C++ Planning and Resource Reasoning (PARR) Shell

James McIntyre, Alan Tuchman, David McLean and Ronald Littlefield
AlliedSignal Technical Services Corporation
Goddard Corporate Park
7515 Mission Drive
Seabrook, MD 20706

ABSTRACT

This paper describes a generic, C++ version of the Planning and Resource Reasoning (PARR) shell which has been developed to supersede the C-based versions of PARR that are currently used to support AI planning and scheduling applications in flight operations centers at Goddard Space Flight Center. This new object-oriented version of PARR can be more easily customized to build a variety of planning and scheduling applications, and C++ PARR applications can be more easily ported to different environments. Generic classes of PARR objects for resources, activities, constraints, strategies, and paradigms are described along with two types of PARR interfaces.

Keywords: AI, Planning, Scheduling, Shell

INTRODUCTION

The Artificial Intelligence (AI) software group at AlliedSignal Technical Services Corporation has been developing expert system software for Goddard Space Flight Center since 1985. These systems use expert system technology to automatically create conflict-free schedules. Many expert systems use commercial programs known as expert system shells written in either Lisp or Prolog. These systems often suffer a performance penalty when they are used on personal computers or engineering workstations. Because we developed our programs using conventional programming languages on readily available computer hardware, they are able to rapidly schedule hundreds of events.

We have adapted an evolutionary prototyping approach to software development. The prototyping aspect of this approach dictates that we gather initial requirements, create or evolve a prototype, evaluate the prototype to update the requirements, and repeat the process until the prototype satisfies the software users. The evolutionary aspect dictates that our prototypes are not developed to be thrown away, but instead are based on generic, reusable software tools. The prototypes are updated, not rebuilt, with each prototyping cycle. As we build a system, we develop new software tools. When the tools appear to be useful outside the project where they were first implemented, we add them to our tool library. In many cases, we are able to base a new system entirely on tools that exist in our library.

BACKGROUND

We delivered the Earth Radiation Budget Satellite (ERBS)-Tracking and Data Relay Satellite System (TDRSS) contact planning system to the ERBS flight operations team in 1987. It was the first AI expert system application at Goddard that supported flight operations. It is used to build schedules of requests to TDRSS for communications contacts to send commands to ERBS and to download data.

The ERBS-TDRSS system created its schedules using the first version of the Planning And Resource Reasoning (PARR) tool (McLean *et al.*, [1]). PARR is an expert system shell designed expressly for scheduling applications. Its knowledge base is organized into activity classes. Each activity class represents something that can be scheduled; in the ERBS-TDRSS system, the activity classes represent the scheduling of communication contacts between ERBS and a Tracking and Data Relay Satellite (TDRS) via a specific antenna. Each activity class contains both strategies which suggest ways that the activity can be scheduled and constraints which specify limitations on when the activity can be scheduled.

PARR works as an intelligent tactical planning tool to put specific activities on a timeline by following the strategies and checking the constraints found in its knowledge base. Unlike traditional scheduling systems that try to anticipate all possible scheduling conflicts before scheduling an activity, PARR uses conflict resolution strategies to reschedule activities that do not meet their constraints (McLean, *et. al.*, [2]). PARR's knowledge base forms a strategic plan, a list of broad strategies used to schedule activities. PARR uses this strategic plan to create a tactical plan, a list of specific activities with specific times and durations.

PARR is unusual in its use of a combination of conflict avoidance and conflict resolution. While many scheduling systems backtrack and change their partial schedules extensively, PARR mimics the way people create schedules. We have found that people obtain insight into a scheduling problem and quickly develop rules like "that requires most of the resources, so schedule it first" or "if we have trouble fitting this into the schedule, we can reduce its duration by a few minutes and make it fit." PARR captures this kind of strategy in its knowledge base along with the more traditional conflict avoidance rules like "this can only be scheduled when that resource is available." By dynamically applying both types of rules, PARR limits the amount of search that is required to build a timeline and is able to produce a conflict-free schedule much faster than other systems.

After developing the ERBS-TDRSS system, we realized we had the appropriate tools in our library to quickly build more complex scheduling systems (McLean, *et. al.*, [3]). In 1991, we delivered the Explorer Platform Planning System (EPPS) to the flight operations team for the Extreme Ultraviolet Explorer (EUVE) spacecraft (McLean, *et. al.*, [4]). EPPS shares much of its implementation with the ERBS-TDRSS system, including a revised version of PARR. Like the ERBS system, EPPS generates schedules of TDRSS requests. In the development of EPPS, we found that our simple way of expressing constraints was insufficient. The ERBS system simply checked whether a conflicting resource or activity was scheduled during the period under consideration. EPPS needed to schedule activities that affected power and tape consumption, and the system needed constraints that could track these resources. We realized that new scheduling problems would need new resource models to support new kinds of constraints. Rather than creating new code to support each new type of resource, we created more general resource models that could be used to model many different kinds of resources. EPPS was the first system to use resource models

The next significant PARR development occurred as we developed the Hubble Space Telescope (HST) Servicing Mission Planning and Replanning Tool (SM/PART) which we delivered in 1992 to help the HST flight operations personnel build integrated timelines and command plans to control the activities of the HST, the space shuttle and the space shuttle crew for the HST servicing mission in December 1993 (Bogovich, *et. al.*, [5]). SM/PART uses PARR in a slightly different way. In our previous systems, each activity class in the knowledge base is scheduled repeatedly over time. In SM/PART, each activity class is scheduled only once during the schedule, but the activity class definitions and resource availability may change, causing the need for replanning. This changed our understanding of the relationship between tactical and strategic planning. Users may not always be able to completely define all of their activities before scheduling and may need to tweak the knowledge base during the tactical planning process.

In addition to these three systems, we also developed several prototype scheduling systems that gave us insight into other useful capabilities. We prototyped systems that merged schedules that were created by other scheduling systems and built a new version of PARR that takes multiple schedules as inputs and merges them into a single conflict-free schedule. Another prototype scheduled observations of stars and represented each potential target star as a different activity class. To create an efficient schedule, we created a system that found the shortest path between the targets stars using a Hopfield net (Yen, *et. al.*, [6]). These prototype systems pointed up the need for a more general system to select the

activity classes to be scheduled, the order in which they are considered for scheduling, and which of their strategies are used.

A NEW PARR

After using PARR to develop three major systems and several prototypes, we began to find limitations with our implementation of PARR. Each new project required new strategies, new constraints, and new resource models to support the new constraints. It was more and more difficult to make changes needed to add the new strategies, constraints, and resource models without affecting other parts of the program. PARR also needed ways to add broader kinds of strategies to encompass more than one activity class.

We also gained insight into PARR's interfaces. PARR has two primary interfaces: an interactive interface and an interface using files. Each new system or prototype required a different interactive interface. Some of the differences were dictated by the fact we implemented each system on different hardware platforms and operating systems; others were driven by the user's needs. We saw we would continue to need to adapt the interactive interface to meet the users' needs. The file-based interface also continued to change. New strategies, resource models, and constraints required changes in the formats of the knowledge base files. The prototypes that explored the relationships between the activity classes also needed changes in the file formats. We saw that if the file structures were extensible, old versions of the files would still be usable after new features were added.

We decided that the best approach to these problems was to reimplement PARR in C++. C++ is an object-oriented language based on C, the language used to implement all previous versions of PARR. The objects defined in C++ contain both data and methods, or functions that operate on the data. Each object has a well-defined interface, so interactions between objects are easy to understand and an object can be changed with little impact on other objects. Objects can inherit data, methods, and interfaces from other objects, so new objects can reuse existing objects. Objects also have the ability to provide their own implementations of methods they inherit, giving them the ability to use the same interface to accomplish different tasks.

We found the functionality provided by C++ to be a good match for our needs. Strategies can be represented by an abstract class that describes a set of general methods that are appropriate for any strategy. Each of the specific strategies can be represented by a class derived from the abstract strategy class. Each derived class can implement the general methods in a way that is appropriate for that specific strategy. New strategies can be added by deriving additional classes without modifying the abstract strategy class. These new derived classes can be integrated into the system without modifying the rest of the program. Constraints and resource models can be represented the same way. The more general strategies that handle interactions between activity classes can also benefit from the same representation. C++ and object-oriented techniques also offer solutions to our interface concerns. Different interactive interfaces for different users can share much of their implementation code by using a class library made up of classes which implement the scheduling algorithms.

IMPLEMENTING PARR IN C++

The Solar and Heliospheric Observatory (SOHO) Experimenters' Operations Facility (EOF) Core System (ECS) requires a scheduling system to find and resolve conflicts between the schedules for each of the satellite's eleven instruments. The scheduling system in the EOF needed the type of schedule merging capability provided by one of the prototype systems. This presented the opportunity to redesign and reimplement PARR using object-oriented methods.

When designing or implementing a program using object-oriented methods, the primary organizing construct is the class. Each class describes a type of object. By definition, an object is anything that can be thought about, a concept. In most object-oriented systems, the classes are abstractions of concrete, real-life objects, like spacecraft, stars, or people. Because PARR is a generic scheduling system that can schedule many kinds of things,

PARR's classes represent abstractions of scheduling objects, such as activities, strategies and constraints.

Resource Classes

PARR refers to any data it does not directly schedule as a resource. Resources include both data that PARR cannot change and data that changes as a result of the schedule PARR is creating. In the satellite scheduling domain, spacecraft daylight is an example of a resource PARR cannot change, and available power is an example of data that is changed by the activities PARR schedules.

PARR has different resource models it uses to track different types of resources. Each resource model is represented by a different resource class. PARR implements an abstract class, ResourceClass, from which each of the classes representing a type of resource is derived. ResourceClass has an interface that is appropriate for any type of resource model. Its interface has a method that accesses the resource's value at any given time. It also has a method that confirms that a constraint can be satisfied and one that updates the data to reflect the application of the constraint.

Subscribable is the resource class used to represent data that PARR cannot change. A class called SubscribableResourceClass is derived from ResourceClass to represent these resources. It implements the access method so that it returns the number "one" when the resource is available and the number "zero" when it is not. The method that confirms a constraint checks the value against the constraint. The update method simply modifies bookkeeping data because the data itself cannot be changed.

Limited capacity is one of the resource classes that can be used to model data that is modified as PARR creates a schedule. It is used to model resources that have a maximum amount that can be used at one time, like water supplied through a pipe from a public utility. It is implemented with a class called LimitedCapacityResourceClass. Each instance of this class contains data stating its maximum level of consumption. It updates the access method from its parent class, ResourceClass, to provide the amount of capacity that remains at any specific time. The constraint checking method verifies the constraint against this value and the update method modifies it to reflect the application of the constraint.

Consumable is the third resource class that PARR provides. It is used to model resources that can be consumed and replenished. The gasoline in a car is a good example of a consumable resource. The class that is derived from ResourceClass to implement this model is called ConsumableResourceClass. Instances of this class contain both a maximum capacity for the resource and a base level that is used when no other data is available to initialize the resource availability. The class implements the access method that gives the amount of resource available at any time. The constraints that are used with this resource class are different in that they can also specify the replenishment of the resource. The constraint validation method checks that the constraints that specify consumption do not consume more of the resource than is available. The update method modifies the data to reflect the amount of consumption or replenishment the constraint indicates.

Activity Class

PARR also needs to represent the types of activities that it is to schedule. They are defined in activity classes. Each activity class represents one type of activity that can be scheduled. Activity classes are represented by a class called ActivityClass. ActivityClass acts primarily as a container for data describing its class and for keeping track of when activities of its class have been scheduled. It has methods for input and output, methods that can tell when activities of its class have been scheduled, and methods to update this information when PARR adds or removes specific activities from the schedule. Instances of ActivityClass contain some simple data including the activity class name, priority, and how activities of this class can be shifted. An activity class can be derived from another activity class in the same way a C++ class can be derived from another C++ class. ActivityClass has a reference to the activity class it is derived from, if any, and it has data access methods that automatically obtain any

data that the ActivityClass inherits from its parent. Most importantly, ActivityClass contains a list of constraints and a set of strategies.

Constraints

Constraints represent PARR's conflict avoidance rules. A constraint can state how an activity must be scheduled in relationship to other activities or resources. For instance, a constraint might say that an activity can only be scheduled during spacecraft daylight, or only when no other activity is scheduled. If the resource represents data that the schedule indirectly modifies, the constraint will say how that data is modified. For example, if the resource is water from a pipe, the constraint will state how much water the activity consumes. PARR implements an abstract class, Constraint, to establish an interface for all of its constraints. Constraint establishes methods to validate the constraint for a given activity, and to update the resource data for an activity when the activity is actually scheduled. There is a different subclass of constraint for each resource class, and one that represents the constraints between different activity classes.

SubscribableConstraint is the class that represents constraints between activity classes and resources represented by a subscribable resource class. Each instance of SubscribableConstraint contains a reference to the SubscribableResourceClass the activity is constrained by and the desired state of the constraint. For example, if the constraint stated that an activity could only be scheduled during spacecraft daylight, the SubscribableConstraint would contain a reference to the SubscribableResourceClass representing spacecraft daylight and the value "one", indicating that the resource should be present. Its methods simply call the corresponding methods in the resource class to which it refers.

LimitedCapacityConstraint is the class that represents constraints between activity classes and resources represented by a limited capacity resource class. The instances of LimitedCapacityConstraint each contain a reference to the specific instance of LimitedCapacityResourceClass that represents the desired resource and the amount of the resource that the activity consumes. For example, if the constraint says that an activity consumes 25 gallons of water, the resource class reference would be set to the instance of LimitedCapacityResourceClass that represents water and the amount would be set to 25. As with the SubscribableConstraint, LimitedCapacityConstraint implements its methods by calling the methods in the resource class to which it refers.

Constraints between activity classes and resources represented by a consumable resource class are handled by the class ConsumableConstraint. Its instances contain a reference to the instance of ConsumableResourceClass that models the resource under consideration and the amount of the resource under consideration. Because activities can either supply or consume resources of this class, the instances also contain an indicator telling whether this is a constraint where the resource is being consumed, or whether the constraint is indicating that the activity supplies more of the resource being modeled. For example, if the constraint was being used for an activity that records data to the data tape recorder on a satellite and it requires 25 feet of tape, the resource class reference would be set to the instance representing tape, the amount would be set to 25, and the indicator would be set to "consume." If the constraint was being used for an activity that downlinks data from that same tape recorder and the activity reads back 200 feet of tape, the resource class reference would remain the same, the amount would be set to 200, and the indicator would be set to "supply." As in the other constraints, the methods in ConsumableConstraint are implemented by calling corresponding methods in the ConsumableResourceClass to which it refers.

All of the constraints which represent relationships between resources and activity classes follow a similar pattern. Because the class ActivityClass keeps track of information on when an activity has been scheduled in a way that parallels the way resource classes keep track of resource consumption, ActivityClassConstraint can be implemented the same way that the other classes derived from Constraint are implemented.

ActivityClass constraints represent constraints between two different activity classes. The first of the two activity classes is implied by the location of the constraint within an activity class. Each instance of ActivityClassConstraint contains a reference to the second ActivityClass and an indicator that represents whether the constraining activity class must be present or absent. In other words, if the constraint stated that an activity could only be scheduled when a certain other activity was not scheduled, the constraint would contain a reference to the ActivityClass representing the constraining activity and its indicator would be set to "avoid," indicating that the constraining activity must not be on the schedule during the same period. Its methods call the methods provided by ActivityClass indicating when an activity has been scheduled.

Strategies

Strategies represent PARR's conflict resolution rules. Strategies are used to place activities on the schedule, and to move activities when the constraint checking process discovers conflicts. For instance, a strategy might suggest scheduling an activity when a certain resource becomes available, or putting an activity after an activity it conflicts with. Although many strategies may be appropriate for rescheduling, only certain strategies make sense when initially adding an activity to a schedule. PARR implements an abstract class, Strategy, to provide an interface for classes that implement strategies. Strategy defines methods that take an activity and a list of conflicts and trys to reschedule the activity to avoid the conflicts.

Because only a limited number of strategies can be used to schedule an activity initially, and they require a different interface when used in this way, PARR provides a second abstract class, InitialStrategy, that is derived from Strategy. InitialStrategy provides methods in addition to those defined by Strategy that can be used when initially scheduling an activity. It defines a method that schedules an activity given no other input than the current schedule. This method returns a list of conflicts if the activity cannot be scheduled, and an indicator that tells if the activity should be scheduled again. InitialStrategy also implements the rescheduling method. It reschedules by calling the scheduling method, ignoring the list of conflicts passed into the method. The three different initial strategies that are provided are the start strategy, the stop strategy and the at strategy.

The start strategy tries to schedule an activity when a resource becomes available. For instance, if having a star in view of a satellite is a resource, a start strategy may indicate that an activity is to be scheduled when the resource becomes available, indicating the activity should be scheduled when the star is in view of the satellite. PARR implements a class, StartStrategy, that is used to represent a start strategy in an activity class. Each instance of StartStrategy contains a reference to the ResourceClass that corresponds to the appropriate class. It also has an offset duration that can be set to indicate that the activity should be scheduled the specified amount of time before or after the start of resource availability.

The start strategy is considered a repeatable strategy; the activity will be scheduled every time the resource becomes available. The instances contain a data item that gives the user the ability to indicate a count that makes the strategy skip a number of instances of resource availability before scheduling again. For instance, if the count was set to 3 in the previous example, the system would try to schedule the activity every third time the star came into view. StartStrategy implements the method for scheduling by searching the current schedule for activities of this type. When it finds the last activity of this type, it searches for when the key resource becomes available. If the count is set, it skips the appropriate number of times the resource becomes available. It then attempts to schedule the activity and checks its constraints. If there are conflicts, it returns a list of them. It also returns an indicator that the strategy should be retried so that the scheduling of this resource continues.

The end strategy is quite similar to the start strategy, except that is attempts to schedule at the end of a resource's availability. It is implemented via the class StopStrategy. StopStrategy's implementation parallels

StartStrategy's. Each instance of StopStrategy contains a reference to the ResourceClass corresponding to the key resource, an offset, and a skip count. Its implementation of the schedule strategy is the same as in StartStrategy, except that it looks for the end of a resource's availability.

The at strategy is the simplest of the initial scheduling strategies. It simply schedules an activity at a given time. The class AtStrategy is used to represent this strategy in an activity class. The instances of AtStrategy store the start time that the user specifies. It implements the scheduling method to put the activity at the given time and check for conflicts. The method returns a list of any conflicts it finds and an indicator that the strategy is not to be applied again.

The initial strategies can also be used as alternative strategies to reschedule activities that could not be initially added to the schedule because of conflicts. In addition, PARR also provides several strategies that can only be used as alternative strategies. These alternative strategies include: the before strategy, the after strategy, the next strategy, the prior strategy, the duration strategy, the bump strategy, the activity strategy, the delete strategy and the shift strategy.

The before and after strategies try to resolve a conflict by scheduling an activity either before or after the conflicts. The before strategy is represented by the class BeforeStrategy and the after strategy by the class AfterStrategy. BeforeStrategy implements the rescheduling method to search through the constraints to find the time before all of the conflicts and attempts to schedule the activity at that time. AfterStrategy implements the method using a similar algorithm to find a time after all the conflicts.

The next and prior strategies look for the next or previous time that a given resource becomes available and tries to reschedule the activity at that time. The next strategy is represented by the class NextStrategy and the prior strategy is represented by the class PriorStrategy. Instances of each of these classes contain a reference to the ResourceClass representing the key resource. NextStrategy implements the rescheduling method to start at the time the activity was to be scheduled, search for the next time the key resource becomes available, and attempt to schedule the activity at that time. PriorStrategy implements the method using the same algorithm but searching for the last time the key resource was available.

The duration strategy tries to make an activity fit into the schedule by reducing its duration. The duration strategy is represented by the class DurationStrategy. Each instance of DurationStrategy contains a duration by which the activity can be reduced. It implements the reschedule method by reducing the duration of the activity and adding the activity to the schedule.

The bump strategy is closely related to the duration strategy. Rather than reducing the duration of the activity, it moves its starting time. The bump strategy is implemented by the class BumpStrategy. The instances of BumpStrategy each contain an amount of time by which the start time of the activity can be moved. It implements the reschedule method by changing the start time of the activity by adding the offset and putting the activity into the schedule.

The activity strategy resolves a conflict by scheduling an activity of a different class. The activity strategy is represented by the class ActivityStrategy. Each instance of ActivityClass contains a reference to the ActivityClass that represents the alternative type of activity that should be scheduled. ActivityStrategy implements the reschedule method by calling the schedule method in the InitialStrategy in the alternative ActivityClass. If it encounters conflicts, it calls the reschedule method of each of the alternative strategies in turn until the conflict is resolved.

The delete strategy deletes the activities that are causing the conflict. Before deleting an activity, PARR checks that the activity causing the conflict is of a lower priority than the activity containing the delete strategy. The delete strategy is represented by the class DeleteStrategy. It implements the reschedule method by finding the activities that cause all of the conflicts. After checking that all of them

are of a lower priority than the activity being scheduled, it deletes them. Then it tries again to add the original activity to the schedule.

The shift strategy moves the activities that are causing the conflict. Before moving an activity, PARR checks that the activity causing the conflict is a shiftable activity and is of a lower priority than the activity containing the shift strategy. The shift strategy is represented by the class ShiftStrategy. It implements the reschedule method using an algorithm similar to the one used by DeleteStrategy. It starts by finding the activities that cause all of the conflicts. After checking that all of them are shiftable and of a lower priority than the activity being scheduled, it moves them either before or after where the original activity was to be scheduled. Then it tries again to add the original activity to the schedule.

Paradigms

The strategies described above each describe ways to schedule individual classes of activities. Although the strategies can delete or move activities that cause conflicts with an activity being scheduled or schedule an activity of another class if the desired activity creates conflicts, they cannot indicate which activity classes are to be scheduled or in what order the activity classes are to be scheduled. They also cannot get activities from alternative sources, such as schedules created outside of PARR. PARR handles these types of problems by using paradigms. Paradigms work at a higher level than regular strategies. PARR provides an abstract class called Paradigm that provides an interface for paradigms. The interface is similar to the Strategy interface; it defines a method that takes the action that the paradigm describes. It calls this method "schedule".

Paradigms are a recent addition to PARR, so only a few of them have been implemented. PARR currently provides the activity paradigm, the merge paradigm, and the delete paradigm.

The activity paradigm is used to select an activity class to be added to the schedule. In pervious versions of PARR, this was the only type of paradigm. This paradigm has been extended to accommodate the kind of tactical planning used in SM/PART. The activity paradigm lets the user override any of the data in the activity class, including the initial strategy, and add new constraints and alternative strategies to the activity class. This gives the user the ability to create slight variations on an activity without creating an entirely new class that may only be used once.

The activity paradigm is represented by the class ActivityParadigm. Instances of this class contain a reference to an instance of ActivityClass that represents the type of activity to be scheduled. The instance also contains most of the data contained in an instance of ActivityClass: the priority, the information on how the activity can be shifted, the constraints, and the strategies. This data is used only if it has been provided. It implements the schedule method by calling the schedule method in the InitialStrategy provided either locally or in the ActivityClass. If it encounters conflicts, it calls the reschedule method of each of the alternative strategies from the ActivityClass in turn until the conflict is resolved. If none of them resolve the conflict, it calls the reschedule method of each of the alternative strategies provided locally in turn until one of them resolves the conflict.

The merge paradigm is used to merge schedules created outside of PARR into a single conflict-free schedule. It expects that this externally created schedule is in a file in a simple, pre-defined format. The merge paradigm is represented by the class MergeParadigm. Each instance of this class stores the name of the file that contains the schedule to be merged. It implements the schedule strategy by first reading an activity from the schedule file. It then finds the ActivityClass that represents the activity and uses the start time and duration from the external schedule file to add the activity to the schedule. It uses the constraints from the ActivityClass to validate that the activity causes no conflicts. If it does cause conflicts, it calls the reschedule method of each of the alternative strategies from the ActivityClass in turn until the conflicts are resolved. It then repeats the process for each activity in the external schedule file.

The delete paradigm is used when the schedule is modified interactively. When the

user directs PARR to delete a specific activity from the schedule, an instance of the delete paradigm is used to represent this action. It may seem that the paradigm that was used to schedule the activity could be removed, but most of the paradigms can add many activities to the schedule. The delete paradigm is represented in the system by the class DeleteParadigm. Each instance of this class contains a reference to the ActivityClass that corresponds to the activity to be deleted and the start time and duration of the activity. This information is enough to uniquely identify a specific activity. DeleteParadigm implements the schedule method (although the name schedule is somewhat of a misnomer in this case) by checking that the removal of the activity will not cause any conflicts, and then deleting it from the schedule.

Files

The set of classes used to implement PARR is completed with classes used to organize the other classes into logical sets that can be saved to or read from files. These classes are derived from an abstract class called File. Having a single abstract class representing the main files simplifies writing the part of the user interface that handles saving and loading information from disk. The File class provides an interface defining methods to transfer the information its derived classes contain between memory and disk. The three classes derived from File are ResourceBase, KnowledgeBase and Plan.

A resource base serves as a repository for resource classes. PARR represents this concept with the class ResourceBase. Each instance or ResourceBase holds a list of instances of the class ResourceClass. It provides methods for retrieving the resource classes by name, adding new resource classes, and deleting existing ones.

A knowledge base is a set of information that can be used to reason about a given topic. In PARR, the knowledge bases are used to store the activity classes. PARR provides a class, KnowledgeBase, that contains instances of ActivityClass to represent the knowledge needed to schedule each type of activity. Instances of KnowledgeBase also contain references to one or more instances of ResourceBase. These resource bases provide the resource classes to which the constraints refer. KnowledgeBase implements methods for retrieving the activity classes by name, adding new activity classes and deleting existing ones. It also provides methods to add or remove resource bases and to access the resource classes contained in its resource bases.

The class that is used to combine all of the information in PARR in order to create the final schedule is called Plan. Plan represents a planning problem in its entirety. Each instance of plan contains the starting date and time for the schedule and its duration. It has references to one or more instances of KnowledgeBase that it uses to provide the knowledge needed to schedule activities. Most importantly, it has references to instances of the class Paradigm. These instances represent the tactical plan and the information needed to select the proper activity classes in the correct ways to create the schedule. It has methods to access and update the data it contains, including methods to access the resource classes and activity classes it can access indirectly. Most importantly, it has a schedule method that calls the schedule method of each of its paradigms in turn to create the final schedule.

PARR'S INTERFACES

As stated earlier, PARR has two primary types of interfaces: a file-based interface and an interactive interface. The current version of PARR uses different approaches to make each of these interfaces easier to modify.

The format for PARR's input files has been changed in the current version. The files that represent resource bases, knowledge bases, and plans now allow the free use of spaces, line breaks and comments between any words in the file. The file formats use keywords to help clarify the meaning of each data item. This helps both the user, who is given more information on what the data in the files represents, and the program, which can determine if any pieces of data are missing. Any new types of data that are added to the file formats will be optional, so older files will not need to be changed to accommodate the new format.

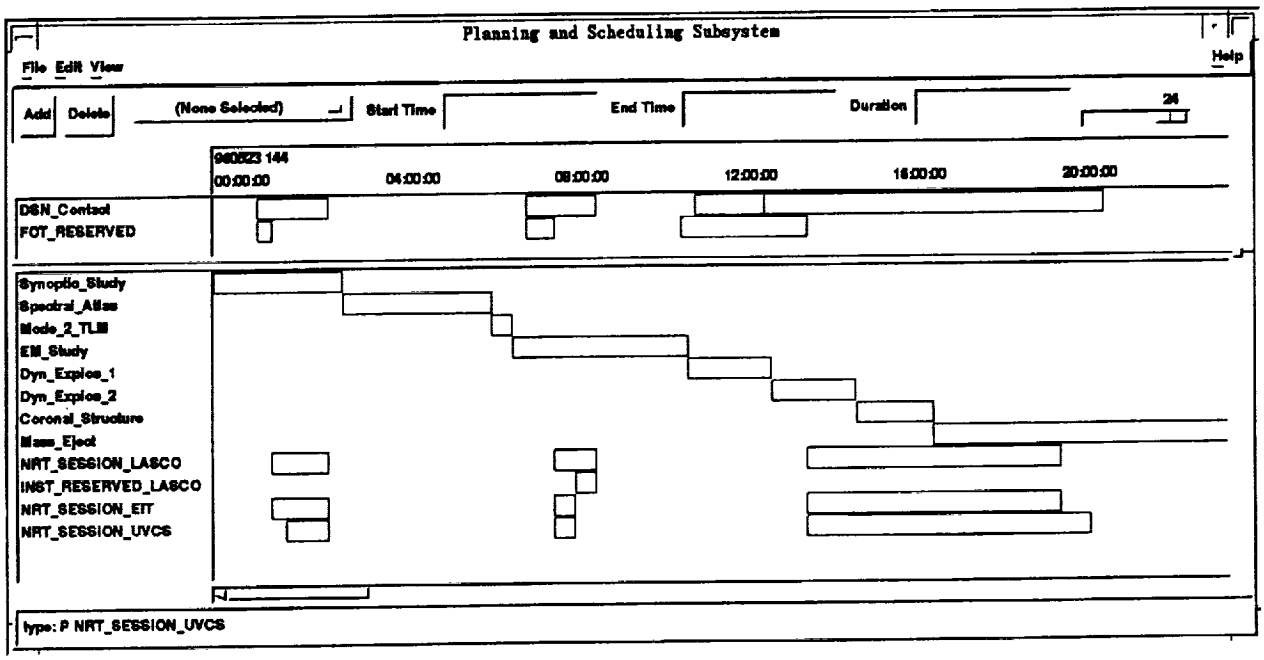

Figure 1. Motif-Style User Interface Display for the PARR Timeline Editor

Each class defined in PARR is responsible for handling its own file data. Every class has methods that can retrieve data from a file, create an instance based on that data, and store an instance to a file. The abstract classes are designed to retrieve data for any class that is derived from them. When derived classes are written, the abstract classes can handle the new data that may appear in a file without modifying their methods.

Because different scheduling systems require different types of interactive interfaces, the classes that implement PARR's scheduling algorithms do not have any interactive interface built into them. They can call outside functions when instances are created, modified or destroyed. They use a mechanism similar to the one used in the X Windows toolkit for accepting functions that are called when their instances change state.

We have developed an interactive user interface to C++ PARR for the SOHO EOF project. It uses the OSF/Motif libraries and a set of X Windows toolkit widgets that we developed in-house. Its primary window is the timeline editor shown in Figure 1. This display lets the user view a graphical timeline that represents the schedule and interactively change it by adding or deleting activities. It also provides a window for editing the activity classes that make up the knowledge base. This window is shown in Figure 2. It displays an activity class textually, but lets the user modify it using the mouse for most of the necessary input.

CONCLUSIONS

The object oriented nature of the new C++ version of PARR allows it to be relatively easily customized to build new planning and scheduling applications. For each new PARR application, the classes of generic objects for resource classes, constraints, and strategies can be supplemented with application-specific types.

In addition, C++ PARR applications can be more easily ported to different environments because the object oriented user interface code has been more completely separated from the algorithmic code.

Finally, the new C++ version of PARR provides paradigm constructs. One of the current paradigms PARR provides lets the user

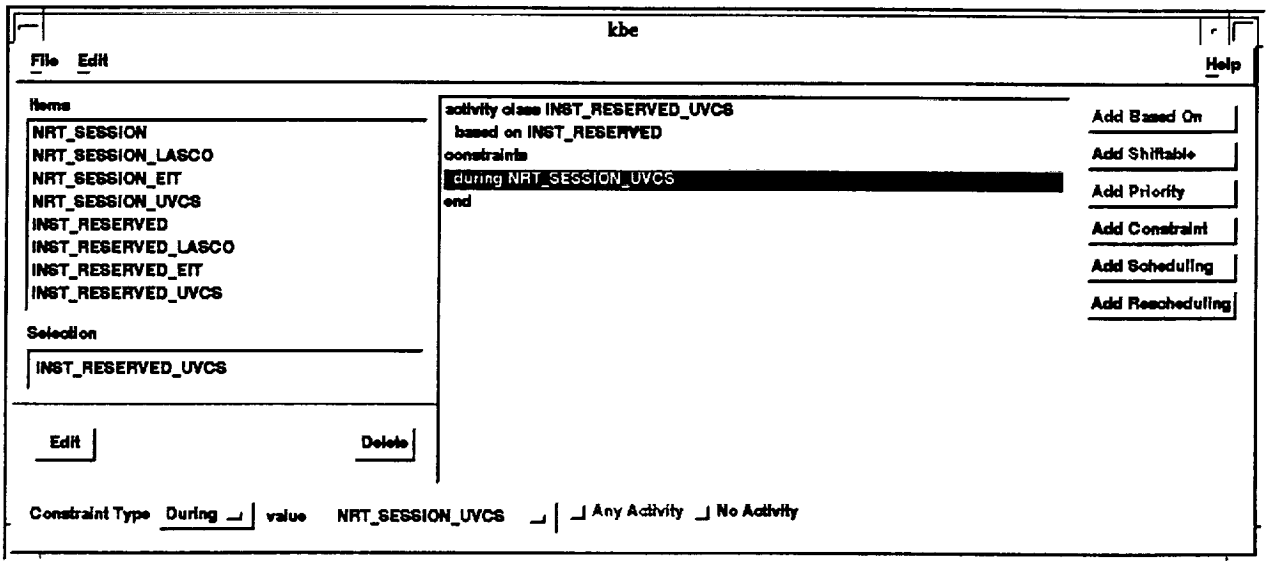

Figure 2. Motif-Style User Interface Display for the PARR Knowledge Base Editor

create variations on the activity classes when creating a tactical plan. Another paradigm provides for the input of schedules created by other systems. The paradigm constructs provide capabilities for extending PARR to handle additional kinds of scheduling problems more easily.

ACKNOWLEDGMENTS

The authors would like to thank Goddard Space Flight Center Code 514, AlliedSignal Technical Services Corporation, and the SOHO Project for their support of this task. This work was supported by NASA Contract NAS5-27772.

REFERENCES

1. McLean, D., Littlefield, R., and Beyer, D., "An Expert System for Scheduling Requests for Communications Links between TDRSS and ERBS," *Proceeding of the 1987 Goddard Conference on Space Applications of Artificial Intelligence (AI) and Robotics,* Goddard Space Flight Center, Greenbelt, MD, May 13-14, 1987.

2. McLean, D., Page, B., Tuchman, A., Kispert, A., Yen, W., and Potter, W., "Emphasizing Conflict Resolution versus Conflict Avoidance during Schedule Generation," *Expert System With Applications* , Vol. 5, Pergamon Press, 1992.

3. McLean, D. and Yen, W., "PST & PARR: Plan Specification Tools and a Planning and Resource Reasoning Shell for use in Satellite Mission Planning," *Proceedings of the 1989 Goddard Conference on Space Applications of Artificial Intelligence,* Greenbelt, MD, 1989.

4. McLean, D., Page, B., and Potter, W., "The Explorer Platform Planning System: An Application of a Resource Reasoning Planning Shell," *Proceedings of the First International Symposium on Ground Data Systems for Spacecraft Control,* Darmstadt, Germany, 1990.

5. Bogovich, L., Johnson, J., Tuchman, A., McLean, D., Page, B., Kispert, A., Burkhardt, C., and Littlefield, R., "Using AI/Expert System Technology to Automate Planning and Replanning for the HST Servicing Missions," *Proceedings of the 1993 Goddard Conference on Space Applications of Artificial Intelligence,* pp. 3-10, Goddard Space Flight Center, Greenbelt, MD, May 10-13, 1993.

6. Yen, W., and McLean, D. "Combining Heuristics for Optimizing a Neural Net Solution to the Traveling Salesman Problem," *Proceedings of the First International Joint Conference on Neural Networks,* San Diego, CA, June, 1990.