Applications of Artificial Intelligence to Mission Planning

INTERIM REPORT

for

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

by

Donnie R. Ford
John S. Rogers
and Stephen A. Floyd
The University of Alabama in Huntsville
Huntsville, AL 35899
(205) 895-6240

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

operational instructions were reported to NASA MSFC in the first Interim Report 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 were requested in order to increase its functionality and performance capabilities. Coupled with the modified software, new Frontier of Feasibility traversing techniques were to be examined. A description of each of the alterations and additions to the prototype software and differing techniques are detailed in the following sections of this paper.

2.0 Software Data Structure Conversion

2.1 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 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.

2.2 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.

2.3 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.

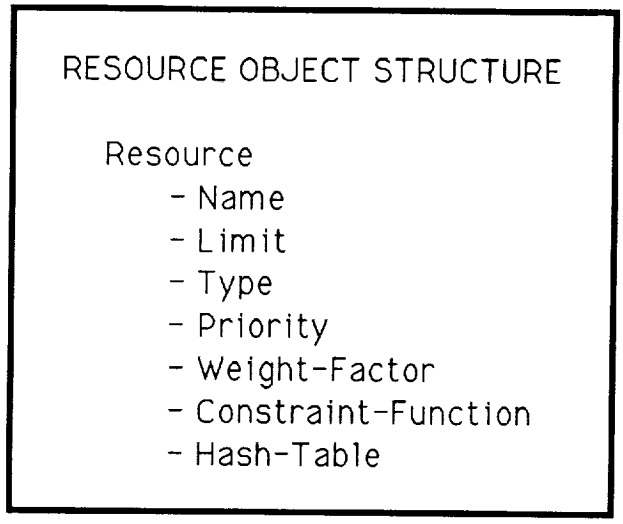

Figure 1

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.

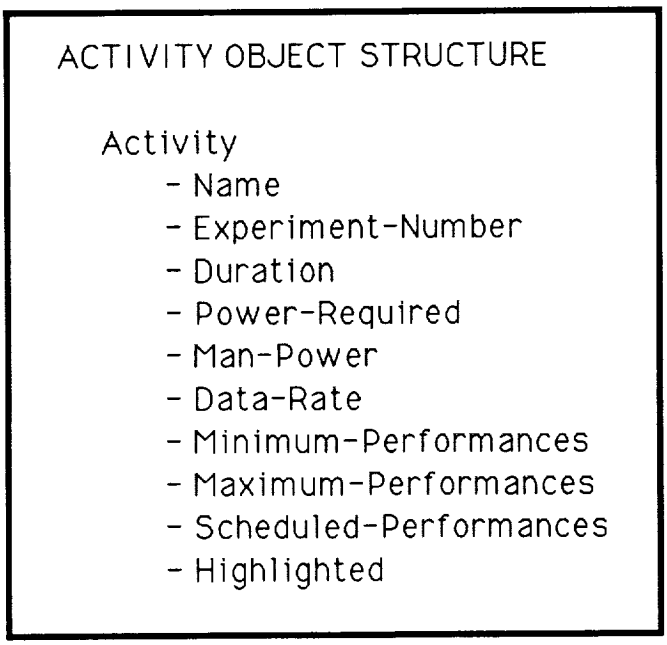

Figure 2

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.

2.4 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.

3.0 Software Functionality Modifications and Enhancements

3.1 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.

3.2 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.

3.3 Task Approach

Although a general description of the resource allocation software system's allocation algorithm is describe 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.

3.4 Task Results

The software system was modified from a multiple allocation to a single allocation step process. 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 price. The system which was already under criticism for the

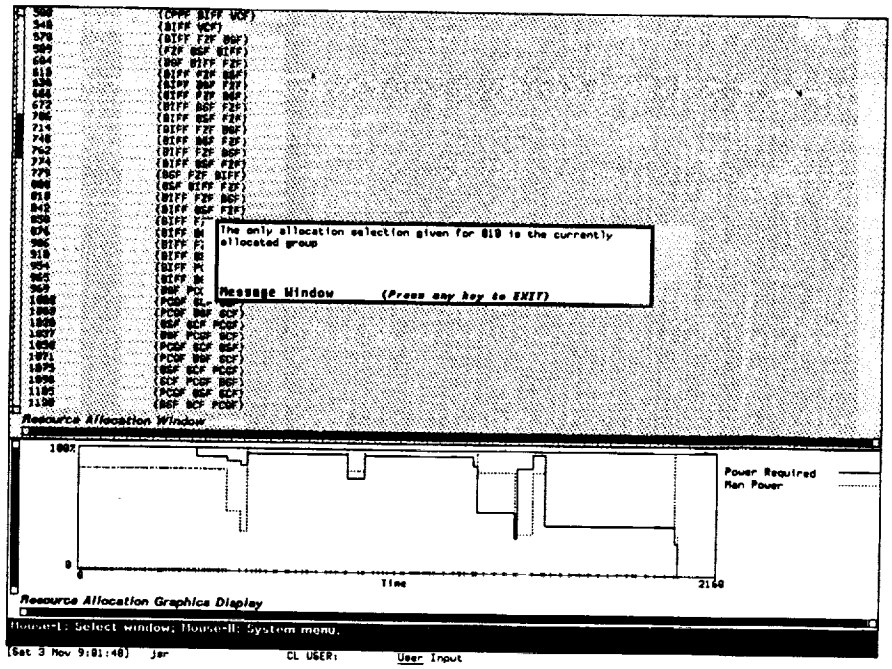

Figure 3

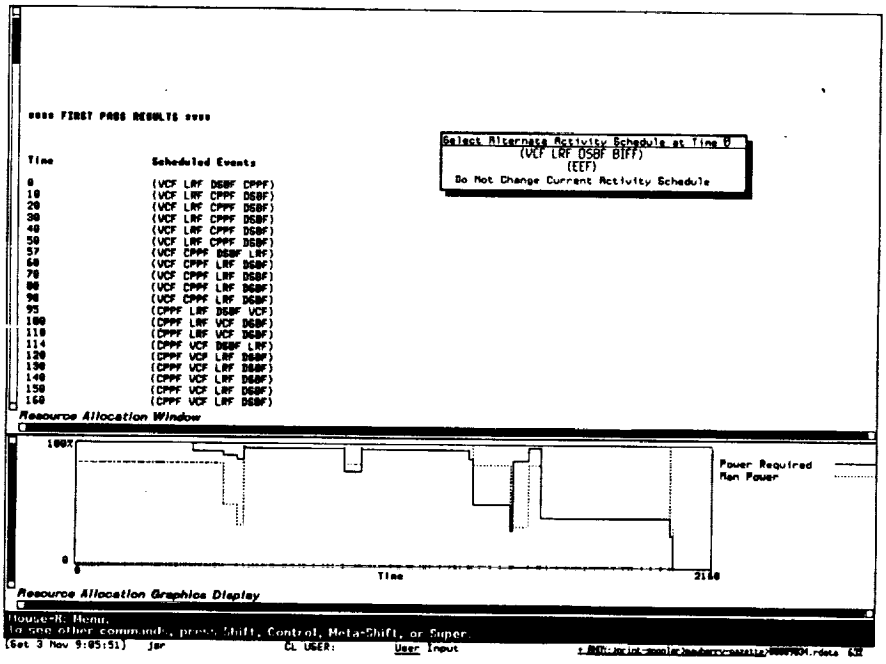

Figure 4

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.

4.0 Portability of Resource Allocation To A TI MicroExplorer

4.1 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.

4.2 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.

4.3 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.

4.4 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.

5.0 Frontier of Feasibility Software System

5.1 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.

5.2 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.

5.3 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.

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 1. 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.

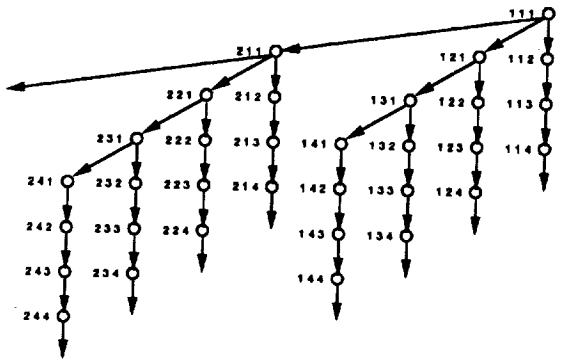

Figure 2. A three activity tree graph.

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.

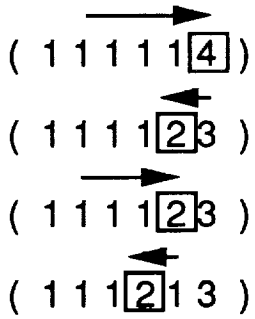

Figure 3. Example of the six stage process.

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.

5.4 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.

Appendix A

```
;;; -*- 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~5 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 B

```
;;; -*- 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~3%"
                   *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~5 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 C

```
;;; -*- 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 "~v€a~5 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~ↄ 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))
```