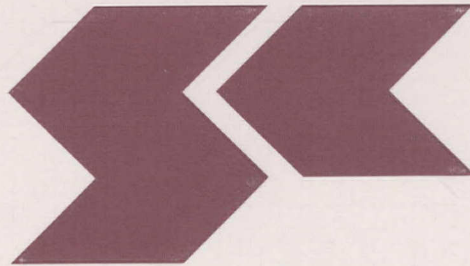
RELEASABLE
DOC-CR-SP-89-004



Software Kinetics

**USERS GUIDE TO VERSION 2
OF THE
CASE BASED PLANNING SYSTEM**

Prepared for: Dept. of Communications
Ottawa, Ontario

SKL Document #1500-19-004.01.0
Copy #5 31 March 1989

TL
797
U74
1989

RELEASABLE
DOC-CR-SP-89-004

TL
797
U74
1989

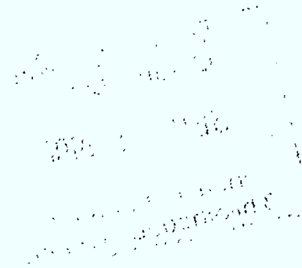
Industry Canada
Library Queen
SEP 16 1998
Industrie Canada
Bibliothèque Queen

COMMUNICATIONS CANADA
JAN 25 1990
LIBRARY -- BIBLIOTHÈQUE

USERS GUIDE TO VERSION 2 OF THE CASE BASED PLANNING SYSTEM

Prepared for: Dept. of Communications
Ottawa, Ontario

SKL Document #1500-19-004.01.0
Copy #5 31 March 1989



TL
797
U74
1989

DD 9319144
DL 9330485

USERS GUIDE TO VERSION 2

OF THE

CASE BASED PLANNING SYSTEM

Contract #36001-8-3580

31 March 1989

Prepared for:

Department of Communications
Ottawa, Ontario

Prepared by:

Software Kinetics Ltd.
65 Iber Road, P.O. Box 680
Stittsville, Ontario Canada
K0A 3G0

Software Kinetics Document #1500-19-004.01.0

Document Approval Sheet

for the

USERS GUIDE TO VERSION 2

OF THE

CASE BASED PLANNING SYSTEM

Document No: 1500-19-004.01.0

Document Name: Users Guide to Version 2 of the Case Based Planning System

Approvals	Signature	Date
Software Specialist:	Stephen Northover S. Northover	31 March 89
Project Manager:	David A Moss D. A. Moss	31 March 1989

Document Revision History

Revision	Reason for Changes	Origin Date
01	New Document Issued	31 March 1989

Government of Canada | Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT **DOC-CR-SP-89-004**

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: Users Guide to Version 2 of the Case Based Planning System

AUTHOR(S): S. F. Northover

ISSUED BY CONTRACTOR AS REPORT NO: 1500-19-004

PREPARED BY: Software Kinetics Ltd.
65 Iber Road,
Stittsville, Ontario,
K0A 3G0.

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 36001-8-3580

DOC SCIENTIFIC AUTHORITY: Peter Adamovits
Computer and Intelligent Systems Group,
Space Mechanics Directorate.

CLASSIFICATION: Releasable

This report presents the views of the author(s). Publication of this report does not constitute DOC approval of the reports findings or conclusions. This report is available outside the department by special arrangement.

DATE: March 31, 1989

Adaptive antennas TK 7871.6
Algorithms - QD 76.27/2

Industry growth Z 699.
~~Bases~~ ~~Banques~~ Banques de données - Canada - Commerci
Droit d'auteur et informatique - Canada
~~Bases de données - Gestion~~

Banques de données - Canada -
Commercialisation

Users Guide to Version 2 of the Case Based Planning System

Author: Stephen Northover (Software Kinetics)
Contract: The Evaluation of a CBPS with Respect to MSS Applications (1500-19)
Date Prepared: March 31, 1989

Executive Summary

This document describes the implementation and operation of version 2 of the Case Based Planning System (CBPS) based on the conceptual design developed at Carleton University by D.L. Deugo and F. Oppacher [2]. Version 2 of the CBPS was implemented in the computer languages Smalltalk and PROLOG from the original specification of the CBPS [2]. Version 2 of the Case Based Planning System consists of a user environment intended for the development of domain independent planning applications.

During development, a simplified test environment from the domain of power management on an orbiting spacecraft as described by Adamovits [4] was used as a test bed. Simple planning problems in the domain of power management have been solved.

The concept of a planning environment has been introduced to allow for the representation of domain dependent knowledge in a domain independent manner. The use of task and plan rules allows the domain independent representation of relationships within tasks and between tasks in a plan.

Special attention has been taken with respect to program modularity and the separation of user-interface from planning code with the view to facilitate future enhancements.

Table of Contents

	Page
1. Introduction	4
2. The CBPS Objects	5
2.1 Tasks, Plans, and the Plan Library	5
2.2 The CBPS Object	12
2.2.1 Specification of a Plan	14
2.2.2 Plan Selection	15
2.2.3 Plan Execution	17
2.2.4 Replanning	19
2.2.5 Plan Evaluation	21
3. The CBPS User Interface Objects	23
3.1 The CBPSBrowser	23
3.2 The PlanBrowser	26
3.3 The LibraryBrowser	28
4. Summary	32
References	
Glossary of Terms	
Appendices	
Appendix A : PROLOG Predicates for Task and Plan Rules	
Appendix B : Sample Demonstration	
Appendix C : The CBPS Planning Classes	
Appendix D : Source Code	

Section 1 - Introduction

This document describes the implementation and operation of version 2 of the Case Based Planning System. It is assumed that the reader is familiar with case based reasoning, Smalltalk and PROLOG, as well as the work of D.L. Deugo and F. Oppacher in the area of case based planning. For a more information on planning systems and case based reasoning see references [1], [2], and [3]. Version 1 of the CBPS, developed by D.L. Deugo [3], bears little resemblance to the system described here. Version 1 was designed to demonstrate concepts and lacked the mechanisms required to model a reasonably complex problem. As robustness and ease of modification were not features of version 1 of the CBPS, it was decided to discard version 1 and reimplement the CBPS from the original design.

Section 2 provides a description of the objects that implement case based planning in version 2 of the CBPS. A discussion of the objects that provide a user interface to the CBPS can be found in section 3. Section 2 begins with a brief review of case based reasoning in order to provide background information for the following subsections.

Section 2.1 describes the concepts of a task, plan and the plan library as implemented in version 2 of the CBPS. These are the basic objects that the CBPS manipulates from the users point of view.

Section 2.2 describes the CBPS object. The CBPS object performs all of the activities normally associated with case based reasoning in version 2 of the CBPS. Section 2.2.1 describes the *specification of a plan*, often referred to as *operator input*. The *specification of a plan* is used to drive the plan selection process. Section 2.2.2, 2.2.3, 2.2.4, and 2.2.5 detail the Plan Selection, Plan

Execution, Replanning, and Plan Evaluation subsystems of the CBPS respectively.

Section 3 provides a description of the objects that implement the user interface to version 2 of the CBPS.

Section 3.1 describes the operation of the CBPSBrowser. The CBPSBrowser provides the user interface to the CBPS object as described in section 2.2. The CBPSBrowser allows the user to manipulate a CBPS object and perform case based reasoning with that object.

Section 3.2 describes the PlanBrowser object. The PlanBrowser object is used to view and edit plans but more importantly allows the input of the *specification of a plan.*

Section 3.3 discusses the LibraryBrowser. The LibraryBrowser allows the user to create and edit plan libraries. A description of the browser and its functionality is provided.

Section 4 is a brief conclusion that touches on the major topics covered by this document.

Section 2 - The CBPS Objects

Introduction

The approach that case based planning systems take to planning is case based and makes use of dynamic memory techniques [1, 2]. With respect to the CBPS, this means that based on planning requirements, plans are selected from a plan library. The plan library consists of a collection of predetermined plans that are expected to work in certain situations with some known level of success. If no plan is available to match the requirements, a plan that partially matches may be modified or an entirely new plan constructed to meet the current requirements. After the plan selection, creation and modification process has occurred, the plan is executed. During plan execution, success and failure information is gathered for the plan. Using past history information for each plan, new planning requests may be better satisfied in the future. Over time, the CBPS should adapt to its surroundings by learning to use the most suitable plan in a given situation.

The following section describes the basic objects used to implement version 2 of the CBPS. Users that are familiar with these objects and wish to explore the user interface to version 2 of the CBPS are encouraged to skip this section and proceed to section 3. For users that wish to see a quick demonstration of the CBPS, Appendix B provides a step by step guide that solves one particular planning problem.

2.1 Tasks, Plans and the Plan Library

This section describes the implementation and functionality of tasks, plans and the plan library. The user is expected to be familiar with the concepts of a task

and a plan as described in reference [2]. A detailed description will not be provided here.

The task, plan and plan library objects are the basic building blocks of the CBPS. It is important that the user have a clear understanding of how these objects function before attempting to use the Case Based Planning System. A LibraryBrowser is provided as a means of creating and editing tasks, plans and the plan library. A detailed discussion of the LibraryBrowser is deferred to section 3.3.

Tasks

Every task has a name, a start time, a duration, resources to be acquired and released during execution, and some task rules to represent relationships internal to the task. Tasks are stored in an OrderedCollection [5] within a Plan. Plans will be discussed in detail in the next section. Figure 1 shows a typical task.

Name	*task1*	
Start Time	*?*	
Duration	*10*	
Resources	*Power*	*20*
Rules	task1 (task1) :- between (1, 20, start) task (task1, start, 10) ... etc	

Figure 1 - Task "task1"

The task **name** is expected to follow the Smalltalk/V convention for valid variable names [5]. It must begin with a lower case letter and contain only letters and digits.

The **start time** and **duration**, when provided, are numbers that represent units of

time along the planning horizon. When asked to execute, the task will expect to start at its start time and last as long as its duration. It is also possible to leave these values unspecified. A more detailed discussion of unspecified values is deferred until section 3.2.

Resources are named values that use numbers to specify the quantity of the resource that is required by the task when it executes. For example, a resource named "Astronaut", with some specified qualifications, that has a value of "2" means that the task requires two astronauts before it can start. When the task starts, it will acquire the two astronauts from the available astronauts in the environment. If none are available, the task will not start. When the task ends, the two astronauts are released back to the environment for use by other tasks.

Task rules are expressed in the form of PROLOG predicates [6]. Task rules can express any relationship internal to the task but are expected to provide numerical bounds on the start time and duration of the task. The bounds for start time and duration usually take the form of a range of acceptable values. These ranges are normally expressed using the "between" and "member" PROLOG predicates. A more complete description of the available PROLOG predicates for use with task rules can be found in Appendix A.

Task rules must have the same name as the corresponding task. It is essential that this naming convention is not violated. The task name is used to locate the PROLOG predicate when the predicate is required. Each predicate must take exactly one parameter. When the predicate is executed, this parameter will be the PROLOG representation of the task. For example, the following is a task rule for a task named "task54":

```
task54 (task54) :-
     member ([0, 20, 40, 60], start),
     between (30, 40, duration),
     task (task54, start, duration).
```

This rule states: *"Task54 must start at time 0, 20, 40, or 60. It has a duration that varies between 30 and 40 time units."*. The first two statements specify a range for the values of the start time and duration of the task. The last statement uses a special predicate called "task" to unify these values with the parameter "task54". Appendix A describes the "task" predicate in detail.

Task rules are used by the CBPS to both verify and generate values for the start time and duration of a task. When a start time or duration is specified for the task, task rules are used to ensure that this value is valid. In the example above, the PROLOG variable "task54" might have a start time of 20 and a duration of 35 when the predicate is evaluated. The last statement in the example will check the values of the variables "start" and "duration", as generated by the back tracking of the "between" and "member" predicates, against the start time and duration found in the variable "task54". Task rules used in this context will verify the correctness of the start time and duration of the task.

When a start time or duration is not specified, task rules are used to generate valid candidates for the missing value. In the above example, start time and duration in the PROLOG variable "task54" would be unbound when the rule was evaluated. The values of "start" and "duration" generated by the "between" and "member" predicates would then get bound to the start and duration required by the variable "task54". On back tracking, new values are generated. Values for start and end time can be used to fit the task into a plan. Task rules that generate values schedule the task within the plan.

Plans

Every plan has a unique name with respect to other plans in the plan library, a start time and end time that are automatically calculated, a collection of tasks that are to be executed, an execution history, and plan rules to specify relationships between tasks in the plan. Plans are stored in an OrderedCollection that is used by

the CBPS to represent the plan library. The plan library will be discussed in detail in the next section. Figure 2 shows a typical plan.

Name	*plan9*
Start Time	*5*
End Time	*61*
Tasks	#(task1 task27 task54 task100)
History	10 success(s), 3 failure(s)
Rules	plan9 ([task1,task27, task54, ...) :- overlaps (task1, task100), ... etc

Figure 2 - Plan "plan9"

The plan **name** is expected to follow the Smalltalk naming conventions for variable names. It must begin with a lower case letter followed by any number of letters or digits.

Plan **start times** and **end times** can not be directly manipulated by the user. It is the start time of the first task and the end time of the last task that define the start and end time of the plan.

The collection of **tasks** in the plan consists of tasks as described in section 2.1. It is possible to add and delete tasks from this collection. Plan start and end times are recalculated every time a task is added or deleted from the plan.

The Plan **history** can not be directly manipulated by the user. It is updated when the plan executes by the Plan Execution module of the CBPS as described in section 2.2.3. The purpose of the Plan history is to record success and failure information for the plan.

Plan rules are used to order tasks within a plan. Plan rules, like task rules, are expressed in the form of PROLOG predicates. The name of the predicate must be the same as the name of the plan. This name is used to locate the predicate when required. Plan rules take exactly one parameter. This parameter is a list of every task in the plan. The order of the tasks in this list is immaterial. For example, a plan rule for a plan called "plan9" that has 4 tasks named "task1", "task54", "task100", and "task27" could be:

```
plan9  ([task1,  task27,  task54,  task100])  :-
       overlaps  (task1,  task100),
       distinct  (task27,  task54),
       precedes  (task27,  task54),
       task  (task1,  _,  _,  end1),
       task  (task54,  end1,  _).
```

This rule states: *"Plan9 is a plan that contains task1, task27, task54, and task100. Task1 and task100 overlap. This implies that some portion of task1 must happen at the same time as task100. Task27 and task54 are distinct and therefore do not overlap. Task27 starts before task54 and the end time of task1 is the same as the start time of task54."*

Most of the predicates in the above expression are self explanatory with the exception of the last two. The statement "task (task1, _, _, end1)" extracts the end time of "task1" and places it in the variable named "end1". The statement "task (task54, end1, _)" compares the end time extracted from "task1" with the start time that would be extracted from "task54" with the variable "end1" for equality. A more complete definition of the available PROLOG predicates for use with plan rules can be found in Appendix A.

Plan rule predicates involving tasks that are no longer present in a plan will always succeed. This is a feature of the implementation. For example, if "task100" was removed from "plan9" in the above example, the predicate that forces "task1" to overlap "task100" will always succeed. This has the effect of removing the restriction that "task100" and "task1" overlap.

A LibraryBrowser (section 3.3) can be used to edit the plans, tasks, and plan rules. The head of the plan rule is automatically generated and updated when tasks are added and deleted. The user is only required to enter the body of the rule. For example, one need never type in "plan9 ([task1, ...". It is sufficient to type "plan9 () :- ..." and then the rule body. When the rule is saved, the rule head will be generated automatically to contain a list of every task in the plan. A restriction on the user of the LibraryBrowser is that the plan is not verified when it is entered into the plan library. The LibraryBrowser assumes that only valid plans will be added to the library. A complete description of the LibraryBrowser can be found in section 3.3.

The Plan Library

The plan library is stored as an OrderedCollection of plans. The plan library is assumed to contain only valid plans. This means that the plan verification process as performed by the Plan Selection module of the CBPS (section 2.2.2) should always succeed on these plans. Task and Plan rules for the plan should never be violated for a plan that is a member of the plan library.

The LibraryBrowser is an object that adds, deletes and updates plans stored in a plan library. A complete description of the LibraryBrowser can be found in section 3.3.

2.2 The CBPS Object

The CBPS (Case Based Planning System) object accesses and coordinates the activities of each of the four subsystems of version 2 of the CBPS. These subsystems are the Plan Selection module, the Plan Execution module, the Replanning module, and the Plan Evaluation module. Figure 3 shows the

relationship of these modules within a CBPS object. For a more detailed description of the behaviour of each of the modules, see reference [2].

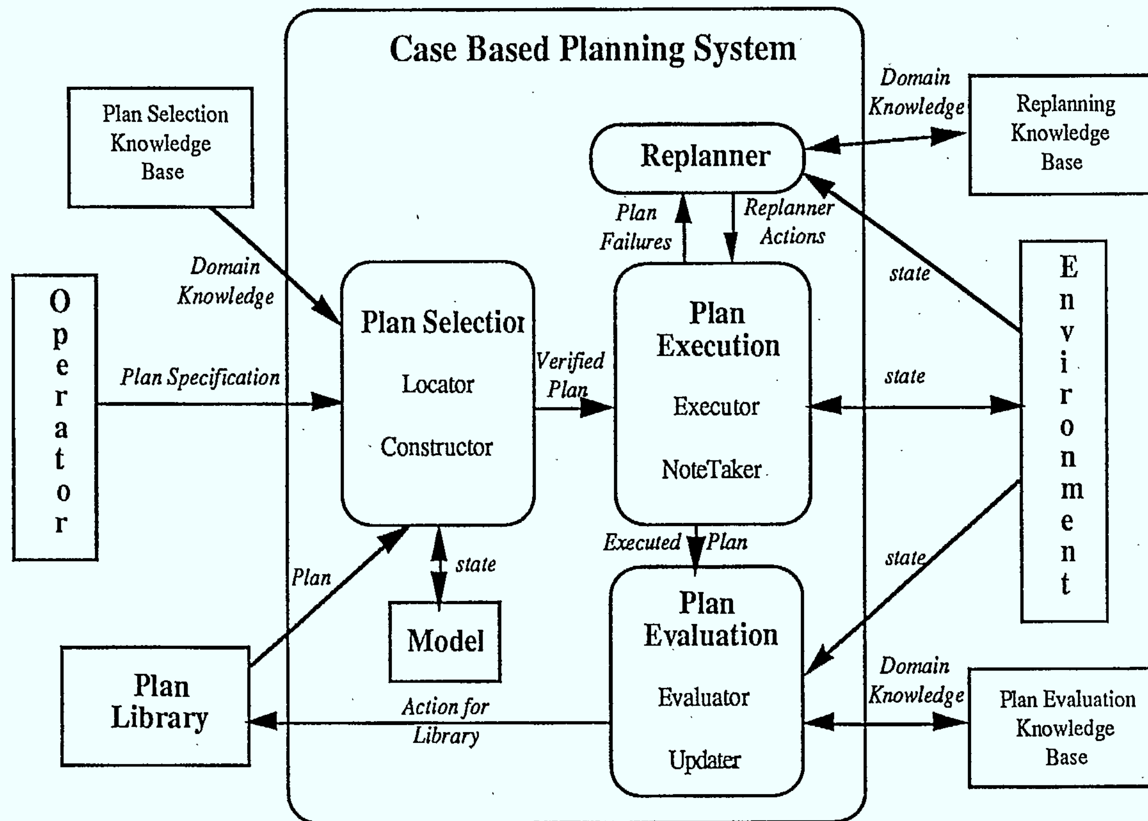


Figure 3 - The Case Based Planning System

A CBPS object accesses the current plan, the current plan library (perhaps created by the LibraryBrowser), the environment of execution, a Selector object to select and verify the plan, an Executor object to execute the plan and gather execution statistics, and an Evaluator object to evaluate these statistics and update the plan library. The Selector object uses a model of the environment to simulate plan execution as part of the plan verification process.

A CBPSBrowser is provided as a means of viewing and manipulating CBPS objects. The CBPSBrowser provides a windowing interface to a CBPS object and is used to manipulate the environment and invoke all the functions of a CBPS object. The CBPSBrowser allows the user to watch the Case Based Planning System in action. The CBPSBrowser will be discussed in detail in section 3.1.

2.2.1 The Specification of a Plan

The specification of a plan is perhaps the most important part of the Case Based Planning System from a users perspective. The specification of a plan is the mechanism that the user employs to tell the CBPS what to do. It is this specification that drives the other modules of the CBPS.

When the user wishes to plan an activity, he specifies what to do by supplying the tasks he wants to execute subject to certain constraints. For example, he may wish to execute "task27" and "task54" such that "task27" starts some time before "task54". He may wish to assign particular start times, durations and resource requirements for these tasks or let the CBPS assign these values.

The specification of a plan in the CBPS takes the form of a special kind of plan called an "unordered" plan. An "unordered" plan is simply a plan that may or may not bear some resemblance to any of the plans in the plan library. An "unordered" plan can contain any number of tasks. The start times, durations and resource requirements of these tasks may or may not be specified. Plan rules can be used to indicate how tasks in the "unordered" plan interrelate. The tasks themselves may be copies of existing tasks or completely new tasks.

An "unordered" plan can best be described as a plan that requires the services of a Selector object to make it ready for execution. A PlanBrowser is provided for creating and editing "unordered" plans. The PlanBrowser is described in section 3.2.

2.2.2 Plan Selection

Plan Selection is performed by the Selector object. Figure 4 shows the functionality of the Selector. A Selector object requires the current environment to access available resources and extract the "unordered" plan and an Executor to be used during the plan verification process. The Selector object performs both the actions of the Locator and Constructor subsystems [2]. There are no explicit Locator or Constructor objects in version 2 of the CBPS. A detailed description of the Selector can be found in reference [2].

The introduction of a Plan Selection Knowledge Base, as implemented by the SelectorRules object, allows domain dependent knowledge to enter into the plan selection process. This knowledge base was not included in the design or implementation of version 1 of the CBPS.

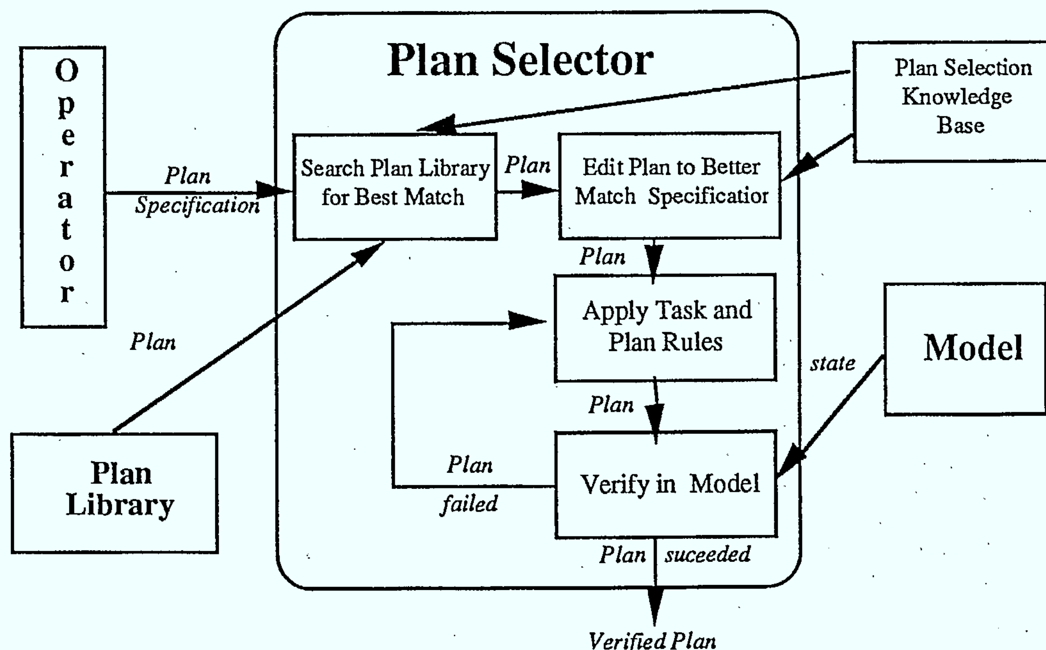


Figure 4 - The Plan Selector

The Locator portion of the Selector is responsible for selecting a plan from the plan library that best matches the operators requirements as specified by the "unordered" plan. The Constructor portion of the Selector is responsible for taking the plan from the Locator and massaging it to better match the requirements of the operator.

The Locator uses the same criteria to locate plans that was used in version 1 of the CBPS. However, these criteria are implemented, in a domain independent manner, as PROLOG predicates in the class SelectorRules. When the plan library is searched, a 3-tuple predicate named "better" is invoked in the SelectorRules object with parameters supplied by the Locator. The first two parameters are plans, the third parameter is the current environment. The "better" predicate succeeds if the first plan is a better match than the second plan with respect to the "unordered" plan and the environment. Using the "better" predicate, domain dependent criteria for matching plans can be introduced into the Locator, in a domain independent manner.

The Constructor will add and remove tasks in a manner similar to version 1 of the CBPS but mechanisms are in place to disallow this practice if necessary. For some plans, removing or adding tasks may be inappropriate. When the Constructor wishes to remove a task from a plan, a 2-tuple predicate named "canBeRemoved" is invoked in the SelectorRules object. The first parameter in the predicate is the task to be removed and the second is the current environment. If the "canBeRemoved" predicate succeeds, the Constructor will remove the task. A similar predicate called "canBeAdded" is invoked when the Constructor wishes to add a task. These two predicates allow domain dependent criteria for modifying plans to be introduced into the Constructor, in a domain independent manner.

Access to the SelectorRules object is provided by the LogicBrowser. The LogicBrowser can be used to edit the plan selection rules stored in the SelectorRules object. The LogicBrowser is discussed in reference [5].

The approach to plan verification used by the Constructor is completely different from the approach used by version 1 of the CBPS implementation. The verification process involves first satisfying plan and task rules, and then simulating the execution of the plan in a model of the environment. Any failures in the simulated execution of a task do not invoke the Replanner as did the version 1 of the CBPS [3]. Instead, the Constructor will try different candidate values for the start time and duration for each of the tasks until the simulation of plan execution completes without failure. Replanning, in version 2 of the CBPS, is the function of the Replanner object (section 2.2.4) and is only performed when the plan executes. Unlike version 1 of the CBPS, version 2 is capable of constructing and verifying a completely new plan using plan and task rules and a model of the environment.

The Selector object answers a plan that is ready to execute or the *nil* object [5] if the plan could not be verified. A plan can fail verification for two reasons:

1) If task or plan rules are specified such that they can never succeed, plan verification can never succeed. For example, a plan rule that states *"task1 must follow task2 and task2 must follow task1"* will always cause plan verification to fail.

2) If an ordering for tasks within a plan cannot be found such that environmental resources are available when needed, plan verification will fail. For example, a task that requires 2 astronauts can never execute in an environment where only one astronaut is ever available.

2.2.3 Plan Execution

Plan Execution is performed by the Executor object. Figure 5 shows the functionality of the Executor. Task success and failures are recorded by the NoteTaker [2] object as they occur during task execution. The plan history found in every plan is actually an instance of a NoteTaker that records execution

information for the plan. A detailed description of the Executor can be found in reference [2].

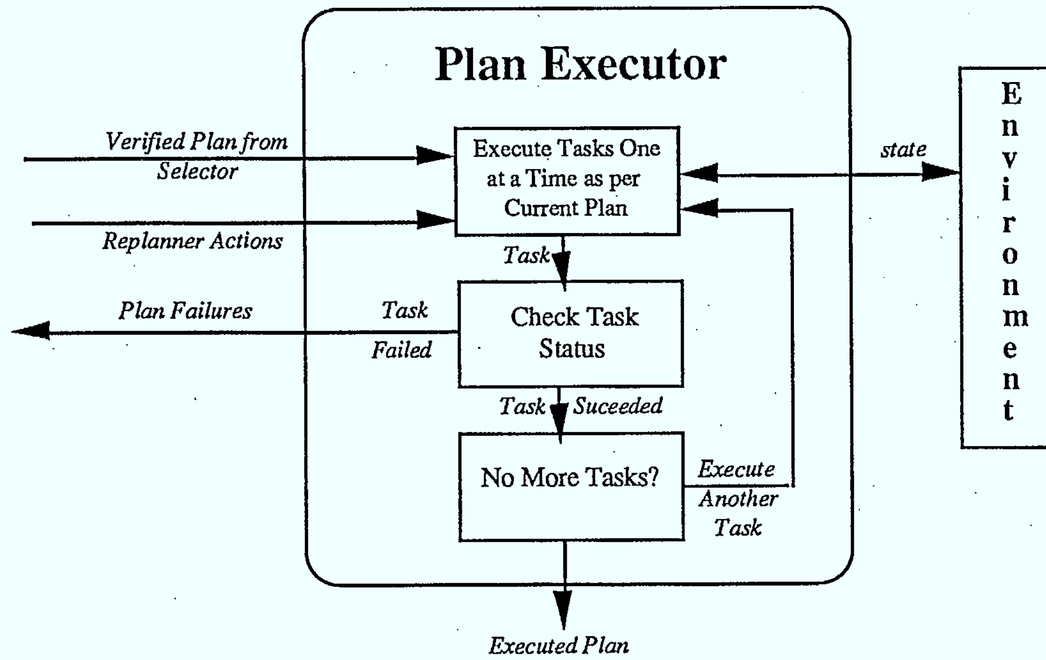


Figure 5 - The Plan Executor

In order to begin execution, the Executor requires a plan and an environment. The Executor keeps track of the task that is about to execute, the tasks that have successfully executed and the tasks that are currently executing. When a task fails to start, the Replanner object is invoked with the name of the resource that was not available, the task that requested this resource, and the Executor itself. By supplying the Executor as a parameter, the Replanner has the opportunity to access information stored in the Executor. This information could be in the form of the current environment or the collection of tasks that are currently executing. Details of the Replanner are presented in the next section.

2.2.4 Replanning

The functionality of the Replanner is provided by the Replanner object. Figure 6 shows the behaviour of the Replanner. A detailed description of the Replanner can be found in reference [2].

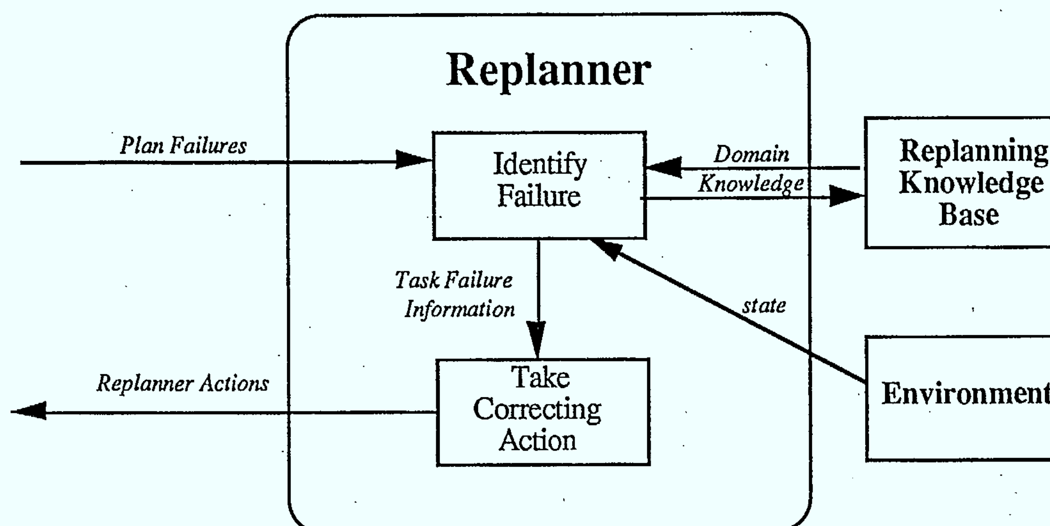


Figure 6 - The Replanner

The job of the Replanner is to perform an action that will allow the plan to continue after a task failure. To do this, a 3-tuple PROLOG predicate named "failure" is invoked in the Replanner object with parameters supplied by the Executor. The first parameter is the name of the resource that could not be acquired. The second parameter is the task that failed. The third parameter is the Executor that attempted to execute the task.

The following rule will retry a failed task one minute later in the plan, if the failed task is named "task54" and it was unable to acquire enough of the resource named #Power from the environment:

```
failure (#Power, task, executor) :-
     is (#task54, task name),
     is (_, executor scheduleTask: task
        atTime: executor time + 1).
```

Currently, The Replanner has a choice of telling the executor to drop the current plan, drop the current task, or reschedule the failed task at some future time. These actions are implemented as the Smalltalk messages #scheduleTask:atTime:, #dropTask: and #dropPlan to be sent to the Executor object.

Because the Executor object contains the current environment as well as task execution information, replanning rules can easily be made more sophisticated than those in the example. Using the special PROLOG predicate "is", one can execute any Smalltalk expression and have the answer imported back into PROLOG. For example, the following expression could be part of a task failure rule:

```
...
is (1, executor environment resources at: #Satellite),
is (true, executor executedTasks includes: task15),
...
```

These two statements will succeed if the environment contains exactly one satellite and the executor has successfully executed "task15".

Presently, any actions taken by the Replanner are not automatically verified by the Replanner or the Executor. It is therefore possible to reschedule a failed task at a time that is not valid for the task or at a time that violates a plan rule. As the Replanner is expected to supply an action quickly, the often lengthly process of plan verification is deferred until plan evaluation (section 2.2.4). The Replanner is primarily intended to provide a "quick fix" in order to get the plan back on its feet. A more sophisticated approach to replanning could be added at a later date.

Access to the Replanner is provided by the LogicBrowser. The

LogicBrowser can be used to edit failure rules for the Replanner class. The LogicBrowser is discussed in reference [5].

2.2.5 Plan Evaluation

An Evaluator object performs the process of plan evaluation as described in reference [2]. Figure 7 shows the relationship of the components within an Evaluator. Updating the library is done by the Evaluator object. There is no Updater object, as described in reference [2], in version 2 of the CBPS. For more information about the Evaluator, see reference [2].

The introduction of a Plan Evaluation Knowledge Base, as implemented by the EvaluatorRules object, allows domain dependent knowledge to enter into the plan evaluation process. This knowledge base was not included in the design or implementation of version 1 of the CBPS.

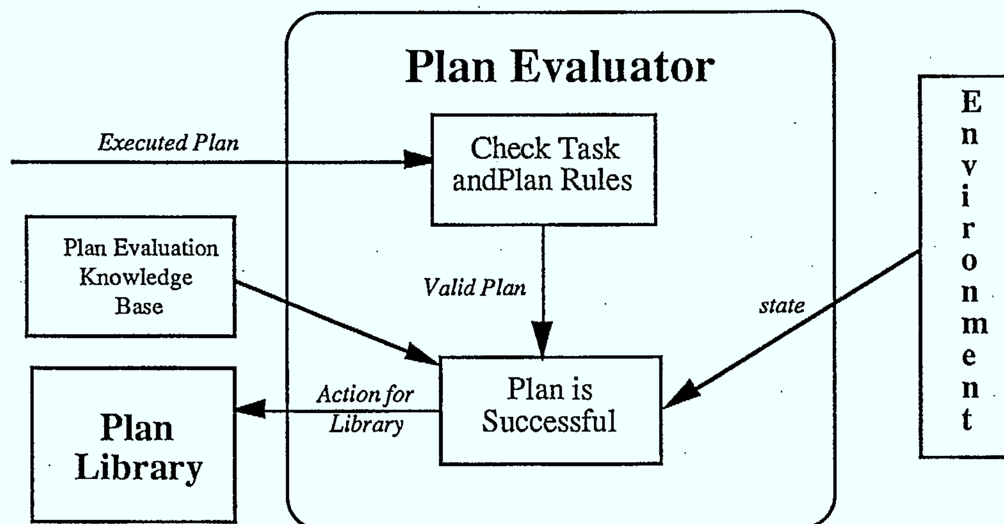


Figure 7 - The Evaluator

The Evaluator requires the executed plan, the plan library, and the current environment. Presently, the state of the environment is not considered when determining the success of the executed plan but is provided for future use. For example, a perfectly good plan with no task or plan rule violations used to capture and repair a satellite could fail miserably if the satellite could not be captured. In this case, the environment could be examined to see if the satellite was captured and a decision to remove the plan from the library could be overturned.

The first action of the Evaluator is to verify that plan and task rules have not been violated during plan execution or replanning. This differs from the implementation of the Evaluator in the version 1 of the CBPS [3]. In version 2 of the CBPS, when a task or plan rule violation was detected, the evaluation process is instantly aborted. A plan that contains tasks that violate any rules for the plan should never be added or updated in the plan library. Such a plan could never pass the verification process of the Selector.

The criteria for plan evaluation are similar to those used by version 1 of the CBPS [2]. The execution history as recorded by the NoteTaker [2] object is used to determine the worth of the plan. Currently, environmental factors are not considered in the decision. However, these criteria are implemented, in a domain independent manner, as PROLOG predicates in the class EvaluatorRules. When a plan is to be evaluated, a 3-tuple predicate called "evaluate" for the EvaluatorRules object is invoked. The first parameter is the plan to be evaluated. The second parameter is the plan library. The last parameter is the evaluator object.

Currently, the evaluate predicate has a choice of telling the evaluator to add the plan to the library, forget the plan, remove the plan from the library, or update the plan in the library. These actions are implemented as the Smalltalk messages #addPlan, #forgetPlan, #removePlan and #dropPlan to be sent to the Evaluator object.

Access to the EvaluatorRules object is provided by the LogicBrowser.

Section 3 - The CBPS User Interface Objects

Introduction

The following section describes the objects used to implement the user interface to version 2 of the Case Based Planning System. Users that are not familiar with the basic CBPS objects described in section 2 should read that section before proceeding. For a quick demonstration of the CBPS, see Appendix B.

3.1 The CBPSBrowser

The CBPSBrowser provides a user interface to a CBPS object. The CBPSBrowser is the main user interface to version 2 of the Case Based Planning System. Figure 8 shows a typical CBPSBrowser. A CBPSBrowser is created by the following expression:

```
CBPSBrowser new openOn: (CBPS new).
```

The expression "CBPSBrowser example" will open a CBPSBrowser that uses the plan library returned by the expression "CBPS exampleLibrary". A default "unordered" plan and a default environment with an available resource called "Power" is also provided. It is a good idea to explore the operation of the CBPS object and CBPSBrowser using this example.

Within the CBPSBrowser, there are 6 subpanes. Four of these panes are labelled "Plans", "Env Vars", "Env Values" and "Status". The other 2 unlabelled panes are used to display tasks and resource usage of the current plan, over the start and end time of the plan. The two unlabelled panes are for output only.

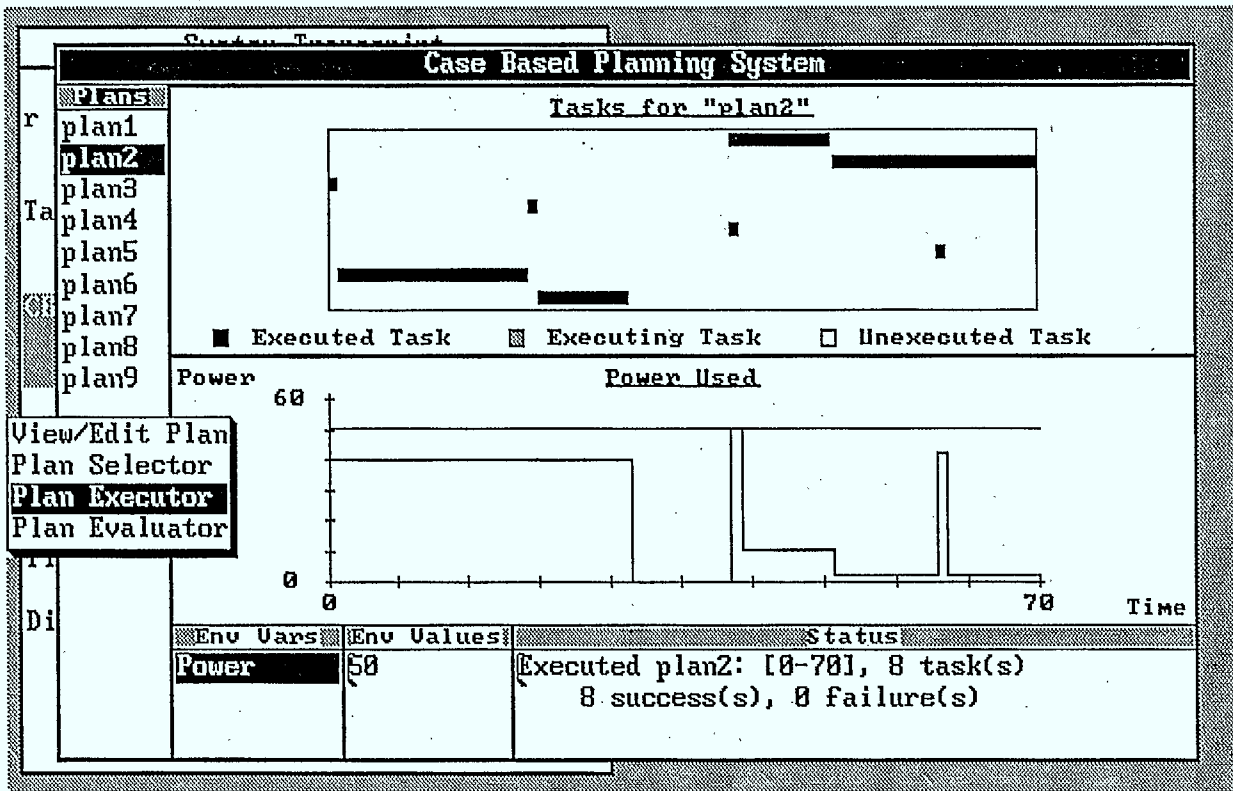


Figure 8 - The CBPSBrowser

The pane labelled "Plans" shows the available plans in the plan library. Selecting one of these plans will set the "operator input" for the CBPS equal to the selected plan.

The "View/Edit Plan" option of the pop-up menu for the "Plans" pane will allow you to edit a current plan to create an "unordered" plan by adding or removing tasks or altering task parameters. When "View/Edit Plan" is selected, a PlanBrowser (section 3.2) is created to do the editing. When you are finished specifying the requirements for the plan, you should close the PlanBrowser window.

The "Plan Selector" option of the pop-up menu for the "Plans" pane will perform the actions of the Selector object with the current "unordered" plan. The selected plan will appear highlighted in the "Plans" subpane to indicate that it is the current plan. New plans constructed from existing plans take the name of the old plan with an "x" appended to the end. For example, if a new plan was constructed by editing a plan from the library called "plan9", the name of the new plan would be "plan9x". During the operation of the Selector, the pane labelled "Status" and the two unlabelled panes will be updated to show the progress of the Selector.

The "Plan Executor" option of the pop-up menu for the "Plans" pane will execute the selected plan. Task execution, replanning, and resource usage are graphically displayed while the plan executes.

The "Plan Evaluator" option of the pop-up menu for the "Plans" pane invokes the Evaluator for the selected plan. The results of plan evaluation are displayed in the pane labelled "Status". The "Plans" pane is updated to show the action that the Evaluator performed on the library.

The subpanes labelled "Env Vars" and "Env Values" show the available resources and the amount of those resources in the current environment. These two panes operate in the same manner as the "Resource Name" and "Resource Values"

subpanes of the PlanBrowser and LibraryBrowser (sections 3.2 and 3.3). Changing the available resources in the environment can be used as a method of introducing plan failures. Reducing the amount of an environment resource may cause a previously good plan to fail during execution, forcing replanning to occur.

The subpane labelled "Status" provides a transcript of the actions of the CBPSBrowser. The plan selection, execution, and evaluation processes write to this pane to show their progress. The "Status" pane should always display the results of the last action that was taken. If the last action was to execute a plan, the "Status" pane will show the plan name, number of tasks, number of task successes and number of task failures.

3.2 The PlanBrowser

The PlanBrowser can be used to manipulate complete and "unordered" plans. Normally, it is used to create or edit "unordered" plans from within the CBPSBrowser object (section 3.1). Figure 9 shows a typical PlanBrowser as invoked from a CBPSBrowser. A PlanBrowser can be created with the following expression:

```
PlanBrowser new openOn: (Plan new name: 'plan1').
```

The PlanBrowser object has a similar appearance and functionality to that of the LibraryBrowser (section 3.3) with the exception that:

1) the "Plans" subpane is missing and,
2) the edit fields labelled "Start Time:" and "Duration:" in the "Task Values" subpane will accept a single "?" as well as integer values.

A "?" in the start time or duration edit field indicates that the actual value for the field is unknown at this time but may be derived during the plan verification process (section 2.2.2) from task rules, plan rules and the simulation of plan execution.

Case Based Planning System

Plans: plan1, plan2

Tasks for "plan2"

Plan Browser

Task	Task Values	Task Rules
load1 load3 Add Task Remove Task load4c load4d load5a load5b	Name: load1 Start Time: 40 Duration: 10	load1 (load1) :- member ([30, 40], start), task (load1, start, 10).
	Resource Name: Power Resource Value: 10	Plan Rules: plan ([load1, load2, load3, load4a, loa not (overlaps (load1, load2)), task (load3, end1, _), task (load1, _, _, end1).

Env Vars	Env Values	Status
Power	50	Executed plan2: [0-70], 8 task(s) 8 success(s), 0 failure(s)

Figure 9 - The PlanBrowser

3.3 The Library Browser

The plan library is implemented as an OrderedCollection [5] of plans. In order to create a plan library, you must first create the OrderedCollection that will contain the plans. Normally, this OrderedCollection is stored in a global variable. This will make the plan library available for use by other CBPS objects at a later time.

A plan library is created using the LibraryBrowser by executing the following expression (and answering "yes" to the question "Declare MyLib as global"):

```
MyLib := OrderedCollection new.
LibraryBrowser new openOn: MyLib
```

The expression "CBPS exampleLibrary" answers an OrderedCollection of plans that can be used as an example library when first exploring the system. Any changes that you make to this example library will not be saved. However, if you execute the following expression instead of the one above, you will be provided with a LibraryBrowser to examine this sample library:

```
LibraryBrowser new openOn: CBPS exampleLibrary
```

Figure 10 shows a typical LibraryBrowser. Each of the subpanes are labelled at the top to indicate their function. The following section will describe the functionality of each of the subpanes of the LibraryBrowser.

The subpane labelled "Plans" shows a complete list of every plan presently in the library. Like all ListPanes [5] in the CBPS, this pane is scrollable. Selecting a plan from the list, indicates that you wish to view or edit this plan. Selecting a plan will cause the pane labelled "Tasks" to display the tasks for the selected plan. The pane labelled "Plan Rules" displays the plan rules associated with the selected plan.

Figure 10 - The LibraryBrowser

The pop-up menu in the "Plans" pane allows you to add new plans or delete the currently selected plan from the library. The other panes in the browser refresh automatically when a plan is added or deleted.

Selecting a task from the pane labelled "Tasks" will cause the panes labelled "Task Values", "Resource Names", "Resource Values", and "Task Rules" to be updated with the values corresponding to the selected task. The pop-up menu for the "Tasks" pane allows you to add new tasks or delete the currently selected task from the current plan. The other panes in the browser refresh automatically when a task is added or deleted.

The pane labelled "Task Values" provides edit fields to enter and modify task name, start time, and duration for the currently selected task. Selecting a field with the mouse will allow you to edit the value in that field. Typing <Return> will accept the new value. The fields labelled "Start Time:" and "Duration:" expect integer values. All values that you enter are verified against the task rules displayed in the pane labelled "Task Rules". If a value is not accepted from an edit field, the value that was entered has violated these rules.

The pane labelled "Resource Name" lists the resources that are required by the task. Selecting a resource from this pane will display the value of the resource in the pane labelled "Resource Values". The pop-up menu for this pane allows you to add a new resource or remove the currently selected resource. The affected panes will refresh accordingly.

After selecting a resource name from the "Resource Names" pane, editing the value found in the "Resource Values" pane and selecting "save" from the pop-up menu in this pane will cause the selected resource to get the new value. If you do not "save" the value in this pane, the new value for the selected resource will not be accepted.

Task rules are edited in the "Task Rules" pane. After editing the rule and

selecting "save" from the pop-up menu in the pane, the system will attempt to accept the rule. You will be informed of any PROLOG syntax errors in the rule that may exist and will be asked to correct them.

Plan rules are edited in the "Plan Rules" pane. After editing the rule and selecting "save" from the pop-up menu in the pane, the system will attempt to accept the rule. You will be informed of any PROLOG syntax errors in the rule that may exist and will be asked to correct them. The head of the plan rule that you enter is automatically updated to accept a list of every task in the plan.

Section 4 - Summary

Version 2 of the Case Based Planning System as described in this document provides the framework for developing domain dependent applications that use case based reasoning in a domain independent manner. The domain of power management on an orbiting spacecraft has provided an initial test environment for the CBPS. Knowledge from this domain has been encoded in an example plan library and was used to pose simple planning problems.

The critical concept of a planning environment allows the representation of domain dependent knowledge within the CBPS as rules and parameter values. Plan and task rules allow the relationships within tasks and plans to be easily expressed. The dynamic nature of the CBPS allows the user to examine and modify these rules on the fly in order to create different planning scenarios.

A CBPS has been implemented that is robust, extensible and independent of any particular user interface. The PlanBrowser, LibraryBrowser, and CBPSBrowser objects, supplied with the CBPS, provide one possible user interface. These objects make use of the Smalltalk windowing interface to provide easy access to the CBPS and present information graphically.

References

[1] **Oppacher F., Deugo D., Thomas D.**, School of Computer Science, Carleton University, *"Planning Techniques Survey: Their Applicability to the Mobile Servicing System"*, Department of Communications, Ottawa, Canada, DOC-CR-SP-88-005, 1988.

[2] **Oppacher F., Deugo D.**, School of Computer Science, Carleton University, *"A Proposed Approach For Scheduling Applications (With Respect to the Mobile Servicing System)"*, Department of Communications, Ottawa, Canada, DOC-CR-SP-88-006, 1988.

[3] **Oppacher F., Deugo D.**, School of Computer Science, Carleton University, *"A Dynamic Case Based Planning System for Space Station Applications: Software and Operation Description"*, DOC-CR-SP-88-007, Department of Communications, Ottawa, Canada, 1988.

[4] **Adamovits P.**, Communications Research Centre, *"Generic Spacecraft Power Subsystem Domain Description"*, Technical Memo, DOC-DSM-86-36, Department of Communications, Ottawa, Canada, January 21, 1987.

[5] **Digitalk Inc.**, *"Smalltalk/V Tutorial and Programming Handbook"*, 9841 Airport Boulevard, Los Angeles, California 90045, 1986 (Smalltalk/V version 1.2, release disks, and update notices).

[6] **Clocksin W.F, Mellish C.S.**, Univeristy of Oxford, University of Edinburgh,*"Programming in PROLOG"*, Springer - Verlag, Berlin, 1981.

Glossary of Terms

action - An *action* is the domain specific activity of a *task* (see *task*). For example, a *task* may represent the *action* of moving a robot arm to capture a satellite.

constraint - A *constraint* is a boolean valued expression that specifies a relationship between *properties* found in *tasks*, *plan*s, or the *environment.* At any particular time, a *constraint* may be satisfied or violated with respect to a *task, plan* or the *environment.*

environment - The *environment* represents the application domain in which the *planner* operates. It may be modified by the execution of a plan or task (see *plan execution* and *task execution*) or by some external force. For example, a *task* may modify the *environment* to indicate that it has completed; or an eclipse may occur, and modify the *environment* to inform the *planner* that it is operating in darkness.

plan - A plan is an ordered sequence of tasks. It is provided to a *planner* to be executed (see *plan execution)* in an environment. A *plan* may have *properties* or *constraints* associated with it that assist the *planner* when ordering tasks. The *plan's properties* and *constraints* often relate the plan to the current state of the *environment.*

plan execution - A *plan* executes by *executing the tasks* in the *plan* in the order defined in the *plan.* The *execution of a plan* may modify the *properties* of the *plan* or the state of the *environment.* The execution of a *plan* will always result in *plan success* or *plan failure.*

plan failure - A *plan failure* means that the *execution of the plan* did not

proceed as expected. This may occur when a *task failed* within the *plan* and could not be corrected by *replanning* or a *constraint* was violated.

plan rule - A *plan rule* is a special type of *constraint* that is associated with a *plan* and specifies relationships between the *tasks* within the *plan*. *Plan rules* can be used to order *tasks* within a *plan*.

plan success - A *plan success* means that the *execution of the plan* proceeded as expected.

planner - A *planner* is the entity that is responsible for producing a *plan* in some manner based on certain requirements. These requirements are provided in the form of *constraints* and *properties* that operate within *tasks*, *plans* and the *environment*.

planning - The activity performed by a *planner*.

property - A *property* is a named data value associated with a *task* , a *plan*, or the *environment*. For example, a *task* may have the *property* that it expects to begin execution at 12:00 (see *task execution*). A *plan* may have the *property* that it includes a *task* named "fred54" that has *failed* ten times. The *environment* may have the *property* that a machine is broken.

replanning - Replanning is the activity performed by a *planner* when a *task* fails (see *task failure* or *plan failure*). This may involve *planning*.

resource - A *resource* is a special type of *property* (see *property*) of a *task* or an *environment*. When associated with a *task*, the named data value is acquired and released from the *environment* when the *task* begins and ends execution. If the *resource* is unavailable, the *task* is not able to start. When associated with an *environment*, the named data value is made available for *tasks* that require *resources* of the same name to start.

task - A *task* is a unit of activity within a *plan.* A *task* represents a domain specific *action* that cannot be further decomposed. A *task* may have *properties* or *constraints* associated with it that assist the *planner* to position it among other *tasks* in a *plan.* The execution of a *task* (see *task execution*) may alter the *properties* of the *task* or the *environment.* A *task* may require resources in the *environment* in order to *execute.* These resources may be *properties* of the *environment* that are subject to some *constraints* imposed by the *task.*

task execution - The activity associated with the *task* is performed. This can be anything from a machine turning on to an astronaut positioning a robot arm. The *execution of a task* may modify the *properties* of a *task* or the state of the *environment.* The execution of a task will always result in a *task success* or a *task failure.*

task failure - A *task failure* means that the activity associated with the *task,* when *executed,* was not performed satisfactorily. This implies that the action associated with the task was not completed, did not start, or did not finish at the expected time. A task failure causes the *planner* to perform a *replanning* action.

task rule - A *task rule* is a special type of *constraint* that is associated with a *task* and is used to specify relationships within the task. *Task Rules* are used to both verify and generate the *task properties* known as start time and duration.

task success - A *task success* means that the activity associated with the *task,* when *executed,* was performed satisfactorily. This implies that the time for the task to perform and complete the associated action was as expected in the plan.

Appendix A: PROLOG Predicates for Task and Plan Rules

Introduction

The following list of predicates have been added to the PROLOG environment for use in task and plan rules. This section will first describe the predicates intended for task rules. These predicates may also be used in the definition of plan rules.

Task Rules

between ***(start, end, number) :-***

1) When *start*, *end* and *number* are bound to integers, the between relation succeeds if (*start* <= *number* <= *end*) and fails otherwise.
2) When *start* and *end* are bound to integers and *number* is unbound, the between relation binds *number* to *start* and succeeds. On back tracking, *number* is bound to (*number* + 1). When (*number* > *end*), the between relation fails.

member ***(list, element) :-***

1) When *list* is bound to a list and *element* is bound to any object, the member relation succeeds if *element* is equal to any *element* in the list.
2) When *list* is bound to a list and *element* is unbound, the between relation binds *element* to the first member in the list and succeeds. On back tracking, *element* is bound to subsequent

elements in the list. When the list becomes empty, member will fail.

multiply ***(a, b, c)* :-**

1) When *a*, *b* and *c* are bound to numbers, the multiply relation succeeds when ($a * b = c$) is true.
2) When any one of *a*, *b*, *c* is unbound, the multiply relation binds this value to a number such that ($a * b = c$) is true.

sum ***(a, b, c)* :-**

1) When *a*, *b* and *c* are bound to numbers, the sum relation succeeds when ($a + b = c$) is true.
2) When any one of *a*, *b*, *c* is unbound, the sum relation binds this value to a number such that ($a + b = c$) is true.

task ***(task, start, duration)* :-**

1) When *task* is unbound, the task/3 relation succeeds.
2) When *task* is bound to a list of the form [*name*, *duration*, *start*], *start* and *duration* are unified with the members of the list. The task/3 relation is therefore capable of getting, setting and testing the values of *start* and *duration*.

task ***(task, start, duration, end)* :-**

1) The task/4 relation behaves the same way as task/3 but uses the sum relation to enforce the constraint (*start* + *duration* = *end*). The task/4 relation is normally used instead of task/3 when the *end* time of a task is required.

Plan Rules

distinct (***task1, task2***) **:-**

1) When either *task1* or *task2* is unbound, distinct succeeds.
2) When both *task1* and *task2* are bound, the distinct relation succeeds when the time periods that *task1* and *task2* execute within do not overlap.

follows (***task1, task2***) **:-**

1) When either *task1* or *task2* is unbound, follows succeeds.
2) When both *task1* and *task2* are bound, the follows relation succeeds when *task1* begins after *task2* has ended.

overlaps (***task1, task2***) **:-**

1) When either *task1* or *task2* is unbound, overlaps succeeds.
2) When both *task1* and *task2* are bound, the overlaps relation succeeds when the time periods that *task1* and *task2* execute within overlap.

precedes (***task1, task2***) **:-**

1) When either *task1* or *task2* is unbound, precedes succeeds.
2) When both *task1* and *task2* are bound, the precedes relation succeeds when *task1* ends before *task2* starts.

Appendix B: Sample Demonstration

Introduction

This document is intended to provide a quick walk through of the CBPS in order to demonstrate some of the features. The user is expected to be familiar with the Smalltalk/V environment, the PROLOG language, Case Based Planning, and this document, the "Users Guide to Version 2 of the Case Based Planning System".

Demonstration

In any text pane, execute the following:

```
CBPSBrowser  example
```

This expression will create a CBPSBrowser with a default plan library of plans named "plan1" to "plan9", a default environment with 50 units of #Power available and a default "unordered" plan containing 5 tasks named "load1", "load2", "load3", "load4a" and "load4b". These tasks are contained in some of the plans in the default library.

1.0 Plan Specification

Select "View/Edit Plan" from the pop-up menu in the "Plans" pane. A PlanBrowser will be created to edit the "unordered" plan (see section 2.2.1). You will see the five tasks in the pane labelled "Tasks". Select the task "load1" and enter a "?" in the start time field if one is not there already. When you enter the "?", you are specifying that the start time of "load1" is not known at this time but can be

derived by the CBPS when required. Do the same thing for the tasks "load2" and "load3". Next, edit the plan rules in the "Plan Rules" pane for the "unordered plan". Enter for the body of the plan rule (if not already there) the clauses:

```
precedes  (load1,  load2),
distinct  (load2,  load3).
```

Select the "save" option from the pop-up menu in the "Plan Rules" pane.

NOTE: Be careful when entering plan rules. There is no check to ensure that the predicates that you intend to call are defined. This type of error is uncovered only when the predicate is invoked.

After performing the above modifications to the "unordered" plan, the CBPS will be required to find a plan that:

1) contains the tasks "load1", "load2", "load3", "load4a" and "load4b",
2) such that "load1", "load2" and "load3" can start at any time that is valid with respect to the task rules for the task and
3) within the plan, "load1" must end before "load2" starts and
4) "load2" and "load3" cannot overlap.

You should close the PlanBrowser and select the CBPSBrowser before preceding.

2.0 Plan Selection

Bring up the pop-up menu for the "Plans" pane and select the option titled "Plan Selection". This will invoke the Plan Selection module (section 2.2.2) of the CBPS. Observe the "Status" and unlabelled panes.

First, the Selector will attempt to locate the plan from the library that best matches your requirements. It should locate "plan6" as the best match and tell you so in the "Status" pane. Next, the Selector will construct a new plan called "plan6x" by deleting any extra tasks found in "plan6". Finally, the unlabelled pane that displays tasks will show intermediate plans, as the plan is verified.

3.0 Plan Execution

Bring up the pop-up menu for the "Plans" pane and select the option titled "Plan Execution". This will invoke the Plan Execution module (section 2.2.3) of the CBPS. Observe the "Status" and unlabelled panes.

As tasks are execute, they will change colour in the unlabelled task pane. A black task is finished executing. A grey task is currently executing. A white task has yet to execute. Any replanning actions would be shown as they occur in this pane. The execution of "plan6x" should not cause replanning.

3.0 Plan Evaluation

Bring up the pop-up menu for the "Plans" pane and select the option titled "Plan Evaluation". This will invoke the Plan Evaluation module (section 2.2.3) of the CBPS. Observe the "Status" pane.

The plan "plan6x" should be added to the library. If "plan6x" was to be executed again and then evaluated, it would be updated in the library to reflect both executions. If "plan6x" was considered to be a bad plan, it might have been removed or forgotten from the plan library.

4.0 Replanning

To get a quick demonstration of plan failure and replanning, select "plan2"

from the "Plans" pane. This will make "plan2" the current "unordered" plan. Next, select the item called "Power" from the pane labelled "Env Vars". The pane labelled "Env Values" should display the number 50. Change this number to 45 and select "save" from the pop-up menu for the pane. When you save this value, the unlabelled pane that plots power usage over time should be redisplayed.

NOTE: Do not forget to select "save" from the pop-up menu in the "Env Values" pane after editing the resource. If you do not "save" the new value, the selected resource will not be altered.

Finally, execute "plan2". You will see "load4b" fail and be rescheduled at a later time in the execution of the plan. This is the default action for any task that fails to start.

5.0 Miscellaneous

You could now try evaluating "plan2". Because the replanning caused task "load4c" to be rescheduled at a time that is not valid for the task, the plan evaluation process should indicate this in the "Status" pane and abort the evaluation.

Please feel free to experiment by adding, removing and editing tasks in the "unordered" plan, changing plan and task rules, and defining new plan libraries.

Appendix C: The CBPS Planning Classes

1 - Introduction

This sections briefly describes each of the objects that implement planning in version 2 of the CBPS. The CBPS user interface objects are not described here. Unfortunately, there was no time to produce a complete programmers reference manual. It is hoped that this guide, along with the rich Smalltalk programming environment, will provide some assistance for future programmers.

2 - PROLOG Classes

Five PROLOG classes are implemented within the CBPS: CommonRules, EvaluatorRules, Replanner, SelectorRules and TaskRules. The PROLOG class EvaluatorRules is used to implement the domain specific knowledge required when evaluating plans. The PROLOG class Replanner is used to implement replanning rules for the CBPS. The PROLOG class SelectorRules is used to implement the domain specific portion of the Selector. The PROLOG class TaskRules implements task and plan rules and contains the PROLOG and Smalltalk code used to generate and verify plans. The PROLOG class CommonRules is the super class of the other four. It contains the PROLOG predicates described in Appendix A of this document. These predicates may be accessed by any of the five subclasses.

2.1 CommonRules

CommonRules implements PROLOG predicates that are intended for use by the classes Replanner and TaskRules. A complete description of these predicates can be found in Appendix A of this document.

Class Name:	CommonRules
SuperClass:	Prolog
SubClasses:	(Replanner TaskRules)
Instance Variables:	()
Class Variables:	()
Pool Dictionaries:	()

Class Methods:

None.

Instance Methods:

between (start, end, number) :-
A general purpose integer generator/tester.

distinct (task1, task2) :-
Succeed if the tasks are distinct (do not overlap) or if there is no task1 or task2.

follows (task1, task2) :-
Succeed if task1 starts after task2 ends.

member (list, element) :-
A general purpose set membership generator/tester.

multiply (a, b, c) :-
A general purpose multiply relation.

overlaps (task1,task2) :-
Succeed if the tasks overlap or if there is no task1 or task2.

precedes (task1, task2) :-
Succeed if task1 ends before task2 starts or if there is no task1 or task2.

sum (a, b, c) :-
A general purpose sum relation.

task (task, start, duration) :-
Get/Set task values.

task (task, start, duration, end) :-
Get/Set task values.

<u>Example</u>:

The following code fragment will computes all integers between **1** and **5** that are also members of the list **[2, 4, 6, 8]** and answer an Array of answers. Each member of the Array is an Array containing one answer, the current value of the PROLOG variable "x". This expression will return ((2) (4)).

```
CommonRules new :?
        between (1, 5, x),              "generate"
        member ([2, 4, 6, 8], x).       "test"
```

2.2 EvaluatorRules

EvaluatorRules implements PROLOG predicates that are intended to contain the domain specific knowledge needed to evaluate plans. The predicate "evaluate" is called by the Evaluator when it is asked to evaluate a plan.

Class Name:	EvaluatorRules
SuperClass:	CommonRules
SubClasses:	()
Instance Variables:	()
Class Variables:	()

Pool Dictionaries: ()

Class Methods:
None.

Instance Methods:
criteria (plan, successes, failures) :-
This predicate succeeds if the plan can provide values for the number of task successes and failures. These values are unified with the PROLOG variables "successes" and "failures".
evaluate (plan, library, evaluator) :-
This predicate evaluates the plan with respect to a plan library. The evaluation action (ie. add plan to library) is a method for the evaluator that is invoked by this predicate.
newPlan (plan, library) :-
This predicate succeeds if the plan is not found in the library.

Example:
None.

2.3 Replanner

Replanner implements PROLOG predicates that are intended for use when a task fails during plan execution. By convention, the 3-tuple predicate failure() is called with the failure information. The first member of the tuple is a Symbol that is the name of the resource that failed. The second is the instance of the Task object that failed to start. The last parameter is the instance of the Executor that was attempting to execute the task that failed.

Class Name: Replanner
SuperClass: CommonRules
SubClasses: ()
Instance Variables: ()
Class Variables: ()
Pool Dictionaries: ()

Class Methods:
None.

Instance Methods:
failure (resource, task, executor) :-
This code gets executed when a task fails to acquire a resource.

Example:
None.

2.4 SelectorRules

SelectorRules implement PROLOG predicates that are intended to contain the domain specific knowledge needed to locate and construct plans. The predicate "better" is called by the Selector when it is asked to compare two plans. The predicates "canBeAdded", "canBeRemoved" and "canOccur" are called by the Selector when asked to construct a plan.

Class Name:	SelectorRules
SuperClass:	CommonRules
SubClasses:	()
Instance Variables:	()
Class Variables:	()
Pool Dictionaries:	()

Class Methods:
None.

Instance Methods:

better (plan1, plan2, environment) :-
This predicate succeeds if the plan1 is better than plan2 with respect to some criteria contained in the environment. The environment contains the "unordered" plan that is matched against both plans.

canBeAdded (task, environment) :-
This predicate succeeds if the task can be added by the Selector.

canBeRemoved (task, environment) :-
This predicate succeeds if the task can be removed by the Selector.

canOccur (task, environment) :-
This predicate succeeds if the task can be occur in the environment.

criteria (plan, environment, extra, missing, failures) :-
This unifies the values of extra, missing, and failures with the extra tasks and missing tasks with respect to the "unordered" plan and the total task failures of the plan.

Example:
None.

2.5 TaskRules

TaskRules implements PROLOG predicates and Smalltalk code that verifies a plan using the task and plan rules for the plan. An Executor is used to simulate

a plan using the task and plan rules for the plan. An Executor is used to simulate the execution of the plan in a copy of the environment. This class is composed of both Smalltalk methods and PROLOG predicates.

The main entry point is the predicate verify(). This predicate is called with the PROLOG representation of a Plan. This is a list of lists. Each sub-list is a List that represents a Task in the Plan. The predicate schedule() is used by verify() to generate/test the values for start time and duration for each sub-list using the task rules for each task. Next, the list of lists is turned back into a Smalltalk Plan and the execution of the plan is simulated in a copy of the environment. The Smalltalk method #executePlan does the simulated execution. This method answers true or false and exits at the first task failure. Finally, the plan rules are executed.

At any time, verify() or any of the predicates it calls may fail and back track. This will attempt to find a new schedule for the tasks within the current plan.

Class Name: TaskRules
SuperClass: CommonRules
SubClasses: ()
Instance Variables : (plan executor environment)
Class Variables: ()
Pool Dictionaries: ()

Class Methods:
None.

Instance Methods:

default (task) :-
This is the default task rule that all tasks will execute if no rule for the task is specified.

environment: anEnvironment
Set the environment of verification.

execute (task) :-
Execute one task triple. If there is no rule for the task, use the default rule. Because cut() does not work quite right in PROLOG/V, the #respondsTo: code is repeated in both predicates.

executePlan
Execute the plan in a copy of the environment. At the first failure of any kind, answer false and abort the plan execution. The original event queue for the simulation is saved in order to preserve any events that are originally scheduled. These events could update the screen to show the progress of the verification.

executor: anExecutor
Set the plan executor.

schedule (list) :-
Execute each of the task triples in the list as a PROLOG predicate. Succeed if every predicate succeeds.

setPlan: aPlan
Set the plan to be verified.

verify (list) :-
Verify that the tasks in the list can execute properly. First create a schedule of tasks, place them in the current plan, then execute the plan. Exit with success with the first valid plan stored in the list.

Example:
The following code fragment will check to see that the task triples (the PROLOG representation of a Smalltalk task) do not overlap. This expression will fail and therefore answer **nil** (task1 and task2 overlap from time 7 to time 10).

```
TaskRules new :?
      distinct ([#task1, 5, 10], [#task2, 7, 15]).
```

3 - Smalltalk Classes

The following 9 nine classes implement the Smalltalk portion of the CBPS: CBPS, Environment, Evaluator, Executor, NoteTaker, Plan, Selector, Simulation and Task.

3.1 CBPS

The CBPS is the main object that implements case based reasoning. In order to plan, it requires only a plan library and environment. Using instances of the Selector, Executor and the Evaluator, the CBPS object can choose and edit a plan from the library, execute the plan and evaluate the results of the execution. Instances of the above objects are created automatically when a CBPS is created.

Class Name:	CBPS
SuperClass:	Object
SubClasses:	()
Instance Variables:	(plan library environment selector executor planner evaluator)
Class Variables:	()
Pool Dictionaries:	()

Class Methods:

example
This method executes a canned CBPS example.

exampleCBPS
This method answers an example CBPS object.

exampleLibrary
This method answers a library of plans. Uncomment the rule code the first time this code is run.

new
Answer a new instance of the receiver and initialize it.

Instance Methods:

doPlanning

Perform the activities that make up Case Based Reasoning. Select a plan from the plan library, execute it (perhaps failing at some tasks), and evaluate the results.

environment

Answer the current CBPS environment.

environment: anEnvironment

Set the current CBPS environment.

evaluatePlan

Evaluate the current plan using an Evaluator.

evaluator

Answer the plan Evaluator ready to evaluate the current plan with respect to the current environment and plan library.

executePlan

Execute the current plan using an Executor.

executor

Answer a plan Executor ready to execute the current plan in the current environment.

initialize

Initialize the instance variables. Create a default empty plan, a default environment, a default empty library and the Selector, Executor, and Evaluator objects that will perform the basic CBPS functions.

library

Answer the current plan library.

library: aPlanLibrary

Set the plan library.

plan

Answer the current plan for the CBPS.

plan: aPlan

Set the current plan for the CBPS.

selector

Answer the plan Selector ready to select a plan based on the current environment and plan library.

selectPlan

Select a plan using a Selector.

<u>Example:</u>

The following code fragment will do one complete iteration of case based reasoning. The initial plan library is created by the expression "CBPS exampleLibrary". This expression answers an OrderedCollection of plans. The required tasks are set to the tasks at location 1, 3, 4 and 5 from the collection of tasks of the first plan in the library. A planning environment is created and 50 units of #Power are made available. The CBPS is instructed to plan using these initial conditions. The last statement invokes an inspector on the CBPS object so

that the CBPS object may be examined to see what happened. A new plan should be created, executed without failure and added to the plan library.

```
| planner env lib required |
planner := CBPS new.                        "create planner"
lib := CBPS exampleLibrary.                 "get example lib"
required := #(1 3 4 5) collect:             "get tasks 1,3,4,5"
        [:i | lib first tasks at: i].
env := Environment new.                     "create new env"
env resources at: #Power put: 50.           "50 units available"
env requiredTasks: required.                "set required tasks"
planner
        library: lib;
        environment: env;
        doPlanning;                         "do the planning"
        inspect                             "inspect the results"
```

3.2 Environment

The class Environment implements the planning environment for the CBPS objects. The environment is used when executing tasks or simulating the execution of tasks to acquire and release resources. It can be used as a "black board" for tasks to communicate or for planning rules to access to determine the current state of the planner. It is used to hold the "unordered" plan.

When a task fails, it is the failBlock within the Environment that is executed. The failBlock is a Block with no arguments. Normally, this block is set to a block that will invoke a Replanner. However, the verification process within TaskRules sets this block to be a block that jumps out of the Environment and answers false. This mechanism is used to abort the simulated execution of the plan after the first task failure (see #executePlan for TaskRules).

The important method #isBetterPlan:thanPlan: is used to compare plans when searching the plan library. This method allows for domain knowledge to enter into the plan selection process by invoking the "better" predicate for the PROLOG class SelectorRules.

Class Name:	Environment
SuperClass:	Object
SubClasses:	()
Instance Variables:	(unorderedPlan resources failBlock)
Class Variables:	()
Pool Dictionaries:	()

Class Methods:

new

Answer a new instance of the receiver and initialize it.

Instance Methods:

acquireAmount: aValue ofResource: aName

Use up some of the available resource named by aName if possible. Answer true if the resource was acquired, else false. Execute the fail block if no resources were available.

copy

Answer a copy of the receiver. Be sure to create a deep copy of the resources of the receiver. Otherwise, copies of the receiver can destructively modify these resources.

failBlock: aBlock

Set the failBlock of the receiver. The failBlock is executed every time a resource cannot be acquired from the receiver.

initialize

Initialize the instance variables. Create a default (empty) unorderedPlan, a failBlock that does nothing and an empty dictionary to hold resources.

isBetterPlan: plan1 thanPlan: plan2

Answer true if plan1 is the better of the two plans with respect to plan2 and the receiver. This method creates an instance of a SelectorRules object to evaluate the PROLOG predicate "better". Answer true if the predicate succeeds.

releaseAmount: aValue ofResource: aName

Release some of the available resource named by aName back to the receiver.

requiredTasks

Answer a collection of the tasks that are required to execute in the receiver. These are the tasks of the unorderedPlan.

requiredTasks: tasks

Set the collection of the tasks that must execute in the receiver. These are the tasks of the unorderedPlan. Any previous task are removed and copies of new tasks are added to the unorderedPlan.

resources

Answer the dictionary of available resources.

resources: aDict

Set the dictionary of available resources.

unorderedPlan

Answer the unordered plan. This plan represents the current requirements of the operator.

Example:

The following code fragment will create a new environment and make 50 units of power available for consumption, make 'task54' a requirement and set the failBlock to issue a message to the user:

```
| env task54 |
env := Environment new.                       "create new env"
env resources at: #Power put: 50.             "50 units available"
env failBlock: [: resource |                  "prompt on failure"
    Menu message:
        'task failed to acquire #', resource].
task54 := Task new name: 'task54'. "create task"
task54
    startTime: 10;                            "starts at 10"
    duration: 20.                             "last 20 units"
task54 resources at: #Power put: 51.
env requiredTasks: (Array with: task54).
CBPS new
    library: OrderedCollection new; "empty library"
    environment: env;
    doPlanning;                               "do planning"
    inspect                                   "inspect results"
```

3.3 Evaluator

The Evaluator object implements the plan evaluation module of the CBPS. It requires the plan to be evaluated, the plan library and the environment of execution.

The actual evaluation is performed in the method #evaluatePlan. This method first verifies that task and plan rules for the plan have not been violated. Domain knowledge is accessed in #evaluatePlan by invoking the "evaluate" predicate in the PROLOG class EvaluatorRules.

Class Name: Evaluator
SuperClass: Object
SubClasses: ()
Instance Variables: (plan library environment)
Class Variables: ()
Pool Dictionaries: ()

Class Methods:
None.

Instance Methods:

addPlan
Add a new plan in the library.

environment: anEnvironment
Set the environment of evaluation.

evaluatePlan
This method evaluates the plan and takes an action with respect to

the plan library. The PROLOG predicate "evaluate" is invoked in the class EvaluatorRules to perform the actual evaluation.

forgetPlan
Do nothing. Do not add, remove, or update the original plan in the plan library.

hasViolations
Answer true if the plan has any kind of plan or task rule violations.

library: aPlanLibrary
Set the plan library.

plan: aPlan
Set the plan to be evaluated.

planViolation
Answer true if the plan rules for the current plan have been violated.

removePlan
Remove the plan from the plan library.

taskViolations
Answer a collection of the tasks that have task rule violations.

updatePlan
Add a the updated plan to the library. First remove the original plan (if any) from the plan library and then add the current plan. The plan that the Evaluator massages is always a new instance of a plan, even if it is equal to a plan already in the library.

Example:

The following code fragment will evaluate a plan:

```
...
Evaluator  new                              "create new instance"
    plan:  aPlan;                           "set the plan"
    environment:  environment;              "set the environment"
    library:  aPlanLibrary;                 "set the plan library"
    evaluatePlan.                           "do the evaluation"
...
```

3.4 Executor

The Executor is responsible for executing every task in a plan. It requires the plan to be executed and the environment of execution. The executor uses an instance of a Simulation to schedule the start and end times of the tasks. When a failure occurs, the Replanner can get invoked depending on the failBlock in the Environment.

The main entry point to the Executor is the method #executePlan. The

methods #replanOnFailure and #doNothingOnFailure set the environmental failBlock to invoke the Replanner or do nothing respectively.

The methods #beginTask: and #endTask: are scheduled to occur in the simulation at the start and end time of each task in the plan.

The methods #dropPlan, #dropTask: and #scheduleTask:atTime: are intended to be called from the replanner.

Class Name: Executor
SuperClass: Object
SubClasses: ()
Instance Variables:
(plan environment simulation currentTask executingTasks executedTasks)
Class Variables: ()
Pool Dictionaries: ()

Class Methods:

new

Answer a new instance of the receiver and initialize it.

Instance Methods:

beginTask: aTask

Start aTask in the receiver. This method is executed by the simulation when aTask starts. Set the current task to be aTask and ask it to start. If aTask cannot start, do a task failure action. If the task starts, update the executing tasks collection and schedule the task end in the simulation.

currentTask

Answer the task that is about to start in the receiver.

doNothingOnFailure

Do nothing if a failure happens in the environment. This method sets the environment failBlock to a block that does nothing.

dropPlan

Forget the executing the rest of the plan. Release any resources that may be acquired by the tasks that are currently executing and reinitialize the receiver.

dropTask: aTask

Drop aTask from the receiver. A task is assumed to be the current task. The task is removed from the collection of tasks executing and tasks executed (if present) and then removed from the plan. Because the task was unable to start, it also did not schedule and end event for itself in the simulation.

endTask: aTask

End aTask in the receiver. This method is executed by the simulation when aTask ends. Remove aTask from the collection of tasks that are currently executing and add it to the collection of tasks

that have been executed. Note that the task has succeeded.

environment
Answer the environment of execution.

environment: anEnvironment
Set the environment of execution.

executePlan
Execute the plan in the receiver. Schedule the start event for each of the tasks in the plan. Run the simulation for the start and end time of the plan.

failTask: aTask
Indicate that aTask has failed. Ask the plan history to record the failure.

initialize
Initialize the receiver. Create a simulation to simulate the execution tasks. Initialize the collections of tasks that are executing and tasks that have been executed to be empty. Set the sortBlock of the simulation to be a block that ensures that when a #startTask: and #endTask: occur at the same time, the #endTask: is processed first. This is done to ensure that resources are released before they are acquired if events happen at the same time.

plan
Answer the current plan.

plan: aPlan
Set the plan for the receiver to execute.

reinitialize
Reinitialize the receiver. Ask the simulation to be reinitialized (clear event queues, current time, etc.) and set the collections of tasks executing and executed to be empty. This method should be called before executing another plan.

replanOnFailure
Set the receiver to invoke the Replanner when a failure occurs. This method sets the failBlock of the environment to invoke the Replanner.

scheduleTask: aTask atTime: aTime
Schedule aTask to occur in the simulation at aTime. Set the start time of aTask to be aTime. Update the start and end times of the current plan.

simulation
Answer the simulation used by the receiver to simulate the execution of tasks.

succeedTask: aTask
Indicate that aTask has succeeded. Tell the plan history to record the success.

tasksExecuted
Answer a collection of the tasks that have been executed.

tasksExecuting
Answer a collection of the tasks that are currently executing.

tasksToExecute

Answer a collection of the tasks that have yet to be executed. This collection is computed from the plan and the other two task collections.

time

Answer the current time. This is the current time in the simulation.

Example:

The following code fragment will execute a plan. When a task fails, the Replanner will be invoked.

```
...
Executor  new                             "create new instance"
    plan:  aPlan;                         "set the plan"
    environment:  environment;            "set the environment"
    replanOnFailure;   "when tasks fails, invoke replanner"
    executePlan.                          "execute the plan"
...
```

3.5 NoteTaker

The NoteTaker is responsible for recording task failures and successes for a plan. It uses the instance variables "successes" and "failures" as counters to record the number of task successes and failures for the plan. A NoteTaker object can be found in the instance variable called "history" for every plan.

Class Name: NoteTaker
SuperClass: Object
SubClasses: nil
Instance Variables: (plan successes failures)
Class Variables: ()
Pool Dictionaries: ()

Class Methods:

new

Answer a new instance of the receiver and initialize it.

Instance Methods:

failTask: aTask in: anEnvironment

Record the fact that aTask has failed in anEnvironment This method increments the failure counter.

failures

Answer the number of failures.

initialize

Initialize the instance variables. Set failures and successes to zero.

plan: aPlan

Set the current plan for the receiver.

printOn: aStream

Append the ASCII representation of the receiver on aStream. Show the number of successes and failures.

succeedTask: aTask in: anEnvironment

Record the fact that aTask has succeeded in anEnvironment. This method increments the success counter.

successes

Answer the number of successes.

<u>Example:</u>

The following code fragment will record that a success in a new instance of a NoteTaker, for the first task in "plan1", in an environment called "env":

```
| history plan env |
env  :=  Environment  new.                    "create env"
plan  :=  CBPS  exampleLibrary  first.        "get first plan"
history  :=  NoteTaker  new  plan:  plan1."create plan"
plan1  history:  history.                     "set the history"
history
   succeedTask:  (plan1  tasks  first)"first  task  succeeds"
   in:  env.
history
```

3.6 Plan

The Plan class implements objects that represent plans in the CBPS. All plans have a name, some tasks, a start time, an end time and a history. Associated with the plan name are a set to plan rules implemented in the PROLOG class TaskRules.

The plan comparison method #= is used when the plan library is searched to see if the plan is present in the library.

Tasks are added and removed from a Plan using the methods #addTask and #removeTask. Both these methods call #calculateTimes to keep the start time and end time of the plan up to date.

The methods #asList and #fromList: convert a plan to and from the PROLOG representation of a plan.

Class Name:	Plan
SuperClass:	Object
SubClasses:	()
Instance Variables:	(name tasks startTime endTime history)
Class Variables:	()

Pool Dictionaries: ()

Class Methods:

new

Answer a new instance of the receiver and initialize it.

Instance Methods:

= aPlan

Answer true if the receiver is equal to aPlan. This method returns true if the receiver and aPlan have equal names.

addTask: aTask

Add aTask to the receiver. Recalculate the start and end times of the receiver.

asList

Answer the receiver as a list of lists. Each of the lists is a task in the receiver that has been converted into a list. This is the PROLOG representation of a plan.

calculateTimes

Calculate the start and end times for the receiver. Select the minimum and maximum times from the tasks that have defined start and end times.

copy

Answer a deep copy of the receiver. It is essential that any copy of the receiver also have its own copy of the plan history. Otherwise, the receiver and the copy will share the exact same history object that is updated by both.

endTime

Answer the endTime of the receiver.

extraTasks: someTasks

Answer a collection of extra tasks in the receiver with respect to the tasks found in someTasks.

failures

Answer the number of failures for the receiver. Ask the plan history.

fromList: aList

Set the tasks in the receiver from a list of tasks where each task is a list. For each list in the list, find the task in the receiver that corresponds to the list and ask it to initialize itself from the list. The list of lists is the PROLOG representation of the receiver.

history

Answer the history of receiver.

history: aNoteTaker

Set the history of receiver.

initialize

Initialize the instance variables. Create an empty collection of tasks, a new history and set the start and end time to zero.

missingTasks: someTasks

Answer a collection of missing tasks in the receiver with respect to the tasks found in someTasks.

name

Answer the name of the receiver.

name: aString

Set the name of the receiver. The name of the receiver must be a symbol since it is used to access the plan rules for the plan. If aString is not a Symbol, it is converted into one.

printOn: aStream

Append the ASCII representation of the receiver on aStream. Print plan name, starttime and end time and the number of tasks on aStream.

removeTask: aTask

Remove aTask from the receiver. Complain if the task cannot be removed or is not found in the receiver. Recalculate the start and end times of the receiver.

replaceHead: aString

Replace the head of the rule (PROLOG horn clause) found in aString with the a new rule head that has the same name as the receiver and takes a single list of the tasks in the receiver as a parameter. Answer the new rule.

ruleHead

Answer a string that is the head of the plan rule (PROLOG horn clause) with the tasks of the plan in a list.

rules

Answer the PROLOG code that is associated with the receiver. This is a horn clause in the class TaskRules that has the same name as the receiver.

rules: aString

Set the PROLOG code that is associated with the receiver. This is a horn clause in the PROLOG class TaskRules that has the same name as the receiver. Replace the head of the rule and compile and install the new rule in the class TaskRules.

startTime

Answer the startTime of the receiver.

tasks

Answer the tasks of the receiver.

verify

Verify the that the plan rules for the receiver succeed. If there is no plan rule for the receiver, succeed. Otherwise, execute the PROLOG horn clause that has the same name as the receiver in the class TaskRules after converting the receiver into a list of lists. Answer true or false.

<u>Example:</u>

The following code fragment will create a new instance of a plan and initialize it:

```
| plan  task  i  |
plan  :=  Plan  new  name:  'plan54'.          "create plan"
i  :=  1.
#(0  20  35  40)  do:  [:start  |               "create tasks"
     task  :=  Task  new
          name:  'task',i  printString.
     task  startTime:  start;  duration:  20.
     plan  addTask:  task.                      "add task to plan"
     i  :=  i  +  1].
plan  rules:    "set plan rules (rule head will be replaced)"
     'plan54  ([])  :-  distinct  (task2,  task3).'.
plan  inspect
```

3.7 Selector

The Selector object implements the Selector module of the CBPS. By extracting the "unordered" plan from the environment and matching it against the plan library, the instance variable plan is initialized to the plan that best matches the requirements. The instance of the executor is supplied for plan verification purposes.

The main entry point to the Selector is the method #selectPlan. This method calls the methods #locatePlan, #constructPlan and #verifyPlan to do the actual work. These three methods may be called independent of #selectPlan.

The plan that the Selector answers is always a new instance of a plan. Even if it is equal to a plan in the plan library. Plans in the library are considered to be read only by the CBPS.

Class Name:	Selector
SuperClass:	Object
SubClasses:	()
Instance Variables:	(plan environment library executor)
Class Variables:	()
Pool Dictionaries:	()
Class Methods:	

Instance Methods:

addMissingTasks: aPlan

Add any missing tasks to aPlan.

constructPlan

Adds missing tasks, removes extra tasks, and replace equal tasks. Answers a new plan with a new name that has an 'x' appended to the end. This plan should next be verified.

environment: anEnvironment

Set the environment of the receiver.

executor: anExecutor

Set the plan executor. This executor will be used during the plan verification process. It gets invoked after the task and plan rules for the receiver have been satisfied.

library: aPlanLibrary

Set the plan library to be searched for the best match of the operator requirements by the receiver.

locatePlan

Answer the best matching plan in the plan library with respect to the requirements. A new (and empty) plan is answered if the library is empty or the requirements are empty. Otherwise, the actual plan from the library is answered. If you modify this plan directly (ie. don't make a copy) the library will get destructively updated because this is the actual plan that can be found in the library. The methods of the receiver are careful not to do this.

removeExtraTasks: aPlan

Remove any extra tasks from aPlan.

replaceEqual: aPlan

Replace any tasks in aPlan with equal tasks from the operator input. This will allow the operator to unbind any variables in the plan that was selected from the library.

selectPlan

Attempt to locate a plan that best matches the operator requirements from the plan library. Add missing tasks, remove extra tasks, and replace equal tasks. Next, verify that the plan will work in the environment. This is the main entry point of the receiver. Answer nil or a new plan.

verifyPlan

Verify that the plan is expected to work in the environment. This involves creating a new instance of the task and plan rule base (called TaskRules), setting the current plan and environment as well as any executor that may be provided in this rule base, and issuing a PROLOG query that calls the verify() predicate in TaskRules. Answer the plan if the verify succeeded, else answer nil.

<u>Example:</u>

The following code fragment will locate, construct and verify a plan:

```
| selector lib env |
lib  := CBPS exampleLibrary.          "get example lib"
env  := Environment new.              "create environment"
env resources at: #Power put: 50.
env requiredTasks:
     (lib first tasks copyFrom: 2 to: 4).
Selector new
     library: lib;                    "set plan library"
```

```
environment: env;              "set environment"
selectPlan                     "select plan"
```

3.8 Simulation

The class Simulation implements a standard discrete event simulation that has a current time, end time, event queue and a "when" queue. The current time is used to store the current simulation time. The end time stores the time that the simulation should end.

The event queue is a SortedCollection of pairs that is sorted on the first element in each pair. The first element of each pair is the time that the event will occur. The second member of the pair is either a Block with no arguments or a triple of the form #(object message arguments). When the event is processed, it is removed from the event queue and the block is evaluated or the triple is executed using the expression "object perform: message withArguments: arguments"

The "when" queue is an OrderedCollection on pairs. Each member of the pairs is a block. After the execution of an event from the event queue, the "when" queue is processed by executing the second block in each pair for every first block of the pair that evaluates to true.

The simulation ends when the current time is greater that the stop time and there are no more events in the event queue.

Class Name:	Simulation
SuperClass:	Object
SubClasses:	()
Instance Variables:	(stopTime currentTime eventQueue whenQueue)
Class Variables:	()
Pool Dictionaries:	()

Class Methods:

new

Answer a new instance of the receiver and initialize it.

Instance Methods:

atEnd

Answer true if the receiver is finished. The receiver is over when current time is greater than stop time and the eventQueue is empty, or the eventQueue runs out.

atTime: aTime doAction: anAction

Schedule an event to occur in the receiver. anAction is added to the events calendar for future processing.

doAction: anAction

Schedule an event to occur now in the receiver. anAction is added

to the events calendar for processing at the current time.

eventQueue
Answer the event queue.

eventQueue: aQueue
Set the event queue.

executeAction: anAction
Execute anAction. anAction can be a Block with no arguments or an Array of the form #(object message arguments). This method will execute either representation.

initialize
Initialize the receiver. Set current time and stop time to zero. Create a new eventQueue and whenQueue.

nextEvent
Answer the nextEvent to be processed by the receiver. Remove it from the events calendar. Check for attempts to set the simulated time backwards or no next event in the queue. Set the current time to be the time of the event.

plusTime: aTime doAction: anAction
Schedule an event to occur in the receiver. anAction is added to the events calendar for future processing at the current time plus aTime.

processEvent: anEvent
Process the next event. Execute the action found in anEvent. Evaluate the condition blocks for each member of the whenQueue. For each of these that evaluates to true, evaluate the actionBlock.

runFrom: startTime to: endTime
Run the receiver from startTime to endTime. Process events while the receiver is not at its end. This is the main entry point for the receiver.

reinitialize
Reinitialize after the simulation finishes in preparation for the next simulation.

sortBlock: aBlock
Add aBlock as an additional sort for the eventQueue to be invoked if the time of the actions in the queue are equal. Resort the queue.

time
Answer the current simulation time.

whenBlock: conditionBlock doBlock: actionBlock
When conditionBlock evaluates to true, execute actionBlock. conditionBlock is evaluated after the execution of every event on the receiver.

<u>Example:</u>

The following code will run a single server single queue simulation from time 0 to time 60. Clients will arrive starting at time 3 with an interarrival time of exactly 5. Service begins when the server is not busy and there is a client waiting in the queue. Service takes exactly 10 time units.

```
| sim queue start end busy |
busy := false.                      "Server start off idle"
queue := OrderedCollection new. "Queue starts off empty"
sim := Simulation new.
start := [
        queue addLast: #client. "EnQueue client"
        sim time < 60 ifTrue: [ "Schedule next client"
            sim plusTime: 5 doAction: start]].
end := [busy := false]. "End of service, free server"
sim
    whenBlock:              "-- can we process client?"
        [queue notEmpty & busy not]
    doBlock: [              "-- yes, so start service"
        queue removeFirst."DeQueue client"
        busy := true.       "Server is now busy"
        sim                 "Schedule end of service"
            plusTime: 10
            doAction: end].
sim atTime: 3 doAction: start.  "Schedule first event"
sim runFrom: 0 to: 60           "Run the simulation"
```

3.9 Task

Task objects implement the basic unit of activity within a plan. Every task has a name, a start time, a duration and some resources. Associated with the task name are a set of task rules implemented in the PROLOG class TaskRules.

The special class method #scheduleForLoads answer a collection of nine tasks that have been initialized for the load management domain.

Tasks are compared using the #= operation.

The methods #beginExecution: and #endExecution: are called at the start and end of the execution of the task. These methods attempt to acquire and release resources from the environment. They answer true or false to indicate that the task could start and end properly.

The methods #asList and #fromList: convert a task from the Smalltalk to the PROLOG representation of a task.

Class Name: Task
SuperClass: Object
SubClasses: ()
Instance Variables: (name startTime duration resources)
Class Variables: ()

Pool Dictionaries: ()

Class Methods:

new

Answer a new instance of the receiver and initialize it.

scheduleForLoads

This method answers a collection of loads that are initialized to solve a load scheduling problem. Uncomment the code that sets load rules the first time this code is executed.

Instance Methods:

<= aTask

Answer true if the receiver is less than or equal to aTask. This method returns true if the receiver starts before aTask.

= aTask

Answer true if the receiver is equal to aTask. This method returns true if the receiver and aTask are considered the same (have the same name)

>= aTask

Answer true if the receiver is greater than or equal to aTask. This method returns true if the receiver ends before aTask.

acquireResources: anEnvironment

Attempt to acquire all the resources from anEnvironment that the receiver needs in order to begin execution. If any one resource is not acquired, release any successfully acquired resources and answer false.

asList

Answer the receiver as a list of instance variable values or unbound PROLOG variables (LogicRefs). The list is always a triple of the form #(name startTime duration). When a start time or duration is nil, an unbound PROLOG variable is placed in the list.

beginExecution: anEnvironment

Begin the execution of the receiver in anEnvironment. Answer true if the receiver can begin in anEnvironment. This method acquires any required environmental resources.

canBeAdded: anEnvironment

Answer the true if the receiver can be added to a plan with respect to anEnvironment. Create an instance of the PROLOG object SelectorRules and invoke the predicate "canBeAdded". Answer true if the predicate succeeds.

canBeRemoved: anEnvironment

Answer the true if the receiver can be removed from a plan with respect to anEnvironment. Create an instance of the PROLOG object SelectorRules and invoke the predicate "canBeRemoved". Answer true if the predicate succeeds.

canOccur: anEnvironment

Answer the true if the receiver can occur in an anEnvironment.

Create an instance of the PROLOG object SelectorRules and invoke the predicate "canOccur". Answer true if the predicate succeeds.

copy

Answer a copy of the receiver. This is reimplemented as a deepCopy to ensure that the resources are also copied. Otherwise the receiver and its copy would share the same instance of a resource dictionary and may be destructively modified.

duration

Answer the duration of the receiver.

duration: aTime

Set the duration of the receiver.

endExecution: anEnvironment

End the execution of the receiver in anEnvironment. This method releases any acquired environment resources.

endsBefore: aTask

Answer true if the receiver ends before aTask ends.

endTime

Answer the endTime of the receiver. This value is calculated by adding startTime and duration.

fromList: aList

Set the receiver from a list of values. The list is always a triple of the form #(name startTime duration). The startTime and duration in the list may be PROLOG variables. If so, they need to be evaluated to get their Smalltalk values.

initialize

Initialize the instance variables. Create an empty resource dictionary.

interval

Answer the interval over which the receiver occurs (start time to end time).

name

Answer the name of the receiver.

name: aString

Set the name of the receiver. The name of the receiver must be a symbol. If aString is not a Symbol, convert it into one.

nonIntersections: anInterval

Answer a collection of intervals that represent the non-intersections of the interval that the receiver occurs on and anInterval.

overlaps: aTask

Answer true if the receivers start and end time overlap aTask's start and end times.

printOn: aStream

Append the ASCII representation of the receiver on aStream. Print task name and start and end time on aStream.

releaseResources: anEnvironment

Attempt to release all the resources that were acquired by the receiver back into anEnvironment.

resources
Answer the resources used by the receiver.

rules
Answer the PROLOG code that is associated with the receiver. This is a horn clause in the class TaskRules that has the same name as the receiver.

rules: aString
Set the PROLOG code that is associated with the receiver. This is a horn clause in the class TaskRules that has the same name as the receiver. Compile and install the code.

startsBefore: aTask
Answer true if the receiver starts before aTask starts.

startTime
Answer the startTime of the receiver.

startTime: aTime
Set the startTime of the receiver.

stopTime
Answer the endTime of the receiver.

unbind
Unbind instance variables. Set the start time and duration of the receiver to nil. When these are nil, the #asList method for the receiver will replace them with unbound PROLOG variables.

verify
Verify the that the task rules for the receiver succeed. If there is no task rule for the receiver, succeed. Evaluate the PROLOG predicate in the class TaskRules of the same name as the receiver. Answer true or false.

Example:

The following code fragment will create a new instance of a task called "task54" with start time 5, duration 20. Valid values for start time will be in the range 5 to 10.

```
| task rule |
task := task new name: 'task54'.        "create task"
rule :=                                  "create rule string"
'task54 (task54) :-
     between (5, 10, start),
     task (task54, start, 20).'.

task
     startTime: 5;         "task starts at time 5"
     duration: 20;         "task lasts for 20 time units"
     rules: rule.          "set the task rule"
task
```

Appendix D: Source Code

The Smalltalk objects that implement cased based reasoning within the Case Based Planning System can be found in the following files on the CBPS Version 2 source disk:

CBPS	-	cbps.cls
CommonRules	-	commnrls.cls
Environment	-	envrnmnt.cls
Evaluator	-	evaluatr.cls
EvaluatorRules	-	evltrrls.cls
Executor	-	executor.cls
NoteTaker	-	notetakr.cls
Plan	-	plan.cls
Replanner	-	replannr.cls
Selector	-	selector.cls
SelectorRules	-	slctrrls.cls
Simulation	-	simulatn.cls
Task	-	task.cls
TaskRules	-	taskruls.cls
file in file	-	cbps.st
misc. methods	-	cbps.mth

The Smalltalk objects that implement a user interface to the Case Based Planning System can be found in the following files on the CBPS Version 2 source disk:

CBPSBrowser	-	cbpsbrws.cls
DialogBox	-	dialogbx.cls
FieldEditor	-	fildedtr.cls
Field	-	field.cls
LibraryBrowser	-	lbrrybrw.cls
PlanBrowser	-	plnbrwsr.cls
PlotPane	-	plotplane.cls
TaskBrowser	-	tskbrwsr.cls
file in file	-	cbpsuser.st
misc. methods	-	cbpsuser.mth
title pane goodie	-	titlepan.prj

CBPS
Planning Objects
Source Code
Listing

```
Object subclass: #CBPS
  instanceVariableNames:
    'plan library environment selector executor planner evaluator '
  classVariableNames: ''
  poolDictionaries: '' !

!CBPS class methods !

example
        "CBPS example."
    (self exampleCBPS)
        doPlanning;
        inspect!

exampleCBPS
        "CBPS example."
    | env t |
    t := Task scheduleForLoads.
    env := Environment new.
    env resources at: #Power put: 50.
    env requiredTasks:
        (t copyFrom: 1 to: 5) deepCopy.
    ^self new
        environment: env;
        library: self exampleLibrary!

exampleLibrary
        "CBPSLibraryBrowser exampleLibrary
        answers a library of plans.  Uncomment
        the rule code the first time this code
        is run."
    | lib plan sched name |
    sched := Task scheduleForLoads.
    lib := OrderedCollection new.
    1 to: 9 do: [:i |
        plan := Plan new
            name: (name := 'plan', i printString).
        sched do: [:l |
            plan addTask: l copy].
        lib add: plan.
        plan removeTask: (plan tasks at: i)].
    ^lib!

new
        "Answer a new instance of the
        receiver and initialize it."
    ^super new initialize! !


!CBPS methods !

doPlanning
        "Perform the activities that make
        up Case Based Reasoning.  Select a
        plan from the plan library, execute
```

```
        it (perhaps failing at some tasks),
        and evaluate the results."
    self
        selectPlan;
        executePlan;
        evaluatePlan.
    ^plan!

environment
        "Answer the current CBPS environment."
    ^environment!

environment: anEnvironment
        "Set the current CBPS environment."
    environment := anEnvironment!

evaluatePlan
        "Evaluate the current plan
        using an Evaluator."
    self evaluator evaluatePlan!

evaluator
        "Answer the plan Evaluator ready to evaluate
        the current plan with respect to the current
        environment and plan library."
    evaluator
        environment: environment;
        library: library;
        plan: plan.
    ^evaluator!

executePlan
        "Execute the current plan
        using an Executor."
    self executor executePlan!

executor
        "Answer a plan Executor ready to execute
        the current plan in the current environment."
    executor
        environment: environment;
        plan: plan.
    ^executor!

initialize
        "Initialize the instance variables.  Create
        a default empty plan, a default environment,
        a default empty library and the Selector,
        Executor, and Evaluator objects that will
        perform the basic CBPS functions."
    plan := Plan new.
    environment := Environment new.
    library := OrderedCollection new.
    selector := Selector new.
    executor := Executor new.
    "planner := Planner new."
    evaluator := Evaluator new!
```

```
library
        "Answer the plan library."
    ^library!

library: aPlanLibrary
        "Set the plan library."
    library := aPlanLibrary!

plan
        "Answer the current plan for the CBPS."
    ^plan!

plan: aPlan
        "Set the current plan for the CBPS."
    plan := aPlan!

selector
        "Answer the plan Selector ready to select
        a plan based on the current environment
        and plan library."
    selector
        executor: executor;
        environment: environment;
        library: library.
    ^selector!

selectPlan
        "Select a plan using a Selector."
    plan := self selector selectPlan.
    ^plan! !
```

```
Prolog subclass: #CommonRules
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!CommonRules class logicMethods ! !


!CommonRules logicMethods !

"A general purpose integer generator/tester."

"This is the tester part of the between relation.
If the arguments are all known, simply do the test.
This is reimplemented only for efficiency."

between (start, end, next) :-
    nonvar (start), nonvar (end), nonvar(next),
    le (start, next), le (next, end), !!.

"This is the generator portion of the between
relation.  It will answer successive values of
'next' when called with 'start' and 'end' bound."

between (next, end, next).
between (start, end, value) :-
    is (next, start + 1), le (next, end),
    between (next, end, value).!

"Succeed if the tasks are distinct
(do not overlap) or if there is no
task1 or task2."

distinct (task1, task2) :-
    or (var (task1), var (task2)), !!.
distinct (task1, task2) :-
    not (overlaps (task1, task2)).!

"Succeed if task1 starts after task2 ends."

follows (task1, task2) :-
    precedes (task2, task1).!

"A general purpose set membership generator/tester."

member ([first | rest], first).
member ([first | rest], element) :-
    member (rest, element).!

"A general purpose multiply relation."

multiply (a, b, c) :-
    nonvar (c), nonvar (b), is (a, c / b), !!.
multiply (a, b, c) :-
    nonvar (c), nonvar (a), is (b, c / a), !!.
```

```
multiply (a, b, c) :-
    nonvar (a), nonvar (b), is (c, a * b), !!.!

"Succeed if the tasks overlap or
if there is no task1 or task2."

overlaps (task1, task2) :-
    or (var (task1), var (task2)), !!.
overlaps (task1, task2) :-
    task (task1, s1, _, e1),
    task (task2, s2, _, e2),
    not (or (le (e2, s1), ge (s2, e1))).!

"Succeed if task1 ends before task2
starts or if there is no task1 or task2."

precedes (task1, task2) :-
    or (var (task1), var (task2)), !!.
precedes (task1, task2) :-
    task (task1, _, _, e1),
    task (task2, s2, _, _),
    le (e1, s2).!

"A general purpose sum relation."

sum (a, b, c) :-
    nonvar (c), nonvar (b), is (a, c - b), !!.
sum (a, b, c) :-
    nonvar (c), nonvar (a), is (b, c - a), !!.
sum (a, b, c) :-
    nonvar (a), nonvar (b), is (c, a + b), !!.!

"Get/Set task values."

task (task, start, duration) :- var (task), !!.
task ([name, start, duration], start, duration) :- !!.
task (task, start, duration, end) :- var (task), !!.
task ([name, start, duration], start, duration, end) :-
    sum (start, duration, end).! !
```

```
Object subclass: #Environment
  instanceVariableNames:
    'unorderedPlan resources failBlock '
  classVariableNames: ''
  poolDictionaries: '' !

!Environment class methods !

new
        "Answer a new instance of the
        receiver and initialize it."
    ^super new initialize! !


!Environment methods !

acquireAmount: aValue ofResource: aName
        "Use up some of the available resource
        named by aName if possible.  Answer true
        if the resource was acquired, else false.
        Execute the fail block if none available."
    | available |
    available := resources at: aName ifAbsent: [^false].
    available - aValue < 0
        ifTrue: [
            (failBlock value: aName) notNil
                ifTrue: [^false]].
    available := available - aValue.
    resources at: aName put: available.
    ^true!

copy
        "Answer a copy of the reciever.  Be sure
        to supply a deep copy of the resources in
        the receiver.  Otherwise, copies of the
        reciever can destructively modify these
        resources."
    | copy |
    copy := super copy.
    copy resources: resources deepCopy.
    ^copy!

failBlock: aBlock
        "Set the failBlock of the receiver.
        The failBlock is executed every time
        a resource cannot be acquired from
        the reciever."
    failBlock := aBlock!

initialize
        "Initialize the instance variables.
        Create a default (empty) unorderedPlan,
        a failBlock that does nothing and an
        empty dictionary to hold resources."
    failBlock := [:resource | ].
```

```
    unorderedPlan := Plan new
        name: 'plan' asSymbol.
    resources := Dictionary new!

isBetterPlan: plan1 thanPlan: plan2
        "Answer true if plan1 is the better of
        the two plans with respect to plan2 and
        the receiver.  This method uses the same
        criteria that Dwight uses."
    "
    ----- OLD CODE -----
    | e1 m1 e2 m2 requiredTasks |
    requiredTasks := self requiredTasks.
    e1 := (plan1 extraTasks: requiredTasks) size.
    e2 := (plan2 extraTasks: requiredTasks) size.
    m1 := (plan1 missingTasks: requiredTasks) size.
    m2 := (plan2 missingTasks: requiredTasks) size.
    (e1 = e2 and: [m1 = m2]) ifTrue: [
        ^plan1 failures < plan2 failures].
    ^(e1 <= e2) and: [m1 <= m2]
    ----- END OLD CODE -----"

    | p1 p2 |
    p1 := plan1.  p2 := plan2.
    ^(SelectorRules new :?
        better (p1, p2, self),
        exit ()) notNil!

releaseAmount: aValue ofResource: aName
        "Release some of the available resource
        named by aName back to the receiver."
    | available |
    available := resources at: aName ifAbsent: [^false].
    available := available + aValue.
    resources at: aName put: available.
    ^true!

requiredTasks
        "Answer a collection of the tasks
        that are required to execute in
        the receiver.  These are the tasks
        of the unorderedPlan."
    ^unorderedPlan tasks!

requiredTasks: tasks
        "Set the collection of the tasks
        that must execute in the receiver.
        These are the tasks of the unorderedPlan.
        Any previous task are removed and copies
        of new tasks are added."
    unorderedPlan tasks copy do: [:aTask |
        unorderedPlan removeTask: aTask].
    tasks do: [:aTask |
        unorderedPlan addTask: aTask copy]!

resources
        "Answer the dictionary of available resources."
```

```
    ^resources!

resources: aDict
        "Set the dictionary of available resources."
    resources := aDict!

unorderedPlan
        "Answer the unordered plan.  This plan
        represents the current requirements of the
        operator."
    ^unorderedPlan! !
```

```
Object subclass: #Evaluator
  instanceVariableNames:
    'plan library environment '
  classVariableNames: ''
  poolDictionaries: '' !

!Evaluator class methods ! !


!Evaluator methods !

addPlan
        "Add a new plan in the library."
    | code |
    code := (Plan new name: #plan) rules.
    library add: (plan rules: code)!

environment: anEnvironment
        "Set the environment of evaluation."
    environment := anEnvironment!

evaluatePlan
        "This method evaluates the plan and
        takes an action with respect to the
        plan library."
    | notes |
    self hasViolations
        ifTrue: [^self].
    "
    ----- OLD CODE -----
    notes := plan history.
    plan tasks isEmpty
        ifTrue: [^self forgetPlan].
    (library includes: plan) not
        ifTrue: [
            notes successes >= notes failures
                ifTrue: [^self addPlan].
            ^self forgetPlan].
    notes failures > notes successes
        ifTrue: [^self removePlan].
    self updatePlan
    ----- END OLD CODE -----"

    EvaluatorRules new :?
        evaluate (plan, library, self),
        exit().!

forgetPlan
        "Do nothing.  Do not add, remove, or update
        the original plan in the plan library."!

hasViolations
        "Answer true if the receiver has any
        kind of plan or task violations."
    ^(self taskViolations notEmpty or:
```

```
        [self planViolation])!

library: aPlanLibrary
        "Set the plan library."
    library := aPlanLibrary!

plan: aPlan
        "Set the plan to be evaluated."
    plan := aPlan!

planViolation
        "Answer true if the plan rules
        for the current plan have been
        violated."
    ^plan verify not!

removePlan
        "Remove an old plan from the plan library."
    library remove: plan ifAbsent: [
        self error: plan name, ' is missing from lib']!

taskViolations
        "Answer a collection of the tasks
        that have task rule violations."
    ^plan tasks reject: [:aTask | aTask verify]!

updatePlan
        "Add a the updated plan to the library.
        First remove the original plan (if any)
        from the plan library and then add the
        current plan."
    library remove: plan ifAbsent: [].
    library add: plan! !
```

```
CommonRules subclass: #EvaluatorRules
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!EvaluatorRules class logicMethods ! !


!EvaluatorRules logicMethods !

"Succeed if plan can provide the
criteria for evaluation."

criteria (plan, successes, failures) :-
    is (successes, plan history successes),
    is (failures, plan history failures).!

"Evaluate the plan with respect to
a plan library and an evaluator."

evaluate (plan, library, evaluator) :-
    newPlan (plan, library),
    criteria (plan, successes, failures),
    ge (successes, failures),
    is (_, evaluator addPlan), !!.

evaluate (plan, library, evaluator) :-
    newPlan (plan, library),
    is (_, evaluator forgetPlan), !!.

evaluate (plan, library, evaluator) :-
    criteria (plan, successes, failures),
    gt (failures, successes),
    is (_, evaluator removePlan), !!.

evaluate (plan, library, evaluator) :-
    is (_, evaluator updatePlan).!

"Succeed if plan is a new plan with
respect to the plans in the library."

newPlan (plan, library) :-
    is (true, (library includes: plan) not).! !
```

```
Object subclass: #Executor
  instanceVariableNames:
    'plan environment simulation currentTask executingTasks executedTasks '
  classVariableNames: ''
  poolDictionaries: '' !

!Executor class methods !

new
        "Answer a new instance of the
        receiver and initialize it."
    ^super new initialize! !


!Executor methods !

beginTask: aTask
        "Start aTask in the reciever.  This method is
        executed by the simulation when aTask starts.
        Set the current task to be aTask and ask it to start.
        If aTask cannot start, take a task failure action.
        If the task starts, update the executing tasks
        collection and schedule the task end in the
        simulation."
    | action |
    currentTask := aTask.
    (aTask beginExecution: environment)
        ifFalse: [^self failTask: aTask].
    executingTasks add: aTask.
    action := OrderedCollection new.
    action
        add: self; add: #endTask:;
        add: (Array with: aTask).
    simulation
        atTime: aTask endTime
        doAction: action!

currentTask
        "Answer the task that is about
        to start in the receiver."
    ^currentTask!

doNothingOnFailure
        "Do nothing if a failure happens
        in the environemnt.  This method
        sets the environment failBlock
        if a block that does nothing."
    environment failBlock: [:resource | ]!

dropPlan
        "Forget the executing the rest of the plan.
        Release any resources that may be acquired
        by the tasks that are currenly executing and
        reinitialize the reciever."
    executingTasks do: [:aTask |
```

```
        aTask releaseResources: environment].
    self reinitialize!

dropTask: aTask
        "Drop aTask from the receiever.  A task is
        assumed to be the current task.  The task is
        removed from the collection of tasks executing
        and tasks executed (if present) and then removed
        from the plan.  Because the task was unable to
        start, it also did not schedule and end event
        for itself in the simulation."
    self tasksExecuting
        remove: aTask ifAbsent: [].
    self tasksExecuted
        remove: aTask ifAbsent: [].
    plan removeTask: aTask!

endTask: aTask
        "End aTask in the reciever.  This method
        is executed by the simulation when aTask
        ends.  Remove aTask from the collection of
        tasks that are currently executing and add
        it to the collection of tasks that have
        been executed.  Note that the task has
        succeeded."
    aTask endExecution: environment.
    executingTasks remove: aTask ifAbsent: [].
    executedTasks add: aTask.
    self succeedTask: aTask.!

environment
        "Answer the environment of execution."
    ^environment!

environment: anEnvironment
        "Set the environment of execution."
    environment := anEnvironment!

executePlan
        "Execute the plan in the receiver.  Schedule
        the start event for each of the tasks in the
        plan.  Run the simulation from the start and
        end time of the plan."
    plan tasks do: [:aTask |
        self scheduleTask: aTask
            atTime: aTask startTime].
    simulation
        runFrom: 0 "plan startTime"
        to: plan endTime!

failTask: aTask
        "Indicate that aTask has failed.  Ask the
        plan history to remember the failure."
    plan history
        failTask: aTask
        in: environment!
```

```
initialize
        "Initialize the receiver.  Create a simulation
        to simulate the execution tasks.  Initialize the
        collections of tasks that are executing and tasks
        that have been executed to be empty.  Set the
        sortBlock of the simulation to be a block that
        ensures that when a #startTask: and EendTask: occur
        at the same time, the #endTask: is processed first.
        This is done to ensure that resources are released
        before they are acquired if events happen at the
        same time."
    simulation := Simulation new.
    simulation sortBlock: [:a :b |
        ((a at: 2) = #endTask:)].
    executingTasks := OrderedCollection new.
    executedTasks := OrderedCollection new!

plan
        "Answer the current plan."
    ^plan!

plan: aPlan
        "Set the plan for the reciever to execute."
    plan := aPlan!

reinitialize
        "Reinitialize the receiver.  Ask the
        simulation to be reinitialized (clear
        event queues, current time, etc.) and
        set the collections of tasks executing
        and executed to be empty.  This method
        should be called before executing another
        plan."
    simulation initialize.
    simulation sortBlock: [:a :b |
        ((a at: 2) = #endTask:)].
    executingTasks := OrderedCollection new.
    executedTasks := OrderedCollection new.
    currentTask := nil!

replanOnFailure
        "Set the receiver to invoke the
        Replanner when a failure occurs.
        This method sets the failBlock of
        the environment to invoke the
        Replanner."
    | task |
    environment failBlock: [:resource |
        task := self currentTask.
        Replanner new :?
            failure (resource, task, self),
            exit ()]!

scheduleTask: aTask atTime: aTime
        "Schedule aTask to occur in
        the simulation at aTime.  Set
        the start time of aTask to
```

```
        be aTime.  Update the start
        and end times of the current
        plan."
    | action |
    aTask startTime: aTime.
    plan calculateTimes.
    action := OrderedCollection new.
    action
        add: self; add: #beginTask:;
        add: (Array with: aTask).
    simulation atTime: aTime
        doAction: action!

simulation
        "Answer the simulation used by the
        receiver to simulate the execution
        of tasks."
    ^simulation!

succeedTask: aTask
        "Indicate that aTask has succeeded.  Tell
        the plan history to remember the success."
    plan history
        succeedTask: aTask
        in: environment.!

tasksExecuted
        "Answer a collection of the tasks
        that have been executed."
    ^executedTasks!

tasksExecuting
        "Answer a collection of the tasks
        that are currently executing."
    ^executingTasks!

tasksToExecute
        "Answer a collection of the tasks
        that have yet to be executed.  This
        collection is computed from the plan
        and the other two task collections."
    | toExecute |
    toExecute := OrderedCollection new
    plan tasks do: [:aTask |
        ((executingTasks includes: aTask) or:
            [executedTasks includes: aTask])
                ifFalse: [toExecute add: aTask]].
    ^toExecute!

time
        "Answer the current time.  This is
        the current time in the simulation."
    ^simulation time! !
```

```
Object subclass: #NoteTaker
  instanceVariableNames:
    'plan successes failures '
  classVariableNames: ''
  poolDictionaries: '' !

!NoteTaker class methods !

new
        "Answer a new instance of the
        receiver and initialize it."
    ^super new initialize! !


!NoteTaker methods !

failTask: aTask in: anEnvironment
        "Record the fact that aTask has
        failed in anEnvironment."
    failures := failures + 1!

failures
        "Answer the number of failures."
    ^failures!

initialize
        "Initialize the instance variables.
        Set failures and successes to zero."
    successes := failures := 0!

plan: aPlan
        "Set the current plan for the reciever."
    plan := aPlan!

printOn: aStream
        "Append the ASCII representation
        of the reciever on aStream.  Show
        the number of successes and failures."
    aStream
        nextPutAll: successes printString, ' success(s), ';
        nextPutAll: failures printString, ' failure(s)'!

succeedTask: aTask in: anEnvironment
        "Record the fact that aTask has
        succeeded in anEnvironment."
    successes := successes + 1!

successes
        "Answer the number of successes."
    ^successes! !
```

```
Object subclass: #Plan
  instanceVariableNames:
    'name tasks startTime endTime history '
  classVariableNames: ''
  poolDictionaries: '' !

!Plan class methods !

new
        "Answer a new instance of the
        receiver and initialize it."
    ^super new initialize! !


!Plan methods !

= aPlan
        "Answer true if the receiver is
        equal to aPlan.   This method returns
        true if the receiver and aPlan have
        the equal names and equal collections
        of tasks."
    | block |
    block := [:a :b | a name <= b name].
    ^name = aPlan name and:
        [(tasks asSortedCollection: block) =
            (aPlan tasks asSortedCollection: block)]!

addTask: aTask
        "Add aTask to the receiver.  Recalculate
        the start and end times of the receiver."
    tasks add: aTask.
    self calculateTimes!

asList
        "Answer the receiver as a list of
        lists.  Each of the lists is a tasks
        in the reciever that has been converted
        into a list."
    ^(tasks collect: [:aTask | aTask asList]) asList!

calculateTimes
        "Calculate the start and end times
        for the receiver.  Select the minimum
        and maximum times from the tasks that
        have defined start and end times."
    startTime := endTime := nil.
    tasks do: [:aTask |
        (aTask startTime notNil and: [
            aTask duration notNil]) ifTrue: [
                startTime := (startTime isNil
                    ifTrue: [ aTask startTime]
                    ifFalse: [startTime min: (aTask startTime)]).
                endTime := (endTime isNil
                    ifTrue: [aTask endTime]
```

```
				ifFalse: [endTime max: (aTask endTime)])]].
	startTime isNil ifTrue: [startTime := 0].
	endTime isNil ifTrue: [endTime := 0]!

copy
		"Answer a deep copy of the receiver.  It is
		essential that any copy of the receiver also
		have its own copy of the plan history.
		Otherwise, the receiver and the copy will share
		the exact same history."
	| copy |
	copy := self class new
		name: name;
		history: history copy.
	tasks do: [:aTask |
		copy addTask: aTask copy].
	^copy!

endTime
		"Answer the endTime of the receiver."
	^endTime!

extraTasks: someTasks
		"Answer a collection of extra tasks
		in the receiver with respect to the
		tasks found in someTasks."
	^tasks select: [:aTask |
		(someTasks includes: aTask) not]!

failures
		"Answer the number of failures for
		the reciever.  Ask the plan history."
	^history failures!

fromList: aList
		"Set the tasks in the receiver from a list of
		tasks where each task is a list.  For each list,
		find the task in the reciever that corresponds
		to the list and ask it to initialize itself from
		the list."
	| newTasks newTask aTask |
	newTasks := aList asArray.
	newTasks do: [:list |
		newTask := list asArray.
		aTask := tasks
			detect: [:t | t name = newTask first]
			ifNone: [].
		aTask notNil ifTrue: [
			aTask fromList: list]]!

history
		"Answer the history of receiver."
	^history!

history: aNoteTaker
		"Set the history of receiver."
	history := aNoteTaker!
```

```
initialize
        "Initialize the instance variables.
        Create an empty collection of tasks,
        a new history and set the start and
        end time to zero."
    name := '*Unknown*'.
    tasks := OrderedCollection new.
    history := NoteTaker new plan: self.
    startTime := endTime := 0!

missingTasks: someTasks
        "Answer a collection of missing tasks
        in the receiver with respect to the
        tasks found in someTasks."
    ^someTasks select: [:aTask |
        (tasks includes: aTask) not]!

name
        "Answer the name of the receiver."
    ^name!

name: aString
        "Set the name of the receiver.  The name
        of the receiver must be a symbol. "
    name := aString asSymbol!

printOn: aStream
        "Append the ASCII representation
        of the reciever on aStream.  Print
        plan name, start time and end time
        and the number of tasks."
    aStream
        nextPutAll: name,': ';
        nextPutAll: '[',startTime printString,'-',
            endTime printString,'], ';
        nextPutAll: tasks size printString, ' task(s) '", ';
        nextPutAll: history printString"!

removeTask: aTask
        "Remove aTask to the receiver.  Complain
        if the task cannot be removed.  Recalculate
        the start and end times of the receiver."
    tasks remove: aTask ifAbsent: [
        ^self error: 'removing unknown task'].
    self calculateTimes!

replaceHead: aString
        "Replace the head of the rule found
        in aString with the a new rule head
        that has the same name as the receiver
        and takes a single list of the tasks
        in the receiver as a parameter.  Answer
        the new rule."
    | aStream exit code |
    exit := false.
    aStream := ReadStream on: aString.
```

```
    [exit not & aStream atEnd not] whileTrue: [
        [aStream atEnd not and: [aStream next ~= $:]]
            whileTrue: [].
        aStream peek = $- ifTrue: [exit := true].
        aStream atEnd ifFalse: [aStream next]].
    exit ifFalse: [''].
    code := self ruleHead, (aStream copyFrom:
        aStream position + 1 to: aString size).
    ^code!

ruleHead
        "Answer a string that is the head of
        the plan rule with the tasks of the
        plan in a list."
    | head |
    head := name, ' (['.
    tasks do: [:aTask |
        head := head, aTask name.
        aTask = tasks last ifFalse: [
            head := head, ', ']].
    head := head, ']) :-'.
    ^head!

rules
        "Answer the PROLOG code that is associated
        with the receiver.  This is a horn clause
        in the class TaskRules that has the same
        name as the receiver."
    ^TaskRules sourceCodeAt: (self name,':') asSymbol!

rules: aString
        "Set the PROLOG code that is associated
        with the receiver.  This is a horn clause
        in the class TaskRules that has the same
        name as the receiver.  Replace the head
        of the rule and compile and install the
        new rule in the class TaskRules."
    | result code aStream |
    (code := self replaceHead: aString) isEmpty
        ifTrue: [^self].
    (code = self ruleHead) ifTrue: [
        aStream := WriteStream on: ''.
        aStream
            nextPutAll: code; cr;
            nextPutAll: '    !!.'.
        code := aStream contents].
    result := TaskRules compileLogic: code.
    result isNil ifTrue: [^self].
    Smalltalk
        logPrologSource: code
        forSelector: result key
        inClass: TaskRules!

startTime
        "Answer the startTime of the receiver."
    ^startTime!
```

```
tasks
        "Answer the tasks of the receiver."
    ^tasks!

verify
        "Verify the that the plan rules for
        the reciever succeed.  If there is
        no plan rule for the receiver, succeed.
        Answer true or false."
    | list |
    (TaskRules canUnderstand: (name,':') asSymbol)
        ifFalse: [^true].  "no rules => values OK"
    list := self asList.
    ^(TaskRules new :? plan (list), exit()) notNil! !
```

```
CommonRules subclass: #Replanner
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!Replanner class logicMethods ! !


!Replanner logicMethods !

"This code gets executed when a task fails to
acquire a resource."

"Drop the entire plan if #Power fails for any task."
"failure (#Power, task, executor) :-
    is (_, executor dropPlan)."

"Drop the task if #Power for #load3 fails."
"failure (#Power, task, executor) :-
    is (#load3, task name asSymbol),
    is (_, executor dropTask: task)."

"Default is to wait 1 minute and reschedule
the failed task.  This may violate task and
plan rules.  No check is made on the new task
start time until plan evaluation is done."
failure (resource, task, executor) :-
    is (_, executor
            scheduleTask: task value
            atTime: executor time + 1).! !
```

```
Object subclass: #Selector
  instanceVariableNames:
    'plan environment library executor '
  classVariableNames: ''
  poolDictionaries: '' !

!Selector class methods ! !


!Selector methods !

addMissingTasks: aPlan
        "Add any missing tasks to aPlan."
    | missing |
    missing := aPlan missingTasks:
        (environment requiredTasks).
    missing do: [:aTask |
        (aTask canBeAdded: environment)
            ifTrue: [aPlan addTask: aTask]]!

constructPlan
        "Adds missing tasks, removes extra tasks, and
        replace equal tasks.  Answers a new plan with
        a new name that has an 'x' appended to the end
        to be verified."
    | newPlan names |
    newPlan := plan copy.
    self
        addMissingTasks: newPlan;
        removeExtraTasks: newPlan.
    plan = newPlan ifFalse: [
        newPlan
            name: (plan name, 'x');
            history: NoteTaker new.
        names := library collect: [:p | p name].
        [names includes: newPlan name] whileTrue: [
            newPlan name: (newPlan name, 'x')]].
    self replaceEqual: newPlan.
    plan := newPlan.
    ^plan!

environment: anEnvironment
        "Set the environment of the reciever."
    environment := anEnvironment!

executor: anExecutor
        "Set the plan executor.  This executor
        will be used during the plan verification
        process.  It gets invoked after the task
        and plan rules for the reciever have been
        satisfied."
    executor := anExecutor!

library: aPlanLibrary
        "Set the plan library to be
```

```
        searched for the best match
        by the receiver."
    library := aPlanLibrary!

locatePlan
        "Answer the best matching plan in the plan library
        with respect to the requirements.  A new (and empty)
        plan is answered if the library is empty or the
        requirements are empty.  Otherwise, the actual plan
        from the library is answered.  If you modify this
        plan directly (dont't make a copy) the library will
        get destructively updated.  The methods of the receiver
        are careful not to do this."
    | bestPlan requiredTasks |
    (library isEmpty or: [
        (requiredTasks := environment
            requiredTasks) isEmpty])
                ifTrue: [^plan := Plan new].
    bestPlan := library first.
    library do: [:aPlan |
        (environment isBetterPlan: aPlan
            thanPlan: bestPlan) ifTrue: [
                bestPlan := aPlan]].
    plan := bestPlan.
    ^plan!

removeExtraTasks: aPlan
        "Remove any extra tasks from aPlan."
    | extra |
    extra := aPlan extraTasks:
        (environment requiredTasks).
    extra do: [:aTask |
        (aTask canBeRemoved: environment)
            ifTrue: [aPlan removeTask: aTask]]!

replaceEqual: aPlan
        "Replace any tasks in aPlan with equal
        tasks from the operator input.  This will
        allow the operator to unbind any variables
        in the plan that was selected from the library."
    | tasks |
    tasks := environment requiredTasks.
    tasks do: [:aTask |
        (aPlan tasks includes: aTask)
            ifTrue: [
                aPlan
                    removeTask: aTask;  "remove equal task"
                    addTask: aTask]]!

selectPlan
        "Attempt to locate a plan that best matches
        the operator requirements from the plan library.
        Add missing tasks, remove extra tasks, and replace
        equal tasks.  Next, verfy that the plan will work in
        the environment."
    self
        locatePlan;
```

```
        constructPlan;
        verifyPlan.
    ^plan!

verifyPlan
        "Verify that the plan is expected to
        work in the environment.  This involves
        creating a new instance of the task and
        plan rule base (called TaskRules), setting
        the current plan and environment as well
        as any executor that may be provided, and
        issuing a PROLOG query that calls the
        verify() predicate.  Answer the plan if
        the verify succeeded, else answer nil."
    | list ruleBase |
    list := plan asList.
    ruleBase := TaskRules new.
    ruleBase
        setPlan: plan;
        environment: environment;
        executor: executor.
    (ruleBase :? verify (list)) isNil
        ifTrue: [^nil].
    ^plan! !
```

```
CommonRules subclass: #SelectorRules
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!SelectorRules class logicMethods ! !


!SelectorRules logicMethods !

"Succeed if plan1 is better than
plan2 with respect to an environment."

better(plan1, plan2, env) :-
    criteria (plan1, env, extra, missing, fail1),
    criteria (plan2, env, extra, missing, fail2),
    !!, lt (fail1, fail2).
better(plan1, plan2, env) :-
    criteria (plan1, env, extra1, missing1, _),
    criteria (plan2, env, extra2, missing2, _),
    le (extra1, extra2), le (missing1, missing2).!

"Succceed if the task can be added
to a plan in the environment."

canBeAdded (task, environment).!

"Succceed if the task can be removed
from a plan in the environment."

canBeRemoved (task, environment).!

"Succceed if the task can
occur in the environment."

canOccur (task, environment).!

"Succeed if the criteria for plan
selection can be obtained from the
plan and the environment."

criteria (plan, env, extra, missing, failures) :-
    is (extra, (plan extraTasks: env requiredTasks) size),
    is (missing, (plan missingTasks: env requiredTasks) size),
    is (failures, plan failures).! !
```

```
Object subclass: #Simulation
  instanceVariableNames:
    'stopTime currentTime eventQueue whenQueue '
  classVariableNames: ''
  poolDictionaries: '' !

!Simulation class methods !

new
        "Answer a new instance of the
        receiver and initialize it."
    ^super new initialize! !


!Simulation methods !

atEnd
        "Answer true if the receiver is finished.
        The receiver is over when current time is
        greater that stop time and the eventQueue
        is empty, or the eventQueue runs out."
    ^(currentTime > stopTime
        and: [eventQueue isEmpty])
            or: [eventQueue isEmpty]!

atTime: aTime doAction: anAction
        "Schedule an event to occur in
        the receiver.  anAction is added
        to the events calendar for future
        processing."
    eventQueue add:
        (Array
            with: aTime
            with: anAction)!

doAction: anAction
        "Schedule an event to occur now in
        the receiver.  anAction is added to the
        events calendar for processing at the
        current time."
    self
        atTime: currentTime
        doAction: anAction!

eventQueue
        "Answer the event queue."
    ^eventQueue!

eventQueue: aQueue
        "Set the event queue."
    eventQueue := aQueue!

executeAction: anAction
        "Execute anAction.  anAction can be a
        Block with no arguments or an Array of
```

```
        of the form #(object selector arguments).
        This method will execute either."
    (anAction isKindOf: Context)
        ifTrue: [^anAction value].
    (anAction at: 1) perform: (anAction at: 2)
        withArguments: (anAction at: 3)!

initialize
        "Initialize the receiver.  Set current
        time and stop time to zero.  Create a
        new eventQueue and whenQueue."
    currentTime := stopTime := 0.
    whenQueue := OrderedCollection new.
    eventQueue := SortedCollection sortBlock:
        [:a :b | a first <= b first]!

nextEvent
        "Answer the nextEvent to be processed
        by the receiver.  Remove it from the
        events calendar.  Check for attempts
        to set the simulated time backwards or
        no next event in the queue.  Set the
        current time to be the time of the
        event."
    | nextEvent nextTime |
    eventQueue isEmpty ifTrue: [
        ^self error: 'No next simulation event'].
    nextEvent := eventQueue removeFirst.
    nextTime := nextEvent first.
    nextTime < currentTime ifTrue: [
        ^self error: 'Attempt to set clock back'].
    currentTime := nextTime.
    ^nextEvent!

plusTime: aTime doAction: anAction
        "Schedule an event to occur in
        the receiver.  anAction is added
        to the events calendar for future
        processing at the current time
        plus aTime."
    self
        atTime: currentTime + aTime
        doAction: anAction!

processEvent: anEvent
        "Process the next event.  Execute the
        action found in anEvent.   Evaluate the
        condition blocks for each member of the
        whenQueue.  For each of these that evaluates
        to true, evalute the conditionBlock."
    self executeAction: (anEvent at: 2).
    whenQueue do: [:pair |
        pair first value ifTrue: [
            (pair at: 2) value]]!

runFrom: startTime to: endTime
        "Run the receiver from startTime to
```

```
        endTime.  Process events while the
        receiver is not at its end.

        Reinitialize after when the simulation
        finishes in preparation for the next
        simulation."

    | nextEvent |
    stopTime := endTime.
    currentTime := startTime.
    [self atEnd] whileFalse: [
        nextEvent := self nextEvent.
        self processEvent: nextEvent].
    self initialize!

sortBlock: aBlock
        "Add aBlock as an additional sort
        for the eventQueue to be invoked
        if the time of the actions in the
        queue are equal.  Resort the queue."
    | value |
    eventQueue := eventQueue
        asSortedCollection:  [:a :b |
            value := a first <= b first.
            a first = b first ifTrue: [
                value := (aBlock value: (a at: 2)
                    value: (b at: 2))].
            value]!

time
        "Answer the current simulation time."
    ^currentTime!

whenBlock: conditionBlock doBlock: actionBlock
        "When conditionBlock evaluates to true, execute
        actionBlock.  conditionBlock is evaluated after
        the execution of every event on the receiver."
    whenQueue add:
        (Array
            with: conditionBlock
            with: actionBlock)! !
```

```
Object subclass: #Task
  instanceVariableNames:
    'name startTime duration resources '
  classVariableNames: ''
  poolDictionaries: '' !

!Task class methods !

new
        "Answer a new instance of the
        receiver and initialize it."
    ^super new initialize!

scheduleForLoads
        "Load scheduleForLoads answers a collection
        of loads that are initialized to solve a
        load scheduling problem.  Uncomment the code
        that sets load rules the first time this
        code is executed."

    | l1 l2 l3 l4 l5 loads c name |
    loads := OrderedCollection new.
    loads add: (l1 := Task new
        name: 'load1';
"
        rules:
'load1 (load1) :-
    member ([30, 40], start),
    task (load1, start, 10).';
"
        startTime: 40;
        duration: 10).
    l1 resources at: #Power put: 10.
    loads add: (l2 := Task new
        name: 'load2';
"
        rules:
'load2 (load2) :-
    member ([30, 40], start),
    task (load2, start, 10).';
"
        startTime: 30;
        duration: 10).
    l2 resources at: #Power put: 10.
    loads add: (l3 := Task new
        name: 'load3';
"
        rules:
'load3 (load3) :-
    member ([40, 50], start),
    task (load3, start, 10).';
"
        startTime: l1 endTime;
        duration: 20).
    l3 resources at: #Power put: 2.
```

```
    c := 0.
    #(0 20 40 60) do: [:st |
        loads add: (14 := Task new
            name: (name := 'load4', (#(a b c d) at: (c := c + 1)));
"
        rules:
(name, ' (',name,') :-
    task (',name,',',st printString,',1).');
"
            startTime: st;
            duration: 1).
        14 resources at: #Power put: 40].
    c := 0.
    #((1 19) (21 9)) do: [:pair |
        loads add:
            (15 := Task new
                name: (name := 'load5', (#(a b) at: (c := c + 1)));
"
                rules:
(name, ' ('name,') :-
    task (',name,',',pair first printString,',',
        (pair at: 2) printString,').');
"
                startTime: pair first;
                duration: (pair at: 2)).
            15 resources at: #Power put: 40].
    ^loads! !


!Task methods !

<= aTask
        "Answer true if the receiver
        is less than or equal to aTask."
    ^self startsBefore: aTask!

= aTask
        "Answer true if the receiver is
        equal to aTask.   This method returns
        true if the receiver and aTask are
        considered the same."
    name ~= aTask name ifTrue: [^false].
    "(startTime notNil and: [aTask startTime notNil])
        ifTrue: [
            startTime = aTask startTime
                ifFalse: [^false]].
    (duration notNil and: [aTask duration notNil])
        ifTrue: [
            duration = aTask duration
                ifFalse: [^false]]."
    ^true!

>= aTask
        "Answer true if the receiver
        is greater than or equal to aTask."
    ^self endsBefore: aTask!
```

```
acquireResources: anEnvironment
        "Attempt to acquire all the resources
        from anEnvironment that the receiver
        needs in order to begin execution.  If
        any one resource is not acquired, release
        any successfully acquired resources and
        answer false."
    | acquired |
    acquired := OrderedCollection new.
    resources associationsDo: [:assoc |
        (anEnvironment
            acquireAmount: assoc value
            ofResource: assoc key) ifFalse: [
                acquired do: [:name |
                    (anEnvironment releaseAmount:
                        (resources at: name)
                        ofResource: name)].
                ^false].
        acquired add: assoc key].
    ^true!

asList
        "Answer the receiver as a list of
        instance variable values or unbound
        PROLOG variables (LogicRefs).  The
        list is always a triple of the form
        #(name startTime duration).  When a
        start time or duration is nil, an unbound
        PROLOG variable is placed in the list."
    | start dur |
    (start := startTime) isNil
        ifTrue: [start := LogicRef new].
    (dur := duration) isNil
        ifTrue: [dur := LogicRef new].
    ^(Array
        with: name
        with: start
        with: dur) asList!

beginExecution: anEnvironment
        "Begin the exectution of the receiver
        in anEnvironment.   Answer true if the
        reciever can begin in anEnvironment.
        This method acquires any required
        environmental resources."
    ^(self canOccur: anEnvironment) and:
        [self acquireResources: anEnvironment]!

canBeAdded: anEnvironment
        "Answer the true if the receiver can
        be added to an anEnvironment."
    | env |
    env := anEnvironment.
    ^(SelectorRules new :?
        canBeAdded (self, env),
        exit()) notNil!
```

```
canBeRemoved: anEnvironment
        "Answer the true if the receiver
        can be removed from an anEnvironment."
    | env |
    env := anEnvironment.
    ^(SelectorRules new :?
        canBeRemoved (self, env),
        exit()) notNil!

canOccur: anEnvironment
        "Answer the true if the receiver
        occur in an anEnvironment."
    | env |
    env := anEnvironment.
    ^(SelectorRules new :?
        canOccur (self, env),
        exit()) notNil!

copy
        "Answer a copy of the receiver.  This is
        reimplemented as a deepCopy to ensure that
        the resources are also copied.  Otherwise
        the receiver and its copy would share the
        same instance of a resource dictionary and
        may be destructively modified."
    ^self deepCopy!

duration
        "Answer the duration of the receiver."
    ^duration!

duration: aTime
        "Set the duration of the receiver."
    duration := aTime!

endExecution: anEnvironment
        "End the exectution of the receiver
        in anEnvironment.  This method releases
        any acquired environment resources."
    ^self releaseResources: anEnvironment!

endsBefore: aTask
        "Answer true if the receiver
        ends before aTask ends."
    ^self endTime <= aTask endTime!

endTime
        "Answer the endTime of the receiver.  This
        value is calculated by adding startTime and
        duration."
    ^self startTime + self duration!

fromList: aList
        "Set the receiver from a list of instance
        variables values.  The list is always a
        triple of the form #(name startTime duration).
        The startTime and duration in the list may
```

```
        be PROLOG variables.  If so, they need to
        be evaluated to get their values."
    | values |
    "evaluate any LogicRefs"
    values := aList asArray collect: [:v |
        (v isKindOf: LogicRef) ifTrue: [
            v value] ifFalse: [v]].
    self
        name: (values at: 1);
        startTime: (values at: 2);
        duration: (values at: 3)!

initialize
        "Initialize the instance variables.
        Create an empty resource dictionary."
    name := self class name asLowerCase, 'X'.
    resources := Dictionary new!

interval
        "Answer the interval over which the
        receiver occurs (start time to end
        time)."
    ^startTime to: self endTime!

name
        "Answer the name of the receiver."
    ^name!

name: aString
        "Set the name of the receiver.  The name
        of the receiver must be a symbol. "
    name := aString asSymbol!

nonIntersections: anInterval
        "Answer a collection of intervals that
        represent the non-intersections of the
        interval that the receiver occurs on
        and anInterval."
    | myInterval |
    myInterval := self interval.
    ^Array
        with: ((myInterval first min: anInterval first)
            to: ((myInterval first max: anInterval first)
                min: (myInterval last min: anInterval last)) - 1)
        with: ((((myInterval last min: anInterval last)
            max: (myInterval first max: anInterval first)) + 1)
                to: (myInterval last max: anInterval last))!

overlaps: aTask
        "Answer true if the recievers start
        and end time overlap aTask's start
        and end times."
    | anInterval |
    anInterval := aTask interval.
    ^(startTime
        between: anInterval first
        and: anInterval last - 1) or: [
```

```
            self endTime
                between: anInterval first + 1
                and: anInterval last]!

printOn: aStream
        "Append the ASCII representation
        of the reciever on aStream.  Print
        task name and start and end time."
    | start end |
    startTime isNil
        ifTrue: [start := '?']
        ifFalse: [start := startTime printString].
    (duration isNil or: [startTime isNil])
        ifTrue: [end := '?']
        ifFalse: [end := self endTime printString].
    aStream
        nextPutAll: name,': ';
        nextPutAll: '[',start,'-',end,']'.
    resources associationsDo: [:assoc |
        aStream
            nextPutAll: ', ';
            nextPutAll: assoc key, ' ';
            nextPutAll: assoc value printString]!

releaseResources: anEnvironment
        "Attempt to release all the resources
        that were acquired by the receiver back
        into anEnvironment."
    resources associationsDo: [:assoc |
        (anEnvironment
            releaseAmount: assoc value
            ofResource: assoc key)
            ifFalse: [^false]].
    ^true!

resources
        "Answer the resources used by the receiver."
    ^resources!

rules
        "Answer the PROLOG code that is associated
        with the receiver.  This is a horn clause
        in the class TaskRules that has the same
        name as the receiver."
    ^TaskRules sourceCodeAt: (self name,':') asSymbol!

rules: aString
        "Set the PROLOG code that is associated
        with the receiver.  This is a horn clause
        in the class TaskRules that has the same
        name as the receiver.  Compile and install
        the code."
    | result |
    result := TaskRules compileLogic: aString.
    result isNil ifTrue: [^self].
    Smalltalk
        logPrologSource: aString
```

```
        forSelector: result key
        inClass: TaskRules!

startsBefore: aTask
        "Answer true if the receiver
        starts before aTask starts."
    ^startTime <= aTask startTime!

startTime
        "Answer the startTime of the receiver."
    ^startTime!

startTime: aTime
        "Set the startTime of the receiver."
    startTime := aTime!

stopTime
        "Answer the endTime of the receiver."
    ^self endTime!

unbind
        "Unbind instance variables.  Set the start
        time and duration of the receiver to nil.
        When these are nil, the #asList method for
        the receiver will replace them with unbound
        PROLOG variables."
    startTime := duration := nil!

verify
        "Verify the that the task rules for
        the reciever succeed.  If there is
        no task rule for the receiver, succeed.
        Answer true or false."
    | list |
    (TaskRules canUnderstand: (name,':') asSymbol)
        ifFalse: [^true].  "no rules => values OK"
    list := self asList.
    ^(TaskRules new :? schedule ([list]), exit()) notNil! !
```

```
CommonRules subclass: #TaskRules
  instanceVariableNames:
    'plan executor environment '
  classVariableNames: ''
  poolDictionaries: '' !

!TaskRules class logicMethods ! !


!TaskRules logicMethods !

"This is the default task rule that
all tasks will execute if no rule for
the task is specified."

default (aTask) :-
    member ([0, 20, 40, 60], start),
    task (aTask, start, 15).!

"Execute one task triple.  If there is
no rule for the task, use the default
rule.  Because cut() does not work quite
right in PROLOG/V, the #respondsTo: code
is repeated in both predicates."

execute ([pred | rest]) :-
    is (true, self respondsTo: (pred, ':') asSymbol),
    univ (rule, [pred, [pred | rest]]),
    call (rule).
execute ([pred | rest]) :-
    is (false, self respondsTo: (pred, ':') asSymbol),
    univ (rule, [#default, [pred | rest]]),
    call (rule).!

load10 (load1) :-
    member ([30, 40], start),
    task (load1, start, 10).!

load1 (load1) :-
    member ([30, 40], start),
    task (load1, start, 10).!

load2 (load2) :-
    member ([30, 40], start),
    task (load2, start, 10).!

load3 (load3) :-
    member ([30, 40, 50, 60], start),
    task (load3, start, 20).!

load4a (load4a) :-
    task (load4a, 0, 1).!

load4b (load4b) :-
    task (load4b, 20, 1).!
```

```
load4c (load4c) :-
    task (load4c, 40, 1).!

load4d (load4d) :-
    task (load4d, 60, 1).!

load5a (load5a) :-
    task (load5a, 1, 19).!

load5b (load5b) :-
    task (load5b, 21, 9).!

plan1 ([load2, load3, load4a, load4b, load4c, load4d, load5a, load5b]) :-
    distinct (load1, load2),
    task (load3, end1, _),
    task (load1, _, _, end1).!

plan ([load1, load2, load3, load4a, load4b]) :-
    !!.!

"Execute each of the task triples in the
list as a PROLOG predicate.  Succeed if
every predicate succeeds."

schedule ([]).
schedule ([task | tail]) :-
    execute (task),
    schedule (tail).!

"Verify that the tasks in the list
can execute properly.  First create
a schedule of tasks, place them in
the current plan, then execute the
plan.  Exit with success with the
first valid plan stored in the list."

verify (list) :-
    schedule (list),
    is (_, plan fromList: list),
    is (true, self executePlan),
    plan (list),
    exit ().! !

!TaskRules methods !

environment: anEnvironment
        "Set the environment of verification."
    environment := anEnvironment!

executor: anExecutor
        "Set the plan executor."
    executor := anExecutor!

executePlan
        "Execute the plan in a copy of the environment.
        At the first failure of any kind, answer false
```

```
        and abort the plan execution.  The original event
        queue for the simulation is saved in order to
        preserve any events that are originally scheduled.
        These events could update the screen to show the
        progress of the verication."
    | env exec history originalQueue |
    env := environment copy.
    history := plan history copy.
    (exec := executor) isNil
        ifTrue: [exec := Executor new].
    originalQueue := exec simulation eventQueue.
    originalQueue := originalQueue asArray
        asSortedCollection: originalQueue sortBlock.
    env failBlock: [:resource |
        exec reinitialize.
        exec simulation eventQueue: originalQueue.
        plan history: history.
        ^false].
    exec
        plan: plan;
        environment: env;
        executePlan;
        reinitialize;
        environment: environment.
    exec simulation eventQueue: originalQueue.
    plan
        history: history;
        calculateTimes.
    ^true!

setPlan: aPlan
        "Set the plan to be verified."
    plan := aPlan! !
```

```
"This file will install in all of the source
files for the Case Based Planning System (CBPS)
objects.  The user-interface to CBPS objects
can be found in the file 'cbpsui.st'.

Edit #FileIn be the path were the source files
are contained and then execute the following:

     | dir |
     dir := Smalltalk at: #FileInDir put:
          (Directory pathName: 'a:\').
     (dir file: 'cbps.st')
         fileIn;
         close.
     Smalltalk removeKey: #FileInDir.

For an example that tests to test the code
try:

     CBPS example

This expression will create a CBPS, set a
default unordered plan and library, and
invoke the Selector, Executor and Evaluator.

"!

| bytes stream |
Transcript cr; show: 'Filing in CBPS objects '.
bytes := 0.
#('simulatn.cls'
'commnrls.cls'
'slctrrls.cls'
'evltrrls.cls'
'replannr.cls'
'executor.cls'
'taskruls.cls'
'task.cls'
'notetakr.cls'
'plan.cls'
'envrnmnt.cls'
'selector.cls'
'evaluatr.cls'
'cbps.cls'
'cbps.mth') do: [:name |
    Transcript show: '.'.
    (stream := FileInDir file: name)
        fileIn; close.
    bytes := bytes + stream size].

Transcript cr; show: 'CBPS objects (',
    bytes printString, ' bytes) installed.'.
!
```

```
!Prolog logicMethods !

"Convert a structure to a list (=..)."
univ(structure, list) :-
    nonvar(structure),
    is(true, structure value class == Relation),
    is(x, "self allValue:" structure),
    is(list, List
        head: x value head
        tail: x value tail).
univ(structure, [head | list]) :-
    atom(head),
    is(x, "self allValue: "list),
    is(structure, Relation
        head: head value
        tail: x value).! !
```

CBPS
User Interface Objects
Source Code
Listing

```
Object subclass: #CBPSBrowser
  instanceVariableNames:
    'aCBPS currentTask currentEnvResource taskPicture executing text '
  classVariableNames: ''
  poolDictionaries: '' !

!CBPSBrowser class methods !

example
        "Open up an example CBPSBrowser."
    self new openOn: CBPS exampleCBPS! !


!CBPSBrowser methods !

acceptEnv: aString from: aDispatcher
        "Accept aString as the new contents
        of the environment resource text pane.
        Answer true if the string was an
        acceptable value, else answer false."
    | newValue currentEnv |
    currentEnv := self currentEnv.
    (currentEnv isNil or: [currentEnvResource isNil])
        ifFalse: [
            newValue := Compiler evaluate: aString.
            currentEnv resources
                at: currentEnvResource put: newValue].
    executing := false.
    self
        changed: #envText;
        updatePictures.
    ^true!

addEnvResource
        "Add a new resource to the current environment."
    | name currentEnv |
    (currentEnv := self currentEnv) isNil
        ifTrue: [^self].
    (name := Prompter prompt: 'Name ?'
        default: 'Power') isNil
            ifTrue: [^self].
    currentEnv resources at: name asSymbol put: 5.
    currentEnvResource := name asSymbol.
    self
        updateEnvResources;
        changed: #plot!

allTasksBound
        "Answer true if all tasks in
        the current plan are bound."
    | plan |
    plan := self currentPlan.
    plan tasks do: [:aTask |
        (aTask startTime isNil or: [
            aTask duration isNil]) ifTrue: [
```

```
                ^false]].
    ^true!

cbpsMenu
        "Answer the main menu that allows the
        user to maipulate the CBPS object."
    ^Menu
        labelArray: #('View/Edit Plan' 'Plan Selector' 'Plan Executor' 'Plan Evaluator')
        lines: #()
        selectors: #(editPlan selectPlan executePlan evaluatePlan)!

currentEnv
        "Answer the current environment of the CBPS."
    ^aCBPS environment!

currentLib
        "Answer the current plan library of the CBPS."
    ^aCBPS library!

currentPlan
        "Answer the current plan in the CBPS."
    ^aCBPS plan!

displayPlanOn: aForm
        "Answer a Form that has each of the tasks
        in the current plan displayed as a bar and
        has a nice title."
    | plan planForm h w scanner blt
    title label labels p box curFont |
    plan := self currentPlan.
    scanner := CharacterScanner new
        initialize: aForm boundingBox
        font: (curFont := Font eightLine)
        dest: aForm.
    (plan isNil or: [plan tasks isEmpty])
        ifTrue: [title := '* No Tasks *']
        ifFalse: [title := 'Tasks for "', plan name,'"'].
    scanner
        display: title
        at: (p := (aForm width - (curFont
            stringWidth: title)) // 2 @ 4).
        (Pen new: aForm)
            place: (p + (0 @ (curFont basePoint y + 1)));
            goto: (p + ((curFont stringWidth: title)
                @ (curFont basePoint y + 1))).
    planForm := Form
        width: (w := aForm width * 7 // 10)
        height: (h := aForm height * 7 // 10).
    plan isNil ifFalse: [
        self displayTasksOn: planForm].
    planForm
        border: planForm boundingBox
        rule: Form over
        mask: Form black.
    (blt := BitBlt destForm: aForm sourceForm: planForm)
        destX: (aForm width - w // 2);
        destY: (aForm height -h // 2);
```

```
        copyBits.
    executing ifFalse: [^self].
    blt destX: 20;
        destY: (aForm height - 2 - curFont height).
    labels := #('Executed Task' 'Executing Task' 'Unexecuted Task').
    1 to: labels size do: [:i |
        box := Form width: curFont height
            height: curFont height.
        box perform: (#(black gray white) at: i).
        box
            border: box boundingBox
            rule: Form over
            mask: Form black.
        label := labels at: i.
        blt
            sourceForm: box;
            copyBits;
            destX: (blt destX + 20).
        scanner display: label
            at: (blt destX @ blt destY).
        blt destX: (blt destX + 30 +
            (curFont stringWidth: label))].
    ^aForm!

displayTasksOn: aForm
        "Display each of the tasks on aForm.  If
        we are executing the plan, shade the task
        bars to indicate executed, executing, and
        unexecuted tasks."
    | currentPlan scale height blt bar tasks start duration |
    (currentPlan := self currentPlan) isNil
        ifTrue: [^self].
    tasks := currentPlan tasks.
    tasks isEmpty ifTrue: [^self].
    blt := (BitBlt destForm: aForm sourceForm: nil).
    currentPlan endTime = currentPlan startTime
        ifTrue: [^self].
    scale := (aForm width / (currentPlan endTime
        - currentPlan startTime)).
    height := ((aForm height // tasks size) - 4) min: 10.
    tasks do: [:aTask |
        (duration := aTask duration) isNil
            ifTrue: [duration := 1].
        bar := Form
            width: (duration * scale) truncated
            height: height.
        executing ifTrue: [
            (aCBPS executor tasksExecuted includes: aTask)
                ifTrue: [bar black].
            (aCBPS executor tasksExecuting includes: aTask)
                ifTrue: [bar gray]].
        bar
            border: bar boundingBox
            rule: Form over
            mask: Form black.
        (start := aTask startTime) isNil
            ifTrue: [start := 0].
```

```
        blt
            sourceForm: bar;
            destY: (blt destY + 2);
            destX: (start - currentPlan
                startTime * scale) truncated;
            copyBits;
            destY: (blt destY + bar height + 2)]!

editPlan
        "Allow the user to input the specification
        for a plan.  This is an unordered plan that
        is placed in the current environment.  Open
        a PlanBrowser to do the actual editing."
    self changed: #plans.
    PlanBrowser new openOn:
        (self currentEnv unorderedPlan)!

envResourceMenu
        "Answer the Menu for the pane that
        lists the resources of the current
        environment."
    ^Menu
        labelArray: #('Add Resource' 'Remove Resource')
        lines: #()
        selectors: #(addEnvResource removeEnvResource)!

envResources
        "Answer a collection of the names of
        the resources in the current environment."
    | env |
    env := self currentEnv.
    ^env resources keys asArray!

envText
        "Answer the contents of the
        environment resources text pane."
    | currentEnv |
    currentEnv := self currentEnv.
    (currentEnv isNil or: [currentEnvResource isNil])
        ifTrue: [^''].
    ^(currentEnv resources
        at: currentEnvResource
        ifAbsent: [^'']) printString!

evaluatePlan
        "Evaluate the current plan.  Invoke the
        evaluator associated with the CBPS and
        show partial results in the status pane."
    | eval lib aStream |
    (self allTasksBound) ifFalse: [
        ^self text: 'You must first invoke Plan Selector.'].
    lib := self currentLib.
    lib remove: aCBPS plan ifAbsent: [
        ^self text: 'You must first invoke Plan Selector.'].
    eval := aCBPS evaluator.
    self
        changed: #plans;
```

```
        updateStatus: 'Evaluating'.
    CursorManager execute change.
    eval hasViolations
        ifTrue: [
            aStream := WriteStream on: ''.
            aStream
                nextPutAll: 'Plan Violation : ', eval planViolation printString; cr.
            eval taskViolations do: [:aTask |
                aStream nextPutAll: 'Task Violation : ', aTask printString; cr].
            self text: aStream contents.
            Menu message: 'Continue ...']
        ifFalse: [eval evaluatePlan].
    self changed: #plans.
    (lib includes: aCBPS plan)
        ifFalse: [
            self updateStatus: 'Forgot/Removed'.
            aCBPS plan: Plan new.
            aCBPS environment requiredTasks: #()]
        ifTrue: [
            self updateStatus: 'Added/Updated'].
    executing := false.
    self updatePictures.
    CursorManager normal change!

executePlan
        "Execute the current plan using the
        executor associated with the current
        CBPS.  Show partial results in the
        status and picture panes."
    (self allTasksBound) ifFalse: [
        ^self text: 'You must first invoke Plan Selector.'].
    executing := true.
    self updateStatus: 'Executing'.
    CursorManager execute change.
    self updatePictures.
    CursorManager normal change.
    self updateStatus: 'Executed'!

initWindowSize
        "Answer the initial window size of
        the receiver.  This method is used
        by v286."
    ^(Display width * 7 // 8) @
        (Display height * 6 // 7)!

openOn: aCaseBasedPlanner
        "Open the receiver on aCaseBasedPlanner.
        Allow the user to perform the activities
        of Case Based Planning using the reciever
        as a user interface."
    | aTopPane |
    text := ''.
    executing := false.
    aCBPS := aCaseBasedPlanner.
    aTopPane := TopPane new.
    aTopPane
        label: 'Case Based Planning System';
```

```
        minimumSize: (self initWindowSize);
        model: self.
    aTopPane
        addSubpane:
            (ListPane new
                title: 'Plans';
                model: self;
                name: #plans;
                change: #selectPlan:;
                menu: #cbpsMenu;
                framingRatio: (0@0 corner: (1/10)@1)).
    aTopPane
        addSubpane:
            (GraphPane new
                model: self;
                name: #taskPicture:;
                change: #selectTaskPicture:;
                menu: #cbpsMenu;
                framingRatio: ((1/10)@0 corner: 1@(2/5))).
    aTopPane
        addSubpane:
            (PlotPane new
                model: self;
                name: #plot;
                change: #selectPlot;
                menu: #cbpsMenu;
                font: Font eightLine;
                title:  'Resource Usage';
                xTitle: 'Time';
                yTitle: 'Resource';
                xRange: (0 to: 100 by: 10);
                yRange: (0 to: 100 by: 10);
                framingRatio: ((1/10)@(2/5) corner: 1@(4/5))).
    aTopPane
        addSubpane:
            (ListPane new
                title: 'Env Vars';
                model: self;
                name: #envResources;
                change: #selectEnvResource:;
                menu: #envResourceMenu;
                framingRatio: ((10/100)@(4/5) corner: (25/100)@1)).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Env Values';
                model: self;
                name: #envText;
                change: #acceptEnv:from:;
                framingRatio: ((25/100)@(4/5) corner: (40/100)@1)).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Status';
                model: self;
                name: #text;
                framingRatio: ((40/100)@(4/5) corner: 1@1)).
```

```
    aTopPane dispatcher open scheduleWindow!

plans
        "Answer the a collection of the names
        of the plans in the current plan library."
    | library |
    library := self currentLib.
    ^(library collect: [:plan | plan name])
        asSortedCollection!

plot
        "Answer a collection of collections that
        are points for each of the lines in the
        plot pane.  These lines are the resource
        usage and maximum resources available over
        time for each of the resources."
    | env rawPoints points result plan history pens i
      available start end time max amount somePoints mask |
    ((plan := self currentPlan) isNil or: [
        self allTasksBound not]) ifTrue: [^#()].
    aCBPS executor reinitialize.
    available := (env := self currentEnv) resources deepCopy.
    pens := Dictionary new.
    rawPoints := Dictionary new.
    i := 0.
    available keysDo: [:resource |
        (Smalltalk includesKey: #BiColorForm)
            ifTrue: [
                mask := Display compatibleMask color:
                    (#(1 11 2 8 13 0) at: (i + 1 \\ 6 + 1))]
            ifFalse: [
                mask := (Form perform:
                    (#(black darkGray gray lightGray)
                        at: (i \\ 4 + 1)))].
        i := i + 1.
        pens at: resource put: (Pen new mask: mask).
        rawPoints at: resource put: Dictionary new].
    executing
        ifTrue: [aCBPS executor replanOnFailure]
        ifFalse: [
            history := plan history copy.
            aCBPS executor doNothingOnFailure].
    aCBPS executor simulation
        whenBlock: [true]
        doBlock: [
            executing ifTrue: [self updateTaskPicture].
            available keysDo: [:resource |
                (rawPoints at: resource)
                    at: (time := aCBPS executor time)
                    put: (time @ (env resources at: resource))]].
    aCBPS executePlan.
    executing ifFalse: [aCBPS plan history: history].
    aCBPS executor reinitialize.
    end := max := 0.
    result := OrderedCollection new.
    available keysDo: [:resource |
        amount := available at: resource.
```

```
    somePoints := (rawPoints at: resource)
        asSortedCollection: [:pt1 :pt2 | pt1 x <= pt2 x].
    somePoints := somePoints collect: [:pt |
        pt x @ ((amount - pt y) max: 0)].
    somePoints isEmpty ifFalse: [
        points := OrderedCollection new.
        points add: somePoints first.
        2 to: somePoints size do: [:i |
            points
                add: (somePoints at: i) x
                    @ (somePoints at: i - 1) y;
                add: (somePoints at: i)].
        max := ((points inject: points first y into:
            [:maxSoFar :pt | pt y max: maxSoFar])
                max: amount) max: max.
        result add: points.
        start isNil
            ifTrue: [start := points first x]
            ifFalse: [start := points first x min: start].
        end := points last x max: end.
        points addFirst: (pens at: resource)]].
(start isNil or: [end - start = 0]) ifFalse: [
    available keysDo: [:resource |
        amount := available at: resource.
        result add:
            (Array
                with: (pens at: resource)
                with: start @ amount
                with: end @ amount)].
    self
        changed: #plot
        with: #xRange:
        with: (start to: end by:
            (end - start / 10)).
    self
        changed: #plot
        with: #yRange:
        with: (0 to: max + 10 by: 10)].
^result!

plotOLD
    "Answer a collection of collections that
    are points for each of the lines in the
    plot pane. This method knows too much about
    plotting of the special resource named #Power.
    It should be modified to be able to plot any
    resource."
  | env rawPoints points result plan history power start end time max |
  (plan := self currentPlan) isNil ifTrue: [^#()].
  aCBPS executor reinitialize.
  power := (env := self currentEnv)
      resources at: #Power ifAbsent: [^#()].
  rawPoints := Dictionary new.
  executing
      ifTrue: [aCBPS executor replanOnFailure]
      ifFalse: [
          history := plan history copy.
```

```
            aCBPS executor doNothingOnFailure].
    aCBPS executor simulation
        whenBlock: [true]
        doBlock: [
            executing ifTrue: [
                self updateTaskPicture].
            rawPoints at: (time := aCBPS executor time)
                put: (time @ (env resources at: #Power))].
    aCBPS executePlan.
    executing ifFalse: [
        aCBPS plan history: history].
    aCBPS executor reinitialize.
    rawPoints := rawPoints asSortedCollection:
            [:pt1 :pt2 | pt1 x <= pt2 x].
    rawPoints := rawPoints collect: [:pt |
        pt x @ ((power - pt y) max: 0)].
    rawPoints isEmpty ifTrue: [^#()].
    points := OrderedCollection new.
    points add: rawPoints first.
    2 to: rawPoints size do: [:i |
        points
            add: (rawPoints at: i) x
                @ (rawPoints at: i - 1) y;
            add: (rawPoints at: i)].
    max := points inject: points first y into:
        [:maxSoFar :max | max y max: maxSoFar].
    result := OrderedCollection with: points.
    result add:
        (Array
            with: (start := points first x) @ power
            with: (end := points last x) @ power).
    end - start = 0 ifTrue: [^#()].
    self
        changed: #plot
        with: #xRange:
        with: (start to: end by: (end - start / 10)).
    self
        changed: #plot
        with: #yRange:
        with: (0 to: (max max: power) + 10 by: 10).
    ^result!

removeEnvResource
        "Remove the current resource from the
        current environment.  Update any panes
        affected."
    | currentEnv |
    currentEnv := self currentEnv.
    (currentEnv isNil or: [
        currentEnvResource isNil])
            ifTrue: [^self].
    currentEnv resources removeKey: currentEnvResource.
    currentEnvResource := nil.
    self
        updateEnvResources;
        changed: #plot!
```

```
selectEnvResource: aString
        "The resource pane has been selected so
        show the value of the resource on the
        environment resource text pane."
    currentEnvResource := aString.
    self changed: #envText!

selectPlan
        "There is an unordered plan in the
        environment that is a specification
        of the operators requirements.

        Perform the CBPS plan selection module
        functions using the unordered plan and
        the selector for the CBPS."

    | newPlan env lib selector extra missing aStream |

    env := self currentEnv.
    selector := aCBPS selector.

    CursorManager execute change.
    newPlan := selector locatePlan.
    CursorManager normal change.
    extra := (newPlan extraTasks: env requiredTasks) size.
    missing := (newPlan missingTasks: env requiredTasks) size.
    aStream := WriteStream on: ''.
    aStream
        nextPutAll: 'Located ', newPlan printString; cr;
        nextPutAll: '    ', extra printString, ' extra task(s), ';
        nextPutAll: missing printString, ' missing task(s)'; cr;
        nextPutAll: '    ', newPlan history printString; cr.
    self text: aStream contents.

    Menu message: 'Continue ...'.
    newPlan := selector constructPlan.
    aStream := WriteStream on: ''.
    aStream
        nextPutAll: 'Constructed ', newPlan printString; cr;
        nextPutAll: '    removing ', extra printString, ' extra task(s), '; cr;
        nextPutAll: '    adding ', missing printString, ' missing task(s)'.
    self text: aStream contents.
    ((lib := self currentLib) includes: newPlan)
        ifFalse: [lib add: newPlan].
    self changed: #plans
        with: #restoreSelected:
        with: newPlan name.

    Menu message: 'Continue ...'.
    aCBPS plan: newPlan.
    self updateStatus: 'Verifying'.
    CursorManager execute change.
    aCBPS executor reinitialize.
    aCBPS executor simulation
        atTime: (newPlan startTime)
        doAction: (Array with: self
            with: #updateTaskPicture with: #()).
```

```
    selector verifyPlan isNil
        ifTrue: [
            self text: 'Verify failed : ', newPlan printString.
            Menu message: 'Continue ...'].
    CursorManager normal change.

    executing := false.
    aCBPS executor reinitialize.
    self updateStatus: 'Selecting'.
    self
        updateEnvResources;
        updatePictures!

selectPlan: aString
        "The plan library pane has been selected.
        Search the plan library for the plan that
        has the same name as aString name.  Update
        the other panes accordingly."
    | plan library env |
    CursorManager execute change.
    library := self currentLib.
    plan := library
        detect: [:plan |
            plan name = aString]
        ifNone: [^self].
    env := self currentEnv.
    env requiredTasks: plan tasks.
    env unorderedPlan rules: plan rules.
    aCBPS plan: plan copy.
    executing := false.
    self
        updateStatus: 'Selecting';
        updatePictures.
    CursorManager normal change!

selectPlot
        "The plot pane has been selected.
        Do nothing."!

selectTaskPicture: aPoint
        "The task picuture pane has been selected.
        Do nothing."!

taskPicture: aRect
        "Answer a Form for the task picture pane
        and display the form as the contents of
        the pane."
    | aForm |
    aForm := Form
        width: aRect width
        height: aRect height.
    self displayPlanOn: aForm.
    aForm displayAt: aRect origin.
    taskPicture := aForm.
    ^aForm!

text
```

```
        "Answer the contents of the
        environment resouce text pane."
    ^text!

text: aString
        "Set the contents of the environment
        resource text pane and update the pane
        to show aString."
    text := aString.
    self changed: #text!

updateEnvResources
        "Refresh the environment resource panes."
    | currentEnv |
    currentEnv := self currentEnv.
    ((currentEnvResource notNil and: [currentEnv notNil]) and: [
        currentEnv resources keys includes: currentEnvResource])
            ifTrue: [
                self changed: #envResources
                    with: #restoreSelected:
                    with: currentEnvResource]
            ifFalse: [self changed: #envResources].
    self changed: #envText!

updatePictures
        "Refresh the picture panes."
    self
        updateTaskPicture;
        changed: #plot!

updateStatus: action
        "Refresh the status pane."
    | aStream plan |
    plan := self currentPlan.
    aStream := WriteStream on: ''.
    aStream
        nextPutAll: action, ' ', plan printString; cr;
        nextPutAll: '     ', plan history printString.
    self text: aStream contents!

updateTaskPicture
        "Refresh the task picture pane."
    self displayPlanOn: taskPicture white.
    self changed: #taskPicture:! !
```

```
SubPane subclass: #DialogBox
  instanceVariableNames:
    'contents '
  classVariableNames: ''
  poolDictionaries:
    'FunctionKeys ' !

!DialogBox class methods ! !


!DialogBox methods !

defaultDispatcherClass
        "Answer GraphDispatcher which is the
         default dispatcher of a DialogBox."
    ^GraphDispatcher!

displayItem: anItem at: aPoint
        "Display the item at aPoint."
    (anItem isKindOf: String)
        ifTrue: [
            paneScanner display: anItem at:
                (aPoint - frame origin)]
        ifFalse: [anItem displayAt: aPoint]!

displayItems
        "Show the items in the pane."
    | offset max w h |
    Pane windowClip: paneScanner frame.
    offset := frame origin + 2.
    contents do: [:line |
        max := line inject: 0 into: [:maxSoFar :item |
            maxSoFar max: (self itemHeight: item)].
        line do: [:item |
            w := self itemWidth: item.
            h := self itemHeight: item.
            self
                displayItem: item
                at: (offset x @ (offset y
                    + (max - h // 2))).
            offset x: (offset x + w)].
        offset x: frame left + 2;
            y: offset y + max + 2].
    Pane initWindowClip!

initialize
        "Initialize the pane instance variables."
    super initialize.
    topCorner := 1@1.
    curFont := ListFont!

itemHeight: anItem
        "Answer the height of the item."
    (anItem isKindOf: String)
        ifTrue: [^curFont height].
```

```
    ^anItem extent y!

itemWidth: anItem
        "Answer the width of the item."
    (anItem isKindOf: String)
        ifTrue: [^curFont stringWidth: anItem].
    ^anItem extent x!

open
        "Open the pane."
    contents := (name notNil and: [model notNil])
        ifTrue: [model perform: name]
        ifFalse: [Array new].
    contents := contents collect: [:line |
        line collect: [:item |
            (item class == String) ifTrue: [item]
                ifFalse: [model perform: item]]]!

scrollHand: oldPoint to: newPoint
        "Do nothing.  This method
        is used by v286."!

scrollLeft: anInteger
        "Do nothing.  This method
        is used by v286."!

scrollTopCorner: anInteger
        "Do nothing.  This method
        is used by v286."!

scrollUp: anInteger
        "Do nothing.  This method
        is used by v286."!

selectAtCursor
        "Select at the current location in the pane.
        Search for an item that contains the current
        location of the cursor and inform it that it
        has been selected."
    contents do: [:line |
        line do: [:item |
            (item respondsTo: #containsPoint:)
                ifTrue: [
                    (item containsPoint: Cursor offset)
                        ifTrue: [
                            Pane windowClip: frame.
                            item selectAtCursor.
                            ^Pane initWindowClip]]]]!

showWindow
        "Display the receiver pane
         and the selection."
    Display white: paneScanner clipRect.
    self
        displayItems;
        border: frame;
        border!
```

```
topCorner
        "Answer the topCorner."
    ^topCorner!

totalLength
        "Answer the height of the pane.
        This method is used by v286."
    ^frame height // curFont height!

update
        "Update the contents of the
         receiver pane."
    self
        open;
        showWindow! !
```

```
PromptEditor subclass: #FieldEditor
  instanceVariableNames:
    'strokeBlock '
  classVariableNames: ''
  poolDictionaries:
    'FunctionKeys CharacterConstants ' !

!FieldEditor class methods ! !


!FieldEditor methods !

processFunctionKey: aCharacter
        "Private - Process function keys
         from the keyboard or mouse."
    aCharacter == CycleFunction
        ifTrue: [^self accept].
    aCharacter == WindowMenuRequest
        ifTrue: [^self accept].
    (aCharacter == SelectFunction
    or: [aCharacter == PaneMenuRequest])
        ifTrue: [
            pane hasCursor ifFalse: [
                ^self accept]].
    super processFunctionKey: aCharacter!

processInputKey: aCharacter
        "Private - Check to see whether the
         character is permissable."
    (strokeBlock isNil or:
        [(strokeBlock value: aCharacter)])
            ifTrue: [super processInputKey: aCharacter]
            ifFalse: [Terminal bell]!

strokeBlock: aValue
        "Set the value of strokeBlock"
    strokeBlock:= aValue! !
```

```
Object subclass: #Field
  instanceVariableNames:
    'model changeSelector offset width default font resultClass acceptBlock strokeBlock '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants ' !

!Field class methods !

new
        "Create a new instance of the
         receiver and initialize it."
    ^super new initialize! !


!Field methods !

acceptBlock: aValue
        "Set the value of acceptBlock"
    acceptBlock:= aValue!

boundingBox
        "Answer the frame that contains
         the field and its contents."
    ^offset extent: ((width + 1 *
        font width) @ (font height + 4))!

change
        "Answer the value of changeSelector"
    ^changeSelector!

change: aValue
        "Set the value of changeSelector"
    changeSelector := aValue!

containsPoint: aPoint
        "Answer true if the receiver
        contains aPoint."
    ^self boundingBox containsPoint: aPoint!

default
        "Answer the value of default"
    ^default!

default: aValue
        "Set the value of default"
    default := aValue!

display
        "Show the field. Draw two
         lines around the aRectangle."
    self displayClipRect: Pane windowClip!

displayAt: aPoint
        "Show the field. Draw two
```

```
        lines around the aRectangle."
    offset := aPoint.
    self display!

displayClipRect: clipRect
        "Show the field. Draw two
         lines around the aRectangle."
    | aRectangle scanner|
    aRectangle := self boundingBox.
    scanner := CharacterScanner new
        initialize: aRectangle
        font: font.
    scanner
        clipRect: (clipRect intersect: aRectangle);
        display: default
        from: 1
        at: 2@2.
    Display border: aRectangle
        clippingBox: clipRect
        rule: Form over
        mask: Form black.
    Display
        border: (aRectangle insetBy: 1@1)
        clippingBox: clipRect
        rule: Form over
        mask: Form white!

edit
        "Allow the user to edit the
         contents of the field."
    | replyPane topPane aString|
    topPane := TopPane new.
    topPane addSubpane:
        (replyPane := TextPane new
            model: self;
            name: #default;
            dispatcher:
                (FieldEditor new
                    strokeBlock: strokeBlock);
            font: font).
    replyPane
        reframe: (self boundingBox
            intersect: Pane windowClip);
        open;
        showWindow;
        selectAtCursor.
    replyPane dispatcher processInput.
    aString := replyPane contents trimBlanks.
    (acceptBlock value: aString)
        ifTrue: [default := aString]
        ifFalse: [replyPane cancel].
    replyPane
        refreshAll;
        selectAfter: 0@1;
        forceSelectionOntoDisplay;
        hideSelection;
        close.
```

```
    Dependents removeKey: self ifAbsent: [].
    self update.
    ^default!

extent
        "Answer the extent of
         the field and its contents."
    ^((width + 1 * font width) @ (font height + 4))!

initialize
        "Initialize the instance variables
         of the receiver"
    default := ''.
    width := 0.
    offset := 0 @ 0.
    font := "Font eightLine" SysFont.
    acceptBlock := [:aValue | true].
    strokeBlock := [:aChar | true]!

model
        "Answer the value of model"
    ^model!

model: aValue
        "Set the value of model"
    model:= aValue!

offset
        "Answer the value of offset"
    ^offset!

offset: aValue
        "Set the value of offset"
    offset:= aValue!

resultClass: aClass
        "Set the value of resultClass"
    resultClass:= aClass!

select
        "Edit the contents of the field."
    ^self edit!

selectAtCursor
        "Edit the contents of the field."
    ^self edit!

strokeBlock: aValue
        "Set the value of strokeBlock"
    strokeBlock:= aValue!

update
        "The default value has changed
         update the model if there is one."
    (model notNil and: [changeSelector notNil])
        ifTrue: [
            model
```

```
                perform: changeSelector
                with: default]!

width
        "Answer the value of width"
    ^width!

width: aValue
        "Set the value of width"
    width:= aValue! !
```

```
PlanBrowser subclass: #LibraryBrowser
  instanceVariableNames:
    'library '
  classVariableNames: ''
  poolDictionaries: '' !

!LibraryBrowser class methods ! !


!LibraryBrowser methods !

addPlan
        "Add a new plan to the current library.  Make
        sure the name is unique with respect to the
        other plans and refresh affected panes."
    | name newPlan |
    (name := Prompter prompt: 'Name ?'
        default: 'PlanX') isNil
            ifTrue: [^self].
    library do: [:aPlan |
        aPlan name = name ifTrue: [
            Menu message: 'The name "', name, '" is already taken, choose another name.'.
            ^self]].
    newPlan := Plan new name: name.
    library add: newPlan.
    currentPlan := newPlan.
    currentTask := currentResource := nil.
    self
        changed: #plans
        with: #restoreSelected:
        with: name;
        changed: #tasks;
        changed: #taskDialog;
        changed: #resources;
        changed: #resourceText;
        changed: #taskText;
        changed: #taskRule;
        changed: #planRule!

openOn: aLibrary
        "Open a browser on a plan library.  Provide
        a user interface to an OrderedCollection of
        Plan objects."
    | aTopPane |
    library := aLibrary.
    aTopPane := TopPane new.
    aTopPane
        label: 'Library Browser';
        minimumSize:  (self initWindowSize);
        model: self.
    aTopPane
        addSubpane:
            (ListPane new
                title: 'Plans';
                model: self;
```

```
                name: #plans;
                change: #selectPlan:;
                menu: #planMenu;
                framingRatio: (0@0 corner: (1/9)@1)).
    aTopPane
        addSubpane:
            (ListPane new
                title: 'Tasks';
                model: self;
                name: #tasks;
                change: #selectTask:;
                menu: #taskMenu;
                framingRatio: ((1/9)@0 corner: (2/9)@1)).
    aTopPane
        addSubpane:
            (DialogBox new
                title: 'Task Values';
                model: self;
                name: #taskDialog;
                framingRatio: ((2/9)@0 corner: (1/2)@(1/2))).
    aTopPane
        addSubpane:
            (ListPane new
                title: 'Resource Name';
                model: self;
                name: #resources;
                change: #selectResource:;
                menu: #resourceMenu;
                framingRatio: ((2/9)@(1/2) corner: (1/2)@(3/4))).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Resource Value';
                model: self;
                name: #resourceText;
                change: #acceptResource:from:;
                framingRatio: ((2/9)@(3/4) corner: (1/2)@1)).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Task Rules';
                model: self;
                name: #taskRule;
                change: #acceptTaskRule:from:;
                framingRatio: ((1/2)@0 corner: 1@(1/2))).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Plan Rules';
                model: self;
                name: #planRule;
                change: #acceptPlanRule:from:;
                framingRatio: ((1/2)@(1/2) corner: 1@1)).
    aTopPane dispatcher open scheduleWindow!

planMenu
        "Answer the Menu for the pane that
```

```
        lists the plans in the plan library."
    ^Menu
        labelArray: #('Add Plan' 'Remove Plan')
        lines: #()
        selectors: #(addPlan removePlan)!

plans
        "Answer the a collection of the names
        of the plans in the plan library."
    ^library collect: [:plan | plan name]!

removePlan
        "Remove the selected plan from the
        plan library.  Refresh affected panes."
    library remove: currentPlan.
    currentTask := currentPlan := nil.
    self
        changed: #plans;
        changed: #tasks;
        changed: #taskDialog;
        changed: #resources;
        changed: #resourceText;
        changed: #taskText;
        changed: #taskRule;
        changed: #planRule!

selectPlan: aString
        "The plan library pane has been selected.
        Search the plan library for the plan that
        has the same name as aString.  Refresh any
        other panes affected and set the selected
        plan."
    | plan |
    plan := library
        detect: [:plan |
            plan name = aString]
        ifNone: [^self].
    currentPlan := plan.
    currentTask := currentResource := nil.
    self
        changed: #tasks;
        changed: #taskDialog;
        changed: #resources;
        changed: #resourceText;
        changed: #taskText;
        changed: #taskRule;
        changed: #planRule! !
```

```
TaskBrowser subclass: #PlanBrowser
  instanceVariableNames:
    'currentPlan '
  classVariableNames: ''
  poolDictionaries: '' !

!PlanBrowser class methods ! !


!PlanBrowser methods !

acceptPlanRule: aString from: aDispatcher
        "Accept aString as the new contents
        of the plan rule text pane.  Answer true
        if the string was acceptable.  Compile
        and install the PROLOG code."
    | result code |
    currentPlan isNil ifTrue: [^false].
    aString isEmpty
        ifFalse: [
            code := currentPlan replaceHead: aString.
            result := TaskRules
                compileLogic: code
                notifying: aDispatcher.
            result isNil
                ifTrue: [^false].
            Smalltalk
                logPrologSource: code
                forSelector: result key
                inClass: TaskRules.
            currentPlan name: (result key copyFrom: 1
                to: result key size - 1)].
    self changed: #planRule.
    ^true!

acceptTaskRule: aString from: aDispatcher
        "If the super class says the string was
        acceptable, refesh some panes and answer
        true."
    (super acceptTaskRule: aString from: aDispatcher)
        ifFalse: [^false].
   self
        changed: #tasks
        with: #restoreSelected:
        with: currentTask name.
    ^true!

addTask
        "Add a new task to the current plan.  Make
        sure the name is unique with respect to the
        other tasks and refresh affected panes."
    | name newTask tasks |
    currentPlan isNil ifTrue: [^self].
    (name := Prompter prompt: 'Name ?'
        default: 'taskX') isNil
```

```
            ifTrue: [^self].
    tasks := currentPlan tasks.
    tasks do: [:aTask |
        aTask name = name asSymbol ifTrue: [
            Menu message: 'The name "', name, '" is already taken, choose another name.'.
            ^self]].
    newTask := Task new name: name.
    currentPlan addTask: newTask.
    currentTask := newTask.
    self
        changed: #tasks
        with: #restoreSelected:
        with: currentTask name.
    self
        changed: #taskDialog;
        changed: #resources;
        changed: #resourceText;
        changed: #taskRule.
    CursorManager execute change.
    self
        acceptPlanRule: self planRule
        from: nil.
    CursorManager normal change!

defaultPlanRule
        "Answer the default rule for the
        plan.  This is a PROLOG horn clause."
    | aStream |
    currentPlan isNil ifTrue: [^''].
    aStream := WriteStream on: ''.
    aStream
        nextPutAll: currentPlan ruleHead; cr;
        nextPutAll: '    !!.'.
    ^aStream contents!

nameFromString: aString
        "Set the duration of the receiver
        from a String representation.  Call
        the super and refresh panes."
    super nameFromString: aString.
    self
        changed: #tasks
        with: #restoreSelected:
        with: currentTask name!

openOn: aPlan
        "Open a browser on aPlan.  Provide a
        user interface to a Plan object."
    | aTopPane |
    currentPlan := aPlan.
    aTopPane := TopPane new.
    aTopPane
        label: 'Plan Browser';
        minimumSize: (self initWindowSize);
        model: self.
    aTopPane
        addSubpane:
```

```
            (ListPane new
                title: 'Tasks';
                model: self;
                name: #tasks;
                change: #selectTask:;
                menu: #taskMenu;
                framingRatio: (0@0 corner: (1/8)@1)).
    aTopPane
        addSubpane:
            (DialogBox new
                title: 'Task Values';
                model: self;
                name: #taskDialog;
                framingRatio: ((1/8)@0 corner: (1/2)@(1/2))).
    aTopPane
        addSubpane:
            (ListPane new
                title: 'Resource Name';
                model: self;
                name: #resources;
                change: #selectResource:;
                menu: #resourceMenu;
                framingRatio: ((1/8)@(1/2) corner: (1/2)@(3/4))).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Resource Value';
                model: self;
                name: #resourceText;
                change: #acceptResource:from:;
                framingRatio: ((1/8)@(3/4) corner: (1/2)@1)).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Task Rules';
                model: self;
                name: #taskRule;
                change: #acceptTaskRule:from:;
                framingRatio: ((1/2)@0 corner: 1@(1/2))).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Plan Rules';
                model: self;
                name: #planRule;
                change: #acceptPlanRule:from:;
                framingRatio: ((1/2)@(1/2) corner: 1@1)).
    aTopPane dispatcher open scheduleWindow!

planRule
        "Answer the contents of the plan rule pane."
    | text |
    (currentPlan isNil or: [
        (text := currentPlan rules) asString
        = (currentPlan name,':')])
            ifTrue: [^self defaultPlanRule].
    ^text!
```

```
removeTask
        "Remove the current selected task from
        the current plan.  Refresh panes."
    (currentPlan isNil or:
        [currentTask isNil])
            ifTrue: [^self].
    currentPlan removeTask: currentTask.
    currentTask := currentResource := nil.
    self
        changed: #tasks;
        changed: #taskDialog;
        changed: #resources;
        changed: #resourceText;
        changed: #taskRule.
    CursorManager execute change.
    self
        acceptPlanRule: self planRule
        from: nil.
    CursorManager normal change!

selectTask: aString
        "The task pane has been selected.
        Display the task information in the text
        pane.  Refresh any other panes and set
        the selected task."
    | tasks |
    currentPlan isNil ifTrue: [^self].
    tasks := currentPlan tasks.
    currentTask := tasks
        detect: [:task |
            task name = aString]
        ifNone: [^self].
    self changed: #taskDialog.
    (currentResource isNil or: [
        (currentTask resources includesKey:
            currentResource) not])
        ifTrue: [self changed: #resources]
        ifFalse: [
            self changed: #resources
                with: #restoreSelected:
                with: currentResource].
    self
        changed: #resourceText;
        changed: #taskText;
        changed: #taskRule!

taskMenu
        "Answer the Menu for the pane that
        lists the task in the current plan."
    ^Menu
        labelArray: #('Add Task' 'Remove Task')
        lines: #()
        selectors: #(addTask removeTask)!

tasks
        "Answer a collection of the task
```

```
    names that belong to the selected
    plan."
currentPlan isNil ifTrue: [^#()].
^(currentPlan tasks
    collect: [:aTask | aTask name])
        asSortedCollection! !
```

```
SubPane subclass: #PlotPane
  instanceVariableNames:
    'title xTitle yTitle xRange yRange plotPen lines graphOrigin '
  classVariableNames: ''
  poolDictionaries: '' !

!PlotPane class methods !

plotTest
        "PlotPane plotTest"
    | aTopPane |
    aTopPane := TopPane new.
    aTopPane
        label: 'Testing Plot Pane';
        model: aTopPane dispatcher;
        minimumSize: 24@48;
        addSubpane:
            (PlotPane new
                model: self;
                title: 'Plot Test';
                xTitle: 'x';
                yTitle: 'y';
                name: #points).
    aTopPane dispatcher open scheduleWindow!

points

    | items |
    items := OrderedCollection new.
    items
        add: 0@0;
        add: 3@3;
        add: 5@5;
        add: 10@5.
    ^items! !


!PlotPane methods !

computeOrigin
        "Compute the origin of the axis."
    | w h |
    w := frame width.  h := frame height.
    graphOrigin := frame origin +
        (w * 3 // 20 @ (h * 17 // 20))!

defaultDispatcherClass
        "Answer GraphDispatcher which is the
         default dispatcher of a PlotPane."
    ^GraphDispatcher!

displayPoint: aPoint
        "Answer aPoint scaled to fit on the
        plotting surface."
    | x y |
```

```
    x := graphOrigin x + ((aPoint x - (xRange first))
        / ((xRange size - 1) * xRange increment) *
            (frame width * 7 // 10)).
    y := graphOrigin y - ((aPoint y - (yRange first))
        / ((yRange size - 1) * yRange increment)
            * (frame height * 7 // 10)).
    ^(x @ y) truncated!

initialize
        "Initialize the pane instance variables."
    super initialize.
    topCorner := 1@1.
    curFont := ListFont.
    plotPen := Pen new.
    xTitle := yTitle := title := ''.
    xRange := yRange := 0 to: 10.
    lines:= Array new!

lines: someLines
        "Set the lines to be plotted by the receiver."
    lines := someLines!

open
        "Open the pane.  Get the lines
        to be plotted by the reciever."
    lines :=
        (name notNil and: [model notNil])
            ifTrue: [model perform: name]
            ifFalse: [Array new]!

plotLines
        "Plot the lines in the pane.  See if
        there is more that one line to be plotted
        and plot them accordingly."
    (lines notEmpty and: [
        lines first isKindOf: Point])
            ifTrue: [^self plotPoints: lines].
    lines do: [:line |
        self plotPoints: line]!

plotPoints: somePoints
        "Plot the points in the pane."
    | pts aPen |
    somePoints isEmpty ifTrue: [^self].
    (somePoints first isKindOf: Pen)
        ifTrue: [
            pts := somePoints copyFrom: 2
                to: somePoints size.
            aPen := somePoints first
                clipRect: plotPen clipRect;
                destForm: plotPen destForm]
        ifFalse: [
            aPen := plotPen.
            pts := somePoints].
    self plotPoints: pts withPen: aPen.!

plotPoints: somePoints withPen: aPen
```

```
        "Plot the points in the pane."
    somePoints isEmpty ifTrue: [^self].
    aPen place: (self displayPoint: somePoints first).
    somePoints do: [:aPoint |
        aPen goto: (self displayPoint: aPoint)]!

plotTitle
        "Plot the title of the pane."
    | w h p |
    w := frame width.  h := frame height.
    title isEmpty ifFalse: [
        paneScanner
            display: title
            at: (p := (w - (curFont stringWidth: title)) // 2 @ 4).
        plotPen
            place: (frame origin + p + (0 @ (curFont basePoint y + 1)));
            goto: (frame origin + p + ((curFont stringWidth: title)
                @ (curFont basePoint y + 1)))]!

plotXScale
        "Plot the x-scale of the pane."
    | w h stringWidth |
    w := frame width.
    h := curFont height // 2.
    plotPen place: graphOrigin.
    xRange do: [:x |
        plotPen
            goto: (graphOrigin x + ((x - xRange first) * (w * 7 // 10)
                // ((xRange size - 1) * xRange increment)) @ graphOrigin y);
            tick].
    paneScanner
        display: xTitle
        at: (frame width - ((curFont stringWidth: xTitle) + 2))
            @ (frame height - (curFont height + 2)).
        "at: (frame width + (curFont stringWidth: xTitle)
            // 2 @ (2 * h + plotPen location y - frame top))."
    stringWidth := curFont stringWidth: xRange last printString.
    paneScanner
        display: xRange last printString
        at: (plotPen location - frame origin
            + (stringWidth // -2 @ h)).
    stringWidth := curFont stringWidth: xRange first printString.
    paneScanner
        display: xRange first printString
        at: (graphOrigin - frame origin
            + (stringWidth // -2 @ h))!

plotYScale
        "Plot the y-scale of the pane."
    | w h stringHeight |
    h := frame height.
    yRange do: [:y |
        plotPen
            goto: graphOrigin x @ (graphOrigin y
                - ((y - yRange first) * (h * 7 // 10)
                // ((yRange size - 1) * yRange increment)));
            tick].
```

```
    paneScanner
        display: yTitle
        at: 2 @ (plotPen location y - (frame top
            + curFont height + 8)).
    stringHeight := curFont height.
    w := ((curFont stringWidth: yRange last
        printString) + curFont width) negated.
    paneScanner
        display: yRange last printString
        at: (plotPen location - frame origin
            + (w @ (stringHeight // -2))).
    w := ((curFont stringWidth: yRange first
        printString) + curFont width) negated.
    paneScanner
        display: yRange first printString
        at: (graphOrigin - frame origin
            + (w @ (stringHeight // -2)))!

scrollHand: oldPoint to: newPoint
        "Do nothing.  This method
        is used by v286."!

scrollLeft: anInteger
        "Do nothing.  This method
        is used by v286."!

scrollTopCorner: anInteger
        "Do nothing.  This method
        is used by v286."!

scrollUp: anInteger
        "Do nothing.  This method
        is used by v286."!

selectAtCursor
        "The pane has been selected.
        Inform the model (if necessary)"
    changeSelector notNil ifTrue: [
        model perform: changeSelector]!

showWindow
        "Display the receiver pane
         and the selection."
    Display white: paneScanner clipRect.
    plotPen clipRect: paneScanner clipRect.
    self
        computeOrigin;
        border;
        plotTitle;
        plotXScale;
        plotYScale;
        plotLines!

title: aString
        "Set the value of the title."
    title := aString!
```

```
topCorner
        "Answer the topCorner."
    ^topCorner!

totalLength
        "Answer the height of the pane.
        This method is used by v286."
    ^frame height // curFont height!

update
        "Update the contents of the
         receiver pane."
    self
        open;
        showWindow!

xRange: aCollection
        "Set the value of the xRange."
    xRange := aCollection!

xTitle: aString
        "Set the value of the xTitle."
    xTitle := aString!

yRange: aCollection
        "Set the value of the yRange."
    yRange := aCollection!

yTitle: aString
        "Set the value of the yTitle."
    yTitle := aString! !
```

```
Object subclass: #TaskBrowser
  instanceVariableNames:
    'currentTask currentResource '
  classVariableNames: ''
  poolDictionaries: '' !

!TaskBrowser class methods ! !


!TaskBrowser methods !

acceptResource: aString from: aDispatcher
        "Accept aString as the new contents
        of the resource text pane.  Answer true
        if the string was acceptable."
    | newValue |
    (currentResource isNil or: [currentTask isNil])
        ifFalse: [
            newValue := Compiler evaluate: aString.
            currentTask resources at: currentResource put: newValue].
    self changed: #resourceText.
    ^true!

acceptTaskRule: aString from: aDispatcher
        "Accept aString as the new contents
        of the task rule pane.  Answer true if
        the string was acceptable.  Compile
        and install the PROLOG code."
    | result |
    currentTask isNil ifTrue: [^false].
    result := TaskRules
        compileLogic: aString
        notifying: aDispatcher.
    result isNil
        ifTrue: [^false].
    Smalltalk
        logPrologSource: aString
        forSelector: result key
        inClass: TaskRules.
    currentTask name: (result key
        copyFrom: 1
        to: result key size - 1).
    self
        changed: #taskDialog;
        changed: #taskRule.
    ^true!

addResource
        "Add a new resource to the task."
    | name |
    currentTask isNil ifTrue: [^self].
    (name := Prompter prompt: 'Name ?'
        default: 'Power') isNil
            ifTrue: [^self].
    currentTask resources at: name asSymbol put: 10.
```

```
    currentResource := name asSymbol.
    self
        changed: #resources
        with: #restoreSelected:
        with: currentResource;
        changed: #resourceText!

default: aSymbol
        "Answer the default value for the
        instance variable of the task that
        is extracted by executing the method
        named aSymbol in the current task."
    | value |
    (value := currentTask perform: aSymbol) isNil
        ifTrue: [^'?'].
    (value isKindOf: String)
        ifTrue: [^value].
    ^value printString!

defaultTaskRule
        "Answer the default task rule for the
        task.  This is a PROLOG horn clause."
    | name |
    currentTask isNil ifTrue: [^''].
    name := currentTask name.
    ^name, ' (', name, ') :-
    member ([0, 20, 40, 60], start),
    task (', name,', start, 10).'!

durationField
        "Answer the field that will be used
        to edit the duration of the current
        task in a Dialog with the user."
    ^Field new
        model: self;
        change: #durationFromString:;
        default: (self default: #duration);
        width: 6!

durationFromString: aString
        "Set the duration of the receiver
        from a String representation.  Verify
        the new value and accept a '?' to mean
        that the variable is to be unbound (nil)."
    | number oldValue |
    aString trimBlanks = '?'
        ifTrue: [^currentTask duration: nil].
    number := aString asInteger.
    number = (oldValue := currentTask duration)
        ifFalse: [currentTask duration: number].
    currentTask verify ifFalse: [
        currentTask duration: oldValue.
        self changed: #taskDialog]!

initWindowSize
        "Answer the initial window size of
        the receiver.  This method is used
```

```
        by v286."
    ^(Display width - 20) @
        (Display height * 2 // 3)!

nameField
        "Answer the field that will be used
        to edit the name of the current task
        in a Dialog with the user."
    ^Field new
        model: self;
        change: #nameFromString:;
        default: currentTask name;
        width: 6!

nameFromString: aString
        "Set the name of the current task from a String."
    aString = currentTask name
        ifFalse: [currentTask name: aString].
    self changed: #taskRule!

openOn: aTask
        "Open a browser on aTask.  Provide
        a user interface to a Task object."
    | aTopPane |
    currentTask := aTask.
    aTopPane := TopPane new.
    aTopPane
        label: 'Task Browser';
        minimumSize: (self initWindowSize);
        model: self.
    aTopPane
        addSubpane:
            (DialogBox new
                title: 'Task Values';
                model: self;
                name: #taskDialog;
                framingRatio: (0@0 corner: (1/2)@(1/2))).
    aTopPane
        addSubpane:
            (ListPane new
                title: 'Resource Name';
                model: self;
                name: #resources;
                change: #selectResource:;
                menu: #resourceMenu;
                framingRatio: (0@(1/2) corner: (1/2)@(3/4))).
    aTopPane
        addSubpane:
            (TextPane new
                title: 'Resource Value';
                model: self;
                name: #resourceText;
                change: #acceptResource:from:;
                framingRatio: (0@(3/4) corner: (1/2)@1)).
    aTopPane
        addSubpane:
            (TextPane new
```

```
                title: 'Task Rules';
                model: self;
                name: #taskRule;
                change: #acceptTaskRule:from:;
                framingRatio: ((1/2)@0 corner: 1@1)).
    aTopPane dispatcher open scheduleWindow!

removeResource
        "Remove the current resource from the
        current task.  Refresh affected panes."
    (currentResource isNil or: [
        currentTask isNil or: [
            (currentTask resources
                includesKey: currentResource) not]])
            ifTrue: [^self].
    currentTask resources removeKey: currentResource.
    currentResource := nil.
    self
        changed: #resources;
        changed: #resourceText!

resourceMenu
        "Answer the Menu for the resource pane that
        lists the resources in the current task."
    ^Menu
        labelArray: #('Add Resource' 'Remove Resource')
        lines: #()
        selectors: #(addResource removeResource)!

resources
        "Answer a collection of the names of
        the resources for the current task."
    currentTask isNil ifTrue: [^#()].
    ^currentTask resources keys asArray!

resourceText
        "Answer the contents of the resource text pane."
    (currentResource isNil or: [currentTask isNil])
        ifTrue: [^''].
    ^(currentTask resources at: currentResource
        ifAbsent: [^'']) printString!

selectResource: aString
        "The resource pane has been selected so
        show the value of the resource that has
        the same name as aString in the resource
        text pane."
    currentResource := aString.
    self changed: #resourceText!

startField
        "Answer the field that will be used
        to edit the start time of the current
        task in a Dialog with the user."
    ^Field new
        model: self;
        change: #startTimeFromString:;
```

```
        default: (self default: #startTime);
        width: 6!

startTimeFromString: aString
        "Set the start time of the receiver
        from a String representation.  Verify
        the new value and accept a '?' to mean
        that the variable is to be unbound (nil)."
    | number oldValue |
    aString trimBlanks = '?'
        ifTrue: [^currentTask startTime: nil].
    number := aString asInteger.
    number = (oldValue := currentTask startTime)
        ifFalse: [currentTask startTime: number].
    currentTask verify ifFalse: [
        currentTask startTime: oldValue.
        self changed: #taskDialog]!

taskDialog
        "Answer the contents of a dialog
        pane that will be used to edit
        the current task."
    currentTask isNil ifTrue: [^#()].
    ^#((' ')
       ('Name:       ' nameField)
       ('Start Time: ' startField)
       ('Duration:   ' durationField))!

taskRule
        "Answer the contents of the task rule pane."
    | text |
    currentTask isNil ifTrue: [^''].
    (text := currentTask rules) asString
        = (currentTask name,':')
            ifTrue: [^self defaultTaskRule].
    ^text! !
```

```
"This file will install in all of the source
files for the Case Based Planning System (CBPS)
user-interface objects.  The CBPS objects
can be found in the file 'cbps.st'.

Edit #FileIn be the path were the source files
are contained and then execute the following:

    | dir |
    dir := Smalltalk at: #FileInDir put:
        (Directory pathName: 'a:\').
    (dir file: 'cbpsuser.st')
        fileIn;
        close.
    Smalltalk removeKey: #FileInDir.

For an example that tests to test the code
try:

    CBPSBrowser example

This expression will create a CBPS, set a
default unordered plan and library, and
invoke the Selector, Executor and Evaluator.

"!

| bytes stream |
Transcript cr; show: 'Filing in CBPS user-interface '.
bytes := 0.
#('fildedtr.cls'
'field.cls'
'dialogbx.cls'
'plotpane.cls'
'titlepan.prj'
'tskbrwsr.cls'
'plnbrwsr.cls'
'lbrrybrw.cls'
'cbpsbrws.cls'
'cbpsuser.mth') do: [:name |
    Transcript show: '.'.
    (stream := FileInDir file: name)
        fileIn; close.
    bytes := bytes + stream size].

Transcript cr; show: 'CBPS user-interface (',
    bytes printString, ' bytes) installed.'.
!
```

```
!Pane methods !

font: aFont
        "Set curFont, the font currently
         associated with the receiver pane."
    curFont := aFont! !

!Pen methods !

tick
        "Draw a cross at the current location."
    | loc tick |
    tick := 2.
    loc := self location.
    self
        goto: loc x - tick @ loc y; goto: loc;
        goto: loc x + tick @ loc y; goto: loc;
        goto: loc x @ (loc y - tick); goto: loc;
        goto: loc x @ (loc y + tick); goto: loc! !

!TextPane methods !

acceptPrompt
        "Private - Accept the prompted string."
    (changeSelector isNil or: [model isNil])
        ifTrue: [^true]
        ifFalse: [
            ^model
                perform: changeSelector
                with: (textHolder lineAt: 1)
                with: dispatcher]!

accept
        "Private - Save the currently edited text."
    name == #yourself
        ifTrue: [
            model := textHolder string.
            ^true].
    (changeSelector isNil or: [model isNil])
        ifTrue: [^true]
        ifFalse: [
            ^model
                perform: changeSelector
                with: textHolder string
                with: dispatcher]! !
```

```
"
****************************************************************************
Project : TitlePane
Date    : Oct 28, 1987
Time    : 16:45:34

Globals :

Classes :

Methods :
    #deactivatePane defined in SubPane.
    #reframe: defined in SubPane.
    #displayTitle defined in SubPane.
    #title: defined in SubPane.
    #displayWindow defined in SubPane.
    #grayTitle defined in SubPane.
    #reverse defined in DisplayMedium.
    #reverse: defined in DisplayMedium.
    #activatePane defined in SubPane.
    #graySelection defined in SubPane.
    #graySelection defined in ListPane.
    #graySelection defined in TextPane.
    #graySelection defined in GraphPane.

****************************************************************************
"!

!SubPane methods !

deactivatePane
        "Reverse the contents of the title."
    | titleFrame clipRect|
    super deactivatePane.
    titleFrame := (frame origin - (0@11)
        extent: (frame width @ 8)).
    clipRect := WindowClip intersect: titleFrame.
    margin isNil
        ifFalse: [Display reverse: clipRect]! !


!SubPane methods !

reframe: aRectangle
        "Change the frame rectangle of the receiver pane
         based on aRectagle. Also initialize the scroll
         bar and the characterScanner."
    | width origin |
    frame := (framingBlock value: aRectangle)
        insetBy: (margin isNil ifTrue: [2@2]
            ifFalse: [2@12 corner: 2@2]).
    paneScanner := CharacterScanner new
        initialize: frame
            font: curFont;
        setForeColor: self topPane foreColor
            backColor: self topPane backColor.
```

```
    width := 10.
    origin := frame corner x - width
        @ frame origin y.
    scrollBar := BitBlt new
        destForm: Display
        sourceForm: (Form new
            width: width height: frame height * 3;
            offset: origin)
        halftone: nil
        combinationRule: Form over
        destOrigin: origin
        sourceOrigin: 0 @ 0
        extent: width @ frame height
        clipRect: (origin extent:
            width @ frame height)! !


!SubPane methods !

displayTitle
        "Display the title of the
        SubPane (if one is present)."
    | titleFrame clipRect|
    titleFrame := (frame origin - (0@11)
        extent: (frame width @ 8)).
    clipRect := (WindowClip intersect: titleFrame).
    margin isNil
        ifFalse: [self border: (titleFrame expandBy: 2@2).
            Display gray: (titleFrame intersect: clipRect).
            CharacterScanner new initialize: titleFrame
                font: Font eightLine;
                "setForeColor: Form white
                backColor: Form black;"
                "blank: 0@0 width: titleFrame width;"
                display: margin
                at: (titleFrame width - (Font eightLine
                    stringWidth: margin) // 2) @ 0]! !


!SubPane methods !

title: aString
        "Set the title of the subpane.
        Here we are being a bit gross
        by using the instance variable
        'margin' (which is not used
        anywhere) to hold the title."
    margin := aString! !


!SubPane methods !

displayWindow
        "Display the portion of the receiver
         pane that intersects with WindowClip."
    (WindowClip intersects: (frame expandBy: 2 @ 2))
```

```
        ifTrue: [
            self displayTitle.
            paneScanner isNil
                ifTrue: [self showWindow]
                ifFalse: [
                    paneScanner clipRect:
                        (WindowClip intersect: frame).
                    self showWindow.
                    paneScanner clipRect: frame]]! !


!SubPane methods !

grayTitle
        "Gray the title of the
        SubPane (if one is present)."
    | titleFrame clipRect|
    titleFrame := (frame origin - (0@11)
        extent: (frame width @ 8)).
    clipRect := WindowClip intersect: titleFrame.
    margin isNil
        ifFalse: [
            CharacterScanner new initialize: clipRect
                font: Font eightLine;
                gray: (0@0 extent: clipRect extent)]! !


!DisplayMedium methods !

reverse
        "Set aRectangle in the receiver to black."
    self
        fill: (0@0 extent: self extent)
        rule: Form reverse
        mask: nil! !


!DisplayMedium methods !

reverse: aRectangle
        "Set aRectangle in the receiver to black."
    self
        fill: aRectangle
        rule: Form reverse
        mask: nil! !


!SubPane methods !

activatePane
        "Reverse the contents of the title."
    | titleFrame clipRect|
    super activatePane.
    titleFrame := (frame origin - (0@11)
        extent: (frame width @ 8)).
    clipRect := WindowClip intersect: titleFrame.
    margin isNil
```

```
        ifFalse: [Display reverse: clipRect]! !


!SubPane methods !

graySelection
        "Change the visual clue of the selection
         to reflect a deactivated window.
         Default is do nothing."
    self grayTitle! !


!ListPane methods !

graySelection
        "Change the visual clue of the selection
         to reflect a deactivated window."
    super graySelection.
    selection notNil
        ifTrue: [
            paneScanner gray:
                (self lineToRect: selection)]! !


!TextPane methods !

graySelection
        "Display the selection in
         gray color."
    super graySelection.
    selection gray! !


!GraphPane methods !

graySelection
        "Private - Window has been deactivated, save
         the pane contents in the backup form."
    (self respondsTo: #saveGraph)
        ifTrue: [self saveGraph]
        ifFalse: [self scrollBarInit].
    ^super graySelection! !
```

Future Work

Author: Stephen Northover (Software Kinetics)
Contract: The Evaluation of a CBPS with Respect to MSS Applications (1500-19)
Date Prepared: April 7, 1989

The following have been identified as potential areas for improvement in Version 2 of the Case Based Planning System:

- add plan execution KB for extra control over when a task can start. For example, a task may need a satellite to begin even though enough power is available. The plan execution KB could encode such rules.

- plan generation and verification using PROLOG rules is inherently inefficient due to the chronological back tracking. Investigate other means for plan generation (island building, dependency directed back tracking, knowledge based approach, constraint based approach). Write a better planner for use by the Constructor.

- plan verification is not performed after a replanning action. If the verify fails, should we call then replanner again or take some other action? This issue needs investigation.

- examine replanning. What is the appropriate method to replan? Should an actual planner be used instead of PROLOG rules? Is this the same kind of planner that could be used by the Constructor during plan generation.

- generalise plans in the library. The plan library may become clogged with many similar plans. The issue of library maintenance could be explored.

- add subtasks and subplans for Plan and Task objects.

- expand the implementation of the Model of the environment. Currently, the model is an exact copy of reality. Create a Model object that knows about recurring tasks, detects patterns in the environment, etc. and makes use of this information when planning.

- use a more complex domain and test the CBPS by simulation. Simulate planning requests, task failures, environmental conditions etc. Attempt to encode the various KB's for the domain. This should point out weaknesses in the CBPS design and implementation.

- add new types of resources. Presently, all resources are renewable (acquired/released). Create a hierarchy of Resource objects to model other types of resources (vary over time, non-renewable, etc.)

- verify plans when using the LibraryBrowser before entering them into the plan library

CACC / CCAC
95180

USERS GUIDE TO VERSION 2 OF THE
CASE BASED PLANNING SYSTEM

TL
797
U74
1989

DATE DUE