselect)
  (send *message-window* :set-cursor-visibility nil)
  (send *message-window* :set-margin-components
     '((dw:margin-scroll-bar :visibility :if-needed)
       (dw:margin-ragged-borders :thickness 4)
       (dw:margin-label
         :margin :bottom
         :string "Message Window       (Press any key to EXIT)"))))


(defun edit-constraint-equation (resource)
  (let ((buffer (tv:kbd-get-io-buffer))
     (equation (format nil "~a" (get resource 'resource-constraint-function))))
    (send *message-window* :clear-input)
    (loop for i from 0 to (- (length equation) 1)
       do
      (tv:io-buffer-put buffer (char equation i)))
    (zl:putprop resource (read-from-string (accept 'string  :stream *message-window*
       :activation-chars '(#\end)
       :prompt nil)) 'resource-constraint-function)))

(defun find-position (type resource)
  (let ((position)
     (data (catch 'resource (accept type
            :prompt nil
            :stream *resource-menu-window*))))
    (case (car data)
      (exp
```

```
      (setq position (position (cadr data)(get 'list-of 'names)))
      (case (read-from-string
      (get-option-list (format nil "Place ~'b∈A~⊃ resource)
       (list (format nil "Above ~'b∈A~⊃ (cadr data))
             (format nil "Below ~'b∈A~⊃ (cadr data)))))
        (ABOVE (+ 1 position))
        (t (+ 2 position))))
       (resource
      (setq position (position (cadr data) *resources* :test #'string-equal))
       (case
         (read-from-string
           (get-option-list (format nil "Place ~'b∈A~⊃ resource)
            (list (format nil "Left of ~'b∈A~⊃ (cadr data))
           (format nil "Right of ~'b∈A~⊃ (cadr data)))))
        (LEFT (+ 1 position))
        (t (+ 2 position)))))))

;;
(defun take-experiment-action (exp action)
  (cond ((string-equal action "Move this Experiment")
      (send-message-to-user (format nil "~2%     Use mouse to SELECT which EXPERIMENT to~
                    ~%     place ~'b∈A~⊃beside." exp))
       (remove-experiment exp nil)
       (let ((position (find-position 'exp-label-type exp)))
         (zl:putprop 'list-of (insert-item-in-list (get 'list-of 'names)
           exp  position) 'names)))
      ((string-equal action "Delete this Experiment" )
       (remove-experiment exp t))
      ((string-equal action "Add an Experiment ABOVE")
       (add-experiment (+ 1 (position exp (get 'list-of 'names)))))
      ((string-equal action "Add an Experiment BELOW")
       (add-experiment (+ 2 (position exp (get 'list-of 'names)))))))

(defun remove-experiment (exp message)
  (zl:putprop 'list-of (remove exp (get 'list-of 'names)) 'names)
  (if message
      (send-message-to-user
     (format nil "~2%~5tThe EXPERIMENT named ~'b∈a~⊃has been deleted." exp))))

(defun add-experiment (position)
  (let ((variable (make-variable-from-string
         (get-stream '((string :prompt "Enter EXPERIMENT NAME"
      :query-identifier jsr))
           "Add Experiment Utility                    "))))
    (zl:putprop 'list-of (insert-item-in-list (get 'list-of 'names) variable position) 'names)
    (loop for item in *resource-variables*
       do
      (zl:putprop variable 0 item))))

;;This function is the top level controller for the input window.
(defun make-window-layout ()
  (send *resource-menu-window* :clear-history)
  (format *resource-menu-window* "~2%~40t~vExperiment Data Editor~⊃4%" *Font*)
  (let* ((space 10))
    (setq *resource-variables* (loop for resource in *resources*
         initially (space-over *resource-menu-window*
      (+ 6 space))
         collect (make-variable-from-string resource) into var
         counting t into place
         finally (return var)
         do
      (space-over *resource-menu-window* space)
      (make-mouse-sensitive-labels ""
        (list 'resource resource place))))
    (format *resource-menu-window* "~%")
    (loop for exp in (get 'list-of 'names)
       counting t into place
       do
     (make-mouse-sensitive-labels "~%"
       (list 'exp exp place))
      (loop for variable in *resource-variables*
         for header in *resources*
         as width = (string-length header)
```

```
        for column first (+ space (/ width 2.0) space)
           then (+ space (/ width 2.0) column)
        do
     (place-variable column variable exp)
     (setq column (+ (/ width 2.0) column))))
    (place-commands)))

;;This command puts the column and row labels as presentations
(defun make-mouse-sensitive-labels (return object &key (stream *resource-menu-window*)
        (type 'label-type))
  (dw:with-output-as-presentation (:single-box t
       :stream stream
       :type type
       :object object)
    (format stream (format nil "~a~A" return (cadr object)))))

;;This command creats the commands at bottom of menu.
(defun place-commands ()
  (format *resource-menu-window* "~6%")
  (loop for command in '("Exit Data Editor" "Save Current Data to File"
      "Load New Data File")
     do
    (space-over *resource-menu-window* 17)
    (dw:with-output-as-presentation (:single-box t
         :stream *resource-menu-window*
         :type 'control-type
         :object command)
      (surrounding-output-with-border (*resource-menu-window* :shape :oval
           :filled t
           :move-cursor nil)
     (format *resource-menu-window* command)))))

;;This function assists in proper relative heading column spacing
(defun space-over (stream space)
  (format stream (format nil "~~~Aa" space) ""))

;;This function takes a string and returns an atom.
(defun make-variable-from-string (str)
  (loop with flag = 1
     for item being the array-elements in str
     if (not (string-equal item " "))
       collect item into var
       and do
          (setq flag 0)
     else if (= flag 0)
          collect "-" into var
          and do
          (setq flag 1)
        finally  (return (read-from-string
      (apply #'string-append
     (cond ((= flag 1)
            (reverse (cdr (reverse var))))
           (t var)))))))

;;This function assists in correct column spacing
(defun place-variable (column variable exp)
  (format *resource-menu-window* (format nil "~~~at" (zl:fix column)))
  (format-item-mouse-sensitive *resource-menu-window* (get exp variable)
          (list (list exp variable)
         (multiple-value-bind (a b)
     (send *resource-menu-window* :read-cursorpos)
          (list a b)))))


;;This function prints the item to the screen with mouse sensitivity
(defun format-item-mouse-sensitive (stream item descriptors)
    (zl:putprop (caar descriptors) item (cadar descriptors))
    (send stream :set-cursorpos (caadr descriptors) (cadadr descriptors))
    (clearspace stream)
    (zl:putprop (caar descriptors)
     (dw:with-output-as-presentation (:single-box t
      :stream stream
      :type 'resource-type
```

```
      :object descriptors)
       (send stream :set-cursorpos (caadr descriptors) (cadadr descriptors))
       (format stream "~8@a" item))
     (read-from-string (format nil "~a-presentation" (cadar descriptors)))))

;;This function removes the typed in values to allow for presentations.
(defun clearspace (stream)
  (loop repeat 8
     do
    (send stream :clear-char)
    (send stream :forward-char)))

;;This function reads in a value, but does not issue a line-feed.
(defun read-without-return (&optional (stream *standard-output*)
        &key (activation-characters '(#\Return #\End )))
  (loop with cursor-position = (list (multiple-value-bind (a b)
       (send stream :read-cursorpos) (list a b)))
     with var2 = nil
     with position = 0
     as var1 = (send stream :tyi)
     as total-length = (length var2)
     until (member var1 activation-characters)
     if var1
       do
         (cond ((and (equal var1 #\rubout) var2)
          (send stream :tyo #\backspace)
          (send stream :clear-char)
          (setq var2 (cdr var2)
      position (1-  position)
      cursor-position (cdr cursor-position)))
       ((and (or (equal var1 #\c-B)(equal var1 #\backspace)) var2)
        (setq position (1- position))
        (send stream :tyo var1))
       ((equal var1 #\c-F)
        (cond ((< position total-length)
       (setq position (1+ position))
       (send stream :tyo var1))))
       ((= position total-length)
        (setq var2 (cons var1 var2)
      position (1+ position)
      cursor-position (cons (multiple-value-bind (a b)
          (send stream :read-cursorpos)
          (list a b)) cursor-position))
        (format stream "~a" var1))
       ((or (equal var1 #\c-B) (equal var1 #\rubout )))
       (t (send stream :insert-char)
          (format stream "~A" var1)
          (setq var2 (reverse (loop for temp = nil
    then (append temp (list (car end)))
           for end = (reverse var2) then (cdr end)
           repeat position
           finally (return
     (append temp (cons var1 end)))))))))
     finally (return (cond (var2 (setq var2 (read-from-string
      (apply #'string-append (reverse var2))))))))))

;;This function allows the data values to be changed.
(defun change-data-point ()
  (let ((data (catch 'resource (accept '((or resource-type control-type
          label-type exp-label-type))
     :prompt nil
     :stream *resource-menu-window*)))
     (original-position (multiple-value-bind (a b)
     (send *resource-menu-window* :read-cursorpos)
           (list a b)))
     (position))
    (cond ((or (atom data) (atom (car data))) data)
       (t
        (setq position (cadar data))
        (send *resource-menu-window* :erase-displayed-presentation (cadr data))
        (send *resource-menu-window* :set-cursorpos (car position)(cadr position))
        (send *resource-menu-window* :set-cursor-visibility :blink)
        (format-item-mouse-sensitive *resource-menu-window*
```

```
      (read-without-return *resource-menu-window*)
      (car data))
         (send *resource-menu-window* :set-cursor-visibility nil)
         (send *resource-menu-window* :set-cursorpos (car original-position)
       (cadr original-position))
         'data))))

;;This function returns the list of data files that can be selected.
(defun get-data-file-list ()
  (loop for directory in (cdr (fs:directory-list *Resource-File-Directory* ))
     as pathname = (cond ((not (string= (send (car directory) :name) "err"))
          (format nil "~A" (send (car directory) :string-for-dired))))
     collect  pathname ))

;;This function allows the modified data to be saved to a data file.
(defun save-new-file ()
  (with-open-file (stream (string-append *Resource-File-Directory*
      (get-stream '((string :prompt "Enter the Filename"
            :query-identifier jsr))
          "Save File Utility                    ")
      ".data")
        :direction :output
        :if-exists :new-version)
    (format stream "~2%(setq *resources* '(")
    (loop for resource in *resources*
       do
      (format stream " ~a~A~a " #\" resource #\"))
    (format stream "))~2%(setq *frames* '(")
    (loop for exp in (get 'list-of 'names)
       do
      (format stream "~%~a" (cons exp (loop for prop in *resource-variables*
       collect (list prop (list (get exp prop)))))))
    (format stream "))")))

;;This function creates a window and prompts the user for a file name.
(defun get-stream (arguments header)
  (dw:accept-values arguments
         :OWN-WINDOW t
         :temporary-p nil
         :prompt header
         :initially-select-query-identifier 'jsr))

;;This function controls the adding of a resource.
(defun add-resource (position)
  (let* ((new-resource (multiple-value-bind (a b)
         (get-stream '((string :prompt "Enter RESOURCE NAME"
      :query-identifier jsr)
      (number :prompt "Initial Value"
      :default 0))
            "Add Resource Utility                        ")
      (list a b)))
      (variable (make-variable-from-string (car new-resource))))
    (cond ((member variable *resource-variables*)
        (send-message-to-user
          (format nil "~2%~5tThe RESOURCE named ~'b∈a~⊃already exists."
          (car new-resource))))
       (t
         (initialize-experiment-resource-value variable (cadr new-resource))
         (setq *resources* (insert-item-in-list *resources* (car new-resource) position)
       *resource-variables* (insert-item-in-list *resource-variables*
         variable position))))))

;;This function puts an initial value in the resource variables.
(defun initialize-experiment-resource-value (new-resource value)
  (loop for item in (get 'list-of 'names)
     do
    (zl:putprop item value new-resource)))

;;This function inserts an item in a list at position.
(defun insert-item-in-list (lst item position)
  (loop for i from 1
     for each on lst
     until (= i position)
```

```
      collecting (car each) into var
      finally (return (append var (list item) each))))

;;This function allows communication between the user and the program.
(defun send-message-to-user (message)
  (send *message-window* :clear-history)
  (send *message-window* :set-cursor-visibility nil)
  (send *message-window* :select)
  (format *message-window* message)
  (send *message-window* :any-tyi)
  (send *message-window* :deselect))

;;This function removes a resource from consideration by program.
(defun remove-resource (resource &optional (message t))
  (setq *resources* (remove resource *resources* :test #'string-equal)
     *resource-variables* (remove (make-variable-from-string resource)
           *resource-variables*))
  (if message
      (send-message-to-user
     (format nil "~2%~5tThe RESOURCE named ~'b<a~>has been deleted." resource))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

;;;;;;;;;;;Input and Variable Initializing Functions;;;;;;;;

(defun open-input-file ()
  (let ((infile (dw:menu-choose (get-data-file-list)
                                :prompt "Data File List")))
    (cond (infile (load (string-append *Resource-File-Directory* infile)
                        :verbose nil)
                  (initialize-frames)
                  (setq *current-file* infile)))))

(defun initialize-frames ()
  (zl:putprop 'list-of nil 'names)
  (loop for frame in *frames*
        as name = (car frame)
        do
     (zl:putprop 'list-of (append (get 'list-of 'names) (list name)) 'names) ))

(defun determine-maximizing-resource ()
 (setq *maximizing-resource-list* (prioritize-resource-list)
       *maximizing-resource-position*
       (loop for resource in *maximizing-resource-list*
             collecting (position resource *resource-variables*))))

(defun reset-lambda-functions ()
  (loop for (resource priority max-val lambda) in *lambda-lists*
        do
    (zl:putprop resource max-val 'resource-limit)
    (zl:putprop resource priority 'resource-priority)
    (zl:putprop resource lambda 'resource-constraint-function)))

(defun initialize-hash-tables ()
  (let ((parameters
          (loop for resource-item-string in *resources*
                as resource = (make-variable-from-string resource-item-string)
                collecting resource into var
                collecting 0 into value
                finally (setq *resource-variables* var)
                        (return (list (append '(*paths* scheduled-items) var)
                                      (append '(nil nil) value))))))
    (loop for resource in (car parameters)
          for val in (cadr parameters)
          do
      (cond ((boundp resource)
             (clrhash (eval resource)))
            (t (set resource (make-hash-table))))
      (swaphash 0 val (eval resource))
      (swaphash *max-time* val (eval resource)))))

(defun initialize-markers-and-variables ()
  (loop for eac in *frames*
        as name = (car eac)
        do
    (loop for each in (cdr eac)
          do
      (zl:putprop name (caadr each) (car each))))
  (setq *time-list* (list 0 *max-time*))
  (initialize-hash-tables)
  (reset-lambda-functions)
  (determine-maximizing-resource))

;;Returns a sorted list based on highest priority resource
;;in form of '(exp1 exp2 exp3 ...)
(defun build-list ()
  (let ((lst (get 'list-of 'names)))
    (loop for resource in (reverse *maximizing-resource-list*)
          as lst2 = (zl:sortcar (loop for exp in lst
                                      collect (list (get exp resource) exp)) #'>)
          do
      (setq lst (loop for each in lst2
                      collecting (cadr each))))
```

```
    lst))

(defun prioritize-resource-list ()
  (sort (remove 0 (copy-list  *resource-variables*) :test #'=
                :key '(lambda (x) (get x 'resource-priority)))
        #'> :key #'(lambda (x) (get x 'resource-priority))))

;;;;;;;;;;;;;;;Top Level Functions;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;MAIN PROGRAM;;;;;;;;;;;;;;;;;

(defun Allocate-Resources ()
  (time (Allocate-Resources-aux)
        (format t "~%~%*** Program Timing ***~%"))


(defun Allocate-Resources-aux ()
  (cond (*second-time* t)
        (t (open-input-file)
           (setq *second-time* t)))
  (initialize-markers-and-variables)
  (examine-data)
  (send *resource-output-window* :clear-history)
  (send *resource-output-window* :select)
  (let ((lst (build-list)))
    (schedule-pass-one lst)
    (display-pass t)
    (show-used)
    (format *resource-output-window* "~%~a"
            (catch 'resource (accept 'label-type :stream *resource-output-window*
                                                                    :prompt nil)))
    (schedule-pass-two lst)
    (display-pass)
    (show-used))
  (format *resource-output-window* "~%~a"
          (catch 'resource (accept 'label-type :stream *resource-output-window*
                                                                  :prompt nil))))
  (zl:readline *resource-output-window*))

;;;;;;;;;;;;;; TOP LEVEL FUNCTIONS ;;;;;;;;;;;;;

(Defun schedule-pass-one (nlst)
  (loop with lst = (copy-list nlst)
        for (start interval-time)=(list 0 *max-time*)
                  then (find-new-parameters start)
        until (or (= start *max-time*)(null lst))
        as group = (find-max-path start (current-status start)
                                  (find-resource-candidates lst interval-time start))
        do
;    (format t "~2~A ~a " group start)
    (cond ((atom (car group)))
          (t
           (update-hash-tables start
                               (loop for item in (car group)
                                     as performances = (get item 'performances)
                                     as duration = (get item 'duration)
                                     as time = (* performances duration)
                                     if (> time interval-time)
                                       do (setq time
                                                (* (setq performances
                                                         (zl:fix (/ interval-time duration)))
                                                   duration))
                                     if (> performances 0)
                                       collect (list item time) into var
                                     finally (return var)
                                     do
                                  (zl:putprop item (+ performances
                                                      (get item 'scheduled-performances))
                                              'scheduled-performances)
                                  (zl:putprop item (- (get item 'performances) performances)
                                              'performances)
                                  (cond ((<= (- (get item 'performances) performances) 0.)
                                         (setq lst (remove-experiment-from-schedule-list
```

```
                                                    iten lst))))))))))

(defun schedule-pass-two (nlst)
    (loop with lst = (copy-list nlst)
          for (start interval-time) = (find-new-parameters)
                    then (find-new-parameters start)
          for current-status = (current-status start)
          until (= start *max-time*)
          as possible-choices = (non-scheduled lst (gethash start scheduled-items))
        do
;      (format t "~3% start = ~A ~20t~a" start current-status)
      (loop with parans = nil
            while interval-time
            while (Parameters-within-range current-status) ;;Need exit condition here
            as group = (find-max-path start current-status
                                      (find-resource-candidates
                                        possible-choices interval-time start))
            do
;       (format t "~2Interval time = ~a ~20t~a~40t~a" interval-time current-status group)
        (cond ((atom (car group))
               (cond ((= (+ start interval-time) *max-time*)
                      (setq interval-time nil))
                     (t
                       (setq parans (find-next-parameter current-status
                                                         (+ start interval-time))
                             possible-choices (remove-next-time-events
                                               (+ start interval-time) possible-choices))
                       (setq current-status (car parans)
                             interval-time (- (cadr parans) start )))))
              (t
                (update-hash-tables start
                                    (loop for iten in (car group)
                                          as duration = (get iten 'duration)
                                          as performances = (zl:fix (/ interval-time duration))
                                          as time = (* performances duration)
                                          collect (list iten time) into var1
                                          minimize time into var2
                                          finally (setq interval-time var2)
                                                  (return var1)
                                          do
                                      (zl:putprop iten (+ performances
                                                          (get iten 'scheduled-performances))
                                                  'scheduled-performances)
                                      (zl:putprop iten (- (get iten 'performances)
                                                          performances)
                                                  'performances)
                                      (setq possible-choices (remove-experiment-from-schedule-list
                                                              iten possible-choices))))
                (setq interval-time nil))))))

(defun complete (self)
  (send self :deactivate))

(defun display-pass (&optional (title nil))
  (dw::with-output-truncation (*resource-output-window* :horizontal t)
    (cond (title
           (format *resource-output-window* "~2%~38t~v<Resource Allocation Results~>~4%"
                   *Font*)
           (cond ((null *resources-output*)
                  (send *display-menu* :set-label "Select Displayed Output")
                  (send *display-menu* :set-iten-list *resources*)
                  (send *display-menu* :choose)
                  (setq *resources-output*
                        (reverse (send *display-menu* :highlighted-values)))))
           (format *resource-output-window* "~4% **** FIRST PASS RESULTS ****~2%"))
          (t
           (format *resource-output-window* "~4% **** SECOND PASS RESULTS ****")))
    (select-graphical-display)
    (let ((x-y-locations (Initialize-Graph-information *graphical-output*))
          (space 10))
      (show-scheduled)
      (loop for resource in *resources-output*
```

```
            initially (space-over *resource-output-window* (+ 6 space))
            do
        (space-over *resource-output-window* space)
        (format *resource-output-window* "~'bt~a~>" resource))
      (loop for time in *time-list*
            for next-time in (cdr *time-list*)
            do
        (setq x-y-locations (display-output-sensitive "~2" time next-time x-y-locations
                                                      :stream *resource-output-window*))
        (loop for variable in (make-variables *resources-output*)
              for header in *resources-output*
              as width = (string-length header)
              for column first (+ space (/ width 2.0) space)
                       then (+ space (/ width 2.0) column)
              do
          (format *resource-output-window* (format nil "~~~at" (zl:fix column)))
          (format *resource-output-window* "~80a" (gethash time (eval variable)))
          (setq column (+ (/ width 2.0) column))))))

(defun display-output-sensitive (return time next-time x-y-locations
                                 &key (stream *resource-menu-window*)
                                      (type 'label-type))
  (dw:with-output-as-presentation (:single-box t
                                   :stream stream
                                   :dont-snapshot-variables t
                                   :type type
                                   :object (list time))
    (print-it stream return time))
;   (print-it *graphics-window* return time))
    (if (and (not (equal *graphical-display* 'none)) x-y-locations)
        (setq x-y-locations (funcall *graphical-display* x-y-locations next-time)))
  x-y-locations)

(defun print-it (stream return time)
  (format stream (format nil "~a~A" return time)))

(defun make-variables (lst)
  (loop for string in lst
        collect (make-variable-from-string string)))

(defun show-used ()
  (format *resource-output-window* "~3I~10TItem~20tRemaining~40tScheduled~I")
  (loop for item  in (get 'list-of 'names)
        do
    (format *resource-output-window* "~2~10T~A~23t~a~43t~a" item (get item 'performances)
            (get item 'scheduled-performances))))

;;;;;;;;;;;;;;; Second Pass Functions ;;;;;;;;;;;;

(defun non-scheduled (lst used)
  (let ((possible lst))
    (loop for item in used
          do
      (setq possible (remove item possible :test #'equal )))
     possible))


;;;;;;;;;;;;;;; Common Pass Functions ;;;;;;;;;;;;


(defun find-new-parameters (&optional (current nil)(parans nil))
  (let ((lst *time-list*))
    (cond ((null current)
           (setq lst (cons 0 lst)))
          (t
            (setq lst (member current *time-list*  :test #'= ))))
    (loop with start = (cadr lst)
          with status = (if parans parans (current-status start))
          for time in (cddr lst)
          while (compare-each-time-status status time)
          finally (return (list start (if time (- time start)
                                       (- *max-time* (cadr lst))))))))
```

```
(defun find-next-parameter (current time)
  (let ((next (mapcar #'(lambda (x y) (if (> x y) x y)) current
                      (current-status time))))
    (list next (cadr (member time *time-list*)))))

(defun remove-next-time-events (time lst)
  (loop for item in (gethash time scheduled-items)
        do
    (setq lst (remove-experiment-from-schedule-list item lst)))
  lst)

(defun compare-each-time-status (status time)
  (loop for pos from 0
        for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (<= (gethash time (eval each))
                   (nth location status))
        finally (return t)))

(defun Parameters-within-range (current-status)
  (loop for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (> (get each 'resource-limit)
                  (nth location current-status))))

(defun update-Hash-tables (start lst)
  (loop for (item1 duration) in lst
        as end-time = (+ start duration)
        do
    (cond ((null (member end-time *time-list* :test #'=))
           (loop for resource in (cons 'scheduled-items *resource-variables*)
                 do
             (swaphash end-time (Get-hash-value end-time resource nil) (eval resource)))
           (setq *time-list* (sort (cons end-time (copy-list *time-list*)) #'<))))
    (loop for time in (member start *time-list*)
          until (= end-time time)
          do
      (swaphash time (append (Gethash time scheduled-items) (list item1))
                scheduled-items)
      (loop for resource in *resource-variables*
            do
        (swaphash time (+ (Get-hash-value time resource)
                          (get item1 resource)) (eval resource))))))

(defun Get-hash-value (time resource &optional (not-new t))
  (let ((value (gethash time (eval resource))))
    (cond (value value)
          (not-new nil)
          (t (gethash (loop with previous = 0
                            for last-time in *time-list*
                            until (>= last-time time)
                            finally (return previous)
                            do
                      (setq previous last-time)) (eval resource))))))

(defun find-resource-candidates (lst endpoint start)
  (loop for exp in (find-interval-candidates lst endpoint)
        if (check-constraints (add-constraint-values (current-status start) exp))
          collect exp into resource-candidate-list
        finally (return resource-candidate-list)))

(defun find-interval-candidates (lst endpoint)
  (loop for exp in lst
        if (feasible-interval exp endpoint)
          collect exp into variable
        finally (return variable)))

(defun feasible-interval (experiment endpoint)
  (< (get experiment 'duration ) endpoint))

(defun find-possible-downward-paths (sv lst)
  (let* ((top (car lst))
         (bottom (cdr lst))
```

```
        (val (add-constraint-values sv top)))
    (cond ((null (check-constraints val)) '(()))
          (bottom
             (loop for down-lst on (cdr lst)
                   append (group-internediate-lists
                             top (find-possible-downward-paths val down-lst)) into var
                   finally (return var)))
          (t (list lst)))))

(defun add-constraint-values (lst exp)
  (loop for resource in *resource-variables*
        for value in lst
        if (null value)
          do (setq value 0)
        collecting (+ value (get exp resource))))

(defun check-constraints (lst)
  (loop for resource in *resource-variables*
        for value in lst
        always (apply (get resource 'resource-constraint-function) (list value))
        finally (return t)))

(defun find-max-path (time sv lst)
  (loop with max-paths = nil
        with max-value = 0
        for new-lst on lst
        as paths = (find-possible-paths sv new-lst)
        as value = (get-time-interval-priority-value (get-group-values (car paths)) sv)
        finally (setq max-paths (sort-max-paths max-paths))
                (swaphash time max-paths *paths*)
                (return (car max-paths))
        do
    (cond ((= max-value value)
           (setq max-paths (append max-paths paths)))
          ((< max-value value) (setq max-paths paths
                                     max-value value)))))

(defun sort-max-paths (paths)
  (let ((lst (loop for path in paths
                   collecting (list path (get-group-values path)))))
    (loop for pos in (reverse *maximizing-resource-position*)
          do
      (setq lst (sort lst #'> :key (lambda (x) (nth pos (cadr x))))))
    lst))

(defun get-time-interval-priority-value (values lst &optional (pos 0))
  (cond (values
         (+ (nth (nth pos *maximizing-resource-position*) values)
            (nth (nth pos *maximizing-resource-position*) lst)))
        (t 0)))

(defun group-internediate-lists (item lst)
  (loop for each in lst
        collect (cons item each)))

(defun remove-experiment-from-schedule-list (exp lst)
  (remove exp (copy-list lst) :test #'equal))

(defun find-possible-paths (val resource-candidates)
  (let ((lst (find-possible-downward-paths val resource-candidates)))
    (cond ((null lst)(return-from find-possible-paths nil))
          (t (get-maximized-sub-path lst)))))


(defun get-maximized-sub-path (paths)
  (loop for resource in *maximizing-resource-list*
        for position in *maximizing-resource-position*
        until (= (length paths) 1)
        do
    (setq paths
          (loop for lst in paths
                with max-val = 0
                with max-lsts = nil
```

```
                as resource-value = (nth position (get-group-values lst))
                finally (return (reverse max-lsts))
                do
            (cond ((> resource-value max-val)
                   (setq max-val resource-value
                         max-lsts (list lst)))
                  ((= resource-value max-val)
                   (setq max-lsts (cons lst max-lsts)))))))
  paths)

(defun get-group-values (group)
  (loop for item in *resource-variables*
        collecting (loop for each in group
                         summing (get each item))))


(defun current-status (time)
  (loop for each in *resource-variables*
        as value = (gethash time (eval each))
        if (null value)
          do (setq value 0)
        collecting value))

(defun show-scheduled ()
  (format *resource-output-window* "~2% Time ~20tScheduled Events~2%")
  (loop for time in *time-list*
        do
    (format *resource-output-window* "~% ~A ~20t~A" time (gethash time scheduled-items)))
  (format *resource-output-window* "~2%"))

(defun show-resource (resource)
  (loop for time in *time-list*
        do
    (format t "~% ~A ~20t~A" time (gethash time resource))))



;(defun make-mouse-sensitive-labels (return object &key (stream *resource-menu-window*)
;                                                       (type 'label-type))
;  (dw:with-output-as-presentation (:single-box t
;                                   :stream stream
;                                   :type type
;                                   :object object)
;    (format stream (format nil "~a~A" return (cadr object)))))
```

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

(defun select-graphical-display ()
  (cond ((null *graphical-display*)
         (let ((choice (dw:menu-choose '("Line Graph" "No Display")
                                       :prompt "Type of Graphical Display"
                                       :center-p t
                                       :mininun-width 275)))
           (setq *graphical-display*
                 (cond ((or (null choice)
                            (string= choice "Line Graph"))
                        'normalized-graphical-display-of-resources)
                       ((string= choice "No Display")
                        'none)
                       (t 'normalized-graphical-display-of-resources)))))
        (t (send *graphics-window* :clear-history)
           (send *graphics-window* :expose)))
  (cond ((equal *graphical-display* 'none) nil)
        (*graphical-output* nil)
        (t (send *display-menu* :set-item-list (max-valued-resources))
           (send *display-menu* :set-label "Select Graphics Output")
           (send *display-menu* :choose)
           (setq *graphical-output*
                 (reverse (send *display-menu* :highlighted-values)))))
  (cond ((and (not (equal *graphical-display* 'none)) *graphical-output*)
         (cond ((send *graphics-window* :exposed-p))
               (t (multiple-value-bind (a b c d)
                      (send *resource-output-window* :edges)
                    (setq *original-screen-size* (list a b c d))
                    (send *resource-output-window* :set-edges a b c (- d 220))
                    (send *graphics-window* :set-edges a (- d 220)c d)
                    (send *graphics-window* :expose))))
         (draw-axis-for-graph))))

(defun max-valued-resources ()
  (loop for variable in *resource-variables*
        for resource in *resources*
        if (get variable 'resource-limit)
          collect resource into var1
        finally (return var1)))

(defun graphical-restart ()
  (cond (*original-screen-size*
         (send *resource-output-window* :set-edges (car *original-screen-size*)
                                                   (cadr *original-screen-size*)
                                                   (caddr *original-screen-size*)
                                                   (cadddr *original-screen-size*))
         (setq *original-screen-size* nil
               *graphical-display* nil
               *graphical-output* nil))))

(defun Initialize-Graph-information (lst)
  (loop for resource-name in lst
        for style in '(nil 2 4 8 12 20 30 50 80)
        with x = 70
        with dy = 1
        as resource = (make-variable-from-string resource-name)
        as max = (get resource 'resource-limit)
        as y = (-  155 (* dy 150 (/ (gethash 0 (eval resource)) max)))
        collecting (list resource-name resource style max x y) into var
        finally (return var)
        counting t into pos
        do
    (show-graph-legend resource-name style (+ 5 (* pos 15)))))

(defun normalized-graphical-display-of-resources (lst time)
  (let ((variable
          (loop with dx = (/ 780 *max-time*)
                with dy = 1.0
                with next-x = (+ 70.0 (* dx time))
                for (resource-name resource style max x y) in lst
                as next-y = (- 155.0 (* 150.0 dy (/ (gethash time (eval resource)) max)))
                collecting (list resource-name resource style max next-x next-y) into var
```

```
                finally (return (cons next-x var))
                do
            (graphics:draw-line x y next-x y :stream *graphics-window*
                                :dashed style :dash-pattern (list style style))
            (graphics:draw-line next-x y next-x next-y :stream *graphics-window*
                                :dashed style :dash-pattern (list style style)))))
  (graphics:draw-line (car variable) 153 (car variable) 157 :stream *graphics-window*)
  (cdr variable)))

(defun draw-axis-for-graph ()
  (graphics:draw-rectangle 70 5 850 155 :filled nil :stream *graphics-window*)
  (send *graphics-window* :set-cursorpos 35 3)
  (format *graphics-window* "100%")
  (send *graphics-window* :set-cursorpos 55 145)
  (format *graphics-window* "0")
  (send *graphics-window* :set-cursorpos 70 158)
  (format *graphics-window* "0")
  (send *graphics-window* :set-cursorpos 830 158)
  (format *graphics-window* "~a" *max-time*)
  (send *graphics-window* :set-cursorpos 442 162)
  (format *graphics-window* "Time"))

(defun show-graph-legend (name style pos)
  (send *graphics-window* :set-cursorpos 860 pos)
  (format *graphics-window* "~a" name)
  (graphics:draw-line 1000 (+ pos 4) 1050 (+ pos 4) :stream *graphics-window*
                      :dashed style :dash-pattern (list style style)))

(define-presentation-type time-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
             (format stream "the selection ~a" (car object))))

(define-presentation-action time-type
   (time-type t
    :gesture :left
    :context-independent t
    :documentation  "Show Additional Information about this Item.")
   (exit)
  (throw 'time exit))
```

Appendix C
Vax Code Listing for the Multiple Pass Single Resource Allocation Program

```lisp
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

;;;Input and Variable Initializing Functions;;;

(defun vax-get-data-file-list ()
  (setq *dir-array* (make-array (length (directory *Resource-File-Directory*))))
  (do* ((dir (directory *Resource-File-Directory*)(cdr dir))
        (path-name (car dir) (car dir))
        (count 0 (1+ count))
        (newpath nil))
       ((null dir) newpath)
    (setq newpath (append newpath (list (file-namestring path-name))))
    (setf (aref *dir-array* count) (file-namestring path-name))))

(defun vax-open-input-file ()
  (format t "~% ~%Data File List ~%~%")
  (let* ((infile)(answ))
  (do* ((infile (vax-get-data-file-list)(cdr infile))
        (file-name (car infile)(car infile))
        (count 0 (1+ count)))
       ((null infile))
     (format t "~% ~A. ~A" count file-name))
  (format t "~% ~%Choice:] ")
  (setq answ (read))
  (setq infile (aref *dir-array* answ))
  (cond (infile (load (string-append *Resource-File-Directory* infile)
                      :verbose nil)
                (vax-initialize-frames)
                (setq *current-file* infile)))))

(defun vax-initialize-frames ()
  (setf (get 'list-of 'names) nil)
  (do* ((flist *frames* (cdr flist))
        (frame (car flist)(car flist))
        (name (car frame) (car frame)))
       ((null flist) (get 'list-of 'names))
    (setf (get 'list-of 'names) (append (get 'list-of 'names) (list name)))))

(defun vax-initialize-markers-and-variables ()
  (do* ((flist *frames* (cdr flist))
        (eac (car flist)(car flist))
        (name (car eac)(car eac)))
       ((null flist) )
    (do* ((elist (cdr eac)(cdr elist))
          (each (car elist)(car elist)))
         ((null elist) )
      (setf (get name (car each)) (cadr each))))
  (setq *energy-list* (list '(0 0) (list *max-time* 0))
        *detailed-energy-list* '((0))))

(defun vax-build-list ()
  (let* ((temp-list nil))
    (do* ((xlist (get 'list-of 'names) (cdr xlist))
          (exp (car xlist)(car xlist)))
         ((null xlist) temp-list)
      (setq temp-list (append temp-list (list (list (get exp 'power-required) exp)))))
    (setq temp-list (sort (copy-alist temp-list) #'> :key #'car))))


;;;;;;;;;;;;;;;;;;;;Top Level Functions;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;MAIN PROGRAM;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun vax-Allocate-Resources ()
  (time (vax-Allocate-Resources-aux)
        (format t "~3%**** Program Timing ****~2%")))

(defun vax-Allocate-Resources-aux ()
  (cond (*second-time* t)
        (t (vax-open-input-file)
           (setq *second-time* t)))
  (vax-initialize-markers-and-variables)
  (let ((lst (vax-build-list)))
    (vax-display-pass-one (vax-schedule-pass-one lst))
    (vax-display-pass-two)
    (vax-show-used)
    (vax-schedule-pass-two lst)
    (vax-display-pass-two t)
    (vax-show-used)))

;;;;;;;;;;;;;;;;;;; TOP LEVEL FUNCTIONS ;;;;;;;;;;;;;;;;;;;;;

(defun vax-schedule-pass-one (nlist)
  (let* ((ti)(parameters)(lst (copy-list nlist))(variable '())(performances)(duration)(time)(var)(start 0)(e(0.0))
    (do* ((interval-time (- *max-time* start) (- *max-time* start))
```

ORIGINAL PAGE IS OF POOR QUALITY

```
        (nax-energy (- $nax-energy$ sv) (- $nax-energy$ sv))
        (group (vax-find-nax-path sv (vax-find-resource-candidates nax-energy lst interval-time) $nax-energy$)
               (vax-find-nax-path sv (vax-find-resource-candidates nax-energy lst interval-time) $nax-energy$))
        (variable (setq variable (append variable (list (cons start group))))
                  (setq variable (append variable (list (cons start group))))))
      ((or (= start $nax-time$)(null lst)(null group)) variable)
    (setq var nil)
    (cond ((atom group))
          (t (setq ti (vax-regroup-ti
                       (dos ((group-list group (cdr group-list))
                             (iten (car group-list)(car group-list)))
                            ((nunberp iten) (return var))
                         (setq performances (get iten 'performances))
                         (setq duration (get iten 'duration))
                         (setq time (* performances duration))
                         (if (> time interval-time)
                             (setq time (* (setq performances (floor (/ interval-time duration))) duration)))
                         (if (> performances 0)
                             (setq var (append var (list (list iten time performances (get iten 'power-required))))))
                         (setf (get iten 'scheduled-performances)(+ performances (get iten 'scheduled-performances)))
                         (setf (get iten 'performances)(- (get iten 'performances) performances))
                         (cond ((<= (- (get iten 'performances) performances) 0.)
                                (setq lst (vax-remove-experiment-from-schedule-list iten lst))))))
             (vax-update-energy-list start ti)))
    (setq parameters (vax-find-new-parameters start)
          start (car parameters)
          sv (cadr parameters)))))


(defun vax-schedule-pass-two (nlst)
  (lets ((lst (copy-list nlst)) (eiten (car $energy-lists$)) (energy-list $energy-lists$) (detailed-list $detailed-energy-list
s)
        (duration)(time)(var)(possible-choices)(temp)(interval-time)(next-energy))
    (dos ((test))
         ((null eiten) (return))
      (lets ((group '((0)))(ti nil)(energy (cadar energy-list)))
      (dos ((test))
           ((null group))
        (if (nunberp (car group))
            (setq energy next-energy))
        (setq possible-choices (vax-non-scheduled lst (cdar detailed-list)))
        (setq temp (vax-get-pass-two-time-interval energy energy-list))
        (setq interval-time (car temp))
        (setq next-energy (cadr temp))
        (setq group (vax-find-nax-path energy (vax-find-resource-candidates
                                               (- $nax-energy$ energy)
                                               possible-choices interval-time) $nax-energy$))
        (cond ((and (nunberp (car group))(<= next-energy energy))
               (return))
              ((nunberp group))
              (t
               (setq energy (+ energy (car (last group)))
                     ti (vax-regroup-ti
                         (let ((var))
                         (dos ((glist group (cdr glist))
                               (iten (car glist)(car glist))
                               (performances nil))
                              ((nunberp iten) (return var))
                           (setq duration (get iten 'duration))
                           (setq time (* (setq performances (floor (/ interval-time duration))) duration))
                           (if (> performances 0)
                               (setq var (append var (list (list iten time performances (get iten 'power-required))))))
                           (setf (get iten 'scheduled-performances)(+ performances (get iten 'scheduled-performances)))
                           (setf (get iten 'performances)(- (get iten 'performances) performances))))))
               (vax-update-energy-list (car eiten) ti)
               (setq energy-list (nenber (car eiten) $energy-lists :test $'= :key $'car)
                     eiten (car energy-list)
                     detailed-list (nenber (cadar detailed-list)
                                           $detailed-energy-lists :test $'= :key $'car))))))
      (setq energy-list (cdr energy-list)
            eiten (car energy-list)
            detailed-list (cdr detailed-list)))))

(defun vax-display-pass-one (lst)
  (format t "~42 $$$$ FIRST PASS RESULTS $$$$~2%")
  (format t "~22~10tTime~20tEnergy~30tExperiment Started~2%")
  (dos ((list lst (cdr list))
        (iten (car list)(car list))
        (time (car iten) (car iten))
        (value (car (last iten)) (car (last iten))))
       ((null iten) )
    (cond ((< 0 value)
           (format t "~2~10t~A~20t~A~30t~A" time value (reverse (cdr (reverse (cdr iten))))))))))

(defun vax-display-pass-two (&optional title)
  (if title (format t "~42 $$$$ SECOND PASS RESULTS $$$$")
```

```
  (format t "~22~5tTime~15tExperiments Currently Being Conducted~60tPower Required~2%")
  (dos ((list #detailed-energy-list# (cdr list))
        (item (car list)(car list))
        (elist #energy-list# (cdr elist))
        (other (car elist)(car elist)))
       ((or (null item)(null other) ))
    (format t "~2~5t~A~15t~A~63t~A" (car item) (cdr item) (cadr other))))


(defun vax-show-used ()
  (format t "~32~10TItem~20tRemaining~40tScheduled~2%")
  (dos ((list (get 'list-of 'nones)(cdr list))
        (item (car list)(car list)))
       ((null item) )
    (format t "~2~10T~A~23t~A~43t~A" item (get item 'performances) (get item 'scheduled-performances))))

;;;;;;;;;;;;;;;;;; SECOND PASS FUNCTIONS ;;;;;;;;;;;;;;;;;;;;;;

(defun vax-non-scheduled (lst used)
  (let ((possible lst))
    (dos ((ilist used (cdr ilist))
          (item (car ilist)(car ilist)))
         ((null ilist))
      (setq possible (remove item possible :test #'equal :key #'cadr)))
    possible))

(defun vax-get-pass-two-time-interval (energy energy-list)
  (let ((start (caar energy-list)))
    (if (= start #max-time#) (return-from vax-get-pass-two-time-interval '(0 0)))
    (dos ((item (cdr energy-list)(cdr item))
          (end (caar item)(caar item))
          (power (cadar item)(cadar item)))
         ((or (null (cdr item))(< energy power)) (return (list (- end start) (cond ((< energy power) power)
                                                                                   (t energy))))))))

;;;;;;;;;;;;;;;;;; Connon Path Functions ;;;;;;;;;;;;;;;;;;;;;

(defun vax-find-new-parameters (current)
   (cadr (member current #energy-list# :test #'= :key #'car)))

(defun vax-regroup-ti (lst)
   (sort (copy-list lst) #'< :key '(lambda (x)(cadr x))))

(defun vax-update-energy-list (start lst)
  (let# ((energy '())(detailed '())(exit t)(time)(power)(old-power)(old-detailed-power)(item1))
    (do ((list lst (cdr list)))
        ((null list))
      (setq item1 (car list)
            old-power nil
            old-detailed-power nil
            time (+ start (cadr item1))
            power (get (car item1) 'power-required))
      (let# ((end-energy (cdr #energy-list#))(end-detailed (cdr #detailed-energy-list#))(etime)(item2)(detailed-item)(exit t
)
             (energy)(detailed))
        (do ((list2 #energy-list# (cdr list2))
             (list3 #detailed-energy-list# (cdr list3))
             (list4 (cdr #energy-list#) (cdr list4))
             (list5 (cdr #detailed-energy-lists) (cdr list5)))
            ((or (null list2) (null list3) (null exit))
             (setq item2 (car list2))
             (setq #energy-list# (append energy end-energy)
                   #detailed-energy-list# (append detailed end-detailed))
             (cond ((not (member time #energy-list# :test #'= :key #'car))
                    (setq #detailed-energy-list#
                          (sort (copy-list (cons (cons time old-detailed-power)
                                                 #detailed-energy-list#)) #'< :key #'car)
                          #energy-list# (sort (copy-list
                                               (cons (list time old-power)
                                                     #energy-list#)) #'< :key #'car)))))
          (setq item2 (car list2)
                detailed-item (car list3)
                etime (car item2))
          (cond ((or (< time etime)(null list4)(null list5))
                 (setq exit nil))
                (t))
          (setq end-energy list4)
          (setq end-detailed list5)
          (setq energy (append energy (cond ((or (= start etime)(< start etime time))
                                             (setq old-power (cadr item2)
                                                   old-detailed-power (cdr detailed-item)
                                                   detailed-item (append detailed-item (list (car item1))))
                                             (list (list time (+ (cadr item2) power))))
                                            (t (list item2)))))
          (setq detailed (append detailed (list detailed-item)))))))
```

```
(defun vax-find-resource-candidates (available-energy lst endpoint)
  (let* ((resource-candidate-list))
  (dos ((list (vax-find-interval-candidates lst endpoint)(cdr list))
        (exp (car list)(car list)))
       ((null list)(return resource-candidate-list))
    (if (<= (car exp) available-energy)
        (setq resource-candidate-list (append resource-candidate-list (list exp)))))))

(defun vax-find-interval-candidates (lst endpoint)
  (let ((variable))
  (dos ((list lst (cdr list))
        (exp (car list)(car list)))
       ((null list) (return variable))
    (if (vax-feasible-interval exp endpoint)
        (setq variable (append variable (list exp)))))))


(defun vax-feasible-interval (experiment endpoint)
  (< (get (cadr experiment) 'duration) endpoint))

(defun vax-find-possible-downward-paths (sv lst max-energy)
  (let ((var))
    (if (null (car lst))(return-from vax-find-possible-downward-paths (list sv)))
    (let ((val (* sv (caar lst)))(top (cadar lst)))
      (cond ((> val max-energy)(return-from vax-find-possible-downward-paths (list (list sv))))
            ((or (= val max-energy)(null (cadr lst)))
             (return-from vax-find-possible-downward-paths (list (list top val)))))
      (dos ((down-lst (cdr lst)(cdr down-lst)))
           ((null (car down-lst)) (return var))
        (setq var (append var (vax-group-intermediate-lists top (vax-find-possible-downward-paths val down-lst max-energy)))
)))))

(defun vax-find-max-path (sv lst max-energy)
  (let ((path))
  (dos ((new-lst lst (cdr new-lst))
        (max-path '(0))
        (path (vax-find-possible-paths sv new-lst max-energy) (vax-find-possible-paths sv new-lst max-energy)))
       ((null new-lst)(return max-path))
    (if (> (car (last path)) (car (last max-path))) (setq max-path path))
    (if (= (car (last max-path)) max-energy) (return max-path)))))

(defun vax-group-intermediate-lists (item lst)
  (let ((newlist nil))
  (dos ((list lst (cdr list))
        (each (car list)(car list)))
       ((null list) newlist)
    (setq newlist (append newlist (list (cons item each)))))))

(defun vax-remove-experiment-from-schedule-list (exp lst)
  (remove exp (copy-list lst) :test #'equal :key #'cadr))

(defun vax-find-possible-paths (val resource-candidates max-energy)
  (let ((lst (vax-find-possible-downward-paths val resource-candidates max-energy)))
    (cond ((atom (car lst))(return-from vax-find-possible-paths nil))
          (t
           (dos ((list lst (cdr list))
                 (item (car list)(car list))
                 (max (car (last (car (sort (copy-list lst) #'> :key '(lambda (x) (car (last x)))))))
                      (car (last (car (sort (copy-list lst) #'> :key '(lambda (x) (car (last x))))))))
                ((= (car (last item)) max)(return item)))))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

(setq *frames* '((alf (experiment-number (1))
                      (power-required (10000.0))
                      (duration (22))
                      (performances (2))
                      (scheduled-performances (0)))
                 (asf (experiment-number (2))
                      (power-required (8500.0))
                      (duration (18))
                      (performances (2))
                      (scheduled-performances (0)))
                 (amf (experiment-number (3))
                      (power-required (1566.7))
                      (duration (18))
                      (performances (3))
                      (scheduled-performances (0)))
                 (aff (experiment-number (4))
                      (power-required (15000.0))
                      (duration (32))
                      (performances (10))
                      (scheduled-performances (0)))
                 (bif (experiment-number (5))
                      (power-required (480.0))
                      (duration (192))
                      (performances (1))
                      (scheduled-performances (0)))
                 (bsf (experiment-number (7))
                      (power-required (5125.0))
                      (duration (48))
                      (performances (1))
                      (scheduled-performances (0)))
                 (cfef (experiment-number (9))
                       (power-required (4000.0))
                       (duration (149))
                       (performances (5))
                       (scheduled-performances (0)))
                 (cppf (experiment-number (10))
                       (power-required (500.0))
                       (duration (274))
                       (performances (1))
                       (scheduled-performances (0)))
                 (dssf (experiment-number (11))
                       (power-required (500.0))
                       (duration (10))
                       (performances (20))
                       (scheduled-performances (0)))
                 (eef (experiment-number (12))
                      (power-required (15000.0))
                      (duration (257))
                      (performances (1))
                      (scheduled-performances (0)))
                 (elf (experiment-number (13))
                      (power-required (725.0))
                      (duration (7))
                      (performances (5))
                      (scheduled-performances (0)))
                 (emlf (experiment-number (14))
                       (power-required (1725.0))
                       (duration (7))
                       (performances (5))
                       (scheduled-performances (0)))
                 (fzf (experiment-number (15))
                      (power-required (8836.7))
                      (duration (34))
                      (performances (5))
                      (scheduled-performances (0)))
                 (fpf (experiment-number (15))
                      (power-required (2080.0))
                      (duration (32))
                      (performances (1))
                      (scheduled-performances (0)))
                 (fff (experiment-number (17))
                      (power-required (1108.3))
                      (duration (6))
                      (performances (5))
```

```
      (scheduled-performances (0)))
(htff (experiment-number (18))
      (power-required (8000.0))
      (duration (13))
      (performances (2))
      (scheduled-performances (0)))
(iff (experiment-number (19))
      (power-required (3000.0))
      (duration (11))
      (performances (5))
      (scheduled-performances (0)))
(lrf (experiment-number (20))
      (power-required (1500.0))
      (duration (57))
      (performances (1))
      (scheduled-performances (0)))
(ofpf (experiment-number (22))
      (power-required (5000.0))
      (duration (24))
      (performances (2))
      (scheduled-performances (0)))
(opcgf (experiment-number (23))
      (power-required (1650.0))
      (duration (13))
      (performances (2))
      (scheduled-performances (0)))
(pgcf (experiment-number (24))
      (power-required (620.0))
      (duration (8))
      (performances (20))
      (scheduled-performances (0)))
(pcgf (experiment-number (25))
      (power-required (6000.0))
      (duration (55))
      (performances (1))
      (scheduled-performances (0)))
(rscf (experiment-number (26))
      (power-required (550.0))
      (duration (12))
      (performances (2))
      (scheduled-performances (0)))
(scf (experiment-number (28))
      (power-required (3160.0))
      (duration (34))
      (performances (1))
      (scheduled-performances (0)))
(vcf (experiment-number (29))
      (power-required (12490.0))
      (duration (95))
      (performances (1))
      (scheduled-performances (0)))
(vfsgf (experiment-number (30))
      (power-required (5710.0))
      (duration (12))
      (performances (3))
      (scheduled-performances (0)))
(zaa (experiment-number (31))
      (power-required (750.0))
      (duration (30))
      (performances (2))
      (scheduled-performances (0)))
(zab (experiment-number (32))
      (power-required (1000.0))
      (duration (15))
      (performances (1))
      (scheduled-performances (0)))
(zac (experiment-number (33))
      (power-required (683.0))
      (duration (150))
      (performances (4))
      (scheduled-performances (0)))
(zad (experiment-number (34))
      (power-required (987.0))
      (duration (10))
      (performances (3))
      (scheduled-performances (0)))
```

```
(zae (experiment-number (35))
     (power-required (10000.0))
     (duration (30))
     (performances (2))
     (scheduled-performances (0)))
(zaf (experiment-number (36))
     (power-required (600.0))
     (duration (15))
     (performances (5))
     (scheduled-performances (0)))
(zag (experiment-number (37))
     (power-required (7000.0))
     (duration (75))
     (performances (1))
     (scheduled-performances (0)))
(zah (experiment-number (38))
     (power-required (500.0))
     (duration (10))
     (performances (9))
     (scheduled-performances (0)))
(zai (experiment-number (39))
     (power-required (1500.0))
     (duration (11))
     (performances (1))
     (scheduled-performances (0)))
(zaj (experiment-number (40))
     (power-required (2075.0))
     (duration (7))
     (performances (1))
     (scheduled-performances (0)))
(zak (experiment-number (41))
     (power-required (15000.0))
     (duration (250))
     (performances (1))
     (scheduled-performances (0)))
(zal (experiment-number (42))
     (power-required (480.0))
     (duration (190))
     (performances (1))
     (scheduled-performances (0)))
(zam (experiment-number (43))
     (power-required (3000.0))
     (duration (11))
     (performances (5))
     (scheduled-performances (0)))
(zan (experiment-number (44))
     (power-required (8000.0))
     (duration (13))
     (performances (2))
     (scheduled-performances (0)))
(zao (experiment-number (45))
     (power-required (1108.3))
     (duration (6))
     (performances (5))
     (scheduled-performances (0)))
(zap (experiment-number (46))
     (power-required (5125.0))
     (duration (48))
     (performances (1))
     (scheduled-performances (0)))
(zaq (experiment-number (47))
     (power-required (725.0))
     (duration (7))
     (performances (1))
     (scheduled-performances (0)))
(zar (experiment-number (48))
     (power-required (10000.0))
     (duration (22))
     (performances (2))
     (scheduled-performances (0)))
(zas (experiment-number (49))
     (power-required (8500.0))
     (duration (18))
     (performances (2))
     (scheduled-performances (0)))
(zat (experiment-number (50))
```

Appendix D
Symbolics Code Listing for the Multiple Pass-Single Resource Allocation Program

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

;;;;;;;;;;;;;;Global Variables;;;;;;;;;;;;

(defvar *max-energy* 15000)

(defvar *frames*)               ;;Loaded from data file.

(defvar *max-time* 2160)

(defvar *energy-list*)

(defvar *detailed-energy-list* '((0)))

(defvar *second-time* nil)

(defvar *current-file* "")

(defvar *Resource-File-Directory* "andy:>jsr>resource-allocation>data-files>")

(defvar *resources*)

(defvar *resource-variables*)

(defvar *resource-menu-window* (tv:make-window 'dw:dynamic-window
                                               :label "Experiment Data Editor Window"
                                               :blinker-p t))

;(defvar *Data-choices-menu* (tv:make-window 'tv:momentary-menu
;                                           :borders 4
;                                           :label "Alternate Data File List"))

(defvar *message-window* (tv:make-window 'dw:dynamic-window
                                 :blinker-p nil
                                 :edges-from '(300 300 850 400)
                                 :margin-components
                                 '((dw:margin-scroll-bar :visibility :if-needed)
                                   (dw:margin-ragged-borders :thickness 4)
                                   (dw:margin-label
                                     :margin :bottom
                                     :string "Message Window       (Press any key to EXIT)."))))

(defvar *Font* (si:backtranslate-font
                (fed:read-font-from-bfd-file "sys:fonts;tv;40vr.bfd.newest" )))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

;;;;;;;;;;;Input and Variable Initializing Functions;;;;;;;;

(defun open-input-file ()
  (let ((infile (dw:menu-choose (get-data-file-list)
                                :prompt "Data File List")))
    (cond (infile (load (string-append *Resource-File-Directory* infile)
                        :verbose nil)
                  (initialize-frames)
                  (setq *current-file* infile)))))

(defun initialize-frames ()
  (zl:putprop 'list-of nil 'names)
  (loop for frame in *frames*
        as name = (car frame)
        do
     (zl:putprop 'list-of (append (get 'list-of 'names) (list name)) 'names) ))

(defun initialize-markers-and-variables ()
  (loop for eac in *frames*
        as name = (car eac)
        do
    (loop for each in (cdr eac)
          do
      (zl:putprop name (caadr each) (car each))))
  (setq *energy-list* (list '(0 0) (list *max-time* 0))
        *detailed-energy-list* '((0))))

(defun build-list ()
   (zl:sortcar (loop for exp in (get 'list-of 'names)
                     collect (list (get exp 'power-required) exp)) '>))

;;;;;;;;;;;;;;Top Level Functions;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;MAIN PROGRAM;;;;;;;;;;;;;;;;

(defun Allocate-Resources ()
  (time (Allocate-Resources-aux)
        (format t "~3%**** Program Timing ****~2%")))


(defun Allocate-Resources-aux ()
  (cond (*second-time* t)
        (t (open-input-file)
           (setq *second-time* t)))
  (initialize-markers-and-variables)
  (let ((lst (build-list)))
    (display-pass-one (schedule-pass-one lst))
    (display-pass-two)
    (show-used)
    (schedule-pass-two lst)
    (display-pass-two t)
    (show-used)))

;;;;;;;;;;;;; TOP LEVEL FUNCTIONS ;;;;;;;;;;;;

(defun schedule-pass-one (nlst)
  (let ((ti)(parameters)(lst (copy-list nlst)))
    (loop with start = 0
          with  sv = 0.0
          until (or (= start *max-time*)(null lst))
          as interval-time = (- *max-time* start)
          as max-energy = (- *max-energy* sv)
          as group = (find-max-path sv (find-resource-candidates
                                        max-energy lst interval-time) *max-energy*)
          until (null group)
          collecting (cons start group) into variable
          finally (return variable)
            do
     (cond ((atom group))
           (t
       (setq ti (regroup-ti
                  (loop for item in group
                        until (numberp item)
```

```
                    as performances = (get item 'performances)
                    as duration = (get item 'duration)
                    as time = (* performances duration)
                    if (> time interval-time)
                      do (setq time
                               (* (setq performances
                                        (zl:fix (/ interval-time duration))) duration))
                    if (> performances 0)
                      collect (list item time performances
                                    (get item 'power-required)) into var
                    finally (return var)
                    do
                  (zl:putprop item (+ performances (get item 'scheduled-performances))
                           'scheduled-performances)
                  (zl:putprop item (- (get item 'performances) performances) 'performances)
                  (cond ((<= (- (get item 'performances) performances) 0.)
                         (setq lst (remove-experiment-from-schedule-list item lst)))))))
      (update-energy-list start ti)))        ;;Modifies the global variable *energy-list*
      (setq parameters (find-new-parameters start)
            start (car parameters)
            sv (cadr parameters)))))

(defun schedule-pass-two (nlst)
  (let ((lst (copy-list nlst))(eitem (car *energy-list*))(energy-list *energy-list*)
        (detailed-list *detailed-energy-list*))
    (loop
       do
      (cond ((null eitem) (return)))
      (loop with group = '((0))
            with ti = nil
            with energy = (cadar energy-list)
            if (numberp (car group))
              do (setq energy next-energy)
            as possible-choices = (non-scheduled lst (cdar detailed-list))
            as (interval-time next-energy) = (get-pass-two-time-interval energy energy-list)
            do
        (setq group (find-max-path energy (find-resource-candidates
                                           (- *max-energy* energy)
                                           possible-choices interval-time) *max-energy*))
        (cond ((and (numberp (car group))(<= next-energy energy))
               (return))
              ((numberp group))
              (t
               (setq energy (+ energy (car (last group)))
                     ti (regroup-ti
                          (loop with performances = nil
                                for item in group
                                until (numberp item)
                                as duration = (get item 'duration)
                                as time = (* (setq performances
                                                   (zl:fix (/ interval-time duration)))
                                             duration)
                                if (> performances 0)
                                  collect (list item time performances
                                                (get item 'power-required)) into var
                                finally (return var)
                                do
                           (zl:putprop item (+ performances
                                               (get item 'scheduled-performances))
                                       'scheduled-performances)
                           (zl:putprop item (- (get item 'performances) performances)
                                       'performances)
                           )))
               (update-energy-list (car eitem) ti) ;;Modifies the global variable *energy-list*
               (setq energy-list (member (car eitem)
                                         *energy-list* :test #'= :key #'car)
                     eitem (car energy-list)
                     detailed-list (member (caar detailed-list)
                                           *detailed-energy-list* :test #'= :key #'car)))))
      (setq energy-list (cdr energy-list)
            eitem (car energy-list)
            detailed-list (cdr detailed-list)))))

(defun display-pass-one (lst)
  (format t "~4% **** FIRST PASS RESULTS ****~%")
  (format t "~2%~10tTime~20tEnergy~30tExperiment Started~%")
```

```
  (loop for item in lst
        as time = (car item)
        as value = (car (last item))
        do
    (cond ((< 0 value)
           (format t "~%~10t~A~20t~a~30t~a" time value (reverse (cdr (reverse item)))))))))

(defun display-pass-two (&optional title)
  (if title (format t "~4% **** SECOND PASS RESULTS ****"))
  (format t "~2%~5tTime~15tExperiments Currently Being Conducted~60tPower Required~%")
  (loop for item in *detailed-energy-list*
        for other in *energy-list*
        do
    (format t "~%~5t~a~15t~A~63t~A" (car item) (cdr item) (cadr other))))

(defun show-used ()
  (format t "~3%~10TItem~20tRemaining~40tScheduled~%")
  (loop for item  in (get 'list-of 'names)
        do
    (format t "~%~10T~A~23t~a~43t~a" item (get item 'performances)
            (get item 'scheduled-performances))))

;;;;;;;;;;;;;;; Second Pass Functions ;;;;;;;;;;;;

(defun non-scheduled (lst used)
  (let ((possible lst))
    (loop for item in used
          do
      (setq possible (remove item possible :test #'equal :key #'cadr)))
     possible))

(defun get-pass-two-time-interval (energy energy-list)
  (let ((start (caar energy-list)))
    (if (= start *max-time*) (return-from get-pass-two-time-interval '(0 0)))
    (loop for (end power) in (cdr energy-list)
          until (< energy power)
          finally (return (list (- end start) (cond ((< energy power) power
                                                    (t energy)))))))

;;;;;;;;;;;;;;; Common Pass Functions ;;;;;;;;;;;;

(defun find-new-parameters (current)
  (cadr (member current *energy-list* :test #'= :key #'car)))

(defun regroup-ti (lst)
  (sort (copy-list lst) #'< :key '(lambda (x)(cadr x))))

(defun update-energy-list (start lst)
  (loop for item1 in lst
        as old-power = nil
        as old-detailed-power = nil
        as time = (+ start (cadr item1))
        as power = (get (car item1) 'power-required)
        do
    (loop for item2 in *energy-list*
          for detailed-item in *detailed-energy-list*
          as etime = (car item2)
          until (< time etime)
          for end-energy = (cdr *energy-list*) then (cdr end-energy)
          for end-detailed = (cdr *detailed-energy-list*) then (cdr end-detailed)
          collecting (cond ((or (= start etime)(< start etime time))
                            (setq old-power (cadr item2)
                                  old-detailed-power (cdr detailed-item)
                                  detailed-item (append detailed-item (list (cadr item1))))
                            (list etime (+ (cadr item2) power)))
                           (t item2))
            into energy
          collecting detailed-item into detailed
          finally (setq *energy-list* (append energy end-energy)
                        *detailed-energy-list* (append detailed end-detailed)
                  (cond ((not (member time *energy-list* :test #'= :key #'car))
                         (setq *detailed-energy-list*
                               (sort (copy-list (cons time old-detailed-power)
                                                *detailed-energy-list*) #'< :key #'car)
                               *energy-list* (sort (copy-list
                                                    (cons (list time old-power
```

```
                                                    *energy-list*)) #'< :key #'car)))))))

(defun find-resource-candidates (available-energy lst endpoint)
  (loop for exp in (find-interval-candidates  lst endpoint)
        if (<= (car exp) available-energy)
        collect exp into resource-candidate-list
        finally (return resource-candidate-list)))

(defun find-interval-candidates (lst endpoint)
  (loop for exp in lst
        if (feasible-interval exp endpoint)
          collect exp into variable
        finally (return variable)))

(defun feasible-interval (experiment endpoint)
  (<  (get (cadr experiment) 'duration ) endpoint))

(defun find-possible-downward-paths (sv lst max-energy)
  (if (null (car lst))(return-from find-possible-downward-paths (list sv)))
  (let ((val (+ sv (caar lst)))(top (cadar lst)))
    (cond ((> val max-energy)(return-from find-possible-downward-paths (list (list sv))))
          ((or (= val max-energy) (null (cadr lst)))
           (return-from find-possible-downward-paths (list (list top val)))))
    (loop for down-lst = (cdr lst) then (cdr down-lst)
          while (car down-lst)
          append (group-intermediate-lists
                   top (find-possible-downward-paths val down-lst max-energy)) into var
          finally (return var))))

(defun find-max-path (sv lst max-energy)
  (loop with max-path = '(0)
        for new-lst = lst then (cdr new-lst)
        while new-lst
        as path = (find-possible-paths sv new-lst max-energy)
        finally (return max-path)
        do
    (if (> (car (last path)) (car (last max-path))) (setq max-path path))
    (if (= (car (last max-path)) max-energy) (return max-path))))

(defun group-intermediate-lists (item lst)
  (loop for each in lst
        collect (cons item each)))

(defun remove-experiment-from-schedule-list (exp lst)
  (remove exp (copy-list lst) :test #'equal :key #'cadr))

(defun find-possible-paths (val resource-candidates max-energy)
  (let ((lst (find-possible-downward-paths val resource-candidates max-energy)))
    (cond ((atom (car lst))(return-from find-possible-paths nil))
          (t
           (loop with max = (car (last (car (sort (copy-list lst) #'>
                                                  :key '(lambda (x) (car (last x))))))))
                 for item in lst
                 until (= (car (last item)) max)
                 finally (return item))))))
```

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

;;;;;;;;;;;;;;Global Variables;;;;;;;;;;;;

(defflavor selection-menu ()
           (tv:drop-shadow-borders-mixin
            tv:multiple-menu))

(defflavor shadowed-tv-window ()
  (tv:drop-shadow-borders-mixin
    dw:dynamic-window))


(defvar *frames*)              ;;Loaded from data file.

(defvar *max-time*)

(defvar *time-list*)

(defvar *lambda-lists*)

(defvar *paths*)

(defvar *original-screen-size* nil)

(defvar *second-time* nil)

(defvar *current-file* "")

(defvar *Resource-File-Directory* "andy:>jsr>resource-allocation>multiple-data-files>")

(defvar *resources*)

(defvar *resource-variables*)

(defvar *resources-output* nil)

(defvar scheduled-items)

(defvar *maximizing-resource-list*)

(defvar *maximizing-resource-position*)

(defvar *graphical-output* nil)

(defvar *graphical-display* nil)

(defvar *resource-output-window* (tv:make-window 'dw:dynamic-window
                                                 :label "Resource Allocation Window"
                                                 :blinker-p nil))


(defvar *display-menu* (tv:make-window
                         'selection-menu
                         :label "Select Displayed Output"
                         :default-character-style '(:fix :roman :large)
                         :special-choices '(("Selection Complete" :funcall-with-self complete))))

(defvar *resource-menu-window* (tv:make-window 'dw:dynamic-window
                                               :label "Experiment Data Editor Window"
                                               :blinker-p t))

;(defvar *Data-choices-menu* (tv:make-window 'tv:momentary-menu
;                                           :borders 4
;                                           :label "Alternate Data File List"))

(defvar *message-window* (tv:make-window 'dw:dynamic-window
                         ;       :blinker-p nil
                                 :edges-from '(300 300 850 400)
                                 :margin-components
                                 '((dw:margin-scroll-bar :visibility :if-needed)
                                   (dw:margin-ragged-borders :thickness 4)
                                   (dw:margin-label
                                     :margin :bottom
                                     :string "Message Window       (Press any key to EXIT)"))))
```

```
(defvar *graphics-window* (tv:make-window 'tv:dynamic-window
                                          :blinker-p nil
                                          :label "Resource Allocation Graphics Display"))

(defvar *Font* (si:backtranslate-font
                (fed:read-font-from-bfd-file "sys:fonts;tv;40vr.bfd.newest" )))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

;;;;;;;;;;;Input and Variable Initializing Functions;;;;;;;;

(defun open-input-file ()
  (let ((infile (dw:menu-choose (get-data-file-list)
                                :prompt "Data File List")))
    (cond (infile (load (string-append *Resource-File-Directory* infile)
                        :verbose nil)
                  (initialize-frames)
                  (setq *current-file* infile)))))

(defun initialize-frames ()
  (zl:putprop 'list-of nil 'names)
  (loop for frame in *frames*
        as name = (car frame)
        do
    (zl:putprop 'list-of (append (get 'list-of 'names) (list name)) 'names) ))

(defun determine-maximizing-resource ()
 (setq *maximizing-resource-list* (prioritize-resource-list)
       *maximizing-resource-position*
       (loop for resource in *maximizing-resource-list*
             collecting (position resource *resource-variables*))))

(defun reset-lambda-functions ()
  (loop for (resource priority max-val lambda) in *lambda-lists*
        do
    (zl:putprop resource max-val 'resource-limit)
    (zl:putprop resource priority 'resource-priority)
    (zl:putprop resource lambda 'resource-constraint-function)))

(defun initialize-hash-tables ()
  (let ((parameters
         (loop for resource-item-string in *resources*
               as resource = (make-variable-from-string resource-item-string)
               collecting resource into var
               collecting 0 into value
               finally (setq *resource-variables* var)
                       (return (list (append '(*paths* scheduled-items) var)
                                     (append '(nil nil) value))))))
    (loop for resource in (car parameters)
          for val in (cadr parameters)
          do
      (cond ((boundp resource)
             (clrhash (eval resource)))
            (t (set resource (make-hash-table))))
      (swaphash 0 val (eval resource))
      (swaphash *max-time* val (eval resource)))))

(defun initialize-markers-and-variables ()
  (loop for eac in *frames*
        as name = (car eac)
        do
    (loop for each in (cdr eac)
          do
      (zl:putprop name (caadr each) (car each))))
  (setq *time-list* (list 0 *max-time*))
  (initialize-hash-tables)
  (reset-lambda-functions)
  (determine-maximizing-resource))

;;Returns a sorted list based on highest priority resource
;;in form of '(exp1 exp2 exp3 ...)
(defun build-list ()
  (let ((lst (get 'list-of 'names)))
    (loop for resource in (reverse *maximizing-resource-list*)
          as lst2 = (zl:sortcar (loop for exp in lst
                                      collect (list (get exp resource) exp)) #'>)
          do
      (setq lst (loop for each in lst2
                      collecting (cadr each))))
    lst))

(defun prioritize-resource-list ()
```

```
   (sort (remove 0 (copy-list  *resource-variables*) :test #'=
                 :key '(lambda (x) (get x 'resource-priority)))
         #'> :key #'(lambda (x) (get x 'resource-priority))))

;;;;;;;;;;;;;;;Top Level Functions;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;MAIN PROGRAM;;;;;;;;;;;;;;;;

(defun Allocate-Resources ()
  (time (Allocate-Resources-aux)
        (format t "~3%**** Program Timing ****~2%")))


(defun Allocate-Resources-aux ()
  (cond (*second-time* t)
        (t (open-input-file)
           (setq *second-time* t)))
  (initialize-markers-and-variables)
;  (examine-data)
  (send *resource-output-window* :clear-history)
  (send *resource-output-window* :select)
  (let ((lst (build-list)))
    (schedule-pass-one lst)
    (display-pass t)
    (show-used)
    (format *resource-output-window* "~3%~a" (catch 'resource (accept 'label-type :stream *resource-outpu
t-window*
                                                                      :prompt nil)))
    (schedule-pass-two lst)
    (display-pass)
    (show-used))
  ;(send *graphics-window* :select)
  (format *resource-output-window* "~3%~a" (catch 'resource (accept 'label-type :stream *graphics-window*
                                                                    :prompt nil)))
  (zl:readline *resource-output-window*))

;;;;;;;;;;;;;; TOP LEVEL FUNCTIONS ;;;;;;;;;;;;

(Defun schedule-pass-one (nlst)
  (loop with lst = (copy-list nlst)
        for (start interval-time)=(list 0 *max-time*)
                  then (find-new-parameters start)
        until (or (= start *max-time*)(null lst))
        as group = (find-max-path start (current-status start)
                                  (find-resource-candidates lst interval-time start))
        do
;    (format t "~%~A ~a " group start)
    (cond ((atom (car group)))
          (t
           (update-hash-tables start
                               (loop for item in (car group)
                                     as performances = (get item 'performances)
                                     as duration = (get item 'duration)
                                     as time = (* performances duration)
                                     if (> time interval-time)
                                       do (setq time
                                                (* (setq performances
                                                         (zl:fix (/ interval-time duration)))
                                                   duration))
                                     if (> performances 0)
                                       collect (list item time) into var
                                     finally (return var)
                                     do
                                 (zl:putprop item (+ performances
                                                     (get item 'scheduled-performances))
                                             'scheduled-performances)
                                 (zl:putprop item (- (get item 'performances) performances)
                                             'performances)
                                 (cond ((<= (- (get item 'performances) performances) 0.)
                                        (setq lst (remove-experiment-from-schedule-list
                                                   item lst)))))))))))


(defun schedule-pass-two (nlst)
    (loop with lst = (copy-list nlst)
          for (start interval-time) = (find-new-parameters)
```

```
                then (find-new-parameters start)
        for current-status = (current-status start)
        until (= start *max-time*)
        as ; possible-choices = (non-scheduled lst (gethash start scheduled-items))
      do
;     format t "~3% start = ~A ~20t~a" start current-status)
      loop with params = nil
           while interval-time
           while (Parameters-within-range current-status) ;;Need exit condition here
           as group = (find-max-path start current-status
                                     (find-resource-candidates
                                       possible-choices interval-time start))
           do
;     format t "~%Interval time = ~a ~20t~a~40t~a" interval-time current-status group)
      cond ((atom (car group))
            (cond ((= (+ start interval-time) *max-time*)
                   (setq interval-time nil))
                  (t
                    (setq params (find-next-parameter current-status
                                                      (+ start interval-time))
                          possible-choices (remove-next-time-events
                                             (+ start interval-time) possible-choices))
                    (setq current-status (car params)
                          interval-time (- (cadr params) start )))))
           (t
             (update-hash-tables start
                                 (loop for item in (car group)
                                       as duration = (get item 'duration)
                                       as performances = (zl:fix (/ interval-time duration))
                                       as time = (* performances duration)
                                       collect (list item time) into var1
                                       minimize time into var2
                                       finally (setq interval-time var2)
                                               (return var1)
                                       do
                                   (zl:putprop item (+ performances
                                                       (get item 'scheduled-performances))
                                               'scheduled-performances)
                                   (zl:putprop item (- (get item 'performances)
                                                       performances)
                                               'performances)
                                   (setq possible-choices (remove-experiment-from-schedule-list
                                                            item possible-choices))))
             (setq interval-time nil))))))

(defun complicate (self)
  (send self :deactivate))

(defun display-pass (&optional (title nil))
  (dw::with-output-truncation (*resource-output-window* :horizontal t)
    (cond (title
           (format *resource-output-window* "~2%~38t~vBesource Allocation Results~%"
                   *Font*)
           (cond ((null *resources-output*)
                  (send *display-menu* :set-label "Select Displayed Output")
                  (send *display-menu* :set-item-list *resources*)
                  (send *display-menu* :choose)
                  (setq *resources-output*
                        (reverse (send *display-menu* :highlighted-values)))))
           (format *resource-output-window* "~4% **** FIRST PASS RESULTS ****~2%"))
          (t
           (format *resource-output-window* "~4% **** SECOND PASS RESULTS ****")))
    (select-graphical-display)
    (let ((x-y-locations (Initialize-Graph-information *graphical-output*))
          (space 10))
      (show-unscheduled)
      (loop for resource in *resources-output*
            initially (space-over *resource-output-window* (+ 6 space))
            do
        (space-over *resource-output-window* space)
        (format *resource-output-window* "~'b~a~3D resource))
      (loop for time in *time-list*
            for next-time in (cdr *time-list*)
            do
        setq x-y-locations (display-output-sensitive "~%" time next-time x-y-locations
                                                     :stream *resource-output-window*))
```

```
        (loop for variable in (make-variables *resources-output*)
              for header in *resources-output*
              as width = (string-length header)
              for column first (+ space (/ width 2.0) space)
                       then (+ space (/ width 2.0) column)
              do
          (format *resource-output-window* (format nil "~~~at" (zl:fix column)))
          (format *resource-output-window* "~8@a" (gethash time (eval variable)))
          (setq column (+ (/ width 2.0) column))))))

(defun display-output-sensitive (return time next-time x-y-locations &key (stream *resource-menu-window*)
                                                                          (type 'label-type))
  (dw:with-output-as-presentation (:single-box t
                                   :stream stream
                                   :dont-snapshot-variables t
                                   :type type
                                   :object (list time))
    (print-it stream return time)
    (print-it *graphics-window* return time))
;   (if (and (not (equal *graphical-display* 'none)) x-y-locations)
;       (setq x-y-locations (funcall *graphical-display* x-y-locations next-time)))
  x-y-locations)

(defun print-it (stream return time)
  (format stream (format nil "~a~A" return time)))

(defun make-variables (lst)
  (loop for string in lst
        collect (make-variable-from-string string)))

(defun show-used ()
  (format *resource-output-window* "~3%~10TItem~20tRemaining~40tScheduled~%")
  (loop for item  in (get 'list-of 'names)
        do
    (format *resource-output-window* "~%~10T~A~23t~a~43t~a" item (get item 'performances)
            (get item 'scheduled-performances))))

;;;;;;;;;;;;;; Second Pass Functions ;;;;;;;;;;;

(defun non-scheduled (lst used)
  (let ((possible lst))
    (loop for item in used
          do
      (setq possible (remove item possible :test #'equal )))
     possible))


;;;;;;;;;;;;;; Common Pass Functions ;;;;;;;;;;;


(defun find-new-parameters (&optional (current nil)(params nil))
  (let ((lst *time-list*))
    (cond ((null current)
           (setq lst (cons 0 lst)))
          (t
            (setq lst (member current *time-list*  :test #'= ))))
    (loop with start = (cadr lst)
          with status = (if params params (current-status start))
          for time in (cddr lst)
          while (compare-each-time-status status time)
          finally (return (list start (if time (- time start)
                                          (- *max-time* (cadr lst))))))))

(defun find-next-parameter (current time)
  (let ((next (mapcar #'(lambda (x y) (if (> x y) x y)) current
                      (current-status time))))
    (list next (cadr (member time *time-list*)))))

(defun remove-next-time-events (time lst)
  (loop for item in (gethash time scheduled-items)
        do
    (setq lst (remove-experiment-from-schedule-list item lst)))
  lst)

(defun compare-each-time-status (status time)
  (loop for pos from 0
```

```
        for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (<= (gethash time (eval each))
                   (nth location status))
        finally (return t)))

(defun Parameters-within-range (current-status)
  (loop for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (> (get each 'resource-limit)
                  (nth location current-status))))

(defun update-Hash-tables (start lst)
  (loop for (item1 duration) in lst
        as end-time = (+ start duration)
        do
    (cond ((null (member end-time *time-list* :test #'=))
           (loop for resource in (cons 'scheduled-items *resource-variables*)
                 do
             (swaphash end-time (Get-hash-value end-time resource nil) (eval resource)))
           (setq *time-list* (sort (cons end-time (copy-list *time-list*)) #'<))))
    (loop for time in (member start *time-list*)
          until (= end-time time)
          do
      (swaphash time (append (Gethash time scheduled-items) (list item1))
                scheduled-items)
      (loop for resource in *resource-variables*
            do
        (swaphash time (+ (Get-hash-value time resource)
                          (get item1 resource)) (eval resource))))))

(defun Get-hash-value (time resource &optional (not-new t))
  (let ((value (gethash time (eval resource))))
     (cond (value value)
           (not-new nil)
           (t (gethash (loop with previous = 0
                             for last-time in *time-list*
                             until (>= last-time time)
                             finally (return previous)
                             do
                       (setq previous last-time)) (eval resource))))))

(defun find-resource-candidates (lst endpoint start)
  (loop for exp in (find-interval-candidates lst endpoint)
        if (check-constraints (add-constraint-values (current-status start) exp))
          collect exp into resource-candidate-list
        finally (return resource-candidate-list)))

(defun find-interval-candidates (lst endpoint)
  (loop for exp in lst
        if (feasible-interval exp endpoint)
          collect exp into variable
        finally (return variable)))

(defun feasible-interval (experiment endpoint)
  (< (get experiment 'duration ) endpoint))

(defun find-possible-downward-paths (sv lst)
  (let* ((top (car lst))
         (bottom (cdr lst))
         (val (add-constraint-values sv top)))
    (cond ((null (check-constraints val)) '(()))
          (bottom
             (loop for down-lst on (cdr lst)
                   append (group-intermediate-lists
                             top (find-possible-downward-paths val down-lst)) into var
                   finally (return var)))
          (t (list lst)))))

(defun add-constraint-values (lst exp)
  (loop for resource in *resource-variables*
        for value in lst
        if (null value)
          do (setq value 0)
        collecting (+ value (get exp resource))))
```

```
(defun check-constraints (lst)
  (loop for resource in *resource-variables*
        for value in lst
        always (apply (get resource 'resource-constraint-function) (list value))
        finally (return t)))

(defun find-max-path (time sv lst)
  (loop with max-paths = nil
        with max-value = 0
        for new-lst on lst
        as paths = (find-possible-paths sv new-lst)
        as value = (get-time-interval-priority-value (get-group-values (car paths)) sv)
        finally (setq max-paths (sort-max-paths max-paths))
                (swaphash time max-paths *paths*)
                (return (car max-paths))
        do
    (cond ((= max-value value)
           (setq max-paths (append max-paths paths)))
          ((< max-value value) (setq max-paths paths
                                     max-value value)))))

(defun sort-max-paths (paths)
  (let ((lst (loop for path in paths
                   collecting (list path (get-group-values path)))))
    (loop for pos in (reverse *maximizing-resource-position*)
          do
      (setq lst (sort lst #'> :key (lambda (x) (nth pos (cadr x))))))
    lst))

(defun get-time-interval-priority-value (values lst &optional (pos 0))
  (cond (values
         (+ (nth (nth pos *maximizing-resource-position*) values)
            (nth (nth pos *maximizing-resource-position*) lst)))
        (t 0)))

(defun group-intermediate-lists (item lst)
  (loop for each in lst
        collect (cons item each)))

(defun remove-experiment-from-schedule-list (exp lst)
  (remove exp (copy-list lst) :test #'equal))

(defun find-possible-paths (val resource-candidates)
  (let ((lst (find-possible-downward-paths val resource-candidates)))
    (cond ((null lst)(return-from find-possible-paths nil))
          (t (get-maximized-sub-path lst)))))


(defun get-maximized-sub-path (paths)
  (loop for resource in *maximizing-resource-list*
        for position in *maximizing-resource-position*
        until (= (length paths) 1)
        do
    (setq paths
          (loop for lst in paths
                with max-val = 0
                with max-lsts = nil
                as resource-value = (nth position (get-group-values lst))
                finally (return (reverse max-lsts))
                do
            (cond ((> resource-value max-val)
                   (setq max-val resource-value
                         max-lsts (list lst)))
                  ((= resource-value max-val)
                   (setq max-lsts (cons lst max-lsts)))))))
  paths)

(defun get-group-values (group)
  (loop for item in *resource-variables*
        collecting (loop for each in group
                         summing (get each item))))


(defun current-status (time)
  (loop for each in *resource-variables*
        as value = (gethash time (eval each))
```

```
        if (null value)
          do (setq value 0)
        collecting value))

(defun show-scheduled ()
  (format *resource-output-window* "~2% Time ~20tScheduled Events~%")
  (loop for time in *time-list*
        do
    (format *resource-output-window* "~% ~A ~20t~A" time (gethash time scheduled-items)))
  (format *resource-output-window* "~2%"))

(defun show-resource (resource)
  (loop for time in *time-list*
        do
    (format t "~% ~A ~20t~A" time (gethash time resource))))



;(defun make-mouse-sensitive-labels (return object &key  stream *resource-menu-window*)
;                                        (type 'label-type))
;  (dw:with-output-as-presentation (:single-box t
;                                   :stream stream
;                                   :type type
;                                   :object object)
;    (format stream (format nil "~a~A" return (cadr object)))))

(defun make-variables (lst)
  (loop for string in lst
        collect (make-variable-from-string string)))

(defun show-used ()
  (format *resource-output-window* "~3%~10TItem~20tRemaining~40tScheduled~%")
  (loop for item  in (get 'list-of 'names)
        do
    (format *resource-output-window* "~%~10T~A~23t~a~43t~a" item (get item 'performances)
            (get item 'scheduled-performances))))

;;;;;;;;;;;;;;; Second Pass Functions ;;;;;;;;;;;;

(defun non-scheduled (lst used)
  (let ((possible lst))
    (loop for item in used
          do
      (setq possible (remove item possible :test #'equal )))
    possible))


;;;;;;;;;;;;;;; Common Pass Functions ;;;;;;;;;;;;


(defun find-new-parameters (&optional (current nil)(params nil))
  (let ((lst *time-list*))
    (cond ((null current)
           (setq lst (cons 0 lst)))
          (t
            (setq lst (member current *time-list*  :test #'= ))))
    (loop with start = (cadr lst)
          with status = (if params params (current-status start))
          for time in (cddr lst)
          while (compare-each-time-status status time)
          finally (return (list start (if time (- time start)
                                          (- *max-time* (cadr lst)))))))))

(defun find-next-parameter (current time)
  (let ((next (mapcar #'(lambda (x y) (if (> x y) x y)) current
                      (current-status time))))
    (list next (cadr (member time *time-list*)))))

(defun remove-next-time-events (time lst)
  (loop for item in (gethash time scheduled-items)
        do
    (setq lst (remove-experiment-from-schedule-list item lst)))
  lst)

(defun compare-each-time-status (status time)
```

```
  (loop for pos from 0
        for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (<= (gethash time (eval each))
                   (nth location status))
        finally (return t)))

(defun Parameters-within-range (current-status)
  (loop for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (> (get each 'resource-limit)
                  (nth location current-status))))

(defun update-Hash-tables (start lst)
  (loop for (item1 duration) in lst
        as end-time = (+ start duration)
        do
    (cond ((null (member end-time *time-list* :test #'=))
           (loop for resource in (cons 'scheduled-items *resource-variables*)
                 do
             (swaphash end-time (Get-hash-value end-time resource nil) (eval resource)))
           (setq *time-list* (sort (cons end-time (copy-list *time-list*)) #'<))))
    (loop for time in (member start *time-list*)
          until (= end-time time)
          do
      (swaphash time (append (Gethash time scheduled-items) (list item1))
                scheduled-items)
      (loop for resource in *resource-variables*
            do
        (swaphash time (+ (Get-hash-value time resource)
                          (get item1 resource)) (eval resource))))))

(defun Get-hash-value (time resource &optional (not-new t))
  (let ((value (gethash time (eval resource))))
     (cond (value value)
           (not-new nil)
           (t (gethash (loop with previous = 0
                             for last-time in *time-list*
                             until (>= last-time time)
                             finally (return previous)
                             do
                       (setq previous last-time)) (eval resource))))))

(defun find-resource-candidates (lst endpoint start)
  (loop for exp in (find-interval-candidates lst endpoint)
        if (check-constraints (add-constraint-values (current-status start) exp))
          collect exp into resource-candidate-list
        finally (return resource-candidate-list)))

(defun find-interval-candidates (lst endpoint)
  (loop for exp in lst
        if (feasible-interval exp endpoint)
          collect exp into variable
        finally (return variable)))

(defun feasible-interval (experiment endpoint)
  (< (get experiment 'duration ) endpoint))

(defun find-possible-downward-paths (sv lst)
  (let* ((top (car lst))
         (bottom (cdr lst))
         (val (add-constraint-values sv top)))
    (cond ((null (check-constraints val)) '(()))
          (bottom
             (loop for down-lst on (cdr lst)
                   append (group-intermediate-lists
                             top (find-possible-downward-paths val down-lst)) into var
                   finally (return var)))
          (t (list lst)))))

(defun add-constraint-values (lst exp)
  (loop for resource in *resource-variables*
        for value in lst
        if (null value)
          do (setq value 0)
        collecting (+ value (get exp resource))))
```

```
(defun check-constraints (lst)
  (loop for resource in *resource-variables*
        for value in lst
        always (apply (get resource 'resource-constraint-function) (list value))
        finally (return t)))

(defun find-max-path (time sv lst)
  (loop with max-paths = nil
        with max-value = 0
        for new-lst on lst
        as paths = (find-possible-paths sv new-lst)
        as value = (get-time-interval-priority-value (get-group-values (car paths)) sv)
        finally (setq max-paths (sort-max-paths max-paths))
                (swaphash time max-paths *paths*)
                (return (car max-paths))
        do
    (cond ((= max-value value)
           (setq max-paths (append max-paths paths)))
          ((< max-value value) (setq max-paths paths
                                     max-value value)))))

(defun sort-max-paths (paths)
  (let ((lst (loop for path in paths
                   collecting (list path (get-group-values path)))))
    (loop for pos in (reverse *maximizing-resource-position*)
          do
      (setq lst (sort lst #'> :key (lambda (x) (nth pos (cadr x))))))
    lst))

(defun get-time-interval-priority-value (values lst &optional (pos 0))
  (cond (values
         (+ (nth (nth pos *maximizing-resource-position*) values)
            (nth (nth pos *maximizing-resource-position*) lst)))
        (t 0)))

(defun group-intermediate-lists (item lst)
  (loop for each in lst
        collect (cons item each)))

(defun remove-experiment-from-schedule-list (exp lst)
  (remove exp (copy-list lst) :test #'equal))

(defun find-possible-paths (val resource-candidates)
  (let ((lst (find-possible-downward-paths val resource-candidates)))
    (cond ((null lst)(return-from find-possible-paths nil))
          (t (get-maximized-sub-path lst)))))


(defun get-maximized-sub-path (paths)
  (loop for resource in *maximizing-resource-list*
        for position in *maximizing-resource-position*
        until (= (length paths) 1)
        do
    (setq paths
          (loop for lst in paths
                with max-val = 0
                with max-lsts = nil
                as resource-value = (nth position (get-group-values lst))
                finally (return (reverse max-lsts))
                do
            (cond ((> resource-value max-val)
                   (setq max-val resource-value
                         max-lsts (list lst)))
                  ((= resource-value max-val)
                   (setq max-lsts (cons lst max-lsts)))))))
  paths)

(defun get-group-values (group)
  (loop for item in *resource-variables*
        collecting (loop for each in group
                         summing (get each item))))


(defun current-status (time)
  (loop for each in *resource-variables*
        as value = (gethash time (eval each))
```

```
        if (null value)
          do (setq value 0)
        collecting value))

(defun show-scheduled ()
  (format *resource-output-window* "~2% Time ~20tScheduled Events~%")
  (loop for time in *time-list*
        do
    (format *resource-output-window* "~% ~A ~20t~A" time (gethash time scheduled-items)))
  (format *resource-output-window* "~2%"))

(defun show-resource (resource)
  (loop for time in *time-list*
        do
    (format t "~% ~A ~20t~A" time (gethash time resource))))
```

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;Presentation types and actions for mouse sensitivity.;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;This defines the label presentation types.
(define-presentation-type label-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
             (format stream "the selection ~a" (car object))))

;;This is what is done when a column or row label is selected.
(define-presentation-action label-type
   (label-type t
    :gesture :left
    :context-independent t
    :documentation  "Resource Operations")
   (exit)
  (throw 'resource exit))

;;This defines the label presentation types.
(define-presentation-type exp-label-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
             (format stream "the selection ~a" (car object))))

;;This is what is done when a column or row label is selected.
(define-presentation-action exp-label-type
   (exp-label-type t
    :gesture :left
    :context-independent t
    :documentation  "Experiment Operations")
   (exit)
  (throw 'resource exit))


;;This defines the item presentation type and documentation line display
(define-presentation-type resource-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
             (format stream "the resource ~A" (car object))))

;;This is what is done when the item is selected
(define-presentation-action choose-type
   (resource-type t
    :gesture :left
    :context-independent t
    :documentation  "Change this value")
   (resource)
  (throw 'resource
    (list resource (get (caar resource)
                        (read-from-string (format nil "~a-presentation" (caar resource)))))))

;;This defines the item presentation type and documentation line display
(define-presentation-type control-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
             (format stream "the selection ~a" (car object))))

;;This is what is done when a command is selected
(define-presentation-action control-type
   (control-type t
    :gesture :left
    :context-independent t
    :documentation  "Execute this Command")
   (exit)
  (throw 'resource (read-from-string exit)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;                          Program functions                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;This is the Driving Function for the Data Editor.
(defun examine-data ()
  (send *resource-menu-window* :select)
  (dw::with-output-truncation (*resource-menu-window* :horizontal t)
    (loop with again = t
          while again
          do
      (make-window-layout)
      (send *resource-menu-window* :set-cursor-visibility nil)
      (setq again
            (loop with finished = nil
                  until finished
                  as choice = (change-data-point)
                  while choice
                  do
              (cond ((atom choice)
                     (case choice
                       (load
                         (open-input-file)
                         (initialize-markers-and-variables)
                         (return t))
                       (save (save-new-file))
                       (exit (return nil))))
                    (t (case (car choice)
                         (exp
                           (take-experiment-action
                             (cadr choice)
                             (get-option-list (format nil "For Experiment ~'b@a~D"
                                                      (cadr choice))
                                              '("Move this Experiment"
                                                "Delete this Experiment"
                                                "Add an Experiment ABOVE"
                                                "Add an Experiment BELOW")))
                           (return t))
                         (resource
                           (take-resource-action
                             (cadr choice)(caddr choice)
                             (get-option-list (format nil "For Resource ~'b@a~D"
                                                      (cadr choice))
                                              (cond ((member (cadr choice)
                                                             '("Duration" "Performances")
                                                             :test #'string-equal)
                                                     '("Set Value Globally"
                                                       "Set Maximum Value"
                                                       "Move this Resource"
                                                       "Add Resource to the LEFT"
                                                       "Add Resource to the RIGHT"
                                                       "Edit Resource Constraints"))
                                                    (t
                                                     '("Set Value Globally"
                                                       "Set Maximum Value"
                                                       "Move this Resource"
                                                       "Delete this Resource"
                                                       "Add Resource to the LEFT"
                                                       "Add Resource to the RIGHT"
                                                       "Edit Resource Constraints"))))))
                           (return t))))))))
    (send *terminal-io* :select))

;;
(defun get-option-list (prompt options)
  (dw:menu-choose options
                  :prompt prompt
                  :center-p t
                  :row-wise nil))

;;
(defun take-resource-action (resource pos action)
  (cond ((string-equal action "Set Value Globally")
         (let ((value (get-stream '((number :prompt "Global Value"
                                            :default 0
                                            :query-identifier jsr))
```

```
          (if value
              (initialize-experiment-resource-value
                (make-variable-from-string resource ) value))))
        ((string-equal action "Set Maximum Value")
         (zl:putprop resource (get-stream '((number :prompt "Maximum Value"
                                                    :default ,(get resource 'resource-limit)
                                                    :query-identifier jsr))
                                          (format nil "Set ~'bA~%Maximum Value      "
                                                  (make-variable-from-string resource )))
                     'resource-limit))
        ((string-equal action "Edit Resource Constraints")
         (modify-resource-constraint-equations (make-variable-from-string resource)))
        ((string-equal action "Move this Resource")
         (send-message-to-user (format nil "~2%      Use mouse to SELECT which RESOURCE to~
                    ~%      place ~'bA~%beside." resource))
         (remove-resource resource nil)
         (let ((position (find-position 'label-type resource)))
           (setq *resources* (insert-item-in-list *resources* resource position)
                 *resource-variables* (insert-item-in-list *resource-variables*
                                        (make-variable-from-string resource) position))))
        ((string-equal action "Delete this Resource")
         (remove-resource resource))
        ((string-equal action "Add Resource to the LEFT")
         (add-resource pos))
        ((string-equal action "Add Resource to the RIGHT")
         (add-resource (+ 1 pos)))))

(defun modify-resource-constraint-equations (resource)
  (send *message-window* :set-margin-components
        '((dw:margin-scroll-bar :visibility :if-needed)
                                (dw:margin-ragged-borders :thickness 4)
                                (dw:margin-label
                                  :margin :bottom
                                  :string "Constraint Editor Window       (Press <END> key to EXI
T)")))
  (send *message-window* :clear-history)
  (send *message-window* :select)
  (format *message-window* "~2%")
  (send *message-window* :set-cursor-visibility :blink)
  (edit-constraint-equation resource)
  (send *message-window* :deselect)
  (send *message-window* :set-cursor-visibility nil)
  (send *message-window* :set-margin-components
        '((dw:margin-scroll-bar :visibility :if-needed)
                                (dw:margin-ragged-borders :thickness 4)
                                (dw:margin-label
                                  :margin :bottom
                                  :string "Message Window       (Press any key to EXIT)"))))


(defun edit-constraint-equation (resource)
  (let ((buffer (tv:kbd-get-io-buffer))
        (equation (format nil "~a" (get resource 'resource-constraint-function))))
    (send *message-window* :clear-input)
    (loop for i from 0 to (- (length equation) 1)
          do
      (tv:io-buffer-put buffer (char equation i)))
    (zl:putprop resource (read-from-string (accept 'string  :stream *message-window*
                           :activation-chars '(#\end)
                           :prompt nil)) 'resource-constraint-function)))

(defun find-position (type resource)
  (let ((position)
        (data (catch 'resource (accept type
                                       :prompt nil
                                       :stream *resource-menu-window*))))
    (case (car data)
      (exp
        (setq position (position (cadr data)(get 'list-of 'names)))
        (case (read-from-string
                (get-option-list (format nil "Place ~'bA~% resource)
                                 (list (format nil "Above ~'bA~% (cadr data))
                                       (format nil "Below ~'bA~% (cadr data)))))
          (ABOVE (+ 1 position))
          (t (+ 2 position))))
```

```
        (setq position (position (cadr data  *resources* :test #'string-equal))
         (case
           (read-from-string
             (get-option-list (format nil "Place ~'bA~D resource)
                              (list (format nil  "Left of ~'bA~D (cadr data))
                                    (format nil  "Right of ~'bA~D (cadr data)))))
                 (LEFT (+ 1 position))
                 (t (+ 2 position)))))))

;;
(defun take-experiment-action (exp action)
  (cond ((string-equal action "Move this Experiment"
         (send-message-to-user (format nil "~2%       Use mouse to SELECT which EXPERIMENT to~
                        ~%     place ~'bA~D here." exp))
         (remove-experiment exp nil)
         (let ((position (find-position 'exp-label-type exp)))
           (zl:putprop 'list-of (insert-item-in-list (get 'list-of 'names)
                                                     exp  position) 'names)))
        ((string-equal action "Delete this Experiment" )
         (remove-experiment exp t))
        ((string-equal action "Add an Experiment ABOVE")
         (add-experiment (+ 1 (position exp (get 'list-of 'names)))))
        ((string-equal action "Add an Experiment BELOW")
         (add-experiment (+ 2 (position exp (get 'list-of 'names)))))))

(defun remove-experiment (exp message)
  (zl:putprop 'list-of (remove exp (get 'list-of 'names)) 'names)
  (if message
      (send-message-to-user
        (format nil "~2%~5tThe EXPERIMENT named  ~'bA~D has been deleted." exp))))

(defun add-experiment (position)
  (let ((variable (make-variable-from-string
                    (get-stream '((string :prompt "Enter EXPERIMENT NAME"
                                          :query-identifier jsr))
                                "Add Experiment Utility                    "))))
    (zl:putprop 'list-of (insert-item-in-list  (get 'list-of 'names) variable position) 'names)
    (loop for item in *resource-variables*
          do
      (zl:putprop variable 0 item))))

;;This function is the top level controller for the input window.
(defun make-window-layout ()
  (send *resource-menu-window* :clear-history)
  (format *resource-menu-window* "~2%~40t~@Experiment Data Editor~34%" *Font*)
  (let* ((space 10))
    (setq *resource-variables* (loop for resource in *resources*
                                     initially  (space-over *resource-menu-window*
                                                            (+ 6 space))
                                     collect  (make-variable-from-string resource) into var
                                     counting t  into place
                                     finally  (return var)
                                     do
                                  (space-over *resource-menu-window* space)
                                  (make-mouse-sensitive-labels ""
                                    (list 'resource resource place))))
    (format *resource-menu-window* "~%")
    (loop for exp in (get 'list-of 'names)
          counting t into place
          do
      (make-mouse-sensitive-labels "~%"
        (list 'exp exp place))
      (loop for variable in *resource-variables*
            for header in *resources*
            as width = (string-length header)
            for column first (+ space (/ width 2.0)  space)
                     then (+ space (/ width 2.0)  column)
            do
        (place-variable column variable exp)
        (setq column (+ (/ width 2.0) column)))
    (place-commands)))

;;This command puts the column and row  labels as presentations .
(defun make-mouse-sensitive-labels (return-object-key (stream *resource-menu-window*)
                                    &rest  label-type))
```

```
                                    :stream stream
                                    :type type
                                    :object object)
    (format stream (format nil "~a~A" return (cadr object)))))

;;This command creats the commands at bottom of menu.
(defun place-commands ()
  (format *resource-menu-window* "~6%")
  (loop for command in '("Exit Data Editor" "Save Current Data to File"
                         "Load New Data File")
        do
    (space-over *resource-menu-window* 17)
    (dw:with-output-as-presentation (:single-box t
                                     :stream *resource-menu-window*
                                     :type 'control-type
                                     :object command)
      (surrounding-output-with-border (*resource-menu-window* :shape :oval
                                                              :filled t
                                                              :move-cursor nil)
        (format *resource-menu-window* command)))))

;;This function assists in proper relative heading column spacing
(defun space-over (stream space)
  (format stream (format nil "~~~Aa" space) ""))

;;This function takes a string and returns an atom.
(defun make-variable-from-string (str)
  (loop with flag = 1
        for item being the array-elements in str
        if (not (string-equal item " "))
          collect item into var
          and do
          (setq flag 0)
        else if (= flag 0)
          collect "-" into var
          and do
          (setq flag 1)
        finally  (return (read-from-string
                           (apply #'string-append
                                  (cond ((= flag 1)
                                         (reverse (cdr (reverse var))))
                                        (t var)))))))

;;This function assists in correct column spacing
(defun place-variable (column variable exp)
  (format *resource-menu-window* (format nil "~~~at" (zl:fix column)))
  (format-item-mouse-sensitive *resource-menu-window* (get exp variable)
                               (list (list exp variable)
                                     (multiple-value-bind (a b)
                                         (send *resource-menu-window* :read-cursorpos)
                                       (list a b)))))


;;This function prints the item to the screen with mouse sensitivity
(defun format-item-mouse-sensitive (stream item descriptors)
    (zl:putprop (caar descriptors) item (cadar descriptors))
    (send stream :set-cursorpos (caadr descriptors) (cadadr descriptors))
    (clearspace stream)
    (zl:putprop (caar descriptors)
                (dw:with-output-as-presentation (:single-box t
                                                 :stream stream
                                                 :type 'resource-type
                                                 :object descriptors)
                  (send stream :set-cursorpos (caadr descriptors) (cadadr descriptors))
                  (format stream "~8@a" item))
                (read-from-string (format nil "~a-presentation" (cadar descriptors)))))

;;This function removes the typed in values to allow for presentations.
(defun clearspace (stream)
  (loop repeat 8
        do
    (send stream :clear-char)
    (send stream :forward-char)))

;;This function reads in a value; but does not issue a line-feed.
```

```
                          &key (activation-characters '(#\Return #\End )))
  (loop with cursor-position = (list (multiple-value-bind (a b)
                                       (send stream :read-cursorpos) (list a b)))
        with var2 = nil
        with position = 0
        as var1 = (send stream :tyi)
        as total-length = (length var2)
        until (member var1 activation-characters)
        if var1
          do
             (cond ((and (equal var1 #\rubout) var2)
                    (send stream :tyo #\backspace)
                    (send stream :clear-char)
                    (setq var2 (cdr var2)
                          position (1-  position)
                          cursor-position (cdr cursor-position)))
                   ((and (or (equal var1 #\c-B)(equal var1 #\backspace)) var2)
                    (setq position (1- position))
                    (send stream :tyo var1))
                   ((equal var1 #\c-F)
                    (cond ((< position total-length)
                           (setq position (1+ position))
                           (send stream :tyo var1))))
                   ((= position total-length)
                    (setq var2 (cons var1 var2)
                          position (1+ position)
                          cursor-position (cons (multiple-value-bind (a b)
                                                  (send stream :read-cursorpos)
                                                  (list a b)) cursor-position))
                    (format stream "~a" var1))
                   ((or (equal var1 #\c-B) (equal var1 #\rubout )))
                   (t (send stream :insert-char)
                      (format stream "~A" var1)
                      (setq var2 (reverse (loop for temp = nil
                                                then (append temp (list (car end)))
                                                for end = (reverse var2) then (cdr end)
                                                repeat position
                                                finally (return
                                                          (append temp (cons var1 end))))))))
        finally (return (cond (var2 (setq var2 (read-from-string
                                                 (apply #'string-append (reverse var2))))))))

;;This function allows the data values to be changed.
(defun change-data-point ()
  (let ((data (catch 'resource (accept '((or resource-type control-type
                                             label-type exp-label-type))
                                       :prompt nil
                                       :stream *resource-menu-window*)))
        (original-position (multiple-value-bind (a b)
                               (send *resource-menu-window* :read-cursorpos)
                             (list a b)))
        (position))
    (cond ((or (atom data) (atom (car data))) data)
          (t
           (setq position (cadar data))
           (send *resource-menu-window* :erase-displayed-presentation (cadr data))
           (send *resource-menu-window* :set-cursorpos (car position)(cadr position))
           (send *resource-menu-window* :set-cursor-visibility :blink)
           (format-item-mouse-sensitive *resource-menu-window*
                                        (read-without-return *resource-menu-window*)
                                        (car data))
           (send *resource-menu-window* :set-cursor-visibility nil)
           (send *resource-menu-window* :set-cursorpos (car original-position)
                 (cadr original-position))
           'data))))

;;This function returns the list of data files that can be selected.
(defun get-data-file-list ()
  (loop for directory in (cdr (fs:directory-list *Resource-File-Directory* ))
        as pathname = (cond ((not (string= (send (car directory) :name) "err"))
                             (format nil "~A" (send (car directory) :string-for-dired))))
        collect  pathname ))

;;This function allows the modified data to be saved to a data file.
(defun save-new-file ()
```

```
                                        (get-stream '((string :prompt "Enter the Filename"
                                                              :query-identifier jsr))
                                                    "Save File Utility                    ")
                                        ".data")
                        :direction :output
                        :if-exists :new-version)
    (format stream "~2%(setq *resources* '(")
    (loop for resource in *resources*
          do
      (format stream " ~a~A~a " #\" resource #\"))
    (format stream ");~2%(setq *frames* '(")
    (loop for exp in (get 'list-of 'names)
          do
      (format stream "~%~a" (cons exp (loop for prop in *resource-variables*
                                            collect (list prop (list (get exp prop)))))))
    (format stream "))")))

;;This function creates a window and prompts the user for a file name.
(defun get-stream (arguments header)
  (dw:accept-values arguments
                    :OWN-WINDOW t
                    :temporary-p nil
                    :prompt header
                    :initially-select-query-identifier 'jsr))

;;This function controls the adding of a resource.
(defun add-resource (position)
  (let* ((new-resource (multiple-value-bind (a b)
                           (get-stream '((string :prompt "Enter RESOURCE NAME"
                                                 :query-identifier jsr)
                                         (number :prompt "Initial Value"
                                                 :default 0))
                                       "Add Resource Utility                    ")
                         (list a b)))
         (variable (make-variable-from-string (car new-resource))))
    (cond ((member variable *resource-variables*)
           (send-message-to-user
             (format nil "~2%~5tThe RESOURCE named ~'bA~a already exists."
                     (car new-resource))))
          (t
           (initialize-experiment-resource-value variable (cadr new-resource))
           (setq *resources* (insert-item-in-list *resources* (car new-resource) position)
                 *resource-variables* (insert-item-in-list *resource-variables*
                                                           variable position))))))

;;This function puts an initial value in the resource variables.
(defun initialize-experiment-resource-value (new-resource value)
  (loop for item in (get 'list-of 'names)
        do
    (zl:putprop  item value new-resource)))

;;This function inserts an item in a list at position.
(defun insert-item-in-list (lst item position)
  (loop for i from 1
        for each on lst
        until (= i position)
        collecting (car each) into var
        finally  (return (append var (list item) each))))

;;This function allows communication between the user and the program.
(defun send-message-to-user (message)
  (send *message-window* :clear-history)
  (send *message-window* :set-cursor-visibility nil)
  (send *message-window* :select)
  (format *message-window* message)
  (send *message-window* :any-tyi)
  (send *message-window* :deselect))

;;This function removes a resource from consideration by program.
(defun remove-resource (resource &optional (message t))
  (setq *resources*  (remove resource *resources* :test #'string-equal)
        *resource-variables* (remove (make-variable-from-string resource)
                                     *resource-variables*))
  (if message
      (send-message-to-user
        (format nil "~2%~5tThe RESOURCE named ~'bA~a has been deleted." resource)))
```

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

(defun select-graphical-display ()
  (cond ((null *graphical-display*)
         (let ((choice (dw:menu-choose '("Line Graph" "No Display")
                                       :prompt "Type of Graphical Display"
                                       :center-p t
                                       :minimum-width 225)))
           (setq *graphical-display*
                 (cond ((or (null choice)
                            (string= choice "Line Graph"))
                        'normalized-graphical-display-of-resources)
                       ((string= choice "No Display")
                        'none)
                       (t 'normalized-graphical-display-of-resources)))))
        (t (send *graphics-window* :clear-history)
           (send *graphics-window* :expose)))
  (cond ((equal *graphical-display* 'none) nil)
        (*graphical-output* nil)
        (t (send *display-menu* :set-item-list (max-valued-resources))
           (send *display-menu* :set-label "Select Graphics Output")
           (send *display-menu* :choose)
           (setq *graphical-output*
                 (reverse (send *display-menu* :highlighted-values)))))
  (cond ((and (not (equal *graphical-display* 'none)) *graphical-output*)
         (cond ((send *graphics-window* :exposed-p))
               (t (multiple-value-bind (a b c d)
                      (send *resource-output-window* :edges)
                    (setq *original-screen-size* (list a b c d))
                    (send *resource-output-window* :set-edges a b c (- d 220))
                    (send *graphics-window* :set-edges a (- d 220)c d)
                    (send *graphics-window* :expose)))
         (draw-axis-for-graph))))

(defun max-valued-resources ()
  (loop for variable in *resource-variables*
        for resource in *resources*
        if (get variable 'resource-limit)
          collect resource into var1
        finally (return var1)))

(defun graphical-restart ()
  (cond (*original-screen-size*
         (send *resource-output-window* :set-edges (car *original-screen-size*)
                                                   (cadr *original-screen-size*)
                                                   (caddr *original-screen-size*)
                                                   (cadddr *original-screen-size*))
         (setq *original-screen-size* nil
               *graphical-display* nil
               *graphical-output* nil))))

(defun Initialize-Graph-information (lst)
  (loop for resource-name in lst
        for style in '(nil 2 4 8 12 20 30 50 80)
        with x = 70
        with dy = 1
        as resource = (make-variable-from-string resource-name)
        as max = (get resource 'resource-limit)
        as y = (-  155 (* dy 150 (/ (gethash 0 (eval resource)) max)))
        collecting (list resource-name resource style max x y) into var
        finally (return var)
        counting t into pos
        do
    (show-graph-legend resource-name style (+ 5 (* pos 15)))))

(defun normalized-graphical-display-of-resources (lst time)
  (let ((variable
          (loop with dx = (/ 780 *max-time*)
                with dy = 1.0
                with next-x = (+ 70.0 (* dx time))
                for (resource-name resource style max x y) in lst
                as next-y = (- 155.0 (* 150.0 dy (/ (gethash time (eval resource)) max)))
                collecting (list resource-name resource style max next-x next-y) into var
                finally (return (cons next-x var))
                do
```

```
                                :dashed style :dash-pattern (list style style))
            (graphics:draw-line next-x y next-x next-y :stream *graphics-window*
                                :dashed style :dash-pattern (list style style)))))
  (graphics:draw-line (car variable) 153 (car variable) 157 :stream *graphics-window*)
  (cdr variable)))

(defun draw-axis-for-graph ()
  (graphics:draw-rectangle 70 5 850 155 :filled nil :stream *graphics-window*)
  (send *graphics-window* :set-cursorpos 35 3)
  (format *graphics-window* "100%")
  (send *graphics-window* :set-cursorpos 55 145)
  (format *graphics-window* "0")
  (send *graphics-window* :set-cursorpos 70 158)
  (format *graphics-window* "0")
  (send *graphics-window* :set-cursorpos 830 158)
  (format *graphics-window* "~a" *max-time*)
  (send *graphics-window* :set-cursorpos 442 162)
  (format *graphics-window* "Time"))

(defun show-graph-legend (name style pos)
  (send *graphics-window* :set-cursorpos 860 pos)
  (format *graphics-window* "~a" name)
  (graphics:draw-line 1000 (+ pos 4) 1050 (+ pos 4) :stream *graphics-window*
                      :dashed style :dash-pattern (list style style)))

(define-presentation-type time-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
             (format stream "the selection ~a" (car object))))

(define-presentation-action time-type
   (time-type t
    :gesture :left
    :context-independent t
    :documentation  "Show Additional Information about this Item.")
   (exit)
  (throw 'time exit))
```

Appendix E
Symbolics Code Listings for Flavor Definitions of Object Structures

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: Lisp -*-


;;;;;; Resource Allocation Flavors ;;;;;;

(defflavor RESOURCE
        ((limit nil)
         (priority nil)
         (constraint-function nil)
         (hash-table nil))
        ()
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)

(defflavor ENVIRONMENT
        ((resources nil)
         (activities nil)
         (total-time nil)
         (expendables nil))
        ()
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)

(defflavor ACTIVITY
        ((duration nil)
         (performances nil)
         (max-performances nil)
         (scheduled-performances nil)
         (Constraint-function nil))
        ()
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)

(defflavor SELECTION-MENU ()
           (tv:drop-shadow-borders-mixin
            tv:multiple-menu))

(defflavor SHADOWED-TV-WINDOW ()
  (tv:drop-shadow-borders-mixin
    dw:dynamic-window))


;;;;;;;;;;;;;;;Special Flavor Functions;;;;;;;;;;;;

(defun revise-flavor-instances (flavor-name instance-variables)
  (let ((current (append (flavor:FLAVOR-ALL-INSTANCE-VARIABLES
                           (flavor:find-flavor flavor-name))))
        (new (mapcar '(lambda (x) (cond ((listp x) (car x))(t x))) instance-variables)))
    (cond ((and (= (length current) (1+ (length instance-variables)))
                (every '(lambda (x) (member x current)) new))
           nil)
          (t
           (flavor:remove-flavor flavor-name)
           (eval `(defflavor ,flavor-name
                             ,(append instance-variables
                                      '(Constraint-function))
                             ()
                     :readable-instance-variables
                     :writable-instance-variables
                     :initable-instance-variables))))))

(defmacro with-modified-flavor-definition (flavor-name instance-variables
                                           flavor-instances &body body)
  `(let ((flavor (flavor:find-flavor ,flavor-name)))
     (revise-flavor-instances ,flavor-name ,instance-variables)
     (loop for each in ,flavor-instances
           do
       (flavor:transform-instance each flavor))
     ,@body))

(defun supply-instance-variables-with-values (variables-and-values instances)
  (cond ((and instances variables-and-values)
         (loop with flavor = (flavor:flavor-name
```

```
                              (flavor::%INSTANCE-FLAVOR
                                (eval (caar variables-and-values))))
                    for (instance value) in variables-and-values
                    as variable = (read-from-string
                                    (format nil "~a-~A"  flavor instance))
                    do
                  (eval `(setf (,variable ,(eval instance)) ,value))))))

;;;;;;;;;;;;;;;Global Variables;;;;;;;;;;;;;

(defvar *activity*)

(defvar *activity-variables* nil)

(defvar *environment*)

(defvar *frames*)               ;;Loaded from data file.

(defvar *max-time*)

(defvar *time-list*)

(defvar *lambda-lists*)

(defvar *paths*)

(defvar *original-screen-size* nil)

(defvar *second-time* nil)

(defvar *current-file* "")

(defvar *Resource-File-Directory* "andy:>jsr>resource-allocation>multiple-data-files>")

(defvar *resources*)

(defvar *resource-variables* nil)

(defvar *resources-output* nil)

(defvar scheduled-items)

(defvar *maximizing-resource-list*)

(defvar *maximizing-resource-position*)

(defvar *graphical-output* nil)

(defvar *graphical-display* nil)

(defvar *resource-output-window* (tv:make-window 'dw:dynamic-window
                                                 :label "Resource Allocation Window"
                                                 :blinker-p nil))

(defvar *display-menu* (tv:make-window
                         'selection-menu
                         :label "Select Displayed Output"
                         :default-character-style '(:fix :roman :large)
                         :special-choices '(("Selection Complete" :funcall-with-self complete))))

(defvar *resource-menu-window* (tv:make-window 'dw:dynamic-window
                                               :label "Experiment Data Editor Window"
                                               :blinker-p t))

;(defvar *Data-choices-menu* (tv:make-window 'tv:momentary-menu
;                                           :borders 4
;                                           :label "Alternate Data File List"))

(defvar *message-window* (tv:make-window 'dw:dynamic-window
                         ;       :blinker-p nil
                                 :edges-from '(300 300 850 400)
                                 :margin-components
                                 '((dw:margin-scroll-bar :visibility :if-needed)
                                   (dw:margin-ragged-borders :thickness 4)
                                   (dw:margin-label
                                     :margin :bottom
                                     :string "Message Window       (Press any key to EXIT)"))))

(defvar *graphics-window* (tv:make-window 'dw:dynamic-window
                                  :blinker-p nil
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

;;;;;;;;;;;;Input and Variable Initializing Functions;;;;;;;;

(defun open-input-file ()
  (let ((infile (dw:menu-choose (get-data-file-list)
                                :prompt "Data File List")))
    (cond (infile (load (string-append *Resource-File-Directory* infile)
                        :verbose nil)
                  (initialize-frames)
                  (setq *current-file* infile)))))

(defun initialize-frames ()
  (loop for frame in *frames*
        collect (car frame) into names
        finally (setf (environment-activities *environment*) names)))

(defun determine-maximizing-resource ()
 (setq *maximizing-resource-list* (prioritize-resource-list)
       *maximizing-resource-position*
       (loop for resource in *maximizing-resource-list*
             collecting (position resource *resource-variables*))))

(defun reset-lambda-functions ()
  (loop for (resource priority max-val lambda) in *lambda-lists*
        do
   (cond ((and (boundp resource)(instancep (eval resource)))
          (setf (resource-limit (eval resource)) max-val)
          (setf (resource-priority (eval resource)) priority)
          (setf (resource-constraint-function (eval resource)) lambda))
         (t
          (set resource (make-instance 'resource
                                       :limit max-val
                                       :priority priority
                                       :constraint-function lambda))))))

(defun initialize-hash-tables ()
  (let ((parameters
          (loop for resource-item-string in *resources*
                as resource = (make-variable-from-string resource-item-string)
                collecting resource into var
                collecting (read-from-string (format nil "activity--a" resource)) into var2
                collecting 0 into value
                finally (setq *resource-variables* var
                              *activity-variables* var2)
                        (return (list (cons 'scheduled-items var)
                                      (append '(nil nil) value))))))
    (loop for resource in (car parameters)
          for val in (cadr parameters)
          do
      (cond ((boundp-in-instance (eval resource) val)
             (clrhash (resource-hash-table (eval resource))))
            (t (setf (resource-hash-table (eval resource))
                     (make-hash-table))))
      (swaphash 0 val (resource-hash-table (eval resource)))
      (swaphash *max-time* val (resource-hash-table (eval resource))))))

;(defun initialize-markers-and-variables ()
;  (loop for eac in *frames*
;        as name = (car eac)
;        do
;    (loop for each in (cdr eac)
;          do
;      (zl:putprop name (caadr each) (car each))))
;  (setq *time-list* (list 0 *max-time*)))

(defun create-object-structures ()
  (define-environmental-structures)
  (loop for eac in *frames*
        as name = (car eac)
        do
    (loop for each in (cdr eac)
          append (list (read-from-string (format nil ":-a" (car each)))
                       (caadr each)) into var-list
          finally (set name (revise-flavor-instances
```

```
                                (make-instance 'activity)
                                               var-list)))
          do
       (zl:putprop name (caadr each) (car each))))
  (setq *time-list* (list 0 *max-time*))
  (initialize-hash-tables)
  (revise-flavor-instances 'activity *resource-variables*)
  (reset-lambda-functions)
  (determine-maximizing-resource))

(defun define-environmental-structures ()
  (if (null *environment*)
      (setq *environment* (make-instance 'environment
                                         :total-time *max-time*))))

;;Returns a sorted list based on highest priority resource
;;in form of '(exp1 exp2 exp3 ...)
(defun build-list ()
  (let ((lst (environment-activities *activity*)))
    (loop for resource in (reverse *maximizing-resource-list*)
          as lst2 = (zl:sortcar (loop for exp in lst
                                      collect (list (funcall resource  exp) exp)) #'>)
          do
       (setq lst (loop for each in lst2
                       collecting (cadr each))))
    lst))

(defun prioritize-resource-list ()
  (sort (remove 0 (copy-list  *resource-variables*) :test #'=
                :key #'resource-priority)
        #'> :key #'resource-priority))

;;;;;;;;;;;;;;;Top Level Functions;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;MAIN PROGRAM;;;;;;;;;;;;;;;;

(defun Allocate-Resources ()
  (time (Allocate-Resources-aux)
        (format t "~3%**** Program Timing ****~2%")))


(defun Allocate-Resources-aux ()
  (cond (*second-time* t)
        (t (open-input-file)
           (setq *second-time* t)))
  (create-object-structures)
  (initialize-markers-and-variables)
  (examine-data)
  (create-object-structures)
  (send *resource-output-window* :clear-history)
  (send *resource-output-window* :select)
  (let ((lst (build-list)))
    (schedule-pass-one lst)
    (display-pass t)
    (show-used)
    (format *resource-output-window* "~3%~a"
            (catch 'resource (accept 'label-type :stream *resource-output-window*
                                                                    :prompt nil)))
    (schedule-pass-two lst)
    (display-pass)
    (show-used))
  ;(send *graphics-window* :select)
  (format *resource-output-window* "~3%~a"
          (catch 'resource (accept 'label-type :stream *graphics-window*
                                                                :prompt nil)))
  (zl:readline *resource-output-window*))

;;;;;;;;;;;;;; TOP LEVEL FUNCTIONS ;;;;;;;;;;;;

(Defun schedule-pass-one (nlst)
  (loop with lst = (copy-list nlst)
        for (start interval-time)=(list 0 *max-time*)
                  then (find-new-parameters start)
        until (or (= start *max-time*)(null lst))
        as group = (find-max-path start (current-status start)
                                  (find-resource-candidates lst interval-time start))
```

```
        do
;     (format t "~%~A ~a " group start)
    (cond ((atom (car group)))
          (t
           (update-hash-tables start
                               (loop for item in (car group)
                                     as performances = (activity-performances item)
                                     as duration = (activity-duration item)
                                     as time = (* performances duration)
                                     if (> time interval-time)
                                       do (setq time
                                                (* (setq performances
                                                         (zl:fix (/ interval-time duration)))
                                                   duration))
                                     if (> performances 0)
                                       collect (list item time) into var
                                     finally (return var)
                                     do
                                  (setf (activity-scheduled-performances item)
                                        (+ performances (activity-scheduled-performances item)))
                                  (setf (activity-performances item)
                                        (- (activity-performances item)))
                                  (cond ((<= (- (activity-performances item) performances) 0.)
                                         (setq lst (remove-experiment-from-schedule-list
                                                    item lst)))))))))


(defun schedule-pass-two (nlst)
    (loop with lst = (copy-list nlst)
          for (start interval-time) = (find-new-parameters)
                    then (find-new-parameters start)
          for current-status = (current-status start)
          until (= start *max-time*)
          as possible-choices = (non-scheduled lst (gethash start scheduled-items))
        do
;       (format t "~3% start = ~A ~20t~a" start current-status)
       (loop with params = nil
             while interval-time
             while (Parameters-within-range current-status) ;;Need exit condition here
             as group = (find-max-path start current-status
                                       (find-resource-candidates
                                        possible-choices interval-time start))
             do
;         (format t "~%Interval time = ~a ~20t~a~40t~a" interval-time current-status group)
          (cond ((atom (car group))
                 (cond ((= (+ start interval-time) *max-time*)
                        (setq interval-time nil))
                       (t
                         (setq params (find-next-parameter current-status
                                                           (+ start interval-time))
                               possible-choices (remove-next-time-events
                                                 (+ start interval-time) possible-choices))
                         (setq current-status (car params)
                               interval-time (- (cadr params) start )))))
                (t
                  (update-hash-tables start
                                      (loop for item in (car group)
                                            as duration = (activity-duration item)
                                            as performances = (zl:fix (/ interval-time duration))
                                            as time = (* performances duration)
                                            collect (list item time) into var1
                                            minimize time into var2
                                            finally (setq interval-time var2)
                                                    (return var1)
                                            do
                                        (setf (activity-scheduled-performances item)
                                              (+ performances (activity-scheduled-performances item)))
                                        (setf (activity-performances item)
                                              (- (activity-performances item) performances))
                                        (setq possible-choices (remove-experiment-from-schedule-list
                                                                item possible-choices))))
                  (setq interval-time nil))))))

(defun complete (self)
  (send self :deactivate))
```

```
(defun display-pass (&optional (title nil))
  (dw::with-output-truncation (*resource-output-window* :horizontal t)
    (cond (title
           (format *resource-output-window* "~2%~38t~vResource Allocation Results~%"
                   *Font*)
           (cond ((null *resources-output*)
                  (send *display-menu* :set-label "Select Displayed Output")
                  (send *display-menu* :set-item-list *resources*)
                  (send *display-menu* :choose)
                  (setq *resources-output*
                        (reverse (send *display-menu* :highlighted-values)))))
           (format *resource-output-window* "~4% **** FIRST PASS RESULTS ****~2%"))
          (t
           (format *resource-output-window* "~4% **** SECOND PASS RESULTS ****")))
    (select-graphical-display)
    (let ((x-y-locations (Initialize-Graph-information *graphical-output*))
          (space 10))
      (show-scheduled)
      (loop for resource in *resources-output*
            initially (space-over *resource-output-window* (+ 6 space))
            do
        (space-over *resource-output-window* space)
        (format *resource-output-window* "~'b@a~%" resource))
      (loop for time in *time-list*
            for next-time in (cdr *time-list*)
            do
        (setq x-y-locations (display-output-sensitive "~%" time next-time x-y-locations
                                                      :stream *resource-output-window*))
        (loop for variable in (make-variables *resources-output*)
              for header in *resources-output*
              as width = (string-length header)
              for column first (+ space (/ width 2.0) space)
                       then (+ space (/ width 2.0) column)
              do
          (format *resource-output-window* (format nil "~~~at" (zl:fix column)))
          (format *resource-output-window* "~8@a" (gethash time (eval variable)))
          (setq column (+ (/ width 2.0) column)))))))

(defun display-output-sensitive (return time next-time x-y-locations &key (stream *resource-menu-window*)
                                 (type 'label-type))
  (dw:with-output-as-presentation (:single-box t
                                   :stream stream
                                   :dont-snapshot-variables t
                                   :type type
                                   :object (list time))
    (print-it stream return time))
;   (print-it *graphics-window* return time))
    (if (and (not (equal *graphical-display* 'none)) x-y-locations)
        (setq x-y-locations (funcall *graphical-display* x-y-locations next-time)))
  x-y-locations)

(defun print-it (stream return time)
  (format stream (format nil "~a~A" return time)))

(defun make-variables (lst)
  (loop for string in lst
        collect (make-variable-from-string string)))

(defun show-used ()
  (format *resource-output-window* "~3%~10TItem~20tRemaining~40tScheduled~%")
  (loop for item in (environment-activities *environment*)
        do
    (format *resource-output-window* "~%~10T~A~23t~a~43t~a" item (activity-performances item)
            (activity-scheduled-performances item))))

;;;;;;;;;;;;;; Second Pass Functions ;;;;;;;;;;;

(defun non-scheduled (lst used)
  (let ((possible lst))
    (loop for item in used
          do
      (setq possible (remove item possible :test #'equal )))
     possible))


;;;;;;;;;;;;;; Common Pass Functions ;;;;;;;;;;;
```

```
(defun find-new-parameters (&optional (current nil)(params nil))
  (let ((lst *time-list*))
    (cond ((null current)
           (setq lst (cons 0 lst)))
          (t
            (setq lst (member current *time-list*  :test #'= ))))
    (loop with start = (cadr lst)
          with status = (if params params (current-status start))
          for time in (cddr lst)
          while (compare-each-time-status status time)
          finally (return (list start (if time (- time start)
                                          (- *max-time* (cadr lst))))))))

(defun find-next-parameter (current time)
  (let ((next (mapcar #'(lambda (x y) (if (> x y) x y)) current
                      (current-status time))))
    (list next (cadr (member time *time-list*)))))

(defun remove-next-time-events (time lst)
  (loop for item in (gethash time scheduled-items)
        do
    (setq lst (remove-experiment-from-schedule-list item lst)))
  lst)

(defun compare-each-time-status (status time)
  (loop for pos from 0
        for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (<= (gethash time (eval each))
                   (nth location status))
        finally (return t)))

(defun Parameters-within-range (current-status)
  (loop for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (> (resource-limit each)
                  (nth location current-status))))

(defun update-Hash-tables (start lst)
  (loop for (item1 duration) in lst
        as end-time = (+ start duration)
        do
    (cond ((null (member end-time *time-list* :test #'=))
           (loop for resource in (cons 'scheduled-items *resource-variables*)
                 do
             (swaphash end-time (Get-hash-value end-time resource nil) (eval resource)))
           (setq *time-list* (sort (cons end-time (copy-list *time-list*)) #'<))))
    (loop for time in (member start *time-list*)
          until (= end-time time)
          do
      (swaphash time (append (Gethash time 'scheduled-items) (list item1))
                scheduled-items)
      (loop for resource in *resource-variables*
            for operation in *activity-variables*
            do
        (swaphash time (+ (Get-hash-value time (resource-hash-table resource))
                          (funcall operation item1)) (resource-hash-table resource) )))))

(defun Get-hash-value (time resource-table &optional (not-new t))
  (let ((value (gethash time resource-table)))
    (cond (value value)
          (not-new nil)
          (t (gethash (loop with previous = 0
                            for last-time in *time-list*
                            until (>= last-time time)
                            finally (return previous)
                            do
                      (setq previous last-time))
                      resource-table)))))

(defun find-resource-candidates (lst endpoint start)
  (loop for exp in (find-interval-candidates lst endpoint)
        if (check-constraints (add-constraint-values (current-status start) exp))
          collect exp into resource-candidate-list
        finally (return resource-candidate-list)))
```

```
(defun find-interval-candidates (lst endpoint)
  (loop for exp in lst
        if (feasible-interval exp endpoint)
          collect exp into variable
        finally (return variable)))

(defun feasible-interval (experiment endpoint)
  (< (get experiment 'duration ) endpoint))

(defun find-possible-downward-paths (sv lst)
  (let* ((top (car lst))
         (bottom (cdr lst))
         (val (add-constraint-values sv top)))
    (cond ((null (check-constraints val)) '(()))
          (bottom
             (loop for down-lst on (cdr lst)
                   append (group-intermediate-lists
                             top (find-possible-downward-paths val down-lst)) into var
                   finally (return var)))
          (t (list lst)))))

(defun add-constraint-values (lst exp)
  (loop for resource in *resource-variables*
        for value in lst
        if (null value)
          do (setq value 0)
        collecting (+ value (get exp resource))))

(defun check-constraints (lst)
  (loop for resource in *resource-variables*
        for value in lst
        always (apply (resource-constraint-function resource) (list value))
        finally (return t)))

(defun find-max-path (time sv lst)
  (loop with max-paths = nil
        with max-value = 0
        for new-lst on lst
        as paths = (find-possible-paths sv new-lst)
        as value = (get-time-interval-priority-value (get-group-values (car paths)) sv)
        finally (setq max-paths (sort-max-paths max-paths))
                (swaphash time max-paths *paths*)
                (return (car max-paths))
        do
    (cond ((= max-value value)
           (setq max-paths (append max-paths paths)))
          ((< max-value value) (setq max-paths paths
                                     max-value value)))))

(defun sort-max-paths (paths)
  (let ((lst (loop for path in paths
                   collecting (list path (get-group-values path)))))
    (loop for pos in (reverse *maximizing-resource-position*)
          do
      (setq lst (sort lst #'> :key (lambda (x) (nth pos (cadr x))))))
    lst))

(defun get-time-interval-priority-value (values lst &optional (pos 0))
  (cond (values
         (+ (nth (nth pos *maximizing-resource-position*) values)
            (nth (nth pos *maximizing-resource-position*) lst)))
        (t 0)))

(defun group-intermediate-lists (item lst)
  (loop for each in lst
        collect (cons item each)))

(defun remove-experiment-from-schedule-list (exp lst)
  (remove exp (copy-list lst) :test #'equal))

(defun find-possible-paths (val resource-candidates)
  (let ((lst (find-possible-downward-paths val resource-candidates)))
    (cond ((null lst)(return-from find-possible-paths nil))
          (t (get-maximized-sub-path lst)))))
```

```
(defun get-maximized-sub-path (paths)
  (loop for resource in *maximizing-resource-list*
        for position in *maximizing-resource-position*
        until (= (length paths) 1)
        do
    (setq paths
          (loop for lst in paths
                with max-val = 0
                with max-lsts = nil
                as resource-value = (nth position (get-group-values lst))
                finally (return (reverse max-lsts))
                do
            (cond ((> resource-value max-val)
                   (setq max-val resource-value
                         max-lsts (list lst)))
                  ((= resource-value max-val)
                   (setq max-lsts (cons lst max-lsts)))))))
  paths)

(defun get-group-values (group)
  (loop for item in *activity-variables*
        collecting (loop for each in group
                         summing (funcall item (eval each)))))


(defun current-status (time)
  (loop for each in *resource-variables*
        as value = (gethash time (resource-hash-table (eval each)))
        if (null value)
          do (setq value 0)
        collecting value))

(defun show-scheduled ()
  (format *resource-output-window* "~2% Time ~20tScheduled Events~%")
  (loop for time in *time-list*
        do
    (format *resource-output-window* "~% ~A ~20t~A" time (gethash time scheduled-items)))
  (format *resource-output-window* "~2%"))

(defun show-resource (resource)
  (loop for time in *time-list*
        do
    (format t "~% ~A ~20t~A" time (gethash time resource))))



;(defun make-mouse-sensitive-labels (return object &key (stream *resource-menu-window*)
;                                                       (type 'label-type))
;  (dw:with-output-as-presentation (:single-box t
;                                   :stream stream
;                                   :type type
;                                   :object object)
;    (format stream (format nil "~a~A" return (cadr object)))))
```

Appendix F
Symbolics Lisp Code for Modified Single Allocation Step Process

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-

(defun open-input-file ()
  (let ((infile (dw:menu-choose (get-data-file-list)
                                :prompt "Data File List")))
    (cond (infile (load (string-append *Resource-File-Directory* infile)
                        :verbose nil)
                  (initialize-frames)
                  (setq *current-file* infile)))))

(defun initialize-frames ()
  (zl:putprop 'list-of nil 'names)
  (loop for frame in *frames*
        as name = (car frame)
        do
     (zl:putprop 'list-of (append (get 'list-of 'names) (list name)) 'names) ))

(defun determine-maximizing-resource ()
 (setq *maximizing-resource-list* (prioritize-resource-list)
       *maximizing-resource-position*
       (loop for resource in *maximizing-resource-list*
             collecting (position resource *resource-variables*))))

(defun reset-lambda-functions ()
  (loop for (resource priority max-val lambda) in *lambda-lists*
        do
    (zl:putprop resource max-val 'resource-limit)
    (zl:putprop resource priority 'resource-priority)
    (zl:putprop resource lambda 'resource-constraint-function)))

(defun initialize-hash-tables ()
  (let ((parameters
          (loop for resource-item-string in *resources*
                as resource = (make-variable-from-string resource-item-string)
                collecting resource into var
                collecting 0 into value
                finally (setq *resource-variables* var)
                        (return (list (append '(*paths* scheduled-items) var)
                                      (append '(nil nil) value))))))
    (loop for resource in (car parameters)
          for val in (cadr parameters)
          do
      (cond ((boundp resource)
             (clrhash (eval resource)))
            (t (set resource (make-hash-table))))
      (swaphash 0 val (eval resource))
      (swaphash *max-time* val (eval resource))))
  (loop for exp in (get 'list-of 'names )
        do
    (zl:putprop exp nil 'when-scheduled)))

(defun initialize-markers-and-variables ()
  (loop for eac in *frames*
        as name = (car eac)
        do
    (loop for each in (cdr eac)
          do
      (zl:putprop name (caadr each) (car each))))
  (setq *time-list* (list 0 *max-time*))
  (initialize-hash-tables)
  (reset-lambda-functions)
  (determine-maximizing-resource))

;;Returns a sorted list based on highest priority resource
;;in form of '(exp1 exp2 exp3 ...)
(defun build-list ()
  (let ((lst (get 'list-of 'names)))
    (loop for resource in (reverse *maximizing-resource-list*)
          as lst2 = (zl:sortcar (loop for exp in lst
                                      collect (list (get exp resource) exp)) #'>)
          do
      (setq lst (loop for each in lst2
                      collecting (cadr each))))
    lst))

(defun Rig-to-subst-gibbys-frontier-nodes-as-minimums ()
```

```
  (with-open-file (stream *Gibbys-frontier-node-file*
                          :if-does-not-exist nil)
    (cond (stream
           (loop for each in (read stream)
                 for value in (read stream)
                 do
             (zl:putprop each value 'performances)))
          (t
           (format t "~3%~vGibby, I need a frontier node!!!~3%" '(:eurex :italic :huge))
           (beep)
           'missing))))

(defun prioritize-resource-list ()
  (sort (remove 0 (copy-list  *resource-variables*) :test #'=
                :key '(lambda (x) (get x 'resource-priority)))
        #'> :key #'(lambda (x) (get x 'resource-priority))))

(defun permanently-store-pass-one-results ()
  (loop for resource in *resource-variables*
        as results = (eval resource)
        do
    (zl:putprop resource results 'pass-one))
  (loop for each  in (get 'list-of 'names)
        do
    (zl:putprop each (get each 'when-scheduled) 'pass-one))
  (setq *Pass-one-time-line* *time-list*))

;;;;;;;;;;;;;;Top Level Functions;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;MAIN PROGRAM;;;;;;;;;;;;;;;;

(defun Allocate-Resources ()
  (time (Allocate-Resources-aux)
        (format t "~3%**** Program Timing ****~2%")))


(defun Allocate-Resources-aux (&key (Gibby nil))
  (cond (*second-time* t)
        (t (open-input-file)
           (setq *second-time* t)))
  (initialize-markers-and-variables)
  (if (and gibby (Rig-to-subst-gibbys-frontier-nodes-as-minimums))
      (return-from Allocate-Resources-aux "Program Terminated Due to File-Not-Found"))
  (examine-data)
  (let ((lst (build-list)))
      (send *resource-output-window* :clear-history)
      (send *resource-output-window* :select)
      (continue-allocation-pass-one lst)
      (permanently-store-pass-one-results)
      (continue-allocation-pass-two lst)))

(defun continue-allocation-pass-one (lst)
  (schedule-pass-one lst)
  (display-pass t)
  (show-used)
  (place-exit-button "Continue to Second Pass")
  (proceed 'continue-allocation-pass-one))

(defun continue-allocation-pass-two (lst)
  (schedule-pass-two lst)
  (display-pass)
  (show-used)
  (place-exit-button "Terminate Program")
  (proceed 'continue-allocation-pass-two))

;;;;;;;;;;;;; Back Tracking Capabilities ;;;;;;;;;;;;;

(defun Proceed (function)
  (let ((response
         (car (catch 'resource (accept 'label-type :stream *resource-output-window*
                                       :prompt nil)))))

    (cond ((numberp response)
           (backtrack function response))
          ((equal response 'proceed)))))
```

```
(defun backtrack (function time-slot)
  (let ((choices (gethash time-slot *paths*)))
    (loop while
          (if (> (length choices) 1)
              (remove-and-restart function time-slot choices)
              (send-message-to-user
                (format nil "The only allocation selection given for ~a is the currently~%allocated gro
up"
                        time-slot))))))

(defun remove-and-restart (func time choices)
  (loop as selection = (get-option-list
                         (format nil "Select Alternate Activity Schedule at Time ~a" time)
                         (append (string-lists (cdr choices))
                                 '("Do Not Change Current Activity Schedule")))
        when selection
          do
            (cond ((listp (read-from-string selection))
                   (reset-data-structures func time choices selection)
                   (funcall func time))
                  (t
                    (return-from remove-and-restart t)))))

(defun reset-data-structures (func time choices selection)
  (let* ((choice (read-from-string selection))
         (common (intersection choice (car choices)))
         (new (intersection common choice :test #'(lambda (x y) (not (eql x y)))))
         (old (intersection common (car choices) :test #'(lambda (x y) (not (eql x y)))))
         (kill-time (cdr (member time *time-list*))))
      (loop for exp in (get 'list-of 'names)
            as scheduled = (get exp 'scheduled-performances)
            as perfs = (get exp 'performances)
            as times = (get exp 'when-scheduled)
            do
        (loop for eac in times
              until (<= eac time)
              counting t into number
              finally
                (zl:putprop exp (subseq times (1- number)) 'when-scheduled)
                (zl:putprop exp (- scheduled number) 'scheduled-performances)
                (zl:putprop exp (+ perfs number) 'performances)))
      (loop for resources in *resource-variables*
            as table = (eval resources)
            do
        (Remove-hash-entries-with-times-greater-than table time))))

(defun Remove-hash-entries-with-times-greater-than (table start-time)
  (maphash '(lambda (time value)
              (if (> time ,start-time)
                  (remhash time ,table)))
           table))

(defun string-lists (lst)
  (mapcar '(lambda (x) (format nil "~a" x)) lst))

(defun Place-exit-button (message)
  (format *resource-output-window* "~2%~20t")
  (dw:with-output-as-presentation (:single-box t
                                   :stream *resource-output-window*
                                   :type 'label-type
                                   :object 'proceed)
      (surrounding-output-with-border (*resource-output-window* :shape :oval
                                                                :filled t
                                                                :move-cursor nil)
        (format *resource-output-window* message))))

;;;;;;;;;;;;; TOP LEVEL FUNCTIONS ;;;;;;;;;;;;

(Defun schedule-pass-one (nlst &key (backtrack-time nil))
  (loop with lst = (copy-list nlst)
        for (start interval-time)= (if backtrack-time
                                       (find-new-parameters backtrack-time)
                                       (list 0 *max-time*))
                 then (find-new-parameters start)
        until (or (= start *max-time*)
                  (null lst))
```

```
        as possible-choices = (non-scheduled lst (gethash start scheduled-items))
        as group = (find-max-path start (current-status start)
                                  (find-resource-candidates
                                        possible-choices interval-time start))
        do
;     (format t "~%~A ~a " group start)
    (cond ((atom (car group)))
          (t
           (update-hash-tables start
                               (loop for item in (car group)
                                     as performances = (get item 'performances)
                                     as time = (get item 'duration)
                                     collect (list item time) into var
                                     finally (return var)
                                     do
                                 (zl:putprop item (cons start (get item 'when-scheduled)) 'when-scheduled
)
                                 (zl:putprop item (+ 1 (get item 'scheduled-performances))
                                             'scheduled-performances)
                                 (zl:putprop item (- performances 1)
                                             'performances)
                                 (cond ((<= performances 1.)
                                        (setq lst (remove-experiment-from-schedule-list
                                                   item lst)))))))))


(defun schedule-pass-two (nlst)
    (loop with lst = (copy-list nlst)
          for (start interval-time) = (find-new-parameters)
                    then (find-new-parameters start)
          for current-status = (current-status start)
          until (= start *max-time*)
          as possible-choices = (non-scheduled lst (gethash start scheduled-items))
        do
;       (format t "~3% start = ~A ~20t~a" start current-status)
      (loop with params = nil
            while interval-time
            while (Parameters-within-range current-status) ;;Need exit condition here
            as group = (find-max-path start current-status
                                      (find-resource-candidates
                                        possible-choices interval-time start))
            do
;         (format t "~%Interval time = ~a ~20t~a~40t~a" interval-time current-status group)
          (cond ((atom (car group))
                 (cond ((= (+ start interval-time) *max-time*)
                        (setq interval-time nil))
                       (t
                         (setq params (find-next-parameter current-status
                                                           (+ start interval-time))
                               possible-choices (remove-next-time-events
                                                   (+ start interval-time) possible-choices))
                         (setq current-status (car params)
                               interval-time (- (cadr params) start )))))
                (t
                  (update-hash-tables start
                                      (loop for item in (car group)
                                            as duration = (get item 'duration)
                                            as performances = (zl:fix (/ interval-time duration))
                                            as time = (* performances duration)
                                            collect (list item time) into var1
                                            minimize time into var2
                                            finally (setq interval-time var2)
                                                    (return var1)
                                            do
                                        (zl:putprop item (+ performances
                                                            (get item 'scheduled-performances))
                                                    'scheduled-performances)
                                        (zl:putprop item (- (get item 'performances)
                                                            performances)
                                                    'performances)
                                        (setq possible-choices (remove-experiment-from-schedule-list
                                                                  item possible-choices))))
                  (setq interval-time nil))))))

(defun complete (self)
  (send self :deactivate))
```

```
(defun display-pass (&optional (title nil))
  (dw::with-output-truncation (*resource-output-window* :horizontal t)
    (cond (title
           (format *resource-output-window* "~2%~38t~vResource Allocation Results~%"
                   *Font*)
           (cond ((null *resources-output*)
                  (send *display-menu* :set-label "Select Displayed Output")
                  (send *display-menu* :set-item-list *resources*)
                  (send *display-menu* :choose)
                  (setq *resources-output*
                        (reverse (send *display-menu* :highlighted-values)))))
           (format *resource-output-window* "~4% **** FIRST PASS RESULTS ****~2%"))
          (t
           (format *resource-output-window* "~4% **** SECOND PASS RESULTS ****")))
    (select-graphical-display)
    (let ((x-y-locations (Initialize-Graph-information *graphical-output*))
          (space 10))
      (show-scheduled)
      (loop for resource in *resources-output*
            initially (space-over *resource-output-window* (+ 6 space))
            do
        (space-over *resource-output-window* space)
        (format *resource-output-window* "~'b@a~%" resource))
      (loop for time in *time-list*
            for next-time in (cdr *time-list*)
            do
        (setq x-y-locations (display-output-sensitive "~%" time next-time x-y-locations
                                                      :stream *resource-output-window*))
        (loop for variable in (make-variables *resources-output*)
              for header in *resources-output*
              as width = (string-length header)
              for column first (+ space (/ width 2.0) space)
                       then (+ space (/ width 2.0) column)
              do
          (format *resource-output-window* (format nil "~~~at" (zl:fix column)))
          (format *resource-output-window* "~8@a" (gethash time (eval variable)))
          (setq column (+ (/ width 2.0) column))))))))

(defun display-output-sensitive (return time next-time x-y-locations
                                 &key (stream *resource-menu-window*)
                                      (type 'label-type))
  (dw:with-output-as-presentation (:single-box t
                                   :stream stream
                                   :dont-snapshot-variables t
                                   :type type
                                   :object (list time))
    (print-it stream return time))
;   (print-it *graphics-window* return time))
    (if (and (not (equal *graphical-display* 'none)) x-y-locations)
        (setq x-y-locations (funcall *graphical-display* x-y-locations next-time)))
  x-y-locations)

(defun print-it (stream return time)
  (format stream (format nil "~a~A" return time)))

(defun make-variables (lst)
  (loop for string in lst
        collect (make-variable-from-string string)))

(defun show-used ()
  (format *resource-output-window* "~3%~10TItem~20tRemaining~40tScheduled~%")
  (loop for item  in (get 'list-of 'names)
        do
    (format *resource-output-window* "~%~10T~A~23t~a~43t~a" item (get item 'performances)
            (get item 'scheduled-performances))))

;;;;;;;;;;;;;; Second Pass Functions ;;;;;;;;;;;

(defun non-scheduled (lst used)
  (let ((possible lst))
    (loop for item in used
          do
      (setq possible (remove item possible :test #'equal )))
    possible))
```

```
;;;;;;;;;;;;;;; Common Pass Functions ;;;;;;;;;;;

(defun find-new-parameters (&optional (current nil)(params nil))
  (let ((lst *time-list*))
    (cond ((null current)
           (setq lst (cons 0 lst)))
          (t
            (setq lst (member current *time-list*  :test #'= ))))
    (loop with start = (cadr lst)
          with status = (if params params (current-status start))
          for time in (cddr lst)
          while (compare-each-time-status status time)
          finally (return (list start (if time (- time start)
                                          (- *max-time* (cadr lst))))))))

(defun find-next-parameter (current time)
  (let ((next (mapcar #'(lambda (x y) (if (> x y) x y)) current
                      (current-status time))))
    (list next (cadr (member time *time-list*)))))

(defun remove-next-time-events (time lst)
  (loop for item in (gethash time scheduled-items)
        do
    (setq lst (remove-experiment-from-schedule-list item lst)))
  lst)

(defun compare-each-time-status (status time)
  (loop for pos from 0
        for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (<= (gethash time (eval each))
                   (nth location status))
        finally (return t)))

(defun Parameters-within-range (current-status)
  (loop for each in *maximizing-resource-list*
        for location in *maximizing-resource-position*
        always (> (get each 'resource-limit)
                  (nth location current-status))))

(defun update-Hash-tables (start lst)
  (loop for (item1 duration) in lst
        as end-time = (+ start duration)
        do
    (cond ((null (member end-time *time-list* :test #'=))
           (loop for resource in (cons 'scheduled-items *resource-variables*)
                 do
             (swaphash end-time (Get-hash-value end-time resource nil) (eval resource)))
           (setq *time-list* (sort (cons end-time (copy-list *time-list*)) #'<))))
    (loop for time in (member start *time-list*)
          until (= end-time time)
          do
      (swaphash time (append (Gethash time scheduled-items) (list item1))
                scheduled-items)
      (loop for resource in *resource-variables*
            do
        (swaphash time (+ (Get-hash-value time resource)
                          (get item1 resource)) (eval resource))))))

(defun Get-hash-value (time resource &optional (not-new t))
  (let ((value (gethash time (eval resource))))
    (cond (value value)
          (not-new nil)
          (t (gethash (loop with previous = 0
                            for last-time in *time-list*
                            until (>= last-time time)
                            finally (return previous)
                            do
                      (setq previous last-time)) (eval resource))))))

(defun find-resource-candidates (lst endpoint start)
  (loop for exp in (find-interval-candidates lst endpoint)
        if (check-constraints (add-constraint-values (current-status start) exp))
          collect exp into resource-candidate-list
        finally (return resource-candidate-list)))
```

```
(defun find-interval-candidates (lst endpoint)
  (loop for exp in lst
        if (feasible-interval exp endpoint)
          collect exp into variable
        finally (return variable)))

(defun feasible-interval (experiment endpoint)
  (< (get experiment 'duration ) endpoint))

(defun find-possible-downward-paths (sv lst)
  (let* ((top (car lst))
         (bottom (cdr lst))
         (val (add-constraint-values sv top)))
    (cond ((null (check-constraints val)) '(()))
          (bottom
             (loop for down-lst on (cdr lst)
                   append (group-intermediate-lists
                             top (find-possible-downward-paths val down-lst)) into var
                   finally (return var)))
          (t (list lst)))))

(defun add-constraint-values (lst exp)
  (loop for resource in *resource-variables*
        for value in lst
        if (null value)
          do (setq value 0)
        collecting (+ value (get exp resource))))

(defun check-constraints (lst)
  (loop for resource in *resource-variables*
        for value in lst
        always (apply (get resource 'resource-constraint-function) (list value))
        finally (return t)))

(defun find-max-path (time sv lst)
  (loop with max-paths = nil
        with max-value = 0
        for new-lst on lst
        as paths = (find-possible-paths sv new-lst)
        as value = (get-time-interval-priority-value (get-group-values (car paths)) sv)
        finally (setq max-paths (sort-max-paths max-paths))
                (Set-back-tracking-paths
                  time (gethash time scheduled-items) max-paths)
                (return (car max-paths))
        do
    (cond ((= max-value value)
           (setq max-paths (append max-paths paths)))
          ((< max-value value)
           (setq max-paths paths
                 max-value value)))))

(defun Set-back-tracking-paths (time prefix suffix)
  (swaphash time
            (remove-duplicates
              (loop for (eac rst) in suffix
                    collect (append prefix eac))
              :test #'equal)
            *paths*))

(defun sort-max-paths (paths)
  (let ((lst (loop for path in paths
                   collecting (list path (get-group-values path)))))
    (loop for pos in (reverse *maximizing-resource-position*)
          do
      (setq lst (sort lst #'> :key (lambda (x) (nth pos (cadr x))))))
    lst))

(defun get-time-interval-priority-value (values lst &optional (pos 0))
  (cond (values
         (+ (nth (nth pos *maximizing-resource-position*) values)
            (nth (nth pos *maximizing-resource-position*) lst)))
        (t 0)))

(defun group-intermediate-lists (item lst)
  (loop for each in lst
        collect (cons item each)))
```

```
(defun remove-experiment-from-schedule-list (exp lst)
  (remove exp (copy-list lst) :test #'equal))

(defun find-possible-paths (val resource-candidates)
  (let ((lst (find-possible-downward-paths val resource-candidates)))
    (cond ((null lst)(return-from find-possible-paths nil))
          (t (get-maximized-sub-path lst)))))


(defun get-maximized-sub-path (paths)
  (loop for resource in *maximizing-resource-list*
        for position in *maximizing-resource-position*
        until (= (length paths) 1)
        do
    (setq paths
          (loop for lst in paths
                with max-val = 0
                with max-lsts = nil
                as resource-value = (nth position (get-group-values lst))
                finally (return (reverse max-lsts))
                do
            (cond ((> resource-value max-val)
                   (setq max-val resource-value
                         max-lsts (list lst)))
                  ((= resource-value max-val)
                   (setq max-lsts (cons lst max-lsts)))))))
  paths)

(defun get-group-values (group)
  (loop for item in *resource-variables*
        collecting (loop for each in group
                         summing (get each item))))


(defun current-status (time)
  (loop for each in *resource-variables*
        as value = (gethash time (eval each))
        collecting (if value value 0)))

(defun show-scheduled ()
  (format *resource-output-window* "~2% Time ~20tScheduled Events~%")
  (loop for time in *time-list*
        do
    (format *resource-output-window* "~% ~A ~20t~A" time (gethash time scheduled-items)))
  (format *resource-output-window* "~2%"))

(defun show-resource (resource)
  (loop for time in *time-list*
        do
    (format t "~% ~A ~20t~A" time (gethash time resource))))



;(defun make-mouse-sensitive-labels (return object &key (stream *resource-menu-window*)
;                                                       (type 'label-type))
;  (dw:with-output-as-presentation (:single-box t
;                                   :stream stream
;                                   :type type
;                                   :object object)
;    (format stream (format nil "~a~A" return (cadr object)))))
```

Appendix G
Symbolics Lisp Code for Frontier of Feasibility System

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

(defvar *Resource-File-Directory* "andy:>jsr>resource-allocation>multiple-data-files>")

(defvar *frames*)

(defvar *max-resource-area* 0)

(defvar *currently-used* 0)

(defvar *current -file* nil)

(defvar *experiments*)

(defvar *max-resource-area* 58000000)

(defvar *Not-Previously-Notified* t)

(defvar *message-window* (tv:make-window 'dw:dynamic-window
                         ;       :blinker-p nil
                                 :edges-from '(300 300 850 400)
                                 :more-p nil
                                 :margin-components
                                 '((dw:margin-scroll-bar :visibility :if-needed)
                                   (dw:margin-ragged-borders :thickness 4)
                                   (dw:margin-label
                                     :margin :bottom
                                     :string "Message Window       (Press any key to EXIT)"))))

(defvar *interface-window* (tv:make-window 'dw:dynamic-window))


(defflavor activity
        (Name
         Experiment-Number
         Duration
         Power-Required
         Man-Power
         Data-Rate
         Performances
         Minimum-Performances
         Maximum-Performances
         Scheduled-Performances
         Presentation
         (Highlighted nil))
        ()
  (:conc-name "")
  :initable-instance-variables
  :readable-instance-variables
  :writable-instance-variables)

(defun set-up-objects ()
 ; (setq *max-resource-area* (* *max-time* *max-resource*))
  (loop for each in *frames*
        as name = (car each)
        collecting  name into name-list
        as lst = (loop for next in (cdr each)
                       collecting (read-from-string (format nil ":~a" (car next))) into args
                       collecting (caadr next) into args
                       finally (return (append (list :name (format nil "~a" name)) args)))
        finally (setq *Experiments* name-list)
        do
    (set name (apply #'make-instance (cons 'activity lst)))
    (set-minimum (eval name)))
  (calculate-area-used))

(defmethod (set-minimum activity) ()
  (setq Minimum-performances Performances))

(defun restart ()
  (setq *current-file* nil *currently-used* 0 *used-lst* nil ij 1))

(defun calculate-area-used ()
  (setq *currently-used*
        (loop for name in *experiments*
              as duration = (duration (eval name))
```

```
            as power = (power-required (eval name))
            as perfs = (performances (eval name))
            summing (* duration power perfs) into tot-area
            finally (return tot-area))))

(defun make-window-layout ()
  (let* ((space 10))
    (format *interface-window* "~%")
    (loop for exp-lst in (subgroup-list *experiments* 12)
          counting t into row
          collecting (loop for exp in exp-lst
                           counting t into column-number
                           as column = (* 10 column-number)
                           collect (list exp row column-number) into headings
                           finally (format *interface-window* "~%")
                                   (return headings)
                           do
                     (format *interface-window* (format nil "~~~at~a" (zl:fix column) exp)) ) into var
          do
      (loop for exp in exp-lst
            counting t into col-num
            as col = (* 10 col-num)
            do
        (place-variable col 'performances exp))
      (format *interface-window* "~2%"))))

;;This defines the item presentation type and documentation line display
(define-presentation-type resource-type ()
   :no-deftype t
   :parser ((stream) (loop do (dw:read-char-for-accept stream)))
   :printer ((object stream)
             (format stream "the resource ~A" (car object))))

;;This is what is done when the item is selected
(define-presentation-action choose-type
   (resource-type t
    :gesture :left
    :context-independent t
    :documentation  "Change this value")
   (resource)
  (throw 'resource
    (list resource (presentation (eval (caar resource))))))

;;This function assists in correct column spacing
(defun place-variable (column variable exp)
  (format *interface-window* (format nil "~~~at" (zl:fix column)))
  (format-item-mouse-sensitive *interface-window* (funcall variable (eval exp))
                               (list (list exp variable)
                                     (multiple-value-bind (a b)
                                         (send *interface-window* :read-cursorpos)
                                       (list a b)))))

;;This function prints the item to the screen with mouse sensitivity
(defun format-item-mouse-sensitive (stream incoming-item descriptors)
  ;(if (> ij 172) (dbg:dbg) (setq ij (+ 1 ij)))
  (let* ((object (eval (caar descriptors)))
         (items (verify-value-range object Incoming-item))
         (font (car items))
         (item (cadr items)))
    (eval `(setf ,(list (cadar descriptors) object) ,item))
    (send stream :set-cursorpos (caadr descriptors) (cadadr descriptors))
    (clearspace stream)
    (setf (presentation object)
          (dw:with-output-as-presentation (:single-box t
                                           :stream stream
                                           :type 'resource-type
                                           :object descriptors)
            (send stream :set-cursorpos (caadr descriptors) (cadadr descriptors))
            (format stream "~vA~⊃ font item)))))

(defmethod (verify-value-range activity) (item)
  ;(if (> ij 172) (dbg:dbg))
  (let* ((font '(:fix :roman :normal))
         (upper maximum-performances)
         (lower minimum-performances) ;; (zl:fix (+ (* 2/3 upper) .9)))
         (state nil)
```

```
        (available (- *max-resource-area* *currently-used*))
        (increment (zl:fix  (/ available (if (> power-required 0)
                                             (* duration power-required) (abs available)))))
        (resource-limit (+ performances
                           (if (> increment 0) increment 0))))
  ;  (dbg:dbg)
    (cond ((and (> item upper)
                (>= resource-limit upper))
           (setq font '(:fix :bold :normal)
                 state 'upper))
          ((< item lower)
           (setq font '(:fix :italic :normal)
                 state 'lower))
          ((and (> item  resource-limit)
                (> upper resource-limit))
           (setq font '(:fix :roman :normal)
                 state 'resource-limit)))
    (case state
      (upper  (setq font '(:fix :bold :normal))
              (send-message-to-user
                (format nil "The value you entered (~a) for the number of~
                             ~%Performances of ~a is above the maximum allowed of ~A~2%~
                              The maximum value will be used." item name upper))
              (setq item upper))
      (lower (setq font '(:fix :italic :normal))
             (send-message-to-user
               (format nil "The value you entered (~a) for the number of~
                            ~%Performances of ~a is below the minimum allowed of ~A~2%~
                            The minimum value will be used." item name lower))
             (setq item lower))
      (resource-limit
        (send-message-to-user
          (format nil "The value you entered (~a) for the number of~
                       ~%Performances of ~a would exceed the available ~%~
                       amount of the resource (~A).~2%~
                       The maximum possible value (~a) will be used."
                  item name available resource-limit))
        (setq item resource-limit)))
    (cond-every ((= item lower)
                 (setq font '(:fix :italic :normal)))
                ((= item upper)
                 (setq font '(:fix :bold :normal))))
    (setq *currently-used* (+ *currently-used* (* (- item performances) duration power-required)))
    (list font item state)))

(defun review-possible-increases ()
  (let ((Frontier-node t))
    (loop for each in *experiments*
          do
          (cond ((no-possible-increase (eval each))
                 (highlight-object (eval each)))
                ((highlighted (eval each))
                 (remove-existing-highlight (eval each))
                 (setq Frontier-node Nil))
                ((not-maximized (eval each))
                 (remove-existing-highlight (eval each))
                 (setq Frontier-node Nil))))
    Frontier-node))

(defmethod (not-maximized activity) ()
  (> maximum-performances performances))

(defmethod (no-possible-increase activity) ()
  (> (* duration power-required)
     (- *max-resource-area* *currently-used*)))

(defmethod (remove-existing-highlight activity) ()
  (let ((box (dw::presentation-displayed-box presentation))
        (original-position (multiple-value-bind (a b)
                               (send *interface-window* :read-cursorpos)
                             (list a b)))
        (font '(:fix :roman :normal)))
    (setq highlighted nil)
    (cond ((= performances maximum-performances)
           (setq font '(:fix :bold :normal)))
          ((= performances minimum-performances)
```

```
          (setq font '(:fix :italic :normal))))
    (graphics:draw-rectangle (dw::box-left box) (dw::box-top box)
                             (dw::box-right box)(dw::box-bottom box)
                             :stream *interface-window* :opaque t :alu :erase)
    (send *interface-window* :set-cursorpos (dw::box-left box) (dw::box-top box))
    (format *interface-window* "~v∈a~⊃ font performances)
    (send *interface-window* :set-cursorpos (car original-position)(cadr original-position))))

(defmethod (highlight-object activity) ()
  (let ((box (dw::presentation-displayed-box presentation)))
    (setq highlighted t)
    (graphics:draw-rectangle (dw::box-left box) (dw::box-top box)
                             (dw::box-right box)(dw::box-bottom box)
                             :stream *interface-window* :opaque nil :gray-level .15)))

(defun clearspace (stream)
  (loop repeat 8
        do
    (send stream :clear-char)
     (send stream :forward-char)))

;;This function returns the list of data files that can be selected.
(defun get-data-file-list ()
  (loop for directory in (cdr (fs:directory-list *Resource-File-Directory* ))
        as pathname = (cond ((not (string= (send (car directory) :name) "err"))
                             (format nil "~A" (send (car directory) :string-for-dired))))
        collect  pathname ))

;;This function allows communication between the user and the program.
(defun send-message-to-user (message)
  (send *message-window* :clear-history)
  (send *message-window* :set-cursor-visibility nil)
  (send *message-window* :select)
  (format *message-window* message)
  (send *message-window* :any-tyi)
  (send *message-window* :deselect))

(defun subgroup-list (lst group-sizes)
  (let* ((group-size (if (>= group-sizes 1)(zl:fix group-sizes) (length lst)))
         (len (length lst))
         (repeats (/ len group-size)))
    (loop repeat (zl:fix (if (not (= (mod len group-size) 0))
                             (+ 1 repeats) repeats))
          as start first 0 then (+ start group-size)
          as finish first group-size then (+ finish group-size)
          collect (if (> finish len)
                      (subseq lst start)
                      (subseq lst start finish)))))


;;This function reads in a value, but does not issue a line-feed.
(defun read-without-return (&optional (stream *standard-output*)
                            &key (activation-characters '(#\Return #\End )))
  (loop with cursor-position = (list (multiple-value-bind (a b)
                                         (send stream :read-cursorpos) (list a b)))
        with var2 = nil
        with position = 0
        as var1 = (send stream :tyi)
        as total-length = (length var2)
        until (member var1 activation-characters)
        if var1
          do
            (cond ((and (equal var1 #\rubout) var2)
                   (send stream :tyo #\backspace)
                   (send stream :clear-char)
                   (setq var2 (cdr var2)
                         position (1-  position)
                         cursor-position (cdr cursor-position)))
                  ((and (or (equal var1 #\c-B)(equal var1 #\backspace)) var2)
                   (setq position (1- position))
                   (send stream :tyo var1))
                  ((equal var1 #\c-F)
                   (cond ((< position total-length)
                          (setq position (1+ position))
                          (send stream :tyo var1))))
                  ((= position total-length)
```

```
                    (setq var2 (cons var1 var2)
                          position (1+ position)
                          cursor-position (cons (multiple-value-bind (a b)
                                                  (send stream :read-cursorpos)
                                                  (list a b)) cursor-position))
                    (format stream "~a" var1))
                   ((or (equal var1 #\c-B) (equal var1 #\rubout )))
                   (t (send stream :insert-char)
                      (format stream "~A" var1)
                      (setq var2 (reverse (loop for temp = nil
                                                then (append temp (list (car end)))
                                            for end = (reverse var2) then (cdr end)
                                            repeat position
                                            finally (return
                                                      (append temp (cons var1 end))))))))
          finally (return (cond (var2 (setq var2 (read-from-string
                                                  (apply #'string-append (reverse var2))))))))))

;;This function allows the data values to be changed.
(defun change-data-point ()
  (cond ((and *Not-Previously-Notified* (review-possible-increases))
         (send-message-to-user (format nil "~%The current selection represents a Frontier Node.~2%~
                                            No possible performance INCREASES exist."))
         (setq *Not-Previously-Notified* nil)
         'Notified)
        (t
         (let ((data (catch 'resource (accept 'resource-type
                                              :prompt nil
                                              :stream *interface-window*)))
               (original-position (multiple-value-bind (a b)
                                      (send *interface-window* :read-cursorpos)
                                    (list a b)))
               (position))
           (setq *Not-Previously-Notified* t)
           (cond ((or (atom data) (atom (car data)))
                  data)
                 (t
                  (setq position (cadar data))
                  (send *interface-window* :erase-displayed-presentation (cadr data))
                  (send *interface-window* :set-cursorpos (car position)(cadr position))
                  (send *interface-window* :set-cursor-visibility :blink)
                  (format-item-mouse-sensitive *interface-window*
                                               (read-without-return *interface-window*)
                                               (car data))
                  (send *interface-window* :set-cursor-visibility nil)
                  (send *interface-window* :set-cursorpos (car original-position)
                        (cadr original-position))
                  'data))))))

(defun frontier-interface ()
  (if (null-string *current-file*)
      (open-input-file))
  (loop with again = t
        while again
        do
    (send *interface-window* :select)
    (send *interface-window* :clear-history)
    (format *interface-window* "~50t~vFrontier Development Interface~%2%" '(:Fix :bold :normal))
    (make-window-layout)
    (send *interface-window* :set-cursor-visibility nil)
    (monitor-usage)
    (loop with finished = nil
          until finished
          as choice = (change-data-point)
          while choice
          do
      (monitor-usage))))

(defun monitor-usage ()
  (send *interface-window* :set-cursorpos 550 670)
  (send *interface-window* :clear-rest-of-line)
  (format *interface-window* "~5,2f% Available (~a Remaining  ~a Used)"
          (* 100.0 (/ (- *max-resource-area* *currently-used*) *max-resource-area*))
          (float (- *max-resource-area* *currently-used*)) (float *currently-used*)))

(defun null-string (str)
```

```
  (= (length str) 0))

(defun open-input-file ()
  (let ((infile (dw:menu-choose (get-data-file-list)
                                :prompt "Data File List")))
    (cond (infile (load (string-append *Resource-File-Directory* infile)
                        :verbose nil)
                  (set-up-objects)
                  (setq *current-file* infile)))))


(defun test ()
  (loop for each in *experiments*
        as eac = (eval each)
        do
    (format t "~%~a~14t~a~20t~a~30t~a~45t~a~60t~A"
            each (performances eac) (minimum-performances eac) (maximum-performances eac)
            (* (power-required eac) (duration eac)) (no-possible-increase eac))))
```

```
;;; -*- Syntax: Common-Lisp; Package: USER; Base: 10; Mode: LISP -*-

(defvar *resource-allocation-graphics-window*
        (tv:make-window 'dw:dynamic-window))

(defvar *objects* nil)

(defflavor activities
        (Value
         Horizontal-location
         vertical-location
         Maximum
         Minimum)
        ()
    :initable-instance-variables
    :readable-instance-variables
    :writable-instance-variables)

(defvar *horizontal-limit* 600)

(defvar *vertical-offset* 75)

(defvar *horizontal-offset* 100)

(defvar *scale-x* 3)

(defmethod (draw-object-mouse-left activities) (xref)
  (let ((x (+ xref *horizontal-offset*)))
    (graphics:draw-string (format nil "~a" value) (+ Horizontal-location 10) vertical-location
                          :stream *resource-allocation-graphics-window* :alu :erase
                          :attachment-y :top :character-style '(:fix :roman :very-small))
    (graphics:draw-rectangle x vertical-location Horizontal-location (+ 5 vertical-location)
                             :stream *resource-allocation-graphics-window* :alu :flip)
    (setq Horizontal-location x
          Value (calc-new-value Horizontal-location))
    (graphics:draw-string (format nil "~a" value) (+ Horizontal-location 10) vertical-location
                          :stream *resource-allocation-graphics-window*
                          :attachment-y :top :character-style '(:fix :roman :very-small))))

(defun calc-new-value (x)
  (/ (- x *horizontal-offset*) *scale-x*))

(defmethod (check-object activities) (y)
  (<= vertical-location y (+ 5 vertical-location)))

(defun create-initial-objects (num)
  (loop repeat num
        for name in '(anfghj ertyuil yupoliu ewyrue ttyyss gsgsgsg iweie83k ieieiokk jfjfjfkl qwernm)
        counting t into down
        as vert = (+ (* down 10) *vertical-offset*)
        as val = (random 200)
        as hori = (zl:fix (+ *horizontal-offset* (* (/ val 200) *horizontal-limit*)))
        collect (make-instance 'activities
                               :vertical-location vert
                               :Horizontal-location hori
                               :Value val
                               :Maximum (zl:fix (+ val (* .5 (- 200 val))))
                               :Minimum (zl:fix (* .5 val))) into vars
        finally (setq *objects* vars)
        do
    (graphics:draw-string (format nil "~a" name) (- *offset* 10) vert :stream *resource-allocation-graphi
cs-window*
                          :attachment-y :top :attachment-x :right :character-style '(:fix :roman :very-sm
all))
    (graphics:draw-rectangle *horizontal-offset* vert Hori (+ 5 vert) :stream *resource-allocation-graphi
cs-window*)
    (graphics:draw-string (format nil "~a" val) (+ 10 Hori) vert :stream *resource-allocation-graphics-wi
ndow*
                          :attachment-y :top :character-style '(:fix :roman :very-small))))

(defun top-level-ii (&optional (num 10))
  (send *resource-allocation-graphics-window* :select)
  (send *resource-allocation-graphics-window* :clear-history)
  (create-initial-objects num)
  (dw:with-output-recording-disabled (*resource-allocation-graphics-window*)
    (loop with previous = nil
```

```
        do
      (dw:tracking-mouse (*resource-allocation-graphics-window*
                          :who-line-documentation-string
                          "Revise allocation of item")
        (:mouse-motion-hold (x y)
         (let ((xloc (* (truncate (- x *horizontal-offset*) *scale-x*) *scale-x*)))
           (if (and previous
                    (validate-object-maximum previous xloc))
               (draw-object-mouse-left previous xloc))))
        (:mouse-click (button x y)
         (if (equal button #\mouse-1)
             (loop for each in *objects*
                   when (check-object each y)
                     do
                        (setq previous each))))
        (:release-mouse ()
                        (setq previous nil))))))

(defmethod (validate-object-maximum activities) (mouse-position)
           (<= minimum (/ mouse-position *scale-x*) maximum))
```