A dynamic case-based planning system for space application : software and operation description

Government of Canada / Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT **DOC-CR-SP-88-007**

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: A Dynamic Case-Based Planning System for Space Application: Software and Operation Description

AUTHOR(S): D.L. Deugo, F. Oppacher

ISSUED BY CONTRACTOR AS REPORT NO:

PREPARED BY: Carleton University
School of Computer Science

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 31098-7-21600

DOC SCIENTIFIC AUTHORITY: Peter Adamovits

CLASSIFICATION: Releasable

DATE: September 1988

2. A Dynamic Case-Based Planning System for Space Application: Software and Operation Description

By

Dwight Deugo
Franz Oppacher

School of Computer Science
Carleton University
Ottawa, Ontario
Canada

Sept 21,1988

Executive Summary

The Case-Based Planning System project - CBPS for short - started with this contract, the first phase of the development of the software system. In phase 1 three major tasks were performed. They were:

1) a survey of the related AI planning techniques,
2) a conceptual design using the survey results and dynamic memory techniques, and
3) the further development and implementation of the design in Smalltalk.

This third and final task, proving the validity of the CPBS concepts and techniques, is summarized in the contained report.

In this report, a system implementation description is provided to give, along with the code, a complete description of the design. An Operation Description section presents the functional operation of the implementation, and provides the user a source for understanding the planner's operation. We conclude with a discussion of possible future extensions and enhancements that were noted during the implementation phase.

This report, along with the initial documentation: Planning Techniques Survey: Their Applicability to the Mobile Servicing System, A Proposed Approach for Scheduling Applications (With Respect to the Mobile Servicing System), A Dynamic Case-Based Planning System for Space Station Application, and the code comprises the complete results of the CBPS. What we feel is a robust, autonomous planner that uses Knowledge-Based reasoning, Case-Based reasoning, and Dynamic Memory techniques to created an unique, efficient planner.

Table of Contents

1 INTRODUCTION

The Case-Based Planning System project - CBPS for short - started with this contract, the first phase of the development of the software system. In phase 1 three major tasks were performed. They were:

1) a survey of the related AI planning techniques,
2) a conceptual design using the survey results and dynamic memory techniques, and
3) the further development and implementation of the design in Smalltalk.

This document summarizes task 3 by providing an overall discussion and description of the CBPS implementation. Section 2 provides a System Implementation description. It includes a description of the software, control flow, classes, and objects used in the system. Section 3 provides a Operation Description for user operation and control of the planner. All browsers used for data entry, control procedures, and the operation control are described. Section 4 provides a discussion of possible future enhancements and extensions that were noted during the implementation of it. Section 5 summarizes the report and the project.

2 SYSTEM IMPLEMENTATION DESCRIPTION

2.1 Software Description

In this section, we discuss the operation, software, class definitions and their important methods, and the important predefined system objects used by the CBPS. Its is not the intent to go into detail on every object's definition and methods, but rather provide an overall description that along with the actual code can help a designer better understand the implementation and design. The reader is expected to have a working knowledge of Smalltalk and be familiar with the CBPS design. Planning Techniques Survey: Their Applicability to the Mobile Servicing System, A Proposed Approach for Scheduling Applications (With Respect to the Mobile Servicing System), and A Dynamic Case-Based Planning System for Space Station Application are good references. A good reference for those unfamiliar with the language is the Smalltalk/V Tutorial and Programming Handbook (IBM Version) by digitalk inc.

Smalltalk uses the Object-Oriented paradigm, allowing a designer to model his system in terms that match human thinking and language, and in terms of objects and actions on objects. It provides an integrated programming environment; one can create, modify, execute, and debug software all from within the same environment. It is an ideal environment for complex problems where fast experimentation and exploration of ideas, structures, and algorithms is essential. It proved to be an important part in the success of the implementation of the CBPS. Its flexible environment was well suited for our research prototype.

The CBPS is implemented on an IBM AT using the digitalk Inc. Smalltalk/V Object-Oriented Programming System (OOPS) version 2.0. The IBM AT should be compatible with the following configuration: a hard disk or 1.2M floppy drive, 640k memory,

monochrome monitor, Hercules monochrome card, DOS 3.0, and a MicroSoft two button mouse.

The software developed for the project is provided on two disks. The Operation disk contains six files: go, v.exe, v2ndpart.exec, image, sources.sml, and change.log. By entering the command 'v' at the DOS prompt, the planner's software can be invoked and executed. The FileOut disk contains a collection of files that represent individual source files of various class definitions and methods created by the implementation. These files are provided for easy porting of the implementation to existing Smalltalk systems that do not wish to use the stand alone software provided on the Operation disk but rather add the new classes and methods to their existing systems. The files and their contents are described in Table 1 in Section 2.3.

To load the CBPS into an existing Smalltalk image, fileIn the file Init.st into the image. This will load in all class definitions twice, contained in the files described in Table 1. On the first pass, you will be asked to declare certain variables as global or undeclared. Please selected undeclared when presented with the choice. On the second pass, this will not occur and the classes will load to completion.

Before the CBPS can be used after filingIn the file Init.st, the icons in Figure 6 in section 2.3.9 must be created. These are the icons shown in the four boxes in Figure 6, consisting of the TasksIcon, the PlanIcon, the PlanLibraryIcon, and the KnowledgeBaseIcon. These are created using the FreeDrawing bit editor. Evaluate the text 'FreeDrawing new open' to open the bit editor. Create the four icons, roughly one inch by one inch, and save using the names: TasksIcon, PlanIcon, PlanLibraryIcon, and KnowledgeBaseIcon. These icons will be retrieved and drawn when the planning environment is opened, as in Figure 6 in section 2.3.9.

The file update.st contains extra methods that were added to existing Smalltalk classes, and the declaration of predefined system objects (see Section 2.4)

2.2 Overview of the CBPS Operation

The operation of the CPBS is controlled through an instance of the Planner object. The object implements a planning environment and window (see Figure 6 in Section 2.3.9) in which all planning operation are available. Accessible through the planner window are the *Plan Library*, the *Knowledge Base*, the *Planning Tasks* , the *Plan,* and the functions for operating on them. As seen in Figure 6 in Section 2.3.9, each one of these objects is displayed in the planner's window. It can be selected by the user to perform the various functions. The PlanLibrary consists of a collection of Plan objects which are used by the planner when locating a plan whose task objects closely match the current Planning Tasks. The Planning Tasks consist of a collection of Task objects which represent the current tasks that are to be planned for. When the user selects the PlanLibrary object in the planner's window, a Plans Browser is opened that provides the ability to view, create, or edit the current plans in the library. The KnowledgeBase provides access to a Logic Browser which is used to store the Knowledge Base rules used in replanning. When the user selects the Plan object in the planner's window, a Tasks Browser is opened that provides the ability to view, create, or edit the current Planning Tasks. By selecting the Plan object, the user may access all of the planning functions: generate, view, execute, evaluate, and clear. When generate is selected, the planner generates a new plan, building one from the planning tasks using a plan located in the Plan Library to guide for the generation process. When view is selected, a Plan Browser is opened permitting the user to view the generated plan. When execute is selected, an Execution Browser is opened permitting the user to simulate the execution of the generated plan. An option is available to fail tasks in the plan in order to cause a replanning exercise. When evaluate is selected, the plan's execution is evaluated and the plan is added, updated, removed, or forgotten from the Plan Library. The evaluation information is added to an Evaluator object

as the plan is generated and executed for later evaluation. The evaluation is initiated by selecting the evaluate option.

Each object and browser above has an associated class description by the same name. The main point here is the Planner object is always in control. It determines what object the user has selected, it opens the appropriate Browsers, it controls the planning process, it stores the current plan object, and it invokes the actions on the plan.

The flow of data through the system is as follows. The user initially builds up a library of plans using the Plan Library Browser. The user initially creates the Knowledge Base replanning rules using the Logic Browser. The user defines his current planning tasks using the Tasks Browser. The user selects the generated option of the plan object. This generates a plan using the Plan Library, the planning tasks, and the Knowledge Base rules for replanning. The new plan can be viewed by selecting the view option from the plan object which opens up a Plan Browser. The same process is used to execute a plan which opens up an Execution Browser for the plan. Finally, the evaluate option is selected and the appropriate action taken with the executed plan by the Evaluator object, using the information stored in it during plan generation and execution. The plan can then be cleared.

2.3 Class Descriptions

In this section, each class is describe in detail to give the reader a better understanding of the CBPS prototype implementation and design. For each class we include a discussion on its inheritance, its instance variables, its methods, and a general description of the class's operation and important methods.

Smalltalk's object classes and methods are defined using the Class Hierarchy Browser. The Class Hierarchy Browser is an interactive browser used for the creation and removal of object

definitions and methods in Smalltalk. The Class Hierarchy Browser consists of three subpanes: the Object subpane which lists the currently defined objects, the Methods subpane which lists the currently defined methods for the object, and the Text subpane which contains the definition or code for the selected object or object's method. By using the *Add SubClass* and *Add Method* menu options, available in the Object and Method subpanes, the CBPS classes in the following section were created.

The source code for each class is located in the files identified in Table 1.

Class File	Name
Task	task.st
PlanningConstraints	planCon.st
Power	power.st
TaskTime	tasktm.st
StartTime	starttm.st
StopTime	stoptm.st
Temperature	temp.st
PlanningResources	planRes.st
RequiredResources	reqRes.st
PowerResource	powerR.st
ReturnResource	returnR.st
PowerReturn	pwrRet.st
PlanningFailure	planFal.st
PlanningReading	planRed.st
PowerReading	pwrRed.st
TemperatureReading	TemRed.st
Plan	plan.st
PlannerDisplay	PlnDis.st
KnowledgeBase	Kb.st
Plan	planIcn.st
PlanLibrary	plnLib.st
PlannerTasks	plntsk.st
Evaluator	eval.st
TaskBrowser	TBrw.st
TasksBrowser	TskBrw.st
PlanBrowser	PlnBrw.st

PlansBrowser	PlnsBr.st
ExecutionBrowser	ExBrw.st
Planner	Planer.st
Updated methods	Update.st
Generate the system	Init.st

Table 1. Source Files

2.3.1 Task Class

Inheritance:

Super Class: Object.
SubClasses: None.

Instance Variables:

name - A string identifying the name of the task.
description - A string identifying the description of the task.
constraints - A dictionary of constraint objects used to store the constraints of the task.
resources - A dictionary of resource objects used to store the resources of the task.
returns - A dictionary of return objects used to store the returns of the task.

Methods:

addConstraint: - Given a new constraint type object, add the constraint to the task's constraint dictionary using the name of the constraint as the index.
addResource: - Given a new resource type object, add the resource to the task's resource dictionary using the name of the resource as the index.

addReturn: - Given a new return type object, add the return to the task's return dictionary using the name of the return as the index.

constraintLabel - This method returns a menu object, with the selection list of the menu composed of the names of all of the the task's constraints found in the constraints dictionary instance variable.

constraints - This method returns the task's constraint instance variable object.

description - This method returns the task's description.

initialize: Given a text string, the task object is initialized with new dictionary objects in the constraint, resources, and returns instance variables, a empty description, and the name instance variable set to the passed text string.

name - This method returns the task's name instance variable object.

resourceLabel - This method returns a menu object, with the selection list of the menu composed of the names of all of the the task's resource objects found in the resources dictionary instance variable.

resource - This method returns the task's resources instance variable object.

returnLabel - This method returns a menu object, with the selection list of the menu composed of the names of all of the the task's return objects found in the returns dictionary instance variable.

returns - This method returns the task's returns instance variable object.

setDescription: - Given a string, this method set the task's description instance variable to the string.

Discussion:

The Task object definition is central to the design of the CBPS. It defines and includes information about a single, unique task the CBPS

must plan for. The task information includes: its name, its description, its constraints, its resources, and its returns. There is an instance variable for each of these information types and is identified by the same name. The method *initialize:* initializes these instance variables and sets the name of the task to the passed string.

The *constraints*, *resources*, and *returns* instance variables are initialized to be Dictionary type objects. Each of these Dictionary objects stores the task's corresponding PlanningConstraints, PlanningResources, and ReturnResources objects (see definitions). All constraints, resources and returns objects are indexed in the corresponding dictionary using their individual names, and for that reason must be unique.

Other methods allow for the manipulation (e.g. *addConstraint:*) and retrieval (e.g. *constraints*) of the above mentioned instance variables.

2.3.2 PlanningConstraints Class

Inheritance:

Super Class: Object
SubClasses: Power, TaskTimes, Temperature

Instance Variables

currentValue - Any type of object that represents the current value of the planning constraint, e.g. integer, time, etc.
name - A string object representing the name of the planning constraint.

Methods

example - This method returns a text string representing the expected text input for the planning constraint on a browser.

get - This method returns the currentValue instance variable object.

getAsText - This method returns the currrentValue instance variable object as a text string.

name - This method returns the name instance variable object.

name: - Given a text string, this method set the name instance variable object to the passed string.

set: - Given a value represented as a string, this method sets the object's currentValue instance variable to the appropriate type passed on the string and constraint type, e.g. if the planning constraint was a Temperature object the current value would be set to an integer.

unifyWith:forPlanner:name - Given a planning constraint and a name, this method unifies the values in the passed planning constraint to those of itself.

Description:

A planning constraint is an item such as power, temperature, start-time, or stop-time, that is used to constrain a task in a plan. The PlanningConstraint object is used to organize the currently defined Constraint type objects in the system, and to supply the common methods that are available to them. The Constraint type objects include: the Power, the TaskTime, and the Temperature objects. All of these objects inherit the instance variables defined for the PlanningConstraint object. These include: *currentValue* and *name*. The *currentValue* and *name* instance variables contain, as their names suggests, the current value of the constraint and its name. Methods are provided to allow for the manipulation (*set:*, *name:*) and retrieval (*get*, *name*) of the instance variables.

It is important that each of the PlanningConstraint subclass objects have the methods *example*, *verifyAt:* and *unifyWith:forPlanner:name* defined, or be willing to use the methods already defined for the PlanningConstraint object. The *example*

method returns a string, providing an example of a text entry for the constraint. The *verifyAt*: method verifies that the current constraint is valid at the passed time. The method *unifyWith:forPlanner:name* unifies the current constraint with that of a passed similar constraint.

Each of the PlanningConstraint type objects is now discussed.

2.2.2.1 Power Class

Inheritance:

Super Class: PlanningConstraints
SubClass: None

Instance Variables:

None

Methods:

example - This method returns a text string representing the expected text input for the power constraint on a browser.
verifyAt: - Given a Time object, this method verifies that the object's current value is within the correct limits at the passed time.

Discussion:

The Power object defines a constraint that contains an integer value representing a number of watts.

2.3.2.1 TaskTime Class

Inheritance:

Super Class: PlanningConstraints
SubClass: StartTime, StopTime

Instance Variables:

None

Methods:

addTime: - Given a Time object, this method increments its current value by the passed time amount.

asSeconds - This methods returns a integer value representing the object's current value as a number of seconds.

example - This method returns a text string representing the expected text input for the time constraint on a browser.

seconds: - Given an integer number of seconds, this method sets its currentValue to a Time object representing the passed number of seconds

verifyAt: - Given a Time object, this method verifies that the object's current values is within the correct limits at the passed time.

Discussion:

The TaskTime object is used to organize the currently defined TaskTime type constraint objects in the system, and to supply the common methods that are available to them. The TaskTime constraint objects defines a constraint that contains a Time object representing a time. Methods are supplied to access the specific Time object (*addTime:, asSeconds, seconds:*) that is the current value of the constraint. There are two types of TaskTime objects, StartTime and

StopTime. Each one supplies its own *unifyWith:forPlanner:name* method and relies on the TaskTime Object's *verifyAt:*.

2.3.2.1.1 StartTime Class

Inheritance:

Super Class: TaskTimes
SubClass: None

Instance Variables:

None

Methods:

unifyWith:forPlanner:name - Given a StartTime object and a name, this method unifies the values in the passed StartTime to those of itself.

Discussion:

The StartTime object defines a constraint that contains a Time object representing a time (e.g. :12:30:01:).

2.3.2.1.2 StopTime Class

Inheritance:

Super Class: TaskTimes
SubClass: None

Instance Variables:

None

Methods:

unifyWith:forPlanner:name - Given a StartTime object and a name, this method unifies the values in the passed StartTime to those of itself.

Discussion:

The StopTime object defines a constraint that contains a Time object representing a time (e.g. :18:45:59:).

2.3.2.2 Temperature Class

Inheritance:

Super Class: PlanningConstraints
SubClass: None

Instance Variables:

None

Methods:

example - This method returns a text string representing the expected text input for the temperature constraint on a browser.

verifyAt: - Given a Time object, this method verifies that the object's current values is within the correct limits at the passed time.

Discussion:

The Temperature object defines a constraint that contains an integer value representing a number of degrees Celsius.

2.3.3 PlanningResources Class

Inheritance:

Super Class: Object
SubClasses: RequiredResources, ReturnResources

Instance Variables

currentValue - Any type of object that represents the current value of the planning resource.
name - A string object representing the name of the planning resource.

Methods

example - This method returns a text string representing the expected text input for the planning resource on a browser.
get - This method returns the currentValue instance variable object.
getAsText - This method returns the currrentValue instance variable object as a text string.
name - This method returns the name instance variable object.
name: - Given a text string, this method sets the name instance variable object to the passed string.
set: - Given a value represented as a string, this method sets the object's currentValue instance variable to the appropriate type passed on the string and resource type, e.g.

if the planning resource was a PowerResource object the current value would be set to an integer.

unifyWith:forPlanner:name - Given a planning resource and a name, this method unifies the values in the passed planning resource to those of itself.

Description:

Planning resources are items such as power that, like PlanningConstraint objects, are also used to constrain a task in a plan. The PlanningResources object is used to organize the currently defined Resource type objects in the system, and to supply the common methods that are available to them. These objects include: the RequiredResources and ReturnResource objects. These objects in turn organize the Required and the Return resources, currently defined as the PowerResource and the PowerReturn objects. These objects inherit the instance variables that are defined for the PlanningResources object, which consist of *currentValue* and *name*. The *currentValue* and *name* instance variables contain, as their names suggest, the current value of the resource and its name. Methods are provided that allow the manipulation (*set:*, *name:*) and retrieval (*get*, *name*) of the instance variables.

It is important that each of the PowerResource and PowerReturn objects have the methods *example*, *set*, and *verifyAt:* defined, or be willing to use the methods already defined for the PlanningResources object. The *example* method returns a string, providing an example of a text entry for the resource. The *verifyAt:* method verifies that the current resource is valid at the passed time. The *set:* method sets the *currentValue* of the resource to the correct type and value, based on the string passed to the method and the resource type. For example, the Power resource sets *currentValue* to and integer value based on the fact that a Power resource should be an integer.

The method *unifyWith:forPlanner:name* unifies the current resource with that of a passed similar resource.

Each of the PlanningResources type objects is now discussed.

2.3.3.1 RequiredResources Class

Inheritance:

Super Class: PlanningResources
SubClasses: PowerResource

Instance Variables

None.

Methods

None.

Description:

The RequiredResources object is used to organize the currently defined Required Resource type constraint objects in the system. The only Required Resource currently defined is the PowerResource object.

2.3.3.1.1 PowerResource Class

Inheritance:

Super Class: RequiredResources
SubClass: None

Instance Variables:

None

Methods:

example - This method returns a text string representing the expected text input for the power resource on a browser.

set: - Given an integer resource value, this method sets the currentValue of the resource to the passed value.

verifyAt: - Given a Time object, this method verifies that the object's currentValue is within the correct limits at the passed time.

Discussion:

The PowerResource object defines a resource that contains an Integer object representing a number of watts (e.g. 20).

2.3.3.1.2 ReturnResource Class

Inheritance:

Super Class: PlanningResource
SubClasses: PowerReturn

Instance Variables

None.

Methods

None.

Description:

The ReturnResource object is used to organize the currently defined Return Resource type constraint objects in the system. The only Return Resource current defined is the PowerResource object.

2.3.3.1.2.1 PowerReturn Class

Inheritance:

Super Class: ReturnResource
SubClass: None

Instance Variables:

None

Methods:

example - This method returns a text string representing the expected text input for the power return on a browser.
set: - Given an integer resource value, this method set the currentValue of the return to the passed value.

Discussion:

The PowerReturn object defines a return resource that contains an Integer object representing a number of watts (e.g.20).

2.3.4 PlanningFailure Class

Inheritance:

Super Class: Object
SubClass: None

Instance Variables:

typeOfFailure - A text string representing a combination of the the failed task's name and failed constraint's name.

description - A text string representing a description of the failure.

numbeOfFailures - A integer representing the number of times the failure has occurred.

failedTask - A symbol representing the failed Task's name.

failedConstraint - A symbol representing the failed Constraint's name.

Methods:

constraint - This method returns the failedConstraint instance variable object.

description - This method returns the description instance variable object.

incrementCounter - This method increments the numberOfFailures instance variable by one.

name - This method return a text string representing the failure's name.

numberOf - This method returns the numberOfFailures instance variable object.

resetCounter - This method sets the numberOfFailures instance variable to zero.

set:task:constraint: - Given the three symbols, this method initializes the typeOfFailures, failedTask, and failedConstraint instance variables to the passed values. It also sets the numberOfFailures instance variable to one and generates an initial value for the description instance variable using the failed task and constraint names.

Discussion:

An essential part of a plan is its failure information. An instance of a PlanningFailure object records an instance of a failure in a plan. A PlanningFailure records the type of failure, its description, the failed task, the failed constraint, and the number of failures of this type. The PlanningFailure object has instance variables for each of these information types. Methods are provided to manipulate (*set:task:constraint*, *setDescription*, resetCounter) and retrieve (*constraint*, *task*, *type*, *description*) the instance variables. The plan's current failures are stored in an instance variable of plan object (see description).

2.3.5 PlanningReading Class

Inheritance:

Super Class: Object
SubClass: PowerReading, TemperatureReading

Instance Variables:

None

Methods:

None.

Discussion:

A Plan reading is the expected value of a constraint at time *t*. The PlanningReading object helps to organize the Plan reading objects. They calculate for example, the expected power and temperature readings at time *t*. The Plan Reading object types currently defined are PowerReading and TemperatureReading. The only method each provides is the class method *atTime:* By sending this method to the

reading class, the expected temperature or power reading at time *t* is returned. Each is now described.

2.3.5.1 PowerReading Class

Inheritance:

Super Class: PlanningReadings
SubClass: None

Instance Variables:

None

Class Methods:

atTime: Given a Time object, this method returns an integer representing the expected power at the passed time.

Discussion:

The expected value reading for a power constraint at time *t*.

2.3.5.2 TemperatureReading Class

Inheritance:

Super Class: PlanningReadings
SubClass: None

Instance Variables:

None

Class Methods:

atTime: Given a Time object, this method returns an integer representing the expected temperature at the passed time.

Discussion:

The expected value reading for a temperature constraint at time *t*.

2.3.6 Plan Class

Inheritance:

Super Class: Object
SubClass: None

Instance Variables:

name - A text string representing the name of the plan.
description - A text string representing the description of the plan.
tasks - A dictionary of Task objects indexed by the task's name.
startTime - A StartTime object representing the start time of the plan.
stopTime - A StopTime object representing the stop time of the plan.
successes - An integer representing the number of times the plan has executed to completion.
failures - A dictionary of PlanningFailure objects indexed by the failure's name.
planOrder - An OrderedCollection representing the task execution order of the plan. The task's start times are used to order the collection.

Methods:

addFailure: - Given a PlanningFailure object, this method adds the object to the failures instance variable.

addMissingTasks:forPlanner: - Given a dictionary of Task objects, this methods adds those tasks in the passed dictionary that are not in its task dictionary to the task dictionary instance variable.

addTask: Given a Task object, this methods adds the object to the task dictionary instance variable.

description - This method returns the description instance variable object.

failureAtTask:withConstraint: This method looks in the plan's failures for a failure identified by the passed task and constraint. If one can be found the method returns the failure object found in the failures instance variable. If none is found it returns nil.

findAPlace:with: - Given a Task object, this method adds the task to the plan in the next available slot that fits the task's start and stop time and does not cause any conflicts with the tasks in the existing plan.

getFailures - This method returns the failures instance variable object.

getStartTime - This method returns the startTime instance variable object.

getStopTime - This method returns the stopTime instance variable object.

getTasks This method returns the tasks instance variable object.

incrSuccesses - This method increments the successes instance variable by one.

matchRatingFor: - Given a dictionary of tasks, this method returns a match rating for the tasks in the passed dictionary and those in the plan's tasks instance variable. The rating is an OrderedCollection that contains three values: the number of extra tasks found, the number of missing tasks not found, and the number of failures of the plan.

moveToTheEnd:with - Given a Task object, this methods adds the task at the end of the plan's task dictionary instance variable.

name - This method returns the name instance variable object.

new: - Given a string, this methods initializes the plan to its initial values and sets its name to the passed string.

numberOfTasks - This method returns an integer representing the number of tasks in the tasks instance variable.

orderPlan - This method orders the tasks in the plan using the plan's tasks start time values, and stores the ordered task collection in the planOrder instance variable.

planOrder - This method returns the planOrder instance variable object.

removeExtraTasks:forPlanner: - Given a dictionary of tasks, this methods remove the tasks in the plan's tasks instance variables that are not in the passed dictionary of tasks.

setDescription: - Given a text string, this method sets the plan's description instance variable to the passed string.

setFailures: - Given a PlanningFailure object, this method sets the plan's failures instance variable to the passed failure.

setName: - Given a text string object, this method sets the plan's name instance variable to the passed string.

setStartTime: - Given a StartTime object, this method sets the plan's startTime instance variable to the passed startTime.

setStopTime: - Given a StopTime object, this method sets the plan's stopTime instance variable to the passed stop time.

setSuccesses: - Given an integer object, this method sets the plan's successes instance variable to the passed integer.

setTasks: - Given a dictionary of Task objects, this method sets the plan's tasks instance variable to the passed dictionary.

successes - This method returns the plan's successes instance variable object.

takeNoAction:with: Given a Task object, do nothing but reporting nothing was done in the browser window.

unifyWith:forPlanner: - Given a dictionary of Task objects, unify them with the plan's tasks and save the unification in the plan's tasks

verifyFor:evaluation: This method verifies that all the plan's task constraints, resources, and returns values are

acceptable at the task's start time. If a failure occurs, the method invokes the replanning mechanism and attempts to correct the plan.

Discussion:

A Plan object stores all of the information associated with a plan. This includes: its name, its description, the tasks associated with the plan, the order the task should be executed (based on each task's start time), the failures associated with the plan, its start time, its stop time, and the number of times the plan has succeeded in executing. There are instance variables for each of these information types, and each is identified by the same name.

The plan's *failures* and *tasks* instance variables are both Dictionary type objects. Each dictionary stores either Task or PlanningFailure objects, using their corresponding names as the index into the dictionary. The plan's *description* and *name* instance variables store a text description and name of the plan. The *startTime* and *stopTime* instance variables hold time objects, representing the start and stop times of the plan. The plan's *successes* instance variable holds an integer number representing the number of times the plan has executed. The *planOrder* instance variable is an OrderedCollection object that contains the plan's tasks. It uses their start time as the basis for ordering the collection of tasks.

The method *new:* initializes a plan object's instance variables to an initial configuration; no tasks, no failures, zero successes, no description, an empty plan order, and a name set to the passed string supplied to the method. Other methods are provided for the manipulation (*addFailure:, addTask:, setStartTime:, setStopTime:, setFailures:, setSuccesses::, setTasks:, setDescription, orderPlan*) and retrieval (*description, getFailures, getStartTime, getStopTime, getTasks, name, planOrder*) of the instance variables.

After a plan from the plan library has been selected to be the base plan, the tasks not required in the plan are removed and any tasks not in the plan that are required are added. The methods *removeExtraTasks:forPlanner:* and *addExtraTasks:forPlanner:* handle these operations. These methods receives a dictionary of tasks to be planned for as a method variable. These are known as the planning tasks. The plan's current tasks dictionary is checked for those tasks that are missing and those that are extra, and the appropriate actions are performed on the base plan's Tasks dictionary.

Another important method is *matchRatingFor:*. This method receives a dictionary of planning tasks as a method variable. These tasks are compared with the plan's current tasks in order to return a match rating (OrderCollection object). The rating identifies the number of extra tasks, the number of missing tasks, and the number of failures the plan has experienced in the past. The method's results are used during plan generation to help determine what plan in the Plan Library is the best match with the current set of planning tasks.

Another method, *unifyWith:forPlanner*, unifies the plan's current tasks with the planning tasks. After the base plan has been selected and the extra and missing tasks removed and added, the plan is ready to by unified with the planning tasks. The base plan's tasks and the planning tasks are similar in name because they have be made identical to begin with. However, their constraints will differ or perhaps be missing altogether. The purpose of this method is to install the unification of the plan's tasks and passed planning task's constraints in the base plan.

The method *verifyFor:*, verifies that the plan will execute given its constraints. It uses the expected constraint values as its basis for determining the validity of the plan. Each of the plan's tasks is individually verified, which involves verifying each of the task's constraints and resources. This process is repeated for each task until

all tasks have been verified or a failure occurs. In the event of a failure, the plan's failure information is indexed using the failing task and constraint to located a replanning action to take on the failing task. If no failure action is located, the Knowledge Base is indexed in a similar manner to locate a replanning action. The replanning action is then executed on the plan which, in the prototype's case, could be either the method *takeNoAction:with:*, *moveToTheEnd:with:*, *or findAPlace:with*. These methods may alter a failed task's location in the plan. After the action is performed, the method returns True or False depending on whether any replanning action was actually done for the failure. If no replanning action was done, plan verification continues from its current task. It is important that a Knowledge Base rule or planning failure conclude with an available replanning action. If new replanning actions are to be concluded, new replanning actions methods must be created. After any replanning action, the complete plan is verified again. Currently only two iterations of replanning are allowed before verification is terminated. This value can be easily increased in the method *verifyFor:*. After all tasks have verified, the plan is ready for execution.

Many methods make reference to a *aPlanner*. This is the Planner object. By referencing it, the method can write to the Planner's text subpane (see the Planner Object definition for further discussion).

2.3.7 PlannerDisplayObjects Class

Inheritance:

Super Class: Object
SubClasses: KnowledgeBase, Plan, PlanLibrary, PlannerTasks

Instance Variables:

form - A form object that stores the bit map of the display object.
rectangle - A rectangle object that defines the display object's display area.

clippingBox - A rectangle object that defines the boundary of the display object.

Methods:

draw - This method draws the display object's form and rectangle on the screen.

drawRectangle - This method draws the display object's rectangle on the screen.

getForm - This method returns the form instance variable object.

getRectangle - This method returns the rectangle instance variable object.

setForm:setRectangle:clipping: - Given a Form, Rectangle, and clipping box Rectangle objects, set the corresponding instance variables to the passed objects.

setRectangle: - Given a Rectangle object, set the rectangle instance variables to the passed object.

Discussion:

The PlannerDisplayObjects object organizes the graphical display objects that appear in the CBPS planner pane (see Planner object), and provides methods that are common to all PlannerDisplayObjects type objects. These objects include: the KnowledgeBase object, the Plan object, the PlanLibrary object, and the PlannerTasks object. Each object has three instance variables: *form*, *rectangle*, and *clipBox*. The *form* variable stores the form object of the display object. The *rectangle* variable stores the rectangle object of the display object. The *clipBox* variables stores the clipping rectangle object of the display object.

The method *setForm:setRectangle:clip* initializes the above three instance variables to the objects passed to the method. The display objects are then drawn using the method *draw*. This method does a BitBlt operation on the display object's definition form to the form

defined by the *form* instance variable, using the clipping box and rectangle instance variables as other parameters for the BitBlt operation. The only method defined for CBPS Display subclass's objects is *form*. This method returns a form which contains the bit map form of the display object. These forms are drawn using the form editor and are stored in its Picture Dictionary under the indexes 'KnowledgeBaseIcon', 'PlanIcon', 'TasksIcon', and 'PlanLibraryIcon'.

Other methods are provided to access the instance variables (*getForm, getRectangle, form*) and to display a bounding box around the display object (*drawRectangle*).

2.3.7.1 KnowledgeBase Class

Inheritance:

Super Class: PlannerDisplayObjects
SubClasses: None

Instance Variables:

None.

Methods:

form - This method returns the form of the display object. It is retrieved from the FreeDrawing's picture dictionary.

Discussion:

The KnowledgeBase object defines the Knowledge base icon shown on the CBPS planner form.

2.3.7.2 Plan Class

Inheritance:

Super Class: PlannerDisplayObjects

SubClasses: None

Instance Variables:

None.

Methods:

form - This method returns the form of the display object. It is retrieved from the FreeDrawing's picture dictionary.

Discussion:

The Plan object defines the Plan icon shown on the CBPS planner form.

2.3.7.3 PlanLibrary Class

Inheritance:

Super Class: PlannerDisplayObjects
SubClasses: None

Instance Variables:

None.

Methods:

form - This method returns the form of the display object. It is retrieved from the FreeDrawing's picture dictionary.

Discussion:

The PlanLibrary object defines the PlanLibrary icon shown on the CBPS planner form.

2.3.7.4 PlannerTasks Class

Inheritance:

Super Class: PlannerDisplayObjects
SubClasses: None

Instance Variables:

None.

Methods:

form - This method returns the form of the display object. It is retrieved from the FreeDrawing's picture dictionary.

Discussion:

The PlannerTasks object defines the PlannerTasks icon shown on the CBPS planner form.

2.3.8 Evaluator Class

Inheritance:

Super Class: Object
SubClasses: None

Instance Variables:

missingTasks - An integer number representing the number of missing tasks in the plan.
extraTasks - An integer number representing the number of extra tasks in the plan.
newFailure - An integer number representing the number of new failures found in plan verification.
oldFailure - An integer number representing the number of previous failures found in plan verification.

replanningOldFailure - An integer number representing the number of previous failures found in replanning during plan execution.

replanningNewFailure - An integer number representing the number of new failures found in replanning during plan execution.

numberOfTasks - An integer number representing the number of tasks in the plan.

Methods:

confirm:for: - Given a Plan and dictionary of Plans, this method performs an evaluation of the plan using the Evaluator's internal data.

evaluate - This method performs the evaluation of the plan in the previous method, and returns a symbol which represents the method (action) to use to complete the evaluation of the plan. The evaluation is based on its internal data which was generated as the plan was generated and executed.

failureCount - This method returns the sum of the newFailure and oldFailure instance variables.

failurePercentage - This method returns the failureCount divided by the number of tasks.

forgetPlan:from: This action method disregards the plan at the users verification.

incrNewFailure - This method increments the newFailure instance variable by one.

incrOldFailure - This method increments the oldFailure instance variable by one.

incrReplanningNewFailure - This method increments the replanningNewFailure instance variable by one.

incrReplanningOldFailure - This method increments the replanningOldFailure instance variable by one.

initialize - This method initializes all instance variables to zero.

newPlan:From: - This action method will add the plan to the plan library under a new name at the users verification.

forgetPlan:From: - This action method will remove the plan from the plan library at the users verification.

replanningCount - This method returns the sum of the replanningNewFailure and replanningOldFailure instance variables.

setExtra: - Given an integer number, this method sets the extraTasks instance variable to the passed number.

setMissing: - Given an integer number, this method sets the missingTasks instance variable to the passed number.

setNumberOfTasks: - Given an integer number, this method sets the numberOfTasks instance variable to the passed number.

updatePlan:From: - This action method will update the plan in the plan library at the users verification.

Discussion:

The Evaluator object records a plan's planning, replanning, and execution information; this enables it to generate and carry out a recommendation for the plan. The recommendation could be to update the plan in the plan library with the new information, add the plan to the plan library under a new name, remove the plan from the plan library, or do nothing.

The method *initialize* initializes all of the object's instance variables to zero. These instance variables include:

- *missingTasks*, which represents the number of tasks that are missing in the base plan and are in the original task requirements.

- *extraTasks*, which represents the number of task that are in the base plan and are not in the original task requirements.

- *numberOfTasks*, which represents the number of tasks in the base plan.

- *newFailure*, which represents the number of new failures experienced in plan's execution.

- *oldFailure*, which represents the number of old failures experienced in plan's execution.

- *replanningNewFailure*, which represents the number of new failures experienced in the plan's verification.

- *replanningOldFailure*, which represents the number of old failures experienced in the plan's verification.

Methods are provided for initializing the *missingTasks, extraTasks, and numberOfTasks* instance variables (*setExtra:, setMissing:, and setNumberOfTasks:*), and for incrementing the failure counters (*incrNewFailure, incrOldFailure, incrReplanningNewFailure, and incrReplanningOldFailure*).

At plan creation, a new Evaluator object is created. As the plan is created, verified, and executed, the corresponding information is sent to the Evaluator object. After the plan has execute, the plan is ready for evaluation; the *confirm:for:* method is sent to the Evaluation object to perform this. This method performs an evaluation of itself using the information that has been previously supplied to it. The method *evaluate* actually performs the evaluation and returns the recommended method to executed. The returned method must be

one of *forgetPlan:from:*, *removePlan:from:*, *newPlan:from:*, or *updatePlan:from:*. The evaluation reviews the number of failures, the types of failure, the number of task, basically all of the instance variables to decide the correct recommendation.

The recommendation will either remove the plan from the Plan Library due to excessive repetitive errors (removePlan:From:), update the plan in the Plan Library due to success or few errors (updatePlan:From:), add a new plan due to the Plan Library to the addition of new tasks to the original plan (newPlan:From:), or do nothing due to the addition of new tasks to the original plan and excessive errors (forgetPlan:From:). The operator is always prompted to confirm the recommendation.

2.3.8 TaskBrowser Class

Inheritance:

Super Class: Object
SubClasses: TasksBrowser

Instance Variables:

textDisplay - A text string that holds the text displayed in the text subpane of the browser.
exampleDisplay - A text string that holds the text displayed in the example subpane of the browser.
viewingType - A symbol that holds the selected viewing type: #Description, #Constraints, #Resources, or Returns.
itemSelected - A symbol representing the current description, constraint, resource, or return picked in the item subpane.
isItemPicked - A boolean representing whether itemSelected contains a valid value.
taskSelected - A symbol representing the current task selected for the browser.

isTaskPicked - A boolean representing whether taskSelected contains a valid value.

Methods:

accept:from: - Given a text string, this method locates the current itemSelected in the browser and using the viewingType updates the itemSelected of the viewingType with the passed string. It is used to accept text from the text subpane of the browser.

acceptExample:from: Given a text string, this method does nothing. It is used to accept text from the example subpane of the browser. No action is the desired action. One does not want any user action in the Example subpane to change its contents.

addConstraint - This methods prompts the user for a constraint name and adds a new planningConstraint object to the selectedTask's constraint dictionary.

addItem - Using the viewingType, this method adds a new planningConstraint, planningResource, or returnResources object to the corresponding dictionary of the selected task.

addResource - This methods prompts the user for a resource name and adds a new planningResource object to the selectedTask's resource dictionary.

addReturn - Ths methods prompts the user for a return name and adds a new ReturnResource object to the selectedTask's return dictionary.

constraints: - This method returns #(Constraints).

constraints: - This method receives the #Constraints symbol and updates the item subpane with a list of the current constraints of the selected task.

deleteConstraint - This methods removes the selected constraint object stored in the itemSelected instance variable from the selectedTask's constraints dictionary, and then updates the appropriate browser subpanes.

deleteItem - This method removes the object stored in itemSelected from the task's dictionary identified by the viewingType instance variable.

deleteResource - This methods removes the selected resource object stored in the itemSelected instance variable from the selectedTask's resources dictionary, and then updates the appropriate browser subpanes.

deleteReturn - This methods removes the selected return object stored in the itemSelected instance variable from the selectedTask's returns dictionary, and then updates the appropriate browser subpanes.

description - This method returns the symbol #(Description).

description: - Given the symbol #Description, this method updates the text subpane with the selectedTask's description.

example - This method return the exampleDisplay instance variable object.

itemMenu - This method return the Menu object for the items subpane.

items - This method return the items to list in the items subpane of the browser based on the current viewingType selected.

items: - This method receives the symbol of the selected item in the item subpane, stores the symbol in the itemSelected instance variable, sets the isItemPicked instance variable to true, and updates the appropriated browser subpanes.

openOn: - Given a task, this method opens a TaskBrowser window for the passed task. The instance variables of the object are initialized to their default values.

resources - This method returns the symbol (#Resources).

resources: - This method receives the #Resources symbol and updates the item subpane with a list of the current resources of the selected task.

returns - This method returns the symbol (#Returns).

returns: - This method receives the #Returns symbol and updates the item subpane with a list of the current returns of the selected task.

text - This method returns the textDisplay instance variable object.

Discussion:

The TaskBrowser object implements a Smalltalk like Browser facility for viewing and modifying a Task object's components. The Task Browser, although not actual required by the prototype, can be used to open a Browser window for a given Task. This is done using the method *openOn:* supplied with the TaskBrowser object. The TaskBrowser object's main purpose is to organize methods and instance variables used by its four subclass objects: TasksBrowser, PlanBrowser, PlansBrowser, and ExecutionBrowser. These objects implement Smalltalk like Browsers for their correspond objects.

Each Browser is described in detail in the following sections.

An example TaskBrowser can be seen in Figure 1. By selecting the Description subpane (1), the task's description is displayed in the Text subpane (7). By selecting either the Constraints (2), Resources (3), or Returns (4) subpane, the corresponding items for that item are shown in the Item subpane (5). By selection one of those items, the corresponding value for that item is displayed in the Text subpane (7), with a sample input for the item shown in the Example subpane (6). In the example below, the task's Power Constraint value is displayed. The interaction between the subpanes, the menus, the menu actions for the subpanes, and the subpane's display data is managed by the TaskBrowser object. All of the TaskBrowser's subclass objects must also display selected Task data, and therefore use the TaskBrowser methods and instance variables for doing so.

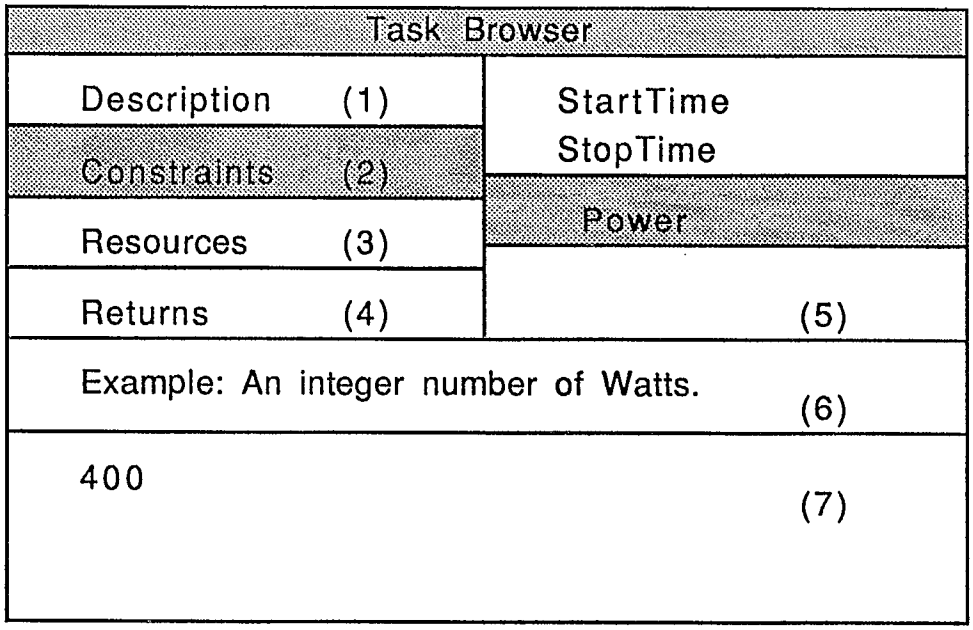

Figure 1. TaskBrowser

Instance variables are used to keep track of when something has been selected, what has been selected, and what is currently displayed. The instance variables *isTaskPicked* and *isItemPicked* indicate whether a task have been selected and whether one of its constraint, resource, or return items has been selected. The instance variables *taskSelected* and *itemSelected* indicate what task and what constraint, resource, or return item has been selected. The instance variable *viewingType* indicates what type of constrain has been selected,e.g. Constraint, Resource, or Return. The final two instance variables, *textDisplay* and *example*, contain the text displayed in the TextDisplay and Example subpanes.

As described previously, the *openOn*: method schedules and opens the TaskBrowser. The method creates the initial window called TopPane, and then adds the seven different subpanes to it: Description, Constraints, Resources, Returns, Items, Example, and Text. When adding a new subpane, three important variables are supplied to the subpanes methods: *name*:, *change*:, and *menu*:. The symbol supplied to the *name*: method defines what method is sent to the TaskBrowser to return the data to display in the subpane. The symbol supplied to the *change*: method defines what method is sent

to the TaskBrowser when an object has been selected in the subpane. The symbol supplied to the *menu*: method defines what method is sent to the TaskBrowser to retrieve the popup menu for the subpane. Different types of subpanes return and receive different objects for display and selection of the subpane. By reviewing the openOn: method, these related subpane methods can be viewed.

Other methods (*addConstraint, addItem, addResource, addReturn, deleteConstraint, deleteItem, deleteResource,* and *deleteReturns*) are sent to the TaskBrowser as a result of a particular subpane's menu selection. These methods either add a new item to a the specific subpane or remove a selected item from a subpane.

One of the object's most important methods is *accept:from:*. This method receives the text that has been entered in the Text Display subpane after the *save* option is selected from the Text Display menu. Depending on the current item type selected, Description, Constraint, Resources, or Return, the corresponding selected item is updated using the passed text string. The string is converted to the internal type of the item. For example, if the Constraint subpane and the Temperature item in the Item subpane have been selected and the save option is chosen, the string in the text pane is converted to an integer and stored as the value of the Temperature constraint. The method *acceptExample:from:* performs the same operation as the Example subpane, however no action is taken with the passed string. It is effectively a nil operation, but must be present.

2.3.8.1 TasksBrowser Class

Inheritance:

Super Class: TaskBrowser
SubClasses: PlanBrowser

Instance Variables:

taskDictionary - A dictionary object that contains Task objects indexed by their names.

Methods:

addNewTask - This methods prompts the user for a task name and adds a new task object to the taskDictionary.

deleteTask - This methods removes the selected task object stored in the taskSelected instance variable from the taskDictionary, and then updates the appropriate browser subpanes.

openOn: - Given a task, this method opens a TaskBrowser window for the passed task. The instance variables of the object are initialized to their default values.

tasks - This method returns an OrderedCollection of the names of the tasks found in the taskDictionary instance variable. The tasks are ordered using their start times.

tasks: - Given a task name, this method sets the taskSelected instance variable to that of the task object in the taskDictionary using the name as an index, sets isTaskPicked to true, and updates the appropriate browser subpanes using the newly selected tasks as its information source.

taskMenu - This method returns a Menu object to be used as the menu for the tasks subpane of the browser.

Discussion:

The TasksBrowser object is a subclass object of the TaskBrowser. It behaves much the same as the TaskBrowser but is used to view a group of Task objects at a single time, not just a single task. Figure 2 provides an example of TasksBrowser. Its appearance is similar to the TaskBrowser, with the addition of a Task subpane (1). This subpane lists the names of tasks that are available for viewing. By selecting a task in the subpane, the user can then select the items he

wishes to view or alter as in the TaskBrowser. The task operations are not permitted unless a task has been selected. In the example below, Task2's Power constraint is displayed.

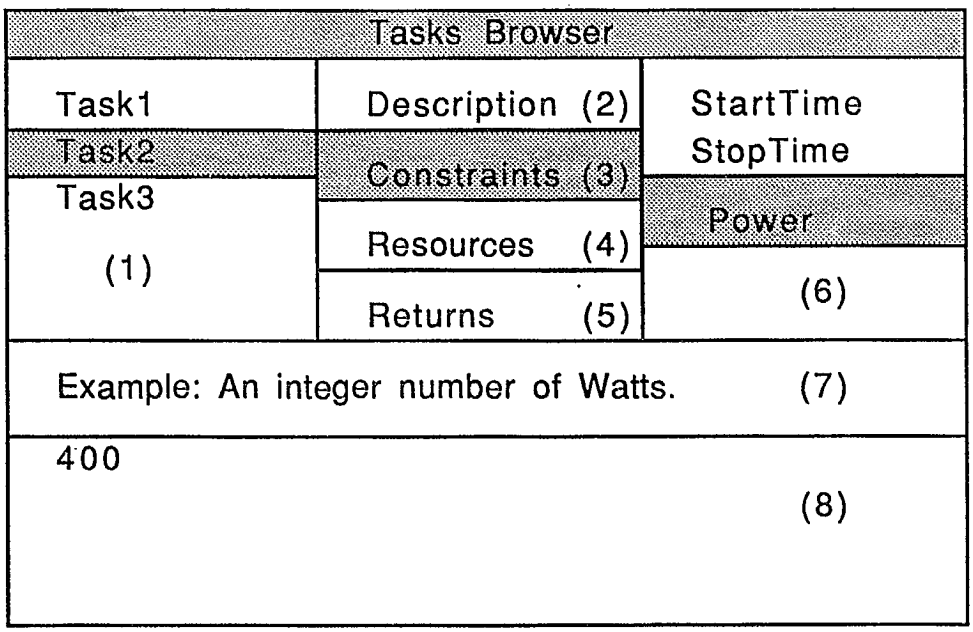

Figure 2. TasksBrowser

A TasksBrowser is opened by using the method *openOn:*. This method receives a dictionary of task objects, and opens the Browser with its eight subpanes: Tasks subpane, Description subpane, Constraints subpane, Resources subpane, Returns subpane, Item subpane, Example subpane, and Text subpane. By reviewing the *openOn:* method, the subpane's related *name: change: and menu:* methods can be viewed. The object's taskDictionary instance variable stores the passed Task dictionary object for the TasksBrowser.

The method *tasksMenu* returns the menu for the Tasks subpane. The available choices are Add and Delete. The methods *addNewTask* and *deleteATask* implement the menu selections. *addNewTask* prompts the user for a task name and then adds a new task object using the supplied name to the task dictionary. The new task is then available for viewing or modification. *deleteATask* removes the

currently selected Task in the Task subpane from the Task dictionary.

2.3.8.1.1 PlanBrowser Class

Inheritance:

Super Class: TasksBrowser
SubClasses: ExecutionBrowser, PlansBrowser

Instance Variables:

currentPlan - This variable contains a Plan object representing the current plan the browser is working with.

planningType - This variable contains a symbol representing the current planning type picked: #Description, #StartTime, #StopTime, #Successes. It indicates if one of these fields is selected in the browser's planItems subpane.

isPlanningTypePicked - A boolean representing if a planning type has been selected.

isPlanPicked - A boolean representing if a plan has been selected for the browser.

failureSelected - A boolean representing if the failure subpane has been selected in the browser.

isFailurePicked - A boolean representing if a failure item has been selected in the failureItems subpane of the browser.

failureTypePicked -This variable contains a symbol representing the current failure type picked: #FailureConstraint, #FailureCount, #FailureDescription, #FailureTask, or #FailureType.

Methods:

accept:from: - Given a text string, this method updates the selected planItem information type of the selectedPlan with the passed string.

deleteAFailure - This methods removes the selected failure object stored in the failureSelected instance variable from the failures dictionary of the selectedPlan, and then updates the appropriate browser subpanes.

failure - This method returns a list of the names of the selectedPlan's current failure objects.

failure: - Given a failure name, this method sets the failureSelected instance variable to that of the failure object in the selected plan's failure dictionary using the name as an index, sets isFailurePicked to true, and updates the appropriate browser subpanes using the newly selected failure as its information source.

failureConstraint: - Given the #failureConstraint: symbol, this method updates the text subpane with the failure constraint for the selectedFailure.

failureCount: - Given the #failureCount: symbol, this method updates the text subpane with the failure count for the selectedFailure.

failureDescription: - Given the #failureDescription: symbol, this method updates the text subpane with the failure description for the selectedFailure.

failureItems - This method returns the symbol #(failureDescription: failureTask: failureConstraint: failureCount: failureType:).

failureItems: Given a symbol as a result of selecting an item from the failureItems subpane, this method performs the method identified by the passed symbol.

failureMenu - This method returns a Menu object to be used as the menu for the failures subpane of the browser.

failureTask: - Given the #failureTask: symbol, this method updates the text subpane with the failure task for the selectedFailure.

failureType: - Given the #failureType: symbol, this method updates the text subpane with the failure type for the selectedFailure.

openOn: - Given a plan, this method opens a PlanBrowser window for the passed plan. The instance variables of the object are initialized to their default values.

planDescription: - Given the symbol #planDescription:, this method displays the selectedPlan's description in the text subpane and updates the appropriate browser subpanes.

planItems - This method returns the symbol #(planDescription: startTime: stopTime: success:).

planItems: Given a symbol as a result of selecting an item from the planItems subpane, this method performs the method identified by the passed symbol.

startTime: - Given the symbol #startTime:, this method displays the selectedPlan's start time in the text subpane and updates the appropriate browser subpanes.

stopTime: - Given the symbol #stopTime:, this method displays the selectedPlan's stop time in the text subpane and updates the appropriate browser subpanes.

success: - Given the symbol #success:, this method displays the selectedPlan's success count in the text subpane and updates the appropriate browser subpanes.

tasks: Given the #Tasks: symbol, the user has selected the tasks subpane, rely on the super class to perform the correct operations and reset the fact the planningType information is no longer the focus of the browser.

Discussion:

A Plan object is composed of a group of plan related data, including: StartTime, StopTime, Description, Success counter, and Failures, and a dictionary of Tasks. The PlanBrowser object is a subclass object of the TasksBrowser. It behaves much the same as the TaskBrowser but is used to view a plan's data including its tasks. Figure 3 provides an example of PlanBrowser. Its appearance is similar to the TasksBrowser, with the addition of PlanItem (1), Failure (2), and FailureItems (3) subpanes. The PlanItem subpane

lists the Plan's data items (Description, StartTime, StopTime, and Success counter) which are available for viewing. By selecting one of these items, the user can view or alter it as in the TasksBrowser. The Failure subpane lists the names of Plan's failures that are available for viewing. By selecting a failure, the user can then select the failure items he wishes to view or alter. The FailureItem subpane lists the Plan's Failure items (Description, Task, Constraint, Type, and Count) which are available for viewing. By selecting an item, it is displayed in the Text subpane permitting the user to view or alter it. The remaining subpanes are identical to the TasksBrowser subpanes and behave in a similar manner.

In the example below, the Plans description is selected and displayed.

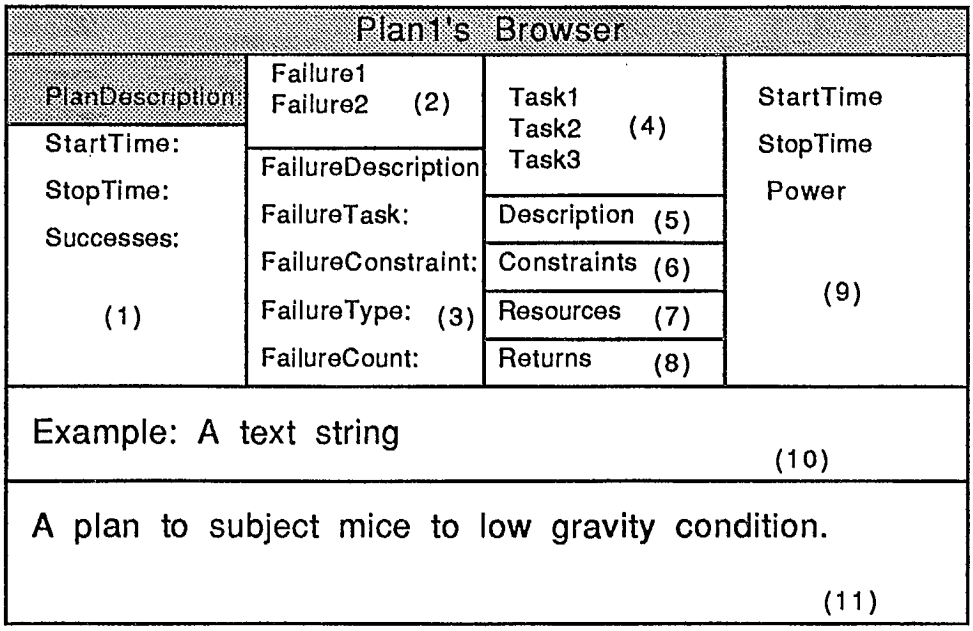

Figure 3. PlanBrowser

A PlanBrowser is opened by using the method *openOn:*. This method opens the Browser on the Plan with its eleven subpanes: PlanItems subpane, Failure subpane, FailureItem subpane, Tasks subpane, Description subpane, Constraints subpane, Resources subpane, Returns subpane, Item subpane, Example subpane, and Text

subpane. By reviewing the *openOn:* method, the subpane's related *name: change: and menu:* methods can be viewed. The object's currentPlan instance variable stores the passed Plan object for the PlanBrowser which is used later to determine the updating procedures.

Additional instance variables are used to keep track of when something has been selected and what has been selected. The instance variables *isFailureSelected* and *failureSelected* indicate whether a plan failure has been selected and what plan failure was selected. The instance variable failureTypePicked indicates which one of the plan's failure items has been selected. The instance variable *planningType* indicate what plan item has been selected.

The method *accept:from:* is enhanced to be able to update the plan's data when a menu *save* option is selected; this is based on what failure, task, or plan items are currently selected. This is done by checking the current planning item selected and if none is the *accept:from:* method of the TasksBrowser is used.

The method *failureMenu* returns the menu for the Failure subpane. The available choice is Delete. The method *deleteAFailure* implements this selection. *deleteAFailure* removes the currently selected Failure in the Failure subpane from the Plan's Failure dictionary.

2.3.8.1.1.1 PlansBrowser Class

Inheritance:

Super Class: PlanBrowser
SubClasses: None

Instance Variables:

planDictionary - A dictionary object that contains Plan objects indexed by their names.

Methods:

addNewPlan - This methods prompts the user for a Plan name and adds a new Plan object to the PlanDictionary.

deletePlan - This methods removes the selected plan object stored in the planSelected instance variable from the planDictionary, and then updates the appropriate browser subpanes.

openOn: - Given a dictionary of plan objects, this method opens a PlansBrowser window for the passed plans. The instance variables of the object are initialized to their default values.

plans - This method returns an OrderedCollection of the names of the plans found in the planDictionary instance variable. The plans are ordered using their start times.

plans: - Given a plan name, this method sets the planSelected instance variable to that of the plan object in the planDictionary using the name as an index, sets isPlanPicked to true, and updates the appropriate browser subpanes using the newly selected plan as its information source.

planMenu - This method returns a Menu object to be used as the menu for the plans subpane of the browser.

Discussion:

The PlansBrowser object is a subclass object of the PlanBrowser. It behaves much the same as the PlanBrowser but is used to view a collection of Plan objects at a single time, not just a single Plan. Figure 4 provides an example of PlansBrowser. Its appearance is similar to the PlanBrowser, with the addition of a Plan subpane (1). This subpane lists the names of plans that are available for viewing. By selecting a plan in the subpane, the user can then select the items he wishes to view or alter as in the PlanBrowser. These operations are not permitted unless a plan has been selected.

In the example, the plan description for Plan1 has been selected and is displayed.

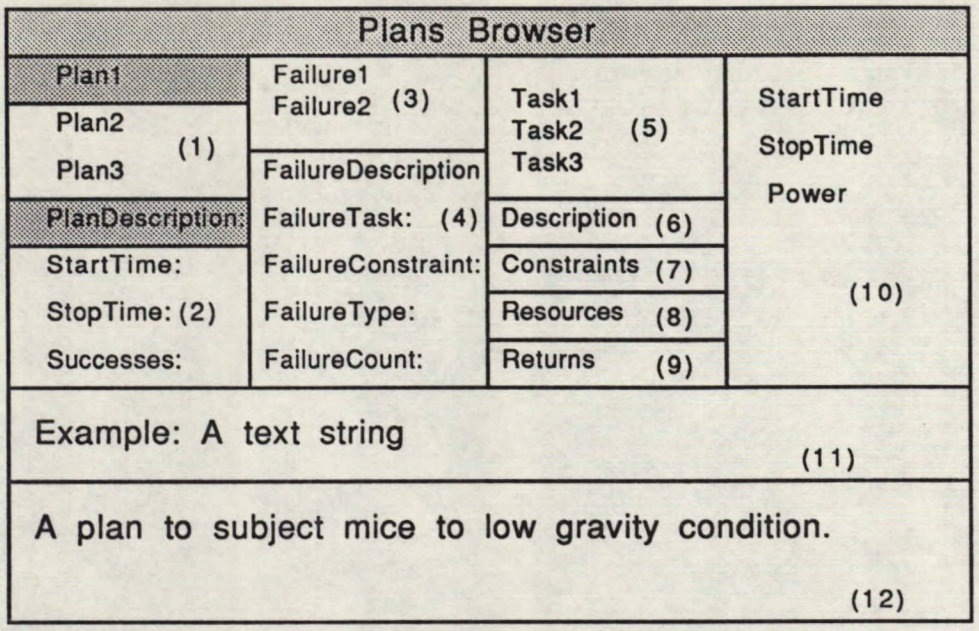

Figure 4. PlansBrowser

A PlansBrowser is opened by using the method *openOn:*. This method receives a dictionary of plan objects and opens the Browser with its twelve subpanes: Plan subpane, PlanItems subpane, Failure subpane, FailureItem subpane, Tasks subpane, Description subpane, Constraints subpane, Resources subpane, Returns subpane, Item subpane, Example subpane, and Text subpane. By reviewing the *openOn:* method, the subpane's related *name: change: and menu:* methods can be viewed. The object's *planDictionary* instance variable stores the passed Plan dictionary object for the PlansBrowser.

The method *plansMenu* returns the menu for the Plans subpane. The available choices are Add and Delete. The methods *addNewPlan* and *deleteAPlan* implement these selections. *addNewPlan* prompts the user for a Plan name and then adds a new plan object using the supplied name to the plan dictionary. This new Plan is then available for viewing or modification. *deleteAPlan* removes the currently selected Plan in the Plans subpane from the Plan dictionary.

2.3.8.1.1.2 ExecutionBrowser Class

Inheritance:

Super Class: PlanBrowser
SubClasses: None

Instance Variables:

remainingTasksToExectute - An Ordered Collection of Task objects, representing the remaining tasks to execute in the browser.

currentPlanner - This variable holds the Planner object who initiated the open of the browser.

currentTask - A task object representing the current task executing.

evaluation - An evaluation object used for storing execution information for later plan evaluation.

Methods:

currentTaskExecuting - This method orders the remaining tasks to execute and returns the task object that is the next task to execute.

executionMenu - This method returns a Menu object to be used as the menu for the taskExecuting subpane of the browser.

exit - This method performs no action - it executes an exit menu selection.

failTask - This method takes the currentTask, prompts the user for a failed constraint and constraint type, and initiates a replanning session for the currentTask.

goToNextTask - This method sets the currentTask to the next task in the remainingTasksToExecute instance variable and updates the appropriate browser subpanes.

openOn:for:evaluation - Given a plan, Planner, and Evaluator object, this method opens a ExecutionBrowser window for

the passed plan using the Planner for information display and adding execution information to the Evaluator object. The instance variables of the object are initialized to their default values.

taskExecuting: Given a symbol, this methods updates the currentTaskExecuting subpane of the browser with the name of the currentTask.

Discussion:

The ExecutionBrowser object is a subclass object of the PlanBrowser. It behaves much the same as the PlanBrowser, but is used to simulate the failure and execution of the tasks in the plan the ExecutionBrowser is opened on. Figure 5 provides an example of ExecutionBrowser. Its appearance is similar to the PlanBrowser, with the addition of a CurrentTask subpane (10). This subpane displays the current task in the plan that can be executed or failed. This is done by opening the menu for the subpane and selecting either the Next or Fail option. All other subpanes operate as in the PlanBrowser.

In the example below the plan's plan description has been selected and is displayed.

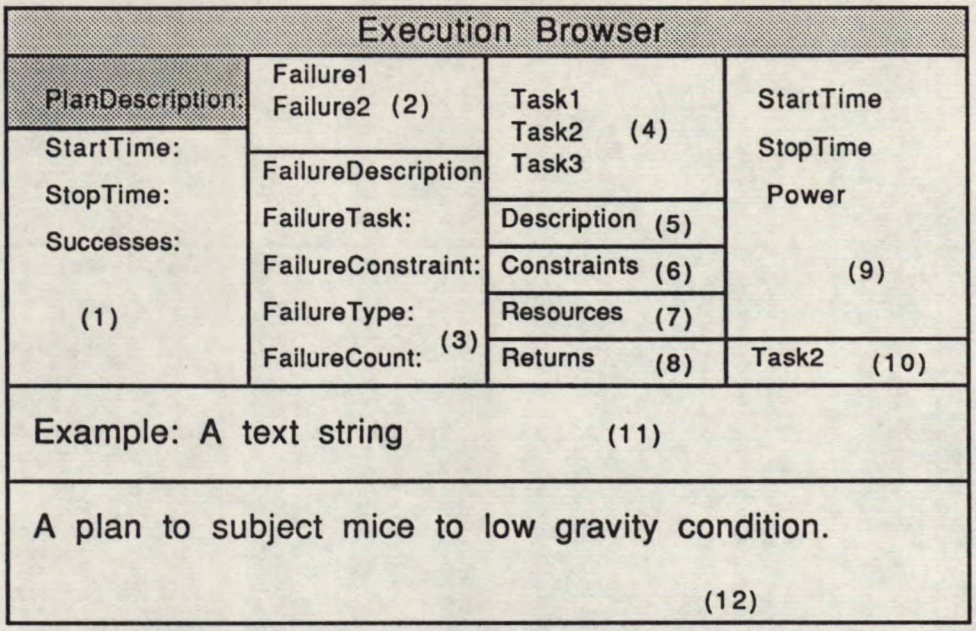

Figure 5. ExecutionBrowser

A ExecutionBrowser is opened by using the method *openOn:*. This method receives a plan object and opens the Browser with its thirteen subpanes: Plan subpane, PlanItems subpane, Failure subpane, FailureItem subpane, Tasks subpane, Description subpane, Constraints subpane, Resources subpane, Returns subpane, Item subpane, CurrentTask, Example subpane, and Text subpane. By reviewing the *openOn:* method, the subpane's related *name: change: and menu:* methods can be viewed. The object's currentPlan instance variable stores the passed Plan object for the ExecutionBrowser. The object's *remainingTasksToExecute* instance variable stores the Plan's collection of tasks that are left to execute. The object's *currentTask* instance variable stores the Plan's current executing task.

The method *executionMenu* returns the menu for the CurrentTask subpane. The available choices are Next and Fail. The methods *goToNextTask* and *failTask* implement these selections. *goToNextTask* sets the *currentTask* instance variable to the next task in *remainingTasksToExecute* instance variable collection. That task is also removed from the remainingTasksToExecute instance variable collection. The new current task is displayed in the CurrentTask subpane. The previous task is considered to of executed successfully

when this action is done. To fail a task, the Fail option is chosen. *failTask* prompts the user for the constraint type (Constraint, Resource, or Return) and the constraint item (e.g. Power, StartTime, etc), and invokes the replanning mechanism using the failed task and user entered failed constraint. The replanning actions are noted in the Planner Text subpane (5). After replanning, the remainingTasksToExecute in the replanned plan are reordered and then made available for execution or failing. After all tasks have been executed, the text "Execution Finished" is displayed in the CurrentTask subpane. The plan is ready for execution.

2.3.9 Planner Class

Inheritance:

Super Class: Object
SubClasses: None

Instance Variables:

plannerTextPane - A TextPane object storing the TextPane object of the window.

taskLibrary - A Dictionary object containing tasks indexed by their names.

boundingBox - A rectangle representing the dimensions of the window.

planLibrary - An instance variable storing the PlanLibrary display object for the window.

evaluation - An instance variable storing the Evaluator object for the window.

knowledgeBase - An instance variable storing the KnowledgeBase display object for the window.

plan - An instance variable storing the Plan display object for the window.

explanationText - A text string that holds the text information for the text subpane of the window.

transcriptCounter - An integer that represents the entry number in the text subpane of the window.

Methods:

acceptExplain:From: - As a result of selecting the save option from the text subpane this method is invoked. It does nothing.

addExplanation: - Given a text string, this method adds the string to the explanationText instance variable and displays the text in the text subpane of the window.

appendToExplanation: - Given a text string, this method adds the string to the explanationText instance variable.

clearPlan - This method sets the ThePlan global variable to nil.

drawConnectionsOn: - Given a form, this methods draws the links between the display objects on the form.

drawEnvironment: - Given a bounding rectangle, this method draws the window's display objects in the window and draws the links.

editPlanningTasks - This method opens a TaskBrowser on the planning tasks defined in the global PlanningTasks dictionary variable.

editPlans - This method opens a PlansBrowser on the plans defined in the global PlanLibrary dictionary variable.

environmentChange: - Given a Point object, this method decides what display object has been selected in the window, pops up the menu for the display object, and performs the choice.

environmentMenu - This method returns a Menu object for when the user selects the background menu for the window.

evaluatePlan - This method performs the evaluation of the plan developed by the planner.

executePlan - This method opens an Execution Browser for the plan developed by the planner.

exit - This method does nothing. It is executed when the exit option is taken on any window menu.

explain - This method returns the explanationText instance variable object.

generate - This method initializes and stores a new Evaluator object for the window, takes the PlanningTasks and locates a Plan in the PlanLibrary that closely matches the PlanningTasks, adds the missing task and removes the extra ones, unifies the PlanningTasks with the tasks of the located plan, verifies the plan, and gets the plan ready for execution. The generated plan is stored in the ThePlan global variable.

initKnowledgeBase:on: - Given a Point and a Form object, this method draws the KnowledgeBase icon at the specified location on the passed form.

initPlan:on: - Given a Point and a Form object, this method draws the Plan icon at the specified location on the passed form.

initPlanLibrary:on: - Given a Point and a Form object, this method draws the PlanLibrary icon at the specified location on the passed form.

initTaskLibrary:on: - Given a Point and a Form object, this method draws the TaskLibrary icon at the specified location on the passed form.

knowledgeBaseMenu - This method returns a Menu object for the menu to use when the KnowledgeBase icon is selected in the window.

open - This method opens an Planner window. The instance variables of the object are initialized to their default values.

openKB - This method opens a LogicBrowser.

planLibraryMenu - This method returns a Menu object for the menu to use when the PlanLibrary icon is selected in the window.

planMenu - This method returns a Menu object for the menu to use when the Plan icon is selected in the window.

reframe: - Given a new Rectangle object, this method handles the reframing of the Planner window to the new rectangle location.

reframeKnowledgeBase: - Given a new Rectangle object, this method handles the reframing of the KnowledgeBase icon to the new rectangle location.

reframePlan: - Given a new Rectangle object, this method handles the reframing of the Plan icon to the new rectangle location.

reframePlanLibrary: - Given a new Rectangle object, this method handles the reframing of the PlanLibrary icon to the new rectangle location.

reframeTaskLibrary: - Given a new Rectangle object, this method handles the reframing of the TaskLibrary icon to the new rectangle location.

taskLibraryMenu - This method returns a Menu object for the menu to use when the TaskLibrary icon is selected in the window.

viewPlan - This method opens an Plan Browser for the plan developed by the planner.

Discussion:

The Planner object defines a CBPS environment window in which all planning activities can be accessed. These activities include generating and modifying the tasks to be planned for, the plans in the library, and the action rules in the Knowledge Base, and the generation, viewing, executing, and evaluation of the plan.

The method *open* opens and schedule the Planner and its two subpanes: Environment subpane (1,2,3,4) and Explanation subpane (5). The Environment subpane displays the Planner display objects; each which may be selected to present a menu of available actions for the object. The Explanation subpane implements a text transcript subpane that the Planner uses to display ongoing planning

information results. Figure 6 provides an example of the CBPS window.

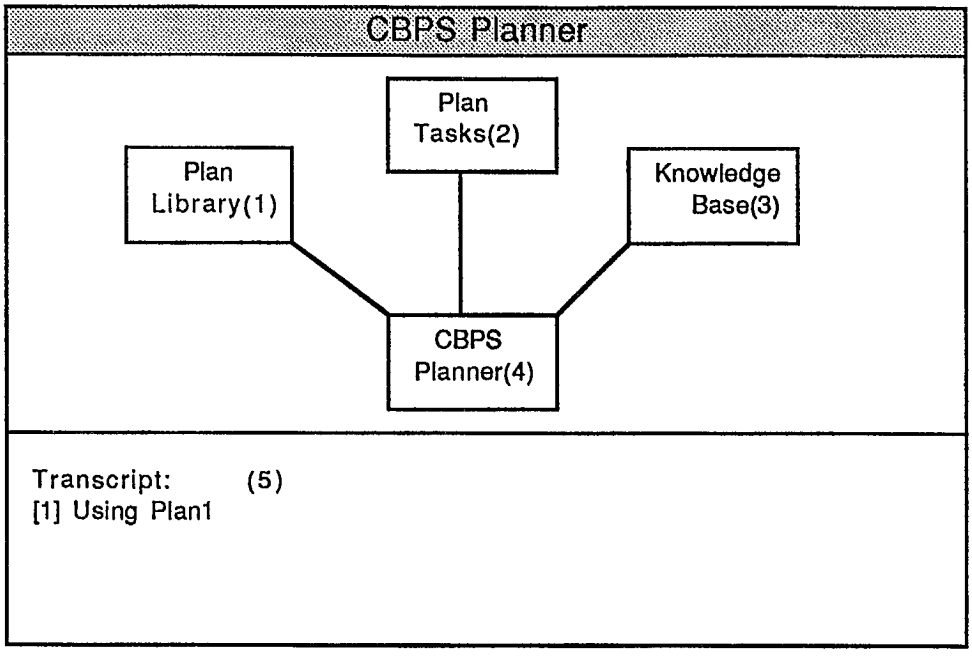

Figure 6. CBPS Window

The method *drawEnvironment* generates the display for the Environment subpane. This method displays the four Planner display objects (PlanLibrary, KnowledgeBase, Plan, and TaskLibrary) by sending the methods *initPlanLibrary*, *initKnowledgeBase*, *initPlan*, and *initTaskLibrary* to itself, with the locations in the Environment subpane where they should be displayed. Each one of these methods draws the graphic object in the display. The method *drawConnectionsOn*: displays the lines that connect the Display objects. The instance variables *taskLibrary*, *planLibrary*, *knowledgeBase*, and *plan*, are used to store the instances of the Planner display objects, reference by other methods that relay on the location of the displayed object.

The method *explain* generates the text display for the Explanation subpane. The physical text displayed is stored in the instance variable explanationText, which is initialized in the *open* method. The

method *addExplanation*: appends a text string to the *explanationText* instance variable and updates the Explanation subpane to display the text. Each update starts a new line in the explanationText and is preceded with a [n], where n is an incrementing integer number. This number is stored in the instance variable *transcriptCounter*, which is incremented after each update by the *addExplanation*: method.

A key method for the Environment subpane is *environmentChange*:. This methods receives the point that has been selected in the Environment subpane with the left mouse button. The method decides what Planner display object has been selected and pops up the menu associated with it. These menus are returned by the methods *taskLibraryMenu*, *planLibraryMenu*, *planMenu*, and *knowledgeBaseMenu*. The choices in each menu correspond to actual Planner methods. Therefore, the actual choice can be, and is, sent to the Planner object using the instruction **self perform: choice**. The methods for the KnowledgeBase display object are *openKb* and *exit*. The methods for the PlanLibrary display object are *editPlans* and *exit*. The methods for the TaskLibrary display object are *editPlanningTasks* and *exit*. The methods for the Plan display object are *generate*, *viewPlan*, *clearPlan*, *executePlan*, *evaluatePlan*, and *exit*. The common *exit* method does nothing. The method *openKb* opens the KnowledgeBase Browser for action rule access. The method *editPlans* opens the Plan's Browser for Plan Library access. The method *editPlanningTasks* opens the Tasks Browser for planning Tasks access. The other planning methods are discussed in the following text.

The method *generate*, using the current planning tasks found in the PlanningTasks system object and the library of plans found in the PlanLibrary system object, generates a plan to execute. The new plan is stored in the *ThePlan* system object. It first locates a plan in the Plan Library that best matches the current planning tasks. The missing tasks in the plan are added and the extra ones removed. The plan tasks and planning tasks are unified, ordered, and verified, as described by the Plan object, and made ready for execution.

The method *viewPlan* opens a Plan Browser for access to the generated plan. The method *clearPlan* sets the ThePlan object to nil. The method *executePlan* opens an Execution Browser on the generated plan for simulated execution of it.

When ever a new plan is generated (using the *generate* method), a new instance of an Evaluator object is created and stored in the evaluation instance variable. Throughout the Planner, the Evaluator object is sent methods (see Evaluator object definition) to record planning, execution, and replanning information. After the plan has executed, the method *evaluatePlan* performs the plan evaluation using the Evaluator object.

2.3.10 Other Classes

Other classes provided by source files supplied on the Smalltalk Tutorial Disk were used. Filing in the source on the file Fredrwng.st added all class descriptions required for form editing. A description of the classes can be found in the file Fredrwng.doc on the same disk. Filing in the source on the file prolog.st adds all class descriptions required for the Logic Browser. A description of the classes can be found in the file prolog.doc on the Tutorial disk.

2.4 Predefined System Objects

Predefined system objects are instances of particular objects that are predefined and used by the CBPS. These objects or used directly by the CBPS, in the case of the PlanningTasks and PlanLibrary objects, or used to generate other objects used by the CBPS, as in the case of the LogicBrowser object which is used to store the action rules. The following is a short description of each of these types of objects.

2.4.1 PlanningTasks Object

The PlanningTasks object is defined as a dictionary object. It stores the planning tasks of the CBPS planner. The TasksBrowser is opened on the PlanningTasks in the CBPS planner to add and remove the various Task information.

2.4.2 PlanLibrary Object

The PlanLibrary object is defined as a dictionary object. It stores the current library of plans that are used by the CBPS planner to determine a plan that meets the current input task requirements. The PlansBrowser is opened on the PlanLibrary in the CBPS planner to add and remove the various Plan information.

2.4.3 ThePlan Object

The ThePlan object is defined as a Plan object. Its stores the current Plan to executed by the CBPS planner.

2.4.4 LogicBrowser Object

The LogicBrowser is used to store the Knowledge Base action rules for the planner. A basic description of the LogicBrowser can be found on the Smalltalk Tutorial disk (Prolog.doc). The important class description in the LogicBrowser is the PlannerExceptionRules. These are prolog clauses that return a replanning action based on a failed task and constraint. These are used in replanning to determine the appropriate action to take with a failure. Please refer to the KnowledgeBase section in the Users Guide for a detailed description.

2.4.5 FreeDrawing (PictureDictionary) Object

The FreeDrawing object implements a form editor for Smalltalk. A basic description of the FreeDrawing object can be found on the Smalltalk Tutorial disk (FreeDrawing.doc). It is used to generate four form icons: the KnowledgBaseIcon, the PlanIcon, the PlanLibraryIcon,

and the TasksIcon. They are stored in the FreeDrawing's class instance variable PictureDictionary. The dictionary stores each icon's form , and is referenced by the icon's text name. These icons are retrieved and displayed in the Planner.

2.5 Summary

The design is compact and centered around three objects: a plan, a task, and a failure. To understand the design, one must first understand the operations and behaviors of these object. The control flow centers around the Smalltalk windowing paradigm. By understanding it, one will get a better feeling for the control flow of the implementation. The main control point is the Planner object window. To view methods that used the described methods, one can use the *senders* menu choice of the Class Hierarchy Browser's Methods subpane. The next section provides some insights into the design flow by describing the CBPS operation in the form of a Operation Description.

By using this section, along with the design documents and the code itself, one is provided a complete information package to understand the concepts of the CBPS and its design.

3 OPERATION DESCRIPTION

3.1 Introduction

The purpose of the section is to describe the operation of the planning system from a operations point of view. The reader is assumed to be familiar with Section 2 and the design and implementation of the CBPS before reading this section.

Preparation for planning, and planning itself, involve several activities:

1. A list of planning tasks must be entered to identify what tasks the planner is planning for.

2. One must develop a library of plans that will be used as past cases for plan generation.

3. A Knowledge Base of replanning rules must be developed in order for the planner to do replanning during plan verification and execution replanning.

4. Finally, a plan can be generated, view, executed, evaluated, and cleared with the above information.

This section describes the user interface, user interaction, and operation of the CBPS to perform the previously outlined operations. The main control point of the CBPS is the Planner Window. This window permits access to various CBPS browsers (Tasks, Plan, PlanLibrary, KnowledgeBase, and Execution) and the planning operations (generations, execution, and evaluation). We begin with

the operation of the planner window, and lead the reader through the process of planning, showing how and when the above activities are performed.

3.2 Environment

To begin a planning session one must open an CBPS window. To do this the expression **Planner new open** must be evaluated in the Smalltalk System Transcript window. This is done by highlighting the expression and selecting the **doIt** option from the System Transcript menu. After evaluating the expression, a CBPS window, as in Figure 6 in Section 2.3.9, is opened.

The CBPS window has two subpanes, an Explanation Transcript subpane (5) and an Environment subpane (1,2,3,4). As planning operations are performed, the Explanation Transcript subpane will be updated with text planning information. Each reference begins with a new number. This subpane is implemented as a Smalltalk text subpane and behaves as such. The Environment subpane displays four objects: a Tasks object, a Plan Library object, a Knowledge Base object, and a Plan object. By moving the cursor to the object and pressing the left mouse button, the corresponding menu for that object is produced. All of these menus have an *exit* selection, which when selected will result in the menu being closed but nothing else. All other menu options perform an action. The following sections describe the resulting actions of selecting the other menu options, and why and when they should be selected.

3.3 Generation, Viewing, or Alteration Planning Tasks

When one wishes to create, view, or alter the planning tasks , the Tasks Display object should be selected in the planner window and the **Edit Task** option chosen. This will open a Tasks Browser for the currently defined tasks. This browser allows for the creation, viewing, or alteration of planning tasks. The task definitions are global and are only removed when deleted using the Tasks Browser.

This mean the CBPS window can be closed and later reopened, with the latest task definitions still available. The TasksBrowser section describes how tasks and a task's attributes are added and deleted from the planning tasks.

3.3.1 Tasks Browser

An example of the Tasks Browser can be seen in figure 2. Subpane (1) displays a list of the currently defined tasks. To add a new task, select the popup the menu for the subpane and choose the **Add Task** option. One is then prompted for a task name. Enter a text string making sure the name is unique. Duplicate task names are not allowed. The new task is displayed in the currently defined task list To delete a Task, select the task, select the name subpane (1), popup the menu for the subpane, and select the **Delete Task** option. The task will be removed from the currently available task list.

To select a task for further viewing, select the task name in the subpane (1). The selected task is known as the currently selected task and all other operation are now performed on it.

By selecting the Description subpane, subpane (3), the description for the current task is displayed in the Text subpane, subpane (8). To change the description, change the text in the Text subpane, which behave like a Smalltalk Text subpane, and select the **save** option from the Text subpane menu. This will update the current Task's description.

To display the current task's constraints, resources, or return resources, select either subpane (3), (4), or (5). Subpane (6) will list the currently defined items for the constraints, resources, or return resources. By selecting one of the items in subpane (6), the corresponding value of that item is displayed in the Text subpane. To update the value, perform the same operation as in updating the task's description. Subpane (7) displays an example of what type of entry is expected in the Text subpane for the currently selected item.

To add a constraint, a resource, or a return resource, first select the item type (either subpane (3) (4) or (5)). Next, popup the menu for subpane (6) and select the **Add Item** option. A list of available entries for the item type is present, choose the constraint, resource, or return resource item type desired. Finally, enter the item name when prompted. The name should not be a duplicate of other item names. The item is now available for selecting in the subpane (6) and updating in the Text subpane.

To delete an item, select the item name in subpane (6), popup the menu for the subpane, and select the **Delete Item** option. The item will be removed from the current task's constraints, resources, or return resources list.

To close the Task Browser, popup the Window menu and select the **close** option. The tasks currently defined are the ones that the CBPS will plan for during plan generation.

3.4 Generation, Viewing, or Alteration of the Plan Library

Before any planning can be done, a library of Plans must be generated to be used as cases for plan generation. Also, periodically the plan library should be maintained to insure its integrity. This involves identifying cases that are similar and generalizing them into a single case covering two or more situations. It could also include identifying cases that are no longer applicable and removing them. The Plans Library Browser section describes how plans and plan attributes are added and deleted from the library.

To create, view, or alter the plans in the Plan Library, the Plan Library Display object should be selected and the **Edit Library** option chosen. This will open a Plans Browser for the currently defined library plans. The plan definitions are global and are only removed when deleted using the Plans Browser. This means the CBPS

window can be closed and later reopened, and the latest plan definitions will still be available.

3.4.1 Plan Library Browser

An example of the Plans Browser can be seen in figure 4. Subpane (1) displays a list of the currently defined plans. To add a new plan, popup the menu for the subpane and select the **Add Plan** option. One will then be prompted for a plan name. Enter a text string making sure the name is unique. Duplicate plan names are not allowed. The new plan is displayed in the currently available plan list. To delete a plan, select the plan name from subpane (1), popup the menu for the subpane, and select the **Delete Plan** option. The plan will be removed from the currently available plan list.

To select a plan for further viewing, select the plan name in the subpane (1). The selected plan is known as the currently selected plan and all other operation are performed on it.

By selecting Description, StartTime, StopTime or Success from subpane (2), the corresponding value for the item is displayed in the Text subpane, subpane (12). To change the value of the item, change the text in the Text subpane and select the **save** option from the Text subpane menu. This subpane behaves like a Smalltalk Text subpane. This will update the current plan's selected information.

Subpane (3) displays the failures defined for the current plan. To select a failure, select one of the failure names in the subpane. To view the failure's description, task, constraint, type, or success count, select one of these items from subpane (4). The corresponding value for the failure item is displayed in the Text subpane, subpane (12). You are not permitted to alter any of these items. Changing the text in the Text subpane and selecting the **save** option from the Text subpane menu has no effect.

One is permitted to delete a failure. To do this, select the failure name from subpane (3), popup the menu for the subpane, and select the **Delete Failure** option. The failure will be removed from the currently available plan failure list.

The remaining subpanes appear as, and are, a Tasks Browser. Subpane (5) displays the current plan's task. Tasks can be added and deleted in a similar manner to the Tasks Browser. By selecting a task, and the Description, Constraints, Resources, or Return Resources subpane, the corresponding item is displayed in subpane (10). By selecting an item in subpane (10) the corresponding value is displayed in the Text subpane. The addition, deletion and update of the selected item is handled as in the Tasks Browser.

To close the Plans Browser, popup the Window menu and select the **close** option. The plans currently defined are the ones that the CBPS will use during plan generation.

3.5 Generation, Viewing, or Alteration of the Knowledge Base

When the user wishes to create, view, or alter the Knowledge Base rules, the Knowledge Base Display object should be selected and the **Edit KB** option chosen. This will open a Logic Browser for the currently defined action rules. The action rule definitions are global and are only removed when deleted using the Logic Browser. This means the CBPS window can be closed and later reopened, and the latest action rule definitions will still be available. The Knowledge Base Browser section describes how action rules are added and deleted.

Action rules are used to decide what methods should be executed in the event of a task's constraint failing during the plan's execution or verification. By reviewing the design and implementation documents, the exact use and operation of these objects is described.

3.5.1 Knowledge Base Browser

An example of the Logic Browser can be seen in Figure 7. It is used to create the action rules used in determining the replanning action when a plan fails. Subpane (1) provides a list of the available rule sets. Depending on the Smalltalk image, there could be several different sets. The one of interest to the CBPS is the **PlannerExceptionRules** set. Selecting this item from the subpane will display **action:** in subpane (2). By selecting the **action:** item in this subpane, the current action rules are displayed in subpane (3). In our example, the PlannerExceptionRules action rules are displayed.

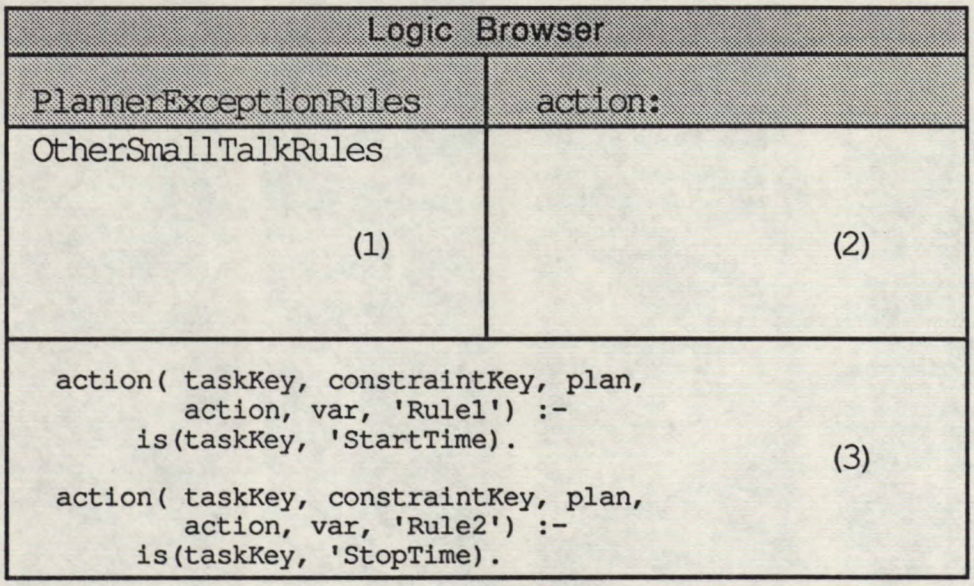

Figure 7. Logic Browser

Subpane (3) is a text display of the action rules. It permits the creation, modification, and deletion of the action rules. By selecting the popup menu for subpane (3) and selecting the save option, the action rules in the subpane are saved as the current action rules.

The action rules are composed of prolog predicates of the form:

```
action (taskKey, constraintKey, plan, action, var, ruleId) :-

    prolog expressions
            "
            "
            .
```

The action rules must start with **action** as the class name. Incoming variables include the taskKey, the constraintKey, and the plan. The *taskKey* is the string name of the task that failed during planning. The *constraintKey* is the string name of the constraint that failure during planning. The *plan* variable contains the current plan object. Return variables include *action* , *var*, and *ruleID*. *var* is an output parameter that can be supplied with any value. *ruleId* should be unified with the rule identifier for the rule. *action* should be unified with the method's symbol name that is executed for replanning - this is the action of the rule.

The action names that can be used to unify with the action variable included: **#findAPlace:with:, #takeNoAction:with:, and #moveToTheEnd:with:.** These methods are called with the **var** instance variable of the action rule as the first parameter at the method. The **#findAPlace:with:** method locates the next available slot in the plan where the failed task's constraints will be satisfied. The **#takeNoAction:with:** methods does no replanning. The method **#moveToTheEnd:with:** moves the failed task to the end of the plan. A further description is located in the System Implementation description.

For a description of Smalltalk's prolog implementation, consult the Prolog.doc file on the Smalltalk tutorial disk.

3.6 Plan Operations

The first three display objects (Task Display Object, Task Library Display Object, and the Knowledge Base Display Object) provide the capability for the user to generate the initial information for planning; the tasks, action rules, and library of plans are now available for planning. The next planning activities are Plan Generation, new Plan Review, Plan Execution, Plan Evaluation, and Plan Clearing. By selecting the Plan Display object, a menu is provided allowing one to select anyone of these activities. The following sections describe that actions taken as a result of making the different menu selections.

3.6.1 Plan Generation

By selecting the **Generate** option from the Plan Display Object's menu selection, the plan generation process is invoked. This option takes the tasks that were entered in the Tasks Browser, locates a base Plan in the Plans Browser, modifies it to meet the current task requirements, verifies the plan, and does any replanning required to verify the plan.

While the generation process is proceeding, Plan generation information is displayed in subpane (2) of the CBPS window. This information includes: what plan was selected from the Plan library to be the base plan, the number of task added to the plan to meet the current task requirements, the number of unrequired tasks removed from the plan, the number of failures the plan has had, what tasks verified in the plan, what tasks failed verification, what failure information was used in replanning, what Action rules were used in the replanning, and whether or not the plan is ready for execution. This information is for the users benefit, and can be used as a source of information to help modify planning tasks or plans in the library for future or current planning.

The verification process will only attempt to verify the plan twice. If it fails again, the user must review the plan generation information and decide what action(s) should be taken with the planning tasks or the chosen base plan. After the plan has been generated and verified the next step is to review the constructed plan.

3.6.2 Plan Reviewer

After the plan has been generated, it can be reviewed by selecting the **View** option from the Plan Display Object's menu selection. This option opens a Plan Browser for the generated Plan. The browser, seen in Figure 3, operates in a similar manner as the Plans Browser, used to view the plans in the Plan Library. The only difference is the first subpane is missing. In the Plans Browser, this subpane was used to select the plan to consider for manipulation or viewing. The Plan Browser is opened for a specific plan, therefor this subpane is not required.

The main purpose of the Browser is to view the generated plan. Although the plan can be altered as in the Plans Browser, this is not advisable. The current version of the plan has been verified and is ready for execution. The Plan will not be reverified if changes are made, it can only be executed.

After the plan has be reviewed, it is ready for execution. Note, the plan can be executed even though it has not been reviewed.

3.6.3 Plan Execution

After the plan has been generated, it can be executed by selecting the **Execute** option from the Plan Display Object's menu selection. This option opens a Execution Browser for the generated Plan. The browser operates in a similar manner as the Plan Browser, used to view the recently generated plan. The only difference is the addition of a new subpane (10), shown in Figure 5.

The new subpane is used to simulate the execution of the plan. The subpane displays the current task of the plan that is ready for execution. By selecting the popup menu for the subpane, the choices **Next** and **Fail** are made available. By choosing the **Next** option, the current task is assumed to of executed successfully, and the next task in the plan is made ready to executed. The next task's name is now displayed in the subpane. By choosing the **Fail** option, the current task is assumed to have failed. One is then presented with a list of the task failures types (Constraints, Resources, and Returns). Select the failure type one desires. After this, one is presented a list of the current items for that failure type. For example, in the Constraint type case these selections could be StartTime, StopTime, Temperature, etc. Select on of the failure items. After this, replanning is done using the failed task and entered failed constraint as the failure information.

Replanning consists of the same activities encounter in plan generation when the plan experiences as failure in plan verification. Replanning information is also displayed in the Transcript subpane of the CBPS window as in plan generation. After the plan has been replanned and reverified, the task execution subpane displays the next task in the plan to execute, and the Plan Browser displays the new plan. Most likely the order of the remaining planning tasks will have changed. When replanning, only the tasks not executed, including the failed task, are replanned. The tasks that have already executed are considered to be alright. The tasks in the Task subpane display the order in which the tasks are to be executed. Due to replanning, this order may change. As in plan generation, only two passes of plan reverification are done. If the plan cannot be verified after two attempts, replanning finishes and one must decide, using the planning information, what operations are required to be done on the planning tasks or the base plan used.

After all tasks have been executed, the message **Plan Ready For Evaluation** will be displayed in the CBPS Transcript window and

Execution Completed is displayed in the Current Task subpane. The plan is now ready for evaluation.

The main purpose of this Browser is to execute the generated plan. Although the plan can be altered as in the Plan Browser, this is not advisable. The current version of the plan has been verified and is ready for execution. The Plan will not be reverified if changes are made, it will only executed from its current execution point.

3.6.4 Plan Evaluation

After the plan has been executed successfully (all task executed), it can be evaluated by selecting the **Evaluate** option from the Plan Display Object's menu selection. This option decides what should be done with the new plan

When ever the plan generation mechanism is invoked, the evaluation information is cleared. New information is added during plan generation, execution, failure, and replanning. This information is then used to determine what is to be done with the executed plan. The for possible actions taken with a plan are:

1) Add the plan under a new name to the Plan Library.
2) Update the Plan Library's plan the current plan is based on with its execution information.
3) Remove the Plan Library's plan the current plan is based on due to excessive errors.
4) Take no action with the plan

The exact logic for the design can be altered or viewed in the method *evaluate* defined for the Evaluator object. To fully understand the context of the logic, review the Evaluator object in the System Implementation description.

3.6.5 Plan Clearing

After the plan has been evaluated, a new planning session can be started. The previously created plan should be erased before starting a new session. This is done by selecting the **Clear** option from the Plan Display Object's menu selection. This option clears the plan and initializes the system for a new planning session.

3.7 Summary

This section has provided a discussion of how to operate the CBPS. It along with the previous section provide the user with an understanding of the design and operation of the CBPS. Each section is designed to compliment the other to provide a useful description of the CBPS.

4 SYSTEM EXTENSIONS AND ENHANCEMENTS

4.1 Introduction

In this section we outline selected areas of the CBPS that could be extended and enhanced, or new areas that could be added to increase the overall power, performance, and efficiency of the planner. The implementation exercise helped to identify these areas by locating sections that where more complicated than expected, and ones that need to be user or application configureable, e.g. the evaluation action rules. Other areas where identified by reviewing the operation of the Planner and locating situations that CBPS could handle better if modifications to the CBPSs control flow where made.

The following sections briefly describe each of the identified areas, and provides suggestions or alternatives that could be included with the CBPS to increase the effectiveness of it.

4.2 Base Plan Locating

The task of locating the plan in the Plan library that most closely resembles the planning tasks is handled by the Planner object's *generate* method. This method initially generates a rating for every plan in the Plan Library in relation to the planning tasks. The rating consists of the number of extra tasks , missing tasks, and the number of failures the plan has had. The plan with the best rating is used as the base plan. The concept best currently means the one with the least number of missing tasks, extra tasks, and task failures, or the one that has the least missing and extra tasks.

The method of rating plans in the Plan Library, and the above simple plan selection rule proved sufficient for the CBPS prototype, but may not be able handle every possible context (situation) or be

efficient enough when the Plan Library grows in size. An example of a situation the rating method does not handle is as follows:

> The Library plan has more extra tasks, less missing tasks, and no failures, in relation to the base plan that was chosen. In some cases, the fact that it has less missing tasks than the base plan may make it a better plan because less tasks have to be added to it. However, the fact that it has more extra tasks than the base plan chosen may make it a worse plan to use. It is a difficult problem to decide exactly when one plan should be used over another, even when it may have more extra tasks, missing tasks, and failures.

This problem points to one obvious extension. Items other than the number of extra task, missing tasks, and failures can be considered when rating and choosing a base plan. Other features such as Start and Stop times, the Constraints, and the Resources can be considered. To an extreme, any comparable plan attribute can be considered.

It is our opinion that you will not be able to devise an algorithm that covers the selection of the best plan from the Plan Library in every case. This is a extremely heuristic activity. For this reason, using an Knowledge Based system to decide the best plan using all of a plan's attributes in relationship to the currently selected best plan, is a good method.

To reduce the initial search space of the Plan library, the Plan library could also be divided into areas of related plans, e.g. based on size, task type, or hierarchy. The base plan locating function could begin in one of the most likely areas and choose between those plans using the current planning tasks as an index. If no suitable

plan is located, the search could be expanded to other possible areas of the Plan Library.

4.3 Base Plan, Planning Task Unification

The task of unifying the selected base plan with the planning tasks is handled by the Plan object's *unifyWith:forPlanner* method. This method takes each task in the base plan and unifies it with the corresponding planning task. There is a one-to-one correspondence between the tasks because the base plan has added the extra tasks and removed the missing tasks before the unification process was invoked. The actual unification process involves the unification of the tasks attributes. These attributes include the task's constraints, resources, and returns.

There are two different types of unification: simple and complex. Simple unification occurs:

1) When a base plan's task has a constraint, resource, or return defined and the planning task does not.

2) When a planing task has a constraint, resource, or return defined and the base plan's task does not.

3) Both the planning task and base plan task have the constraint, resource, or return defined and they are equal in all respects.

In each of these cases, the constraint, resources, or return, is left in tacked and installed in the base plan's task (as in case 2), or just left alone (as in case 1 and 3).

Complex unification occurs when both the planning task and base plan's task have the constraint, resource, or return, defined, and they

are not equal. In this case, each constraint, resource, or return object uses the method *unifyWith:forPlanner:name* to decide whether the planning task's or base plan's task constraint, resource, or return should be used in the base plan's task.

Currently, each one of these methods does a comparison of the two attributes and decides which one to use. For example, if two Temperature Constraints are being unified, it selects the maximum temperature value between the two and uses it in the base plan task's Temperature Constraint value. This rule is appropriate for most tasks, however some tasks would benefit from picking the minimum temperature

To handle this problem, a set of different unification algorithms could be created. Not only dependent on the type of attribute but also on the task id. In the task attributes definition, it could declared what types of unification rules are appropriate for the particular attributes. Now instead of using the standard unification methods for the type, it could select what is best for that particular task's id and attribute.

4.4 Verification and Replanning

The verification process involves verifying that each task's constraints and resources meet with the expected predication about their values at execution time. In the event of a failure to meet the expectation or when an executing task fails, the replanning process is invoked. This process looks first to the plan's failure information to see if it can locate a similar problem in the past. If it can, it then retrieves the planning action used to resolve the problem and executed it. If no similar failure can be found, the Knowledge Base rules are consulted for a planning action. The planning action is retrieved, executed, and the plan is verified again from the start. An enhancement to improve efficiency of this is to only verify the plan from the last task altered by the replanning action, not from the beginning. This is an obvious reduction.

When a task is executing and replanning is done, the replanning information is stored within the task in the plan. If the task experiences another failure, currently the replanning mechanism is invoked again and the process repeated. If the task failed again for the same reason, the previous replanning action was obviously wrong, it should be removed as a valid option when the plan's task fails again. This is currently not done and the addition would enhance the accuracy of the stored failure information for a task in the plan.

An option not currently available is to allow the user to select the replanning action to take in the event of a failure. The user could be shown the suggested action and a list of other available actions, and then allowed to choose the one to use. This would give the user greater input in the replanning process, and his choices would also be recorded and later retrieved when similar problems occur in the future. Thus enhancing the quality and accuracy of the Plan's task knowledge.

4.5 Knowledge Base Rules

The Knowledge Base Rules proved to be an excellent method of encoding what replanning actions are to be taken when various tasks and constraints failed. The decision of what action to take depends on what the failed task and constraint is. The current plan is also available to help decide what action to take, however no supplied methods for accessing it are provide. A user, having an understanding of Prolog and the Plan object, could access this information. However, for ease of use a library of Prolog expressions to access this data could be provide to aid the Knowledge Base programmer. This would permit the Knowledge Base rules to review the tasks before and after the failed task, and to review the task's attributes in order to decide what actions are to be done.

The replanning actions are actual method names defined for the plan object. The currently available actions are *#takeNoAction:with*, *#findAPlace:with*, and *#moveToTheEnd*. For a description of them consult the Users Manual section under Logic Browser. New action methods could be created to enhance the range of available replanning actions. For example, a new one could be added that moves the task to the start of the plan, or one that moves the task after or before a specific task. These additions would help the replanning actions be more flexible, and provide a greater range of options for replanning.

4.6 Evaluation

An Evaluator object is created before the base plan is created, and information added to it as the plan executes. This information includes the number of extra and missing tasks, the number of failures found in verify the initial plan, the number of failures found in the execution of the plan, the types of failures, the number of failures the plan had in the past, and the number of tasks in the plan. When the plan is evaluated, this information is considered and the appropriate action taken with the plan (forget it, add it to the Plan Library, update the base plan in the Plan Library, or remove the base plan from the Plan Library). The evaluation is performed by the Evaluator Object's *evaluate* method.

The evaluation of Evaluator object could be enhanced to consider other variables, rather than the set defined above. These would come from a set of defined variables that are derived as essential to the evaluation of an executed plan. It could be advantageous to have a small expert system decide what is to be done with plan. The current method is set up much like an inference engine already, and this addition would set up the evaluation the way it turned out in the implementation.

4.7 Generalization

Generalization is the process of taking a group of statements, for example plans, and forming a single statement that inherits the concepts and properties found in the group of statements. The process of generalizing similar plans found in the PlanLibrary into a single plan can be done in two ways:

- By performing the generalization operation periodically, the number of plans in the Library can be reduced. Also the plans that are in the Library are more complete - the knowledge used to generate the plan comes from many different sources (plan) not just a single one.

- By having general plans index more specific plans, searching time for the base plan can be reduced. General plans are used to find an initial plan, once found the specific plans it index can be checked and used if they are better suited to the current set of tasks (goals). If the specific plans are not suitable the generalized plan can be used.

The generalization process can be performed in three ways:

1) Incremental - As a new plan is added to the library it is checked to see if it could be generalized into another existing plan. If not, a new plan generalization is created.

2) Collective - Periodically the plans are reviewed , both new and generalized, to generate a new set of generalized plans.

3) Combination - Both previous methods are combined.

The generalization process could also be performed on the Knowledge Base rules in a similar manner to remove duplicates and improve the overall quality of the rules.

4.8 Hierarchical Tasks

In our design, tasks are defined as executable units that can not be further decomposed. This proved to be sufficient to produce a planner that can handle the dynamics of planning and replanning in a changing environment. Plans where manipulated using the tasks they contain and the different attributes of the tasks. The tasks are considered the goals the plan is attempting to achieve. When a task's attributes are satisfied, the task's goal is meet. When the plan's task goals are meet, the plan has succeeded. To be able to further reason about the plan and its tasks and to further decompose the task's goal or goals, a hierarchical approach to task definition could be used.

A hierarchical approach to task definition allows a task to be decomposed into subtasks. These subtasks can be further decomposed into sub-subtasks, an so-on. By doing this, the task's goal can be also decomposed into subgoals. To meet the task's goal, the subtasks used to compose the task can be varied depending on the current executing environment of the plan. This makes plan execution extremely dynamic, the choice of subtask to execute depends on the best subtask to execute given the current situation. This would enable the planner to avoid unexpected dangers and make use of novel opportunities. This is described in detail in the next section.

4.9 Unexpected Dangers and Novel Opportunities

The ability for a planner to avoid unexpected danger and exploit novel opportunities would be an important advancement for any planning system. The key ability of such a planner is to be able to recognize what features about its current execution environment are significant and use them to its advantage. To be be able to determine significance, it must first be noted, but you can not notice all features. The question then is how to recognize significance?

The answer to this question begins with relating the environment's features to the goals that are currently active or ones that will be active in the future. These relations could be that the feature is expected or unexpected. We must determine what impact the features have on the goals and the plan. To do this, the plan and the tasks within the plan should be hierarchical by nature. The reason for this is two fold. First, during planning creation a task can be composed to execute differently depending on expectations about the environment. For example, if a task's temperature is in a particular range it should execute slowly, if it is cooler it can execute quickly. Secondly, by decomposing a plan and tasks into subplans and tasks the current environment features (goals) can be matched against those of future tasks during execution. If a match is found, this unexpected opportunity could be exploited by satisfying one of the future task's goals or subgoals immediately. The method for recognizing them as opportunities is as follows:

1) Notice that certain situation features facilitate the persuit of some goal.

2) Find the goal for which these features are an opportunity and understand why.

3) Decide if the opportunity is important enough to pursue.

As well as waiting for expected or unexpected goals to occur during execution, a planner can be looking ahead in an attempt to make current execution choices based on its current environment that, in addition to meeting its current goals, also lead the planner in a direction that will enhance its chances of successfully executing its other tasks in the future. Goals are strongly activated if they are currently perused, or if a simple feature indicating some relation has been found. It is particularly important to arouse competing or conflicting goals. Once strongly aroused, a goal gains more processing power to determine is subtle features. This is the ability to avoid foreseeable dangers.

4.10 Summary

This section helps show what planning power is possible by building on top of the existing CBPS base. It is from the powerful implementation of the base that makes this possible. The main discovery is the requirement of hierarchical plans to be able to further enhance the reasoning power, making opportunism possible.

5 CONCLUSION

The CBPS implementation has demonstrated that using a combination of case-based and dynamic memory techniques can provide an excellent method of handling planning in a dynamic environment. As can be seen from the implementation, our planner is concerned with many of the different areas of case-based reasoning. We retrieve past cases based on the number and type of tasks in the current situation they match, and use the number of times a plan succeeds in similar situations. We use a knowledge base of rules to aid in the plan transformation of a past plan to the current situation. We use past planning information and current planning operations to explain the planning task. We do dynamic replanning using the same knowledge base to keep the plan executing. We note all of this information and encode it into a past plan or new plan, enabling plan cases to be better utilized on the next planning iteration. The feedback loop enables plans and new plan learning to evolve with the environment over time.

By using dynamic memory and case-based reasoning techniques, combined with knowledge based techniques for replanning, we have presented a design that handles resource constraints, feedback, and achieves both robustness and some degree of autonomy through plan learning. Although not essential to any part, the operator can guide the overall planning process and control the acquisition of new plans and rules for replanning. With the current design and the future enhancements, we feel we are approaching a realistic, efficient planner.

6 REFERENCES

Deugo, D. L., Oppacher, F., Thomas, D., Planning Techniques Survey: Their Applicability to the Mobile Servicing System, TR/SCS, Carleton University, Ottawa, 1988.

Deugo, D. L., Oppacher, F., Thomas, D., A Proposed Approach for Scheduling Applications (With Respect to the Mobile Servicing System), TR/SCS, Carleton University, Ottawa, 1988.

Oppacher, F., Deugo, D. L., A Dynamic Case-Based Planning System for Space Station Application, Proceedings of the Fourth Conference on Artificial Intelligence for Space Applications, 1988.