AGENTBUILDER

An Integrated Toolkit for
Constructing Intelligent
Software Agents

Reference Manual

Version 1.4 Rev. 0

June 16, 2004

Acronymics, Inc.
1301 West 8th St., #28
Mesa, AZ 85201-3841
http://www.agentbuilder.com

Acronymics, Inc.
1301 West 8th St., #28
Mesa, AZ 85201-3841

(480) 615-8543
FAX: (480) 615-1297

http://www.acronymics.com
http://www.agentbuilder.com

AgentBuilder Reference Manual

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Agent Construction Tools

Chapter Overview

You can find the following information in this chapter:

- Overview of the Toolkit
- Project Manager
- Ontology Manager
- Agency Manager
- Agent Manager
- Protocol Manager

A. Overview of the Toolkit

The AgentBuilder toolkit is designed to provide the agent software developer with an integrated environment for quickly and easily constructing intelligent agents.

The construction tools are divided into four major categories. They are described in detail in the following sections. Figure 1 illustrates the tools in the AgentBuilder Toolkit.

Note that not all tools are available in AgentBuilder Lite. For example, the Agency Viewer tools and the Protocol Editor are not a part of the Lite product.

Important Note about Version 1.4

Java Version 1.4 introduced several changes that may impact AgentBuilder users. In particular, agent names can no longer have embedded blank characters. For example, “Buyer Seller” is no longer a valid legal name. Instead, use something like “BuyerSeller” or Buyer_Seller” for the agent name. If you have constructed agents with embedded blank characters in the agent name, you will have to rename them before using them with version 1.4 of AgentBuilder.

Java Version 1.4 also has problems with user names with embedded blanks. If your user name is, for example, “John Doe” then running the Agency Viewer will generate an error. This will likely be a problem only with Windows users. Please see the ReadMe file that came with the distribution for a solution for this problem.

Caution! Some of the examples in this document may have embedded blank characters in agent names. This will be corrected in future revisions to this document.

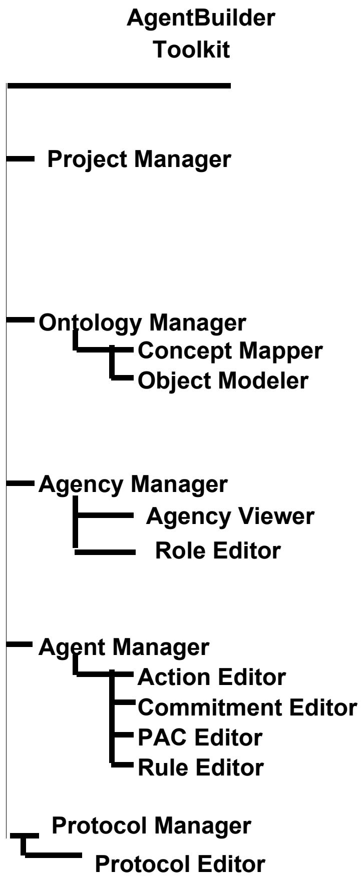


Figure 1. The AgentBuilder Toolkit

Tool Introduction

User Files

Each user is provided a directory where all of their files and directories are stored. The location of the user's directory is system dependent. On UNIX systems the default location for the user's AgentBuilder directory is *user-home-dir*/.AgentBuilder (e.g., /home/flintstone/.AgentBuilder). On Windows systems the default location is *drive*:/Program Files/AgentBuilderLite1.4/users/*user-name* (e.g., C:/Program Files/AgentBuilderLite1.4/users/flintstone). AgentBuilder stores the user's properties file, the error log, and directories for RADL files, generated classes, and the repository in the user's AgentBuilder directory. With the exception of the properties files, the locations of the user's files and directories can be modified using the Project Manager's **Options** dialog.

The user's properties file describes the location of all user files and directories, the user's identity, and appearance preferences. The error log contains any errors generated by any of the AgentBuilder tools. The RADL directory contains agent definitions; the generated classes directory contains classes that have been generated for a particular agent; the repository contains the data for defined ontologies, projects, agencies, and agents.

The structure of the repository must not be modified. The repository contains separate directories for storing information about agencies, agents, ontologies, projects, protocols and the class table. Each directory is labeled *information-type*Store, where *information-type* is used to indicate the type of information stored in the directory. For example, the agentStore contains information about all agents that have been defined by the user. Each time you select the **Save** menu item under the **File** menu, the respective store directory contents are updated.

Performing a **Save** operation in any tool that manipulates agents has special implications. A **Save** operation that updates an agent causes an update in any AgentBuilder tool that has the same agent loaded. This feature is needed because the agent tools are interdependent. This means you can modify an agent's name in the Agent Manager, and then have the project tree automatically refreshed based on the modified agent. This allows you to have any of the agent tools open simultaneously. For example, you can have the Rule Editor open while defining actions, commitments, and templates. When you save this information, the Rule Editor will automatically update itself to display the newly created actions, commitments, and templates. This frees the user from having to manually update the individual agent tools.

Common Interface Features

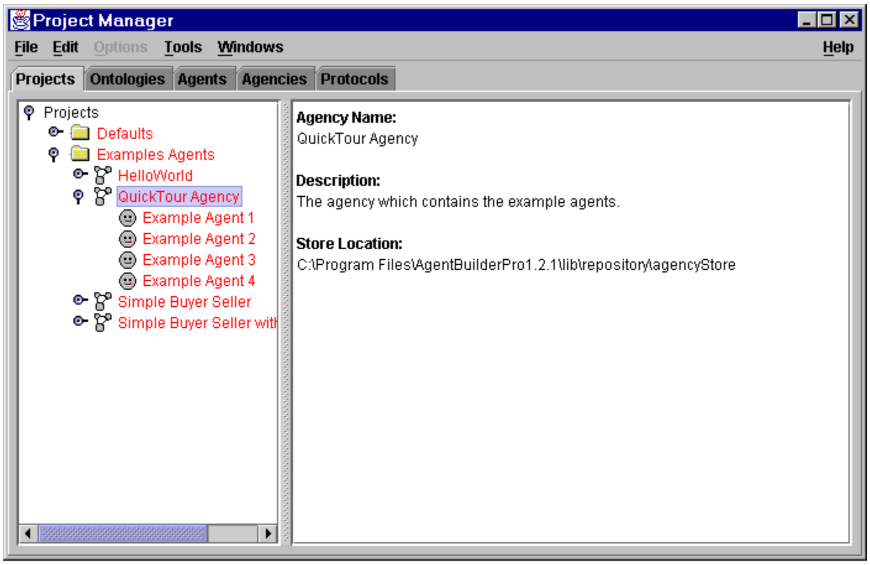
The various AgentBuilder tools share a number of common interface features. The following paragraphs describe common interface features of AgentBuilder. We use the Project Manager as an example. The Project Manager is shown in Figure 2.

Windows menu

Each AgentBuilder tool includes a **Windows** menu. The contents of the **Windows** menu indicate which AgentBuilder tools are visible. The **Windows** menu is managed with a custom window manager. As tools are opened and closed, the **Windows** menu is dynamically updated. Selecting a menu item in the **Windows** menu brings the selected tool to the front. The current implementation limits the selection of windows to windows that are not iconified.

Help Menus

Each AgentBuilder tool contains an identical **Help** menu. Each **Help** menu contains the following menu items:



Agency Name:
QuickTour Agency

Description:
The agency which contains the example agents.

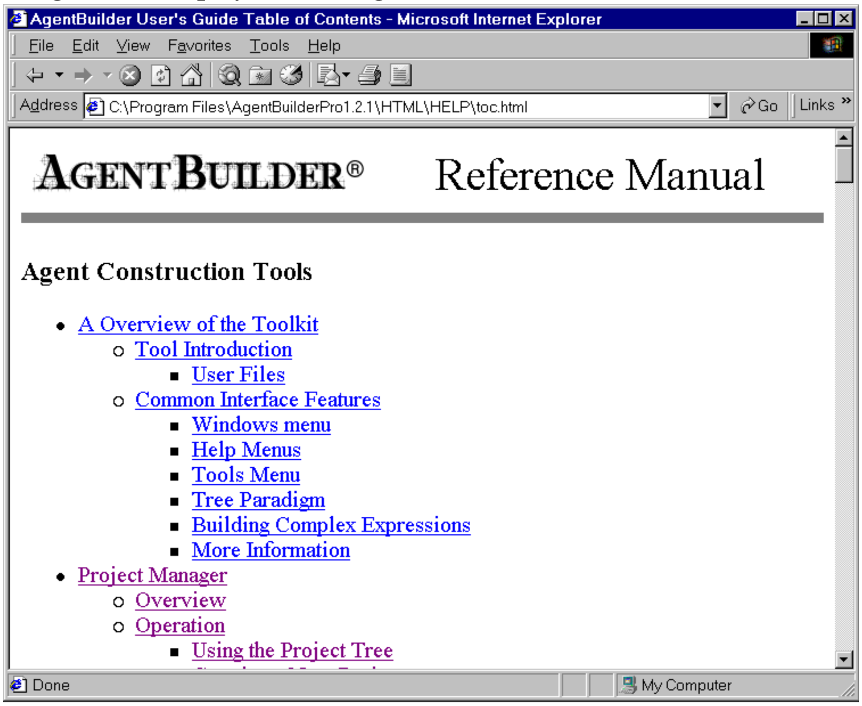
Store Location:
C:\Program Files\AgentBuilderPro1.2.1\lib\repository\agencyStore

Figure 2. AgentBuilder Project Manger

- **About**
- **Index**
- **Search**
- **Tutorial**
- **About AgentBuilder**
- **AgentBuilder Home Page**

The **About** menu item displays information about the currently selected tool. The **Index** menu item provides an index to the help system. The **Search** menu item allows you to search the help system for a specific topic. The **Tutorial** menu item displays an on-line tutorial with information for using the AgentBuilder toolkit. The **About AgentBuilder** menu item brings up a dialog displaying general information about AgentBuilder. The **AgentBuilder Home Page** menu item displays the home page for the AgentBuilder product. With the exception of the **About AgentBuilder** menu item, all other menu items utilize your default web browser to display the **Help** pages.

Figure 3 below shows the **Help** Index for the AgentBuilder tools. The help system is organized as a series of HTML pages. You can navigate the help system using familiar browser controls. .


AgentBuilder User's Guide Table of Contents - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address C:\Program Files\AgentBuilderPro1.2.1\HTML\HELP\toc.html
Go
Links
Done
My Computer

AGENTBUILDER® Reference Manual

Agent Construction Tools

- A Overview of the Toolkit
  - Tool Introduction
    - User Files
  - Common Interface Features
    - Windows menu
    - Help Menus
    - Tools Menu
    - Tree Paradigm
    - Building Complex Expressions
    - More Information
- Project Manager
  - Overview
  - Operation
    - Using the Project Tree

Figure 3. AgentBuilder Help Viewer

Tools Menu

The **Tools** menu can be found in the Project Manager, Ontology Manager, Agent Manager, Agency Manager and Protocol Manager. Unlike the **Help** menu, the contents of the **Tools** menu is dependent on the AgentBuilder tool selected. For the Project Manager, the **Tools** menu provides a menu item for selecting the Agent Engine. The Ontology Manager contains menu items for its tools as does the Agent Manager, Agency Manager and the Protocol Manager.

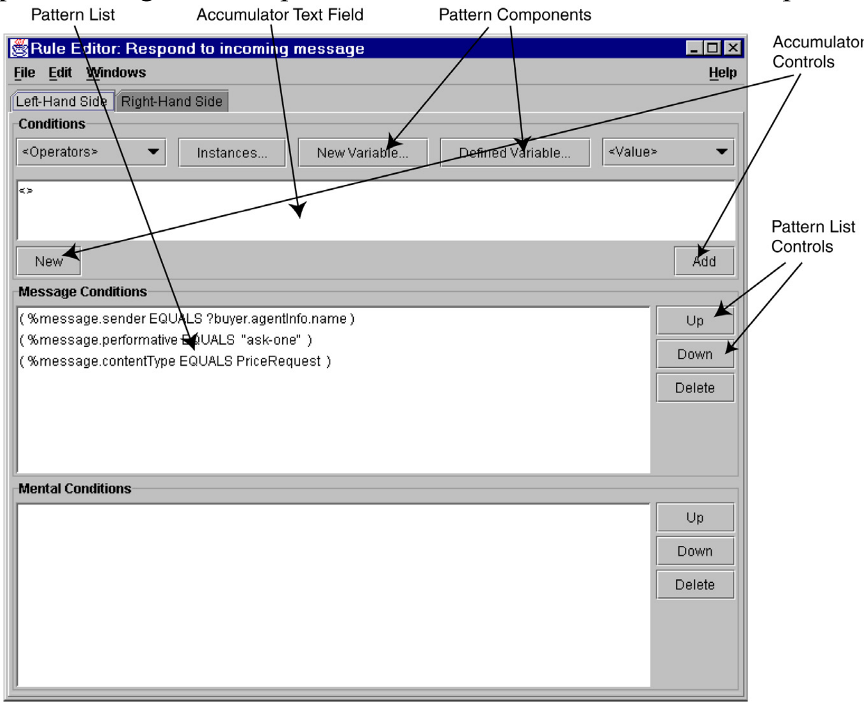
Tree Paradigm

The tree paradigm is used throughout the toolkit. Traditionally, the tree paradigm is used for displaying the contents of file systems. In the AgentBuilder toolkit, the tree paradigm has been extended to describe a containment hierarchy. In the Project Manager, the tree paradigm shows that projects contain agencies, which in turn, contain agents. In the **Attributes** dialog for the PAC Editor, the tree paradigm is used to show the contents of complex attributes that contain simple attributes, and possibly, other complex attributes.

Building Complex Expressions

AgentBuilder provides a very powerful agent construction mechanism that minimizes the amount of typing you must do. The actual agent programming language is generated automatically by the AgentBuilder tools. The objects and attributes you define in the analysis phase of your project are reused by graphical editors. These graphical editors are used to construct complex expressions that give your agents useful behaviors.

AgentBuilder uses an *accumulator* paradigm for constructing complex patterns and expressions. Several editors use an accumulator text field to accumulate pattern components as you enter them, and a pattern list to display the completed patterns (here “patterns” is used in the generic sense and is not restricted to patterns on the left-hand side of a rule). There is usually a row of buttons or pull-down menus above an accumulator text field; these provide the pattern components that you select for insertion into the accumulator. The pattern list for the completed patterns is usually situated below the accumulator text field. Figure 4 shows a portion of the **Condition Pattern** panel that is part of the Rule Editor used in AgentBuilder. It provides a good example of the use of the accumulator concept.



```
( %message.sender EQUALS ?buyer.agentInfo.name )
( %message.performative EQUALS "ask-one" )
( %message.contentType EQUALS PriceRequest )
```

Figure 4. Accumulator Building Complex Patterns

The normal sequence of operations is to specify the components of the pattern (operators, variables, constant values, etc.) by using the row of buttons above the accumulator, then click on the **Add** button to the right of the accumulator text field. Clicking on **Add** will move the pattern from the accumulator to the associated pattern list. Clicking on the **New** button will clear the accumulator text field. (The **New** button does not clear the Pattern List).

In general, pattern components should be specified in a left-to-right order. One important exception to this general rule is the ordering for message conditions and mental conditions in the Rule Editor. To build these conditions you should first specify the conditional operator (e.g., `EQUALS, <=`, etc.) then specify the operands in left-to-right order. For example, to build the message condition:

```
(%message.performative EQUALS achieve)
```

you should first select **EQUALS** from the **Operators** pull-down menu. This will fill the accumulator with the template:

```
(<> EQUALS <>)
```

This template shows the operator and the <> slot markers which indicate that you need to select two operands. In this case, you would select **%message.performative** from the **Defined Variable** dialog (or first you may need to create the variable using the **New Variable** dialog) then select **achieve** from the **Values** dialog. As you select the operands the slot markers in the template will be filled in with the operands from left to right. Finally, click on **Add** to transfer the pattern from the accumulator to the message condition list below the accumulator.

After the patterns are placed in the pattern list, you can change the ordering of the patterns by using the **Up** and **Down** buttons. Clicking on a pattern in the pattern list will highlight the pattern and copy it into the accumulator for modification. Clicking on the **Up** or **Down** button will move the pattern up or down one slot in the list. You can delete a highlighted pattern by clicking on the **Delete** button.

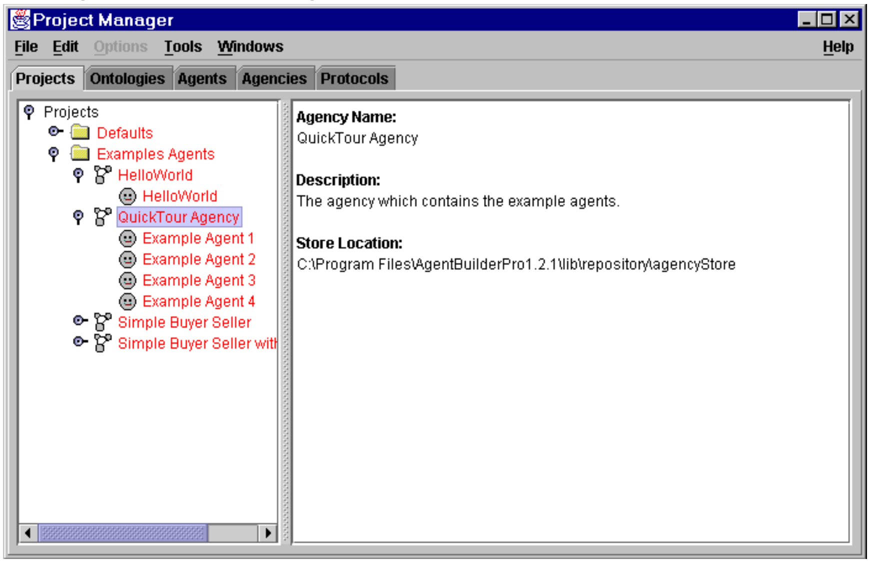
More Information

You can find more detailed information about AgentBuilder in the Appendices to this volume. AgentBuilder intrinsics are described in Appendix A, “Intrinsics” on page 287. A detailed description and grammar for RADL is provided in Appendix B,“Runtime Agent

Definition Language" on page 293. Operators and Patterns supported by the tools are described in Appendix C, "Operators and Patterns" on page 301.

B. Project Manager

The Project Manager is the high-level tool used to create projects, agencies, and agents. Agencies can be created using the Agency Manager, and agents can be created using the Agent Manager. The Project Manager provides an overall view of the development process. The Project Manager allows you to easily see all of the projects, agencies and agents that you have created. The Project Manager is shown in Figure 5.



Agency Name:
QuickTour Agency

Description:
The agency which contains the example agents.

Store Location:
C:\Program Files\AgentBuilderPro1.2.1\lib\repository\agencyStore

Figure 5. The Project Manager

Overview

The **File** menu allows you to create new projects, agencies, and agents. The **File** menu is also used to shut down the program. Using the **Edit** menu, you can delete a selected node in the project tree, cut, copy and paste agents, and view or edit AgentBuilder properties. The **Tools** menu provides access to the Agent Engine. The

Windows menu lets you switch among the various open Agent-Builder tools. The **Help** menu provides access to the AgentBuilder help system.

The project tree allows you to view your repository. The top-level view shows the project folders. Each project folder can contain multiple agencies which, in turn, can contain multiple agents.

The description area provides a textual description of the project, agency, or agent in the project tree. To see a description, select a node in the project tree, and the description area's contents will show a description of the selected project, agency, or agent.

Operation

Using the Project Tree

The project tree is based on a tree structure commonly used to display file systems and the hierarchical relationships between files and directories. Tree nodes that contain children can be opened, or expanded, by double-clicking on the node's label, or by clicking on its *expand* icon to the left of the label. Similarly, a tree nodes that has been expanded can be collapsed by double-clicking on its label or by clicking on its *collapse* icon to the left of the label.

The Agent Manager tool can be opened in the same manner as the Agency Manager. You can select the agent and then click on the Agents tab, or right-click on the agent and select the Edit menu item. You can also open the Agent Manager with no agent loaded, simply click on the agents tab without having selected an agent.

The project tree also provides context-sensitive pop-up menus. The pop-up menus are activated by using the right mouse button. Right-clicking on a project will display the project's pop-up menu. The same can be done with agencies and agents. Right-clicking on an empty area within the project tree will display the project tree's pop-up menu.

Creating a New Project

Projects can be created in two ways. You can select **New** from the **File** menu, or you can right click on the **Projects** folder, then select **New Project** from the pop-up menu. In either case, the **Project Properties** dialog will be displayed. You must supply a name and click on the **OK** button to create the new project.

Creating a New Agency

Agencies can be created in two ways. The first way is to select a project folder in the project tree and then select **New** from the **File** menu. The other method used to create a new agency is to right click on a project folder, then select **New Agency** from the pop-up menu. You must supply a name and click on the **OK** button to create the new agency.

Creating a New Agent

Agents can be created in two ways. The first way is to select an agency folder in the project tree and then select **New** from the **File** menu. The other method used to create a new agent is to right-click on an agency folder and then select **New Agent** from the pop-up menu. You must supply a name and click on the **OK** button to create the new agent.

Cutting, Copying and Pasting an Agent

Agents are the only nodes in the project tree that can be cut, copied, or pasted. There are two ways to use these clipboard functions on an agent. The first way is to use the **Edit** menu's **Cut**, **Copy** and **Paste** menu items. The other method is to use the agent's pop-up menus for **Cut** and **Copy**, and the agency's pop-up menu for **Paste**.

Whichever method is used, an agent must first be selected before a cut or copy operation can be performed. For the paste operation, an agency must be selected. Invalid selections will be ignored. Once an agent has been pasted into an agency, the newly pasted agent will be assigned the current user's name and the current date and time. If the agency being pasted into already contains the name of the agent being pasted, the agent to be pasted will recursively have **CopyOf** prepended to its name.

Modifying Project, Agency and Agent Properties

To modify the properties of a project, agency, or agent, right-click on the node in the project tree. The pop-up menu for the selected node will contain a **Properties** menu item. Selecting the **Properties** menu item will display a properties dialog for the selected node. The properties dialog for projects and agencies allows you to modify the name and description. The properties dialog for an agency is more complex. You can modify an agency's name, description, author, company, and communications. The agent properties dialog is similar to the **Agency Properties** dialog, except that you can also modify the agent's icon, engine cycle time, and agencies. In each of the properties dialogs, you must select the **OK** button in order for the changes to take effect. Selecting the **Cancel** button or closing the window will discard any changes.

Deleting Projects, Agencies and Agents

Projects, agencies, and agents can be deleted in two different ways. The user can first select the tree node to be deleted, then select **Delete** from the **Edit** menu. The other way in which tree nodes can be deleted is by right-clicking on a tree node, then selecting **Delete** from the pop-up menu.

Editing AgentBuilder Properties

To edit user preferences, the **AgentBuilder Options** dialog is accessed through the **Edit** menu's **Options** menu item. The **Agent-Builder Options** dialog allows you to view and edit various preferences such as:

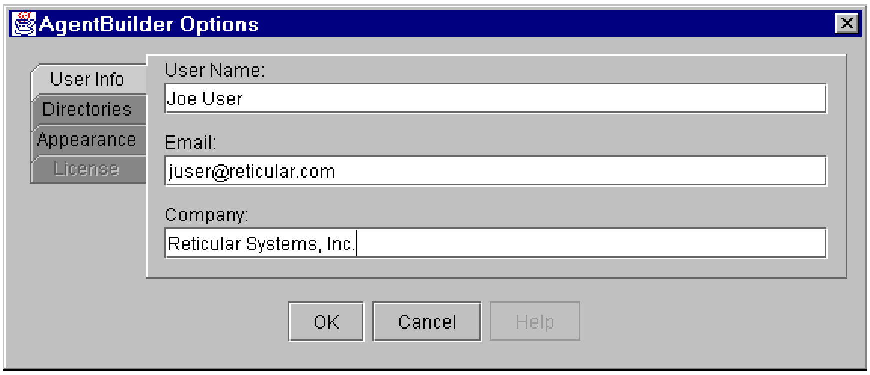
- Name
- Email address
- Company name
- Directory locations
- Error logging
- "Look and Feel"
- Font size
- Background color
- Foreground color
- KQMLConverter class for Sockets

To edit the user's name, email or company name, select the **User Info** tab in the **AgentBuilder Options Dialog**. Figure 6 shows the **AgentBuilder Options Dialog** with the **User Info** tab selected. The **User Info** panel is made up of text fields so that the user can easily change the contents of any or all fields.

To modify the user's directories and error log location, select the **Directories** tab. Figure 7 shows the **AgentBuilder Options Dialog** with the **Directories** tab selected. The **Directories** tab contains a table for viewing and modifying the user's directories.

To modify a directory entry in the table, first select the directory to be modified. Then, select the **Browse** button. Selecting the **Browse** button will bring up a Directory dialog, as shown in Figure 8.

The **Directory** dialog is similar to a file selection dialog, except that it only allows you to view and select directories. The **Directory** dia-

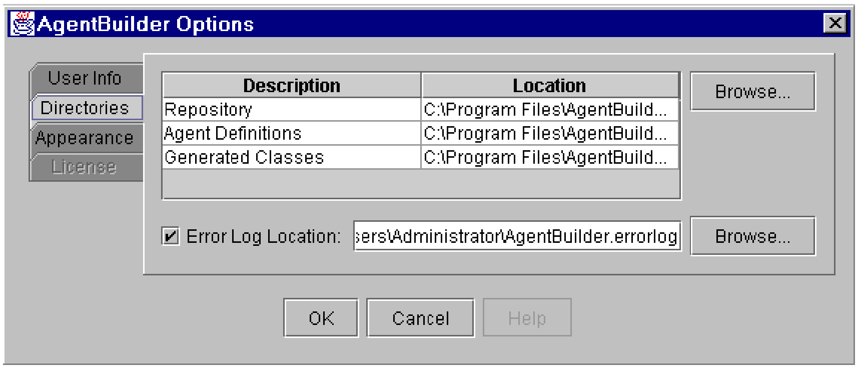


User Name: Joe User

Email: juser@reticular.com

Company: Reticular Systems, Inc.

Figure 6. User Preference Dialog



Description	Location
Repository	C:\Program Files\AgentBuild...
Agent Definitions	C:\Program Files\AgentBuild...
Generated Classes	C:\Program Files\AgentBuild...

☑ Error Log Location: sers\Administrator\AgentBuilder.errorlog

Figure 7. Preferences Dialog Showing Directories

log has several features worth noting. At the top of the dialog, there is a combo box that displays the current directory. The combo box allows you to go up in the directory hierarchy. The dialog also contains icons for going up to the next directory, returning to the home directory, and creating a new directory, respectively. In the center of the dialog is a pane for displaying the contents of the current directory. In this pane, you can double click directories to open them. At the bottom of the dialog, there is and editable text field where you can type the name of the directory. When you find the

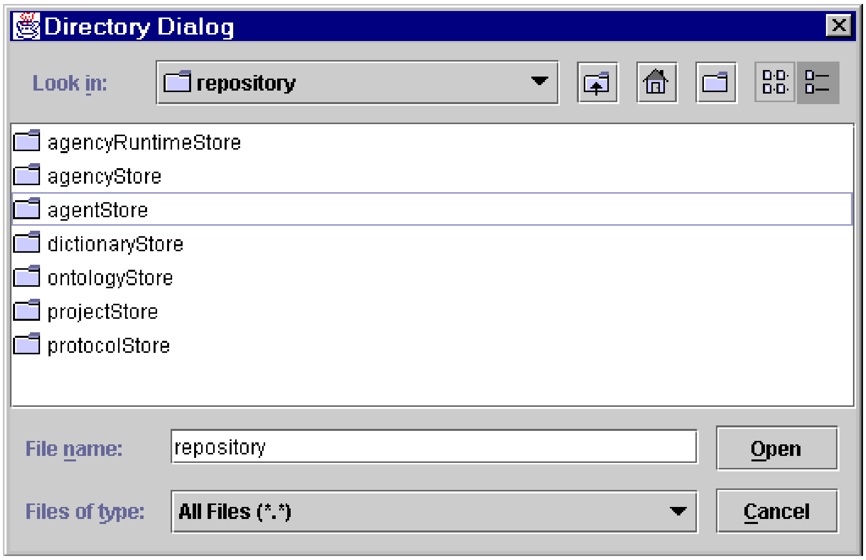
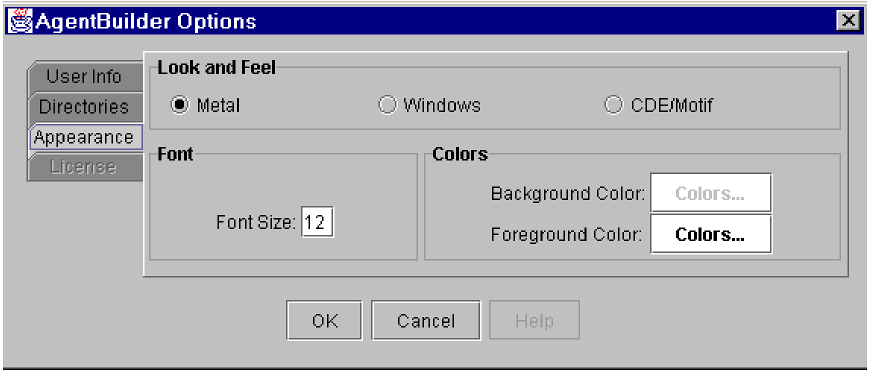


Figure 8. Directory Dialog

desired directory, you can select the directory in the directory pane and then click on the **Open** button. If no directory is selected, the current directory becomes the selected directory. The selected directory will be displayed in the directories table.

You can choose to have error logging turned on or off by selecting the checkbox next to the **Error Log Location** label below the directories table. If error logging is turned on, you must also specify the error log file that will be used. You can specify the error log location by typing in the text field, or by clicking on the **Browse** button. Clicking on the **Browse** button will display the standard File dialog.

The general appearance of the AgentBuilder toolkit can be changed in the **Appearance** panel. Figure 9 shows the **AgentBuilder Options** dialog with the **Appearance** tab selected. The **Appearance** panel is divided into three sub panels. The top panel is the **Look and Feel** panel. The **Look and Feel** panel contains three radio buttons for selecting the **Metal**, **Windows**, and **CDE/Motif** look and feel. The **Metal** look and feel is the default look and feel. Due to licensing restrictions, only Microsoft Windows users can select the **Windows** look and feel. Selecting one of the radio buttons will immediately change the look and feel in all open windows. The **Font Size** panel allows you to change the default size of the fonts used in the application by simply typing in a font size in the text field.


AgentBuilder Options

Look and Feel: [✓ Metal] Windows CDE/Motif

Font Size: 12

Figure 9. Preference Panel for Setting Appearances

The background and foreground colors are shown as the colors of their respective buttons. The background and foreground color are modified by selecting the corresponding **Colors** button. Selecting the **Colors** button will display the **Color Dialog**, shown in Figure 10. The **Color Dialog** contains three sliders for setting the red, green and blue values of the background or foreground colors. You will also find, located on the right of the sliders, a box whose color is set to the current RGB value. Once the desired color has been determined, click on the **OK** button to set the color for the background or foreground color.

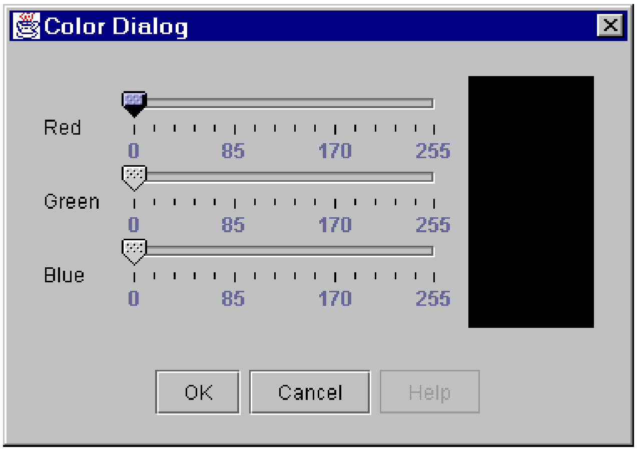


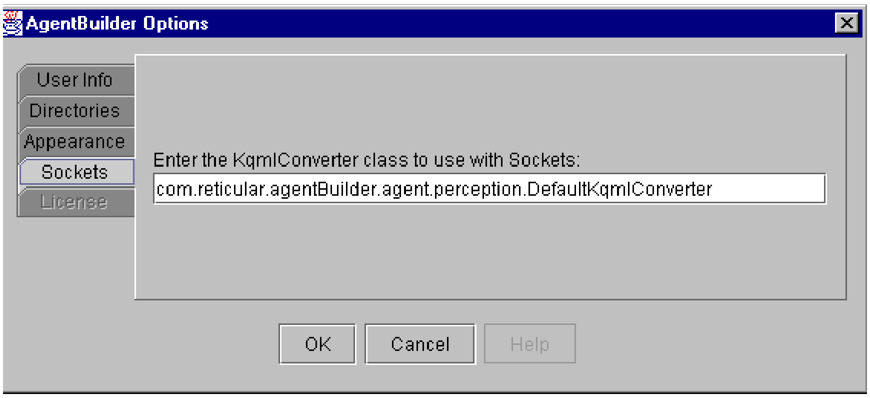
Figure 10. Color Preference Dialog

If you modify any of the preferences, you must select the **OK** button before any of the changes will take effect. Currently, font size and color modifications only take effect the next time the tool is started. Selecting the **Cancel** button or closing the window will discard any preference modifications.

The **Sockets** panel allows you to specify the class to handle the conversion of KQML messages to and from bytes. Figure 11 shows the AgentBuilder **Options** dialog with the **Sockets** tab selected. By default, the class `com.reticular.agentBuilder.agent.perception.DefaultKqmlConverter` will be used whenever you specify that AgentBuilder agents communicate using TCP/IP sockets.

Launching AgentBuilder Tools

Another function performed by the Agency Manager (in addition to providing a graphical display of your projects, agencies and agents) is providing access to the varous tools in AgentBuilder. These tools include the **Agency Manager**, **Agent Manager**, **Ontology Man-**



Enter the KqmlConverter class to use with Sockets:
com.reticular.agentBuilder.agent.perception.DefaultKqmlConverter

Figure 11. TCP/IP Socket Options

ager, **Protocol Manager**, and **Agent Engine**. The following paragraphs describe several methods for opening each of these tools.

The **Agency Manager** can be displayed in three different ways. You can select an agency from the Project tree and then click on the **Agencies** tab. You can also, right-click on the agency and select the **Edit** menu item from the pop-up menu item. The third way to display the **Agency Manager** is to click on the **Agencies** tab without selecting any agency, the **Agency Manager** will be displayed with no agency loaded.

The Agent Engine tool can be opened in two different ways. You can either select the agent and then select the **Tools → Agent Engine** menu item, or by simply right-clicking an agent and selecting the **Run** menu item from the pop-up menu. The Agent Engine cannot be run without first selecting an agent. To run an agency, right-click on the agency and select the **Run** menu item from the pop-up menu. This will launch the **EngineLauncher** for the agency. It will use the agency JVM groups if it exists (See “Creating a JVM Group” for more info).

The **Agent Manager** tool can be opened in the same manner as the **Agency Manager**. You can select the agent and then click on the **Agents** tab, or right-click on the agent and select the **Edit** menu item. You can also open the **Agent Manager** with no agent loaded; simply click on the agents tab without having selected an agent.

There is only one way to open the Ontology Manager and Protocol Manager tools. You must select the **Ontologies** or **Protocols** tab from the **Project Manager.**

Switching AgentBuilder Windows

The **Windows** menu is a dynamic menu that provides a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tools are open. To switch to the desired tool, selects the tool from the **Windows** menu. The selected tool will then be brought to the front on the display.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Exiting AgentBuilder

Because the Project Manager is the top-level tool for the AgentBuilder toolkit, the Project Manager can only be exited, not closed. To exit the system, select **Exit** from the **File** menu.

C. Ontology Manager

Ontology Manager

The Ontology Manager is the tool (shown in Figure 12) that allows you to implement a top-level design of an agent prior to beginning coding or rule definition. Using this tool, you can create new ontologies or modify existing ontologies for the agent under construction. There are two main tools to help you in designing agents and agent-based systems: the *concept mapper* and the *object modeler*. Both of these tools along with the ontology manager tool are described below.

Overview

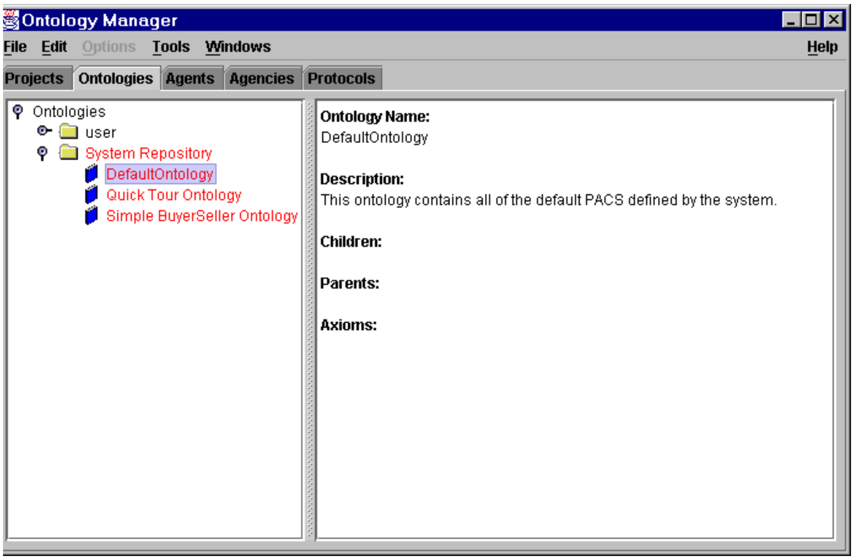
The Ontology Manager has five menus: **File**, **Edit**, **Tools**, **Windows**, and **Help**. The **File** menu is used to create new ontologies and shut down AgentBuilder. The **Edit** menu is used to **Cut**, **Copy**, **Paste** and **Delete** an ontology. You can use the **Tools** menu to select one of the two design tools (i.e. Concept Mapper or Object Modeler). The **Windows** menu allows you to quickly and easily switch between the various open AgentBuilder tools. The **Help** menu gives you access to the AgentBuilder help system. The **Ontology Manager** is shown in Figure 12.

The ontology tree view allows you to view user-defined and system ontologies. You can create and view any number of ontologies. Typically, the system ontologies in the ontology tree are displayed in red. This implies that they are read-only and cannot be altered. A user's personal ontologies are displayed in black. When an ontology is selected in the left panel, the right panel displays general information about the ontology and includes a short textual description of the ontology as well as information about where the ontology is located. Note that the divider between the ontology tree structure and the properties window can be moved horizontally to provide more viewing space for the ontologies.

Operation

Using the Ontology Tree

The Ontology Manager uses the same tree structures found in other AgentBuilder tools. The ontology manager has three levels in its tree structure. The highest level is the Ontologies level, which contains repository folders, which in turn, contain the defined ontologies. The repository folders are represented in the tree by a folder icon. The ontologies are represented by a book icon. Any of the ontologies that are read-only are displayed in red text.



Ontology Name:
DefaultOntology

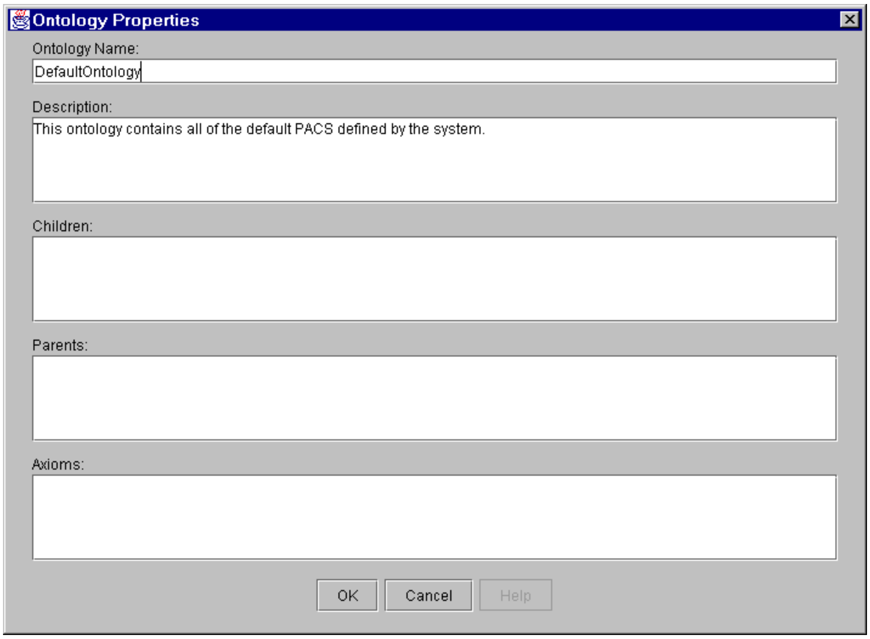
Description:
This ontology contains all of the default PACS defined by the system.

Children:

Parents:

Axioms:

Figure 12. The Ontology Manager



Ontology Name:
DefaultOntology

Description:
This ontology contains all of the default PACS defined by the system.

Children:

Parents:

Axioms:

Figure 13. Ontology Properties

Creating a New Ontology

A new ontology can be added to your user's folder by selecting the user's ontology folder and then either selecting the **New** menu item under **File** or right clicking on the mouse. This will bring up the dialog shown in Figure 13. The properties of the ontology can be entered in this dialog box.

Cutting, Copying and Pasting an Ontology

Ontologies can be cut, copied, or pasted. There are two ways to use the clipboard functions for an ontology. The first way is to use the **Edit** menu's **Cut**, **Copy** and **Paste** menu items. The other method is to use the ontology's pop-up menus for cut and copy, and the repository folder's pop-up menu for paste. Whichever method is used, an ontology must be selected before a cut or copy operation. For the paste operation, a user or system repository folder must be selected. Invalid selections will be ignored. If the folder being pasted into already contains the name of the ontology being pasted, the ontology to be pasted will recursively have **CopyOf** prepended to its name.

Modifying Ontology Properties

The general ontology properties can be modified by right-clicking on the appropriate ontology in the tree structure and selecting the **Properties…** item in the pop-up menu (by right-clicking on the selected ontology).

Deleting Ontologies

To delete an ontology from the ontology tree structure, simply select the desired ontology and choose **Delete** from the **Edit** menu (or from the pop-up menu by right-clicking on the selected ontology). This will prompt AgentBuilder to display a dialog asking if you are sure you want to delete this ontology. If you select **Yes** the ontology will be deleted. If you decide that you do not want to delete the ontology, simply select either **No** or **Cancel** and the deletion operation will not be performed.

Launching Ontology Tools

You can launch either the Concept Mapping tool or the Object Modeler tool. It is necessary to first select the desired ontology before launching either of these tools. If you launch the Concept Mapper or Object Modeler without first selecting an ontology AgentBuilder will remind you by displaying a dialog asking you to select an ontology.

Switching AgentBuilder Windows

To switch between different AgentBuilder windows, select the desired window in the **Windows** menu. This will bring the selected open window to the foreground.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index of the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Pag**e menu item will display the home page for the AgentBuilder product.

Exiting AgentBuilder

To exit from AgentBuilder, select the **Exit** from the **File** menu. AgentBuilder will then display a dialog asking whether you are sure you want to exit. If so, then click on the **Yes** button; if you do not wish to exit, click on the **No** button.

Concept Mapper

The Concept Mapper tool is used to organize and structure an ontology. The Concept Mapper provides a drawing canvas for graphically defining concepts and relationships between concepts. These concepts can then be mapped automatically into the Object Modeling Tool. Note that you are not required to use the Concept Mapper. You can work directly with the Object Modeler if desired.

Overview

The Concept Mapper contains a menu bar with five items: **File**, **Edit**, **Map**, **Windows**, and **Help**. The **File** menu allows you to open ontologies and save the current ontology. You can also print the ontology to obtain a hard copy of the concept information. You can also close the **Concept Mapper** and shut down AgentBuilder using the **File** menu. The **Edit** menu allows you to **Cut**, **Copy**, **Paste** or **Delete** a selected concept or link. The **Map** menu allows you to modify the current concept map by adding new concepts or links. The **Map** menu also provides menu items for clearing the whole canvas or refreshing the map. The **Windows** menu allows you to quickly and easily switch between the various open AgentBuilder tools. The **Help** menu gives you access to the AgentBuilder help system. A conceptual map drawn using the Concept Mapper is shown in Figure 14.

The central feature of the Concept Mapper is the canvas you use to construct an arbitrarily complex representation of concepts and relations between the concepts. You are free to construct as simple or complex a concept map as desired. In addition, the concept properties allow you to create an object that will be used in the Object Modeler. It is easy to rearrange concepts to simplify the structure and minimize overlap of links. Three types of links are supported: undirect, direct, and bidirect. Each link can have a label attached to it that represents the type of relationship between two connected concepts.

Operation

Creating a New Concept

You can create a new concept by right-clicking on an unoccupied region of the canvas and selecting **New Concept** from the pop-up menu. This will cause AgentBuilder to display a dialog where you can enter a name for the concept being represented as well as a description of the concept and any critical information that will be helpful. Also, there is a selection box titled **Make Class**. By clicking on this box, you can create an associated class that will appear in the Object Modeler. Thus, you do not have to enter common information into both tools. Clicking the **OK** button will create a new concept on the canvas. This concept is represented by an oval with the concept name located inside it. Note that the point on the canvas where you right-click is the location where the concept will be placed. If you decide not to create this new concept, you can click on the **Cancel** button and the concept map will remain unchanged. You can also use the **Map** menu to create a **New Concept** in a similar manner.

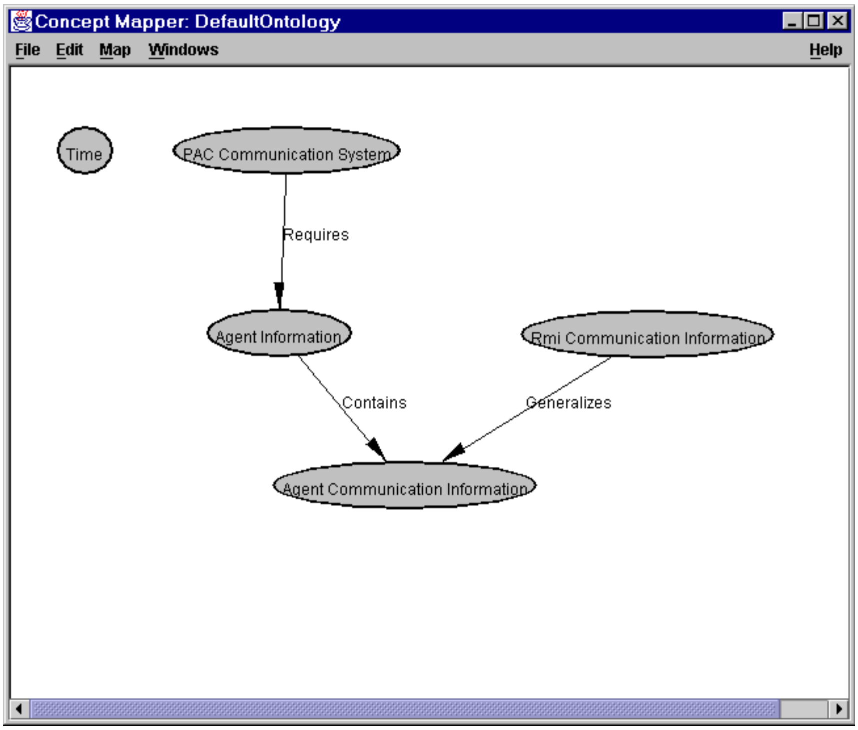


Figure 14. The Concept Mapper

Cutting, Copying and Pasting a Concept

Concepts can be cut, copied, or pasted. There are two ways to use the clipboard functions for a concept. The first way is to use the **Edit** menu's **Cut**, **Copy** and **Paste** menu items. The other method is to use the concept's pop-up menus for cut and copy, and the map's pop-up menu for paste. Whichever method is used, a concept must be selected before a cut or copy operation. If the map being pasted into already contains the name of the concept being pasted, the concept to be pasted will recursively have **CopyOf** prepended to its name.

Creating a New Link

You can create a new link by right-clicking on any unoccupied region of the canvas and selecting the hierarchical **New Link** menu item. You can select between undirected, directed and bidirected links. You can select one of the link types from the **Map → New Link** hierarchical menu. Once you have selected one of the link types, the cursor will change to a cross-hair and your can then click-and-drag on the initial concept and join it to another desired concept. Note that for the undirected and bidirected cases, the order of connection does not matter. However, for the directed case, you must connect them in the direction you wish the arrow to point (the second concept is defined by where you release the mouse button).

Moving an Existing Concept

You can freely move a concept anywhere on the canvas. Do this by clicking on the desired concept and dragging the concept to the new location. Links will adjust themselves to maintain the connection with the relocated concept.

Moving Multiple Concepts

You are able to move multiple concepts at a time. First, you need to select the concepts you wish to move. To do this, hold the **Control** key down and select the concepts with the mouse. Once you have a group of selected concepts, hold the **Control** key down and drag one of the selected concepts to a new location. You will notice that all the selected concepts will move in relation to the mouse cursor. To deselect the concepts, simply click on the canvas.

Deleting an Existing Concept

You can delete an existing concept by clicking on that concept (i.e., selecting it) and then right-clicking and selecting the **Delete** item from the pop-up menu. Likewise, a selected concept can be deleted using the **Delete** item in the **Edit** menu. Note that when a concept is deleted, all links to that concept are also deleted.

Deleting an Existing Link

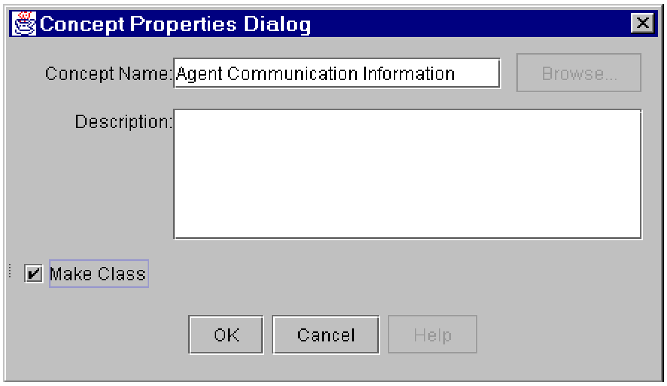
You can delete an existing link either by selecting a link (turning it to a red color) and then right-clicking and selecting **Delete** from the pop-up menu or using the **Delete** menu item in the **Edit** menu.

Changing the Link Type of an Existing Link

You can change an existing link by selecting that link and right-clicking to bring up a pop-up menu. You can then changes the link type by selecting one of the types from the **Link Type** hierarchical menu. The link type can also be changed in the **Link Properties** dialog.

Viewing and Altering the Properties of an Existing Concept

You can view and modify the name and description of an existing concept by right-clicking on the desired concept. You can then select the **Properties** item from the pop-up menu. AgentBuilder will then display a dialog that will allow you to see and alter the name and description of the selected concept. By clicking on the **OK** button, you can enter the changes. Clicking on the **Cancel** button will revert to the original unaltered properties. The **Concept Properties Dialog** is shown in Figure 15.



Concept Name: Agent Communication Information

Description:

[✓ Make Class]

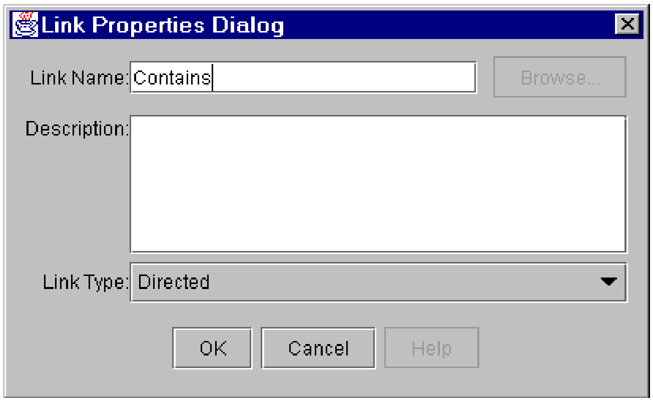
Figure 15. The Concept Properties Dialog

Creating an Associated Class from an Existing Concept

You can create an associated class (that can be viewed inside the Object Modeler tool) by right-clicking on a concept and selecting the **Properties** item from the pop-up menu. AgentBuilder will then display the **Concept Properties** dialog (see “Viewing and Altering the Properties of an Existing Concept” on page 31.). To create the class, simply click the mouse button inside the box labeled **Make Class** and then press the **OK** button to confirm the change. Agent-Builder will then automatically create the class. You can then view and modify the class inside the Object Modeler (see “Object Mod-eler” on page 36).

Viewing and Altering the Properties of an Existing Link

You can view and modify the link label, description and type of a selected link by right-clicking on the selected (red) link. Select the **Properties** item from the pop-up menu. AgentBuilder will then display a dialog that will allow you to see and alter the name and description of the selected link. In addition, the link type can be modified here. By clicking on the **OK** button, you can enter the changes. Clicking on the **Cancel** button will revert to the original unaltered properties. The **Link Properties** dialog is shown in Figure 16.



Link Name: Contains

Description:

Link Type: Directed

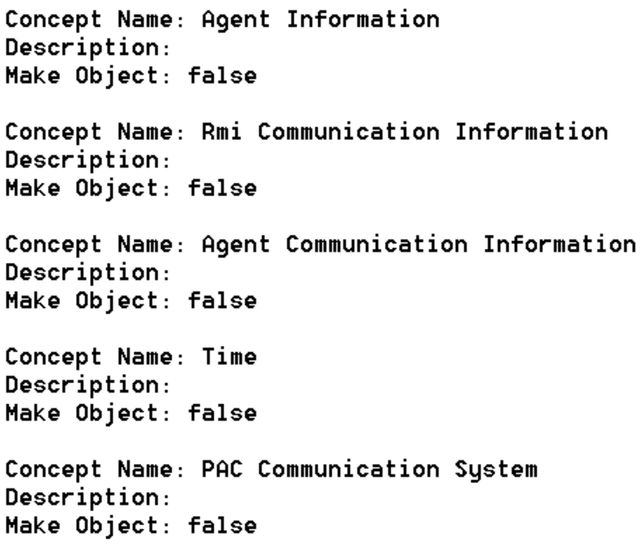
Figure 16. Link Properties Dialog

Saving a Concept Map

You can save a concept map by selecting **Save** under the **File** menu. You can then close the Concept Mapper and return to it at a later time. The save action will save the concept and links as well as their locations relative to each other.

Saving the Concept Map to a File

The concept map can be saved to a text file by selecting **File → Generate Printable**. Selecting this menu item will bring up a file dialog for saving a concept map to a file. By default, the directory is set to the current working directory, and the filename is set to `concept-map-name.txt`. The text file that is generated will contain a text description of all of the concepts in the concept map. Currently, no information is printed about concept links. Figure 17 shows the contents of the DefaultOntology concept map.



```
Concept Name: Agent Information
Description:
Make Object: false

Concept Name: Rmi Communication Information
Description:
Make Object: false

Concept Name: Agent Communication Information
Description:
Make Object: false

Concept Name: Time
Description:
Make Object: false

Concept Name: PAC Communication System
Description:
Make Object: false
```

Figure 17. Concept Map Generate Printable Output

Clearing a Concept Map

If you wish to delete everything on the current concept map and start again, you can do so by selecting **Clear** under the **Map** menu or right-clicking on an unoccupied portion of the canvas and selecting **Clear** in the pop-up menu. Note that this removes all concepts and links that have been entered. AgentBuilder will warn you by displaying a confirmation dialog window. Select **Yes** if you wish to clear the current concept map or **No** if you decide not to clear the canvas. Note that if you later decide you like the previous concept map and have not saved the current concept map since clearing it, you can close the Concept Mapper and not save the changes. However, if you save after clearing the concept map, there is no way to revert to the previous concept map.

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tools are open. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be displayed.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the Concept Mapper

You can close the Concept Mapper by selecting the **Close** item in the **File** menu. If necessary, AgentBuilder will ask you whether you want to save your changes. In selecting **Yes**, you will save all changes since the previous save. Selecting **No** will revert to the concept map configuration last saved.

Exiting AgentBuilder

To exit AgentBuilder, select the **Exit** item from the **File** menu. AgentBuilder will then display a dialog asking whether you are sure you want to exit. If so, then click on **Yes** button; if you do not wish to exit, click on the **No** button.

Object Modeler

The Object Modeler tool is used to describe an object model in an ontology. The Object Modeler provides a canvas where you draw figures representing classes and links representing the relationship between classes. The icons in the Object Modeler represent classes that will be instantiated by the agent to store domain information or perform actions in the domain. The links are used to show the relationships between classes. The classes defined in the object model can be mapped automatically into Java source files, which can then be compiled into Java class files (some source files may require editing, the Object Modeler cannot provide the code for user-defined methods). Note that all Project Accessory Classes (PACs) must first be defined in the Object Modeler before they can be imported into the PAC Editor (See “PAC Editor” on page 92). The Object Modeler is used to define the data attributes and the APIs of all classes that will be used in the agent's mental model. Java class files can be directly imported into an object model, thus saving you the effort of specifying the API for classes that are already coded.

The Object Modeler uses the notation from the Unified Modeling Language (UML). While UML method defines a very rich set of notation for modeling large complex systems, this tool implements a subset of the complete notation. See the UML Resource Center at `http://www.rational.com/uml` for more information.

Overview

The **Object Modeler** contains a menu bar with five items: **File**, **Edit**, **Map**, **Windows** and **Help**. The **File** menu allows you to open previous object models as well as save the current models. The **File** menu also allows you to import class files and update the objects. You can also **Print** the object model to obtain a hard copy. The **Edit** menu allows you to **Cut**, **Copy**, **Paste**, and **Delete** the selected class or link. The **Map** menu allows you to modify the current object model by adding new class or links, or to clear the entire drawing canvas. The **Windows** menu allows you to quickly and easily switch between the various open AgentBuilder tools. The **Help** menu gives you access to the AgentBuilder help system. Figure 18 illustrates a model constructed with the Object Modeler.

The main feature of the Object Modeler is the canvas on which you can construct an arbitrarily complex structure of classes and relations between classes. Utilizing the menus discussed above as well as context-sensitive pop-up menus you can construct simple or complex object models as needed. In addition, the tool allows you to edit a classes' properties. It is easy to rearrange classes to sim-

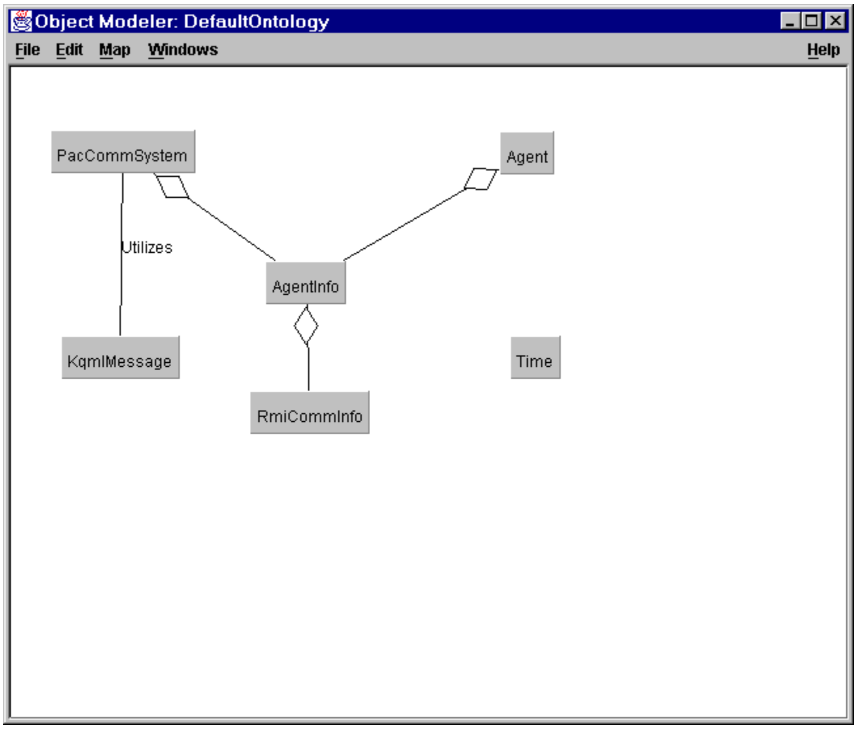


Figure 18. Object Modeler

plify the structure and minimize overlap of links. Three types of class links are supported: *generalization*, *aggregation*, and *association*. Each link has a different symbol that represents the type of relationship between the two connected classes.

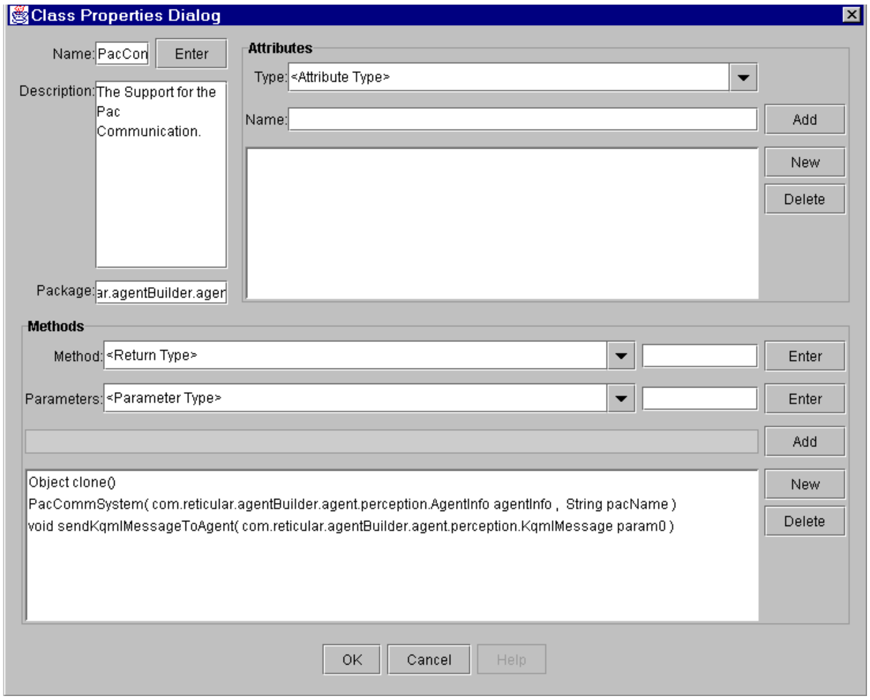
Operation

Creating a New Class

You can create a new rectangular figure representing the new class by right-clicking on an unoccupied region of the canvas and selecting **New Object** from the pop-up menu. This will cause Agent-Builder to display the dialog shown in Figure 19 in which you can enter a name for the class being represented, a description of the class, and method and attribute information. From this dialog you also specify the methods and attributes of the class. The methods are the API of the class. It is important for you to specify all constructors that might be used. It is only necessary to enter the attributes which you are going to read or write; private attributes that are not accessed need not be included. If the type you desire for an attribute is not in the **Types** pull-down menu, you can enter the class in the combo-box text area and then press the **Enter** key on the keyboard after typing the class name.

There are a couple of things to keep in mind when creating a new Class. A global class table keeps track of all classes in the ontologies. Therefore, all classes used in the object model need to specify their fully qualified class name in order to avoid ambiguity. Different objects cannot use the same name, even if they exist in different ontologies. Any classes used as an attribute or parameter must either be a primitive, basic Java type, or defined in the same object model. The basic Java types supported are arrays, hashtables, enumeration, and objects. By default, all classes created in the Object Model will implement the `Serializable` and `Cloneable` interfaces. The tool automatically generates read/write routines for each attribute. Each attribute specified must have read/write methods that follow the Java Bean convention. The Java Bean convention is specified more precisely in the Java literature and requires a "set" and "get" routine for each attribute. For example, if you specify an attribute named price, the tool will generate the methods setPrice and getPrice.

Clicking the **OK** button will create a new rectangular figure on the canvas representing the new class. Note that the point where you right-click is where the figure will be drawn. If you decide not to create this new class, you can press the **Cancel** button and the



Name: PacCon

Description: The Support for the Pac Communication.

Package: ar.agentBuilder.ager

Name:

```
Object clone()
PacCommSystem( com.reticular.agentBuilder.agent.perception.AgentInfo agentInfo , String pacName )
void sendKqmlMessageToAgent( com.reticular.agentBuilder.agent.perception.KqmlMessage param0 )
```

Figure 19. Class Properties Dialog

object model will remain unchanged. You can also use the **Map** menu in order to create a new class in an identical manner.

Cutting, Copying and Pasting a Class

Classes can be cut, copied, or pasted. There are two ways to use the clipboard functions for a class. The first way is to use the **Edit** menu's **Cut**, **Copy** and **Paste** menu items. The other method is to use the class's pop-up menus for cut and copy, and the map's pop-up menu for paste. Whichever method is used, a class must be selected before a cut or copy operation. If the model being pasted into already contains the name of the class being pasted, the class to be pasted will recursively have **CopyOf** prepended to its name.

Viewing and Altering the Properties of an Existing Class

You can view and modify the name and description of an existing class by right-clicking on the desired class. You can then select the **Properties** item from the pop-up menu. AgentBuilder will then display a dialog which will allow you to see and alter the name and properties of the selected class. By clicking on the **OK** button you can enter the changes; clicking on the **Cancel** button will revert to the original unaltered properties.

Creating a New Link

You can create a new link by right-clicking on any unoccupied region of the canvas and selecting between generalization, aggregation and binary association link types. Once again, this can also be initiated by selecting one of the link types from the **Map** menu.

Associations. A group of classes with common structure and common semantics are *associates* of one another. For example, an association link could be *Joe Smith works-at IBM*. An association is something like, *person works-at company.* Associations are always bi-directional and there can be rolenames given for each direction. Also cardinality (UML calls this multiplicity) can be denoted as well as a specific ordering of objects of a Class.

Aggregation (Whole-Part relationship). This relationship relates an assembly class to one component class. For example: Engine is-part-of Car. The aggregation relationship is transitive and it is possible to use the aggregation relationship recursively.

Generalization (Inheritance). Generalization is a relationship between a class (superclass) and one or more refined versions of it (subclass) who share attributes and operations. These are denoted in the superclass from which the subclasses inherits. A subclass may override features defined in a superclass like methods of operations and values of attributes. In this way the subclass is an extension and a restriction of the superclass. A class can inherit from one other class (single inheritance) as well as from several other classes (multiple inheritance). It is mentioned that this can cause conflicts among attributes or methods with the same names that are defined in the different superclasses.

Once you have selected one of the link types, the cursor will change to a cross-hair and you can then click-and-drag on the initial class and join it to the other desired class. Note that the order of connection does matter. The aggregation and generalization links are directed and need to be selected in the appropriate manner: For generation, the first class clicked on is the parent class; for aggregation, the first class must be the class that *contains* the second.

Moving an Existing Class

You can freely move a class anywhere on the canvas that you desire. Do this by clicking on the desired class and dragging the class to the new location. Links will adjust themselves in order to maintain the connection with the moved class.

Moving Multiple Classes

You can move multiple classes in a single operation. First, select the class you wish to move. To do this, hold the **Control** key down and select the classes with the mouse. Once you have a groupof selected classes and while holding the **Control** key down and dragging one of the selected classes to a new location. You will notice that all the selected classes move with the mouse cursor. To deselect the classes, simply click on the canvas.

Deleting an Existing Class

You can delete an existing class by clicking on that class (e.g. selecting it) and then right-clicking and selecting the **Delete** item from the pop-up menu. Likewise, a selected class can be deleted using the **Delete** item in the **Edit** menu. Note that when an class is deleted, all links to that class are also deleted.

Deleting an Existing Link

You can delete an existing link either by selecting a link (a selected link is red in color) and then right-clicking and selecting **Delete** from the pop-up menu or using **Delete** in the **Edit** menu.

Changing the Link Type of an Existing Link

You can change the properties of a link by selecting that link and right-clicking to bring up the pop-up menu. You can then select the link (turning it red) and change the link type by selecting one of the types from the **Link Type** hierarchical menu.

Viewing and Altering the Properties of an Existing Link

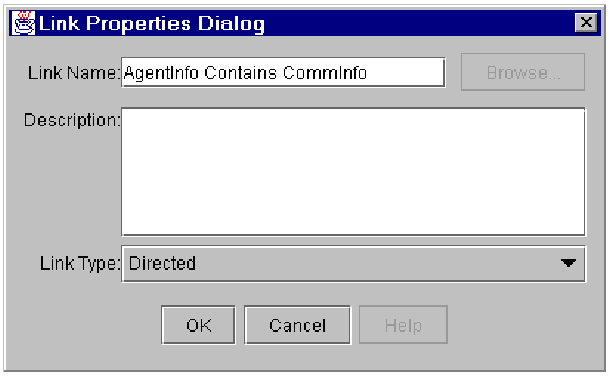
You can view and modify the link label, description and type of a selected link by right-clicking on the selected (red) link. You can then select the **Properties** item from the pop-up menu. AgentBuilder will then display the **Link Properties** dialog that will allow you to see and alter the name, description, and link type of the selected link. Figure 20 shows this dialog for viewing and editing link properties.

Saving a Object Model

You can save an object model by selecting **Save** in the **File** menu. You can then close the Object Modeler and return to it at a later time. The Save action will save the classes and links as well as their locations relative to each other.

Saving the Object Model to a File

The object model can be saved to a text file by selecting **File → Generate Printable**. Selecting this menu item will display a file dialog for saving the object model to a file. By default, the directory is set to the current working directory, and the filename is set to `object-model-name.txt`.The text file that is generated will contain a text description of all of the classes in the object model. Currently, no information is being printed about class links. “Default Ontology Object Model (Printable)” on page 319 shows the contents of the DefaultOntology object model.


Link Properties Dialog

Link Name: AgentInfo Contains CommInfo

Description:

Link Type: Directed

Figure 20. Link Properties Dialog

Clearing an Object Model

If you wish to delete everything from the current object model and start over, you can do so by selecting **Clear** under the **Map** menu. Note that this will remove all classes and links that have been entered. AgentBuilder will ask you to confirm this by displaying a confirmation dialog window. You can either select **Yes** if you are sure that you want to clear the current object model, or select **No** if you decide not to clear the canvas. Note that if the you later decide you want to use the previous object model and have not saved the current object model since clearing it you can close the Object Modeler and not save the changes. However, if you **Save** after clearing the object model, there is no way to revert to the previous object model.

Importing Class Files

You can import class files into your object model. This allows you to automatically create classes in your object model by simply importing selected classes. Figure 21 shows the Object Modeler's **Import Dialog**.

The **Import Dialog** is accessed through the **File → Import Class Files** menu item. To import class files, you must first enter the full package and class name in the class text field. The package name is required, because the Java reflection API can only look up fully qualified class names. Once the package name has been entered into the **Package** text field, the class name must be entered into the **Class** text field.

To add the class to the list, you must either press the **Enter** key while the cursor is inside the **Class** text field, or click on the **Add** button. The new item in the list will be a fully qualified class name. As a convenience, the contents of the **Package** list contains the package names for the classes already defined; this makes it easy to enter additional classes from those packages.

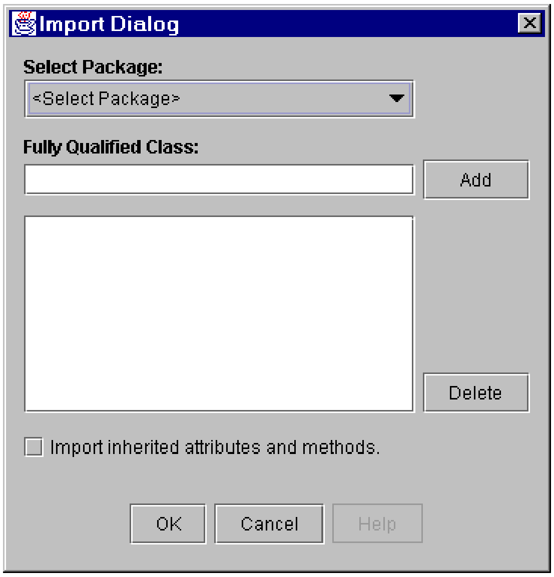
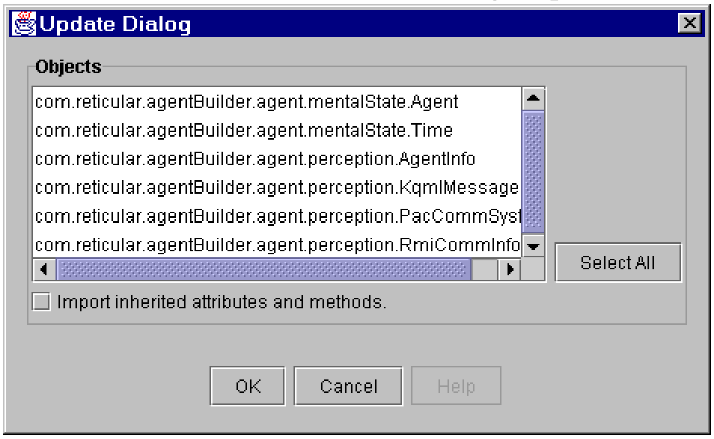


Figure 21. Object Modeler Import Dialog

Once you have finished adding classes to the list, you have the option of importing the inherited attributes and methods by clicking on the checkbox labeled **Import inherited attributes and methods**. Keep in mind that any class being used as a parameter or return value must be a primitive, basic Java type (arrays, hashtables, enumeration, and objects), or defined in the same object model. If this is not the case, a warning dialog will be displayed with a list of undefined classes. You can now click on the **OK** button to import the classes. Any classes that are not included in your classpath will generate a warning and will not get imported into the object model. Clicking on the **Cancel** button will cancel importing classes.

Updating Objects

If at any time the class files for the objects defined in the object model are modified, you will need to recompile the class and update them in the Object Modeler. To update the objects in the object model, you need to select the **Update Objects** menu item from the **File** menu. This will bring up the **Update Dialog**, see Figure 22. You will see the list of objects defined in the object model. You will need to select the objects that you wish to update. The **Select All** button will select all classes listed in the objects pane. Once you have selected the classes to update, you must click on the **OK** button to update the classes. If the new changes of the object include a class that's not defined, it will generate a warning. If you click on the **Cancel** button, no classes will get updated.



com.reticular.agentBuilder.agent.mentalState.Agent
com.reticular.agentBuilder.agent.mentalState.Time
com.reticular.agentBuilder.agent.perception.AgentInfo
com.reticular.agentBuilder.agent.perception.KqmlMessage
com.reticular.agentBuilder.agent.perception.PacCommSyst
com.reticular.agentBuilder.agent.perception.RmiCommInfo

Import inherited attributes and methods.

Figure 22. Update Dialog

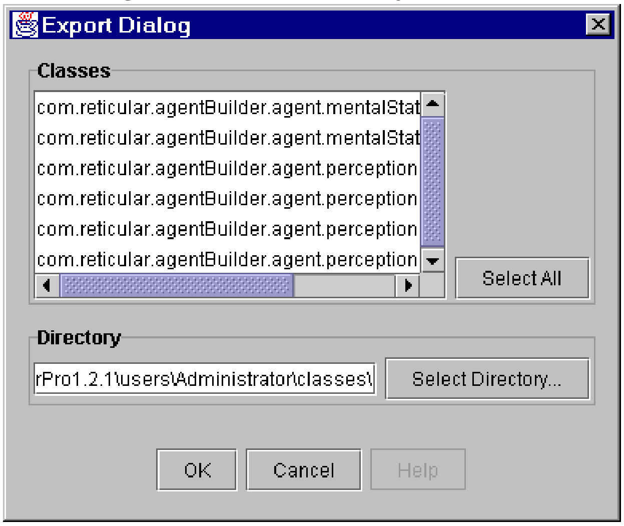
Listing Undefined Classes

Once you have imported objects into the object modeler, you can easily view which classes, if any, need to be defined using the **Undefined Types** dialog. You won't be able to import any objects into the PAC Editor if they contain undefined types.

To access the **Undefined Types** dialog, you need to select the **File** → **List Undefined Classes** menu item. The dialog will provide a list of types that are undefined and point to the class objects using them. Once you have defined all types in the current object model, selecting the **File → List Undefined** menu item will bring up a different dialog informing you that all types have been defined.

Exporting Java Files

Once your object model contains classes, you can export the classes into Java files. Figure 23 shows the Object Modeler's **Export Dialog**.



com.reticular.agentBuilder.agent.mentalStat
com.reticular.agentBuilder.agent.mentalStat
com.reticular.agentBuilder.agent.perception
com.reticular.agentBuilder.agent.perception
com.reticular.agentBuilder.agent.perception
com.reticular.agentBuilder.agent.perception

rPro1.2.1\users\Administrator\classes\

Figure 23. Object Modeler Export Dialog

The **Export Dialog** is accessed through the **File → Generate Java Files** menu item. To export Java files, you must select the classes you want to export. You can make your selections in one of three ways. If you just want to select a single class, you can do so by selecting the class from the list using the left mouse button. To select multiple classes, hold down the **Control** key while selecting classes with the left mouse button. To select all of the classes in the list, click on the **Select All** button.

Once the classes to be exported are selected, you must specify a directory to which the Java files will be exported. The default directory is your current working directory. To change the export directory, click on the **Select Directory** button. Clicking on the **Select Directory** button will display the **Directory Dialog** shown in Figure 24.

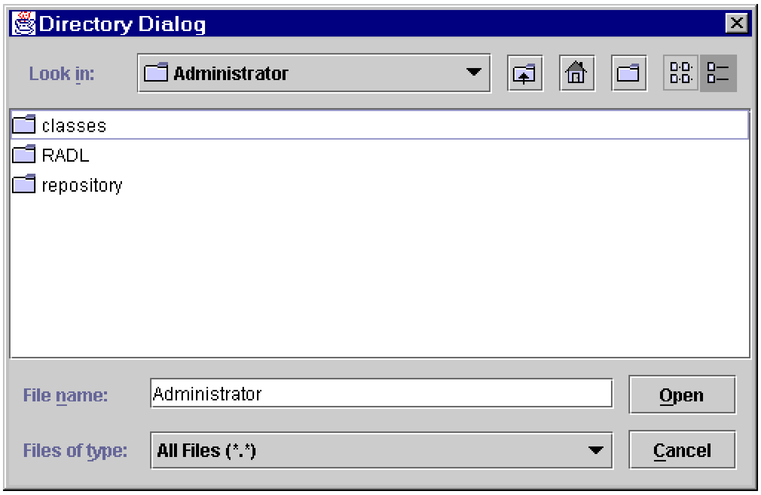


Figure 24. Directory Dialog

Once a directory has been specified, you can click on the **OK** button to export the selected classes into Java files. Alternatively, you can click on the **Cancel** button to cancel the operation. After the selected classes have been exported, Java files will appear in the directory that you specified. The file names will be the class names with the `.java` extension appended to the name.

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tools are open. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be brought to the top of the display.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the Object Modeler

You close the Object Modeler by selecting the **Close** item in the **File** menu. If necessary, AgentBuilder will query whether you want to save your changes. Selecting **Yes** will save all changes since the previous **Save**. Selecting **No** will revert to the object model configuration last saved.

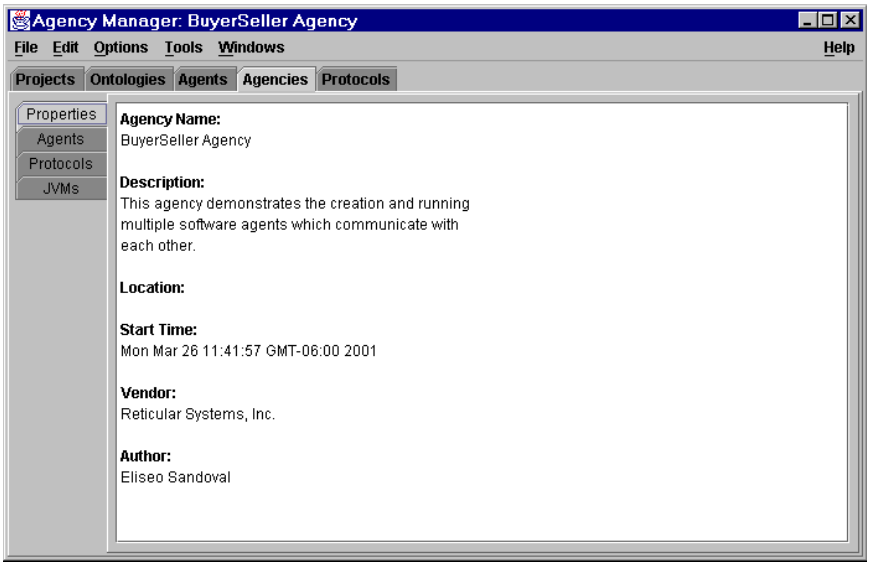
Exiting AgentBuilder

To exit from AgentBuilder, select the **Exit** from the **File** menu. AgentBuilder will then display a dialog asking whether you are sure you want to exit. If so, then click on **Yes** button; if you do not wish to exit, click on the **No** button.

D. Agency Manager

Note: The Agency Manager is part of the AgentBuilder Pro product. The Agency Manager is not provided with the AgentBuilder Lite product. If you are an AgentBuilder Lite user and want to use the tools that are provided with the Agency Manager please contact Acronymics, Inc. for an upgrade to AgentBuilder Pro.

The Agency Viewer allows you to view and run agencies. Using the Agency Manager, you can quickly view an agency's properties, agents, protocols and JVM groups. There are two main tools in the Agency Manager that help you create agents that communicate, cooperate, and negotiate with each other: the Agency Viewer and the Role Editor. Figure 25 shows the Agency Manager panel.



Agency Name:
BuyerSeller Agency

Description:
This agency demonstrates the creation and running multiple software agents which communicate with each other.

Location:

Start Time:
Mon Mar 26 11:41:57 GMT-06:00 2001

Vendor:
Reticular Systems, Inc.

Author:
Eliseo Sandoval

Figure 25. Agency Manager

Agency Manager

The **File** menu allows you to open other agencies, save an agency, create new agents, import protocols, update protocols, create JVM groups, and assign agents to JVM groups. The **File** menu also allows you to exit AgentBuilder. The **Edit** menu allows you to cut, copy, paste, and delete certain properties from the agency. The **Agency Properties** dialog can also be displayed using the **Edit** menu. The **Options** menu allows you to modify the display options for the Agency Manager. The **Tools** menu gives you access to the other agent tools. The **Windows** menu lets you switch between other AgentBuilder tool windows that are open. The **Help** menu gives you access to the AgentBuilder help system.

Using the Agency Manager

Using the Tabbed Pane

You can quickly view an agency's properties, agents, protocols, and JVM groups by using the Agency Manager's tabbed pane. The **Properties** tab displays the agency's name, description, ontologies, location, start time, vendor, and author. The **Agents** tab provides a list of agents that belong to the agency. The agent's properties can be displayed in the description area by selecting an agent. The **Protocols** tab provides a list of protocols that belong to the agency. The protocols name, description and ontologies are displayed when a protocol is selected. The JVM tab provides a list of JVM groups in which agents are assigned. The list of agents assigned to a JVM group is displayed when selecting a JVM group.

You can also modify the placement of the tabs. The **Options** menu contains the **Tab Placement** menu. The **Tab Placement** menu allows you to set tab placement at the top, bottom, right, or left side of the main window.

You can delete items in the list by using the **Edit** menu's **Delete** function. The **Delete** function will delete the selected list item after user confirmation.

List items can also be cut, copied, and pasted within the same agency and between different agencies. To do this, first select a list item from one of the tabbed panes. Then select **Cut** or **Copy** from the **Edit** menu. Before pasting the item in the clipboard, you must decide if you also want the clipboard item to be pasted into another agency. The tool automatically ensures that the list items are only inserted to the appropriate lists. For example, agents can only be pasted into agent lists. When you are ready to paste the clipboard item, select the **Paste** menu item from the **Edit** menu. If the paste is allowed, the clipboard item will be added to the list in the tabbed pane. List items that belong to the **Protocols** tab cannot be copied or pasted into protocol lists.

Opening an Agency

To open an existing agency, select **File → Open…** menu item from the Agency Manager menu bar. The **Open Agency** dialog (Figure 26)will be displayed with a list of agencies that exist for the current repository being used. To select an agency from the list, you can either double-click on the agency, or select an agency from the list and click on the **OK** button. Clicking on the **Cancel** button will cancel opening the agency.

Saving the Current Agency

If the agency has been modified in any way, you must save the current agency information in order to make the changes permanent. Changes that require saving include modifying the agency's properties in the Agency Properties dialog and adding or deleting list items from any of the tabbed panes. To save the current agency, you select the **File → Save** menu item from the **Agency Manager** menu bar.

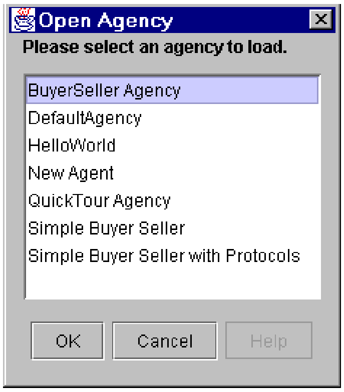


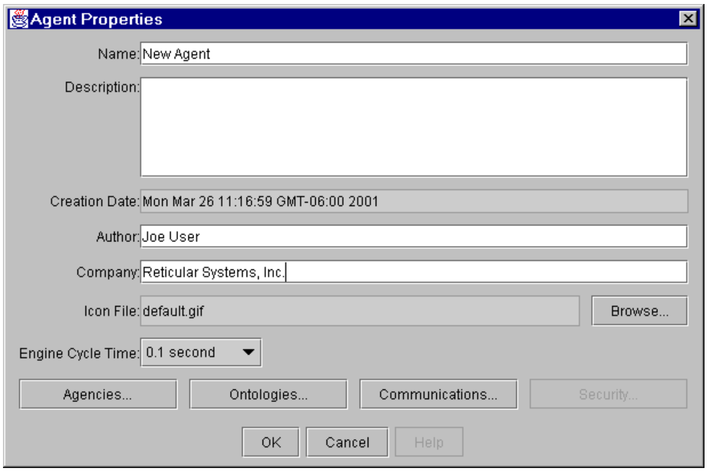
Figure 26. Open Agency Dialog

Creating a New Agent

To create a new Agent using the Agency Manager, you need to select the **New Agent…** menu item from the **File** menu. The **Agent Properties** dialog will be displayed as shown in Figure 27. This dialog will allow you to enter the new agent's name, description, author, company, icon file, engine cycle time, agencies, ontologies, and communications properties. The creation date is set automatically, and is therefore a read-only text field (See “Creating a New Agent” on page 69 for more information on creating new agents).

Importing a Protocol

In order to import an existing protocol, you will need to select **File → Import Protocols…** menu item from the **Agency Manager** menu bar. The **Import Protocol** dialog will be displayed with a list of protocols that exist for the current repository being used. To select a protocol from the list, you can either double-click on the protocol, or select a protocol from the list and click on the **OK** button. Clicking on the **Cancel** button will cancel the **Import Protocol** dialog operation.



Name: New Agent

Description:

Creation Date: Mon Mar 26 11:16:59 GMT-06:00 2001

Author: Joe User

Company: Reticular Systems, Inc.

Icon File: default.gif

Engine Cycle Time: 0.1 second

Figure 27. Agent Properties Dialog

Updating Protocols

To update the protocols in the agency, you need to select the **Update Protocols** menu item from the **File** menu. This will bring up the **Update Protocols Dialog**, see Figure 28. You will see a list of protocols defined in the current agency. You will need to select the protocols that you need to update. The **Select All** button will select all protocols listed in the protocols pane. Once you have selected the protocols to update, you must click on the **OK** button to update the protocols.

Creating a JVM Group

A JVM group is a set of agents from the agency that will run inside the same Java Virtual Machine (JVM). You can start the group of agents from the AgencyViewer or from the Project Manager. If an agent doesn't belong to a JVM group, it will run in it's own JVM. To create a JVM group, select **File →Create JVM…** menu item from the **Agency Manager**. The **Create JVM Dialog** (Figure 29) will prompt you for a name for the JVM group. Once you have entered

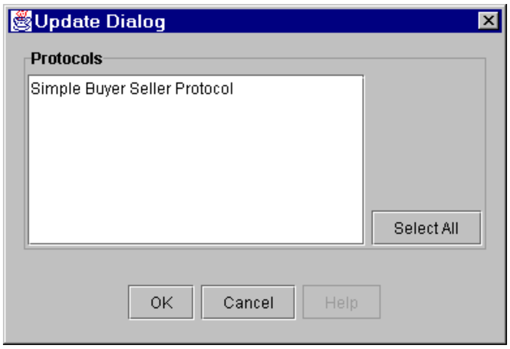


Figure 28. Update Protocols Dialog

the name and click on **OK** button, the new JVM group will get added to the JVM list. The following section explains how to add agents to your new JVM group.

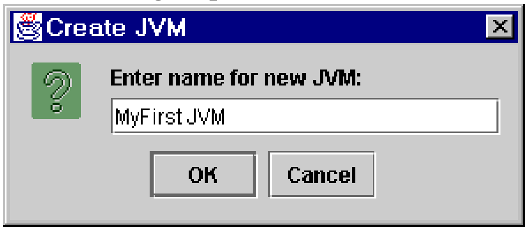


Figure 29. Creating a JVM Group

Assigning Agents to a JVM Group

In order to assign agents to a JVM group, you need to select **File → Assign JVM…** menu item from the Agency Manager. This will bring up the **Assign Agents Dialog**, see Figure 30. You will see a list of available agents in the **Available Agents** panel. Before you add agents to a group, you need to select a JVM group from the drop down list in the **Selected Agents** panel. You can add a single agent by selecting the desired agent and clicking on the add button, or you can add all the agents to the group by clicking on the **Add All** button.

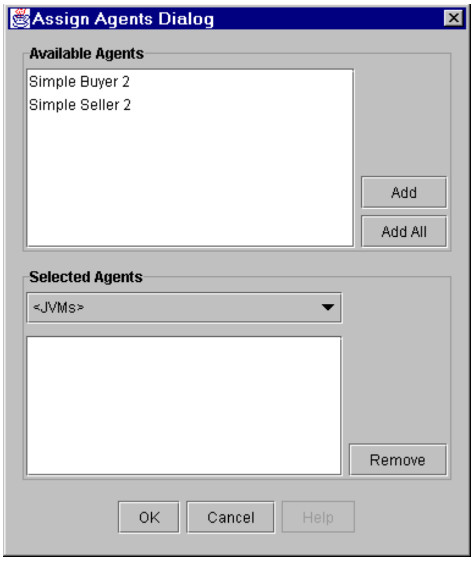


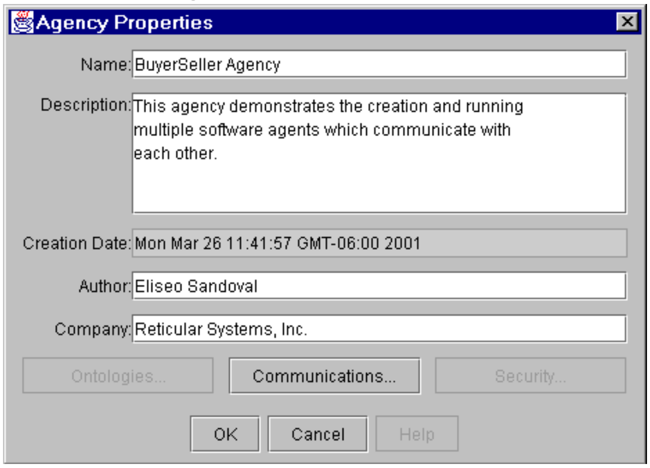
Figure 30. Assigning Agents to a JVM Group

If the **Available Agents** panel contains no agents, that means that all agents already belong to a group, or the agency contains no agents. If the first case is true, you can remove an agent from a JVM group and add it to another JVM group. You can do this by selecting the JVM group the agent belongs to and clicking on the **Remove** button from the **Selected Agents** panel. This will remove the agent from the JVM group and add it to the **Available Agents** panel. You can then add the agent to the desired group using the method described above.

Viewing/Editing Agency Properties

The agency's properties get displayed in the description area when you click on the **Properties** tab in the tabbed pane. To modify any information from the agency properties, you must select the **Edit →**

Properties… menu item from the **Agency Manager** menu bar. This action will display the **Agency Properties** dialog, where you can make your changes. The **Properties** tab must be selected in order to display the **Agency Properties** dialog. In order to apply your new changes, click on the **OK** button. The **Cancel** button will cause the system to ignore any changes that were made. Figure 31 shows the **Agency Properties** dialog.



Name: BuyerSeller Agency

Description: This agency demonstrates the creation and running multiple software agents which communicate with each other.

Creation Date: Mon Mar 26 11:41:57 GMT-06:00 2001

Author: Eliseo Sandoval

Company: Reticular Systems, Inc.

Figure 31. Agency Properties Dialog

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tool windows are open. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be brought to the front on your display.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read help information for the current tool. The **Index** menu item will display an index of the help system's contents. The **Tutorial** menu item will display a Quick Tour of the AgentBuilder Toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Exiting AgentBuilder

The **File** menu's **Exit** menu item should be selected to exit AgentBuilder. This will close all of the tools that are currently open. If you have modified the agency properties, you will be given an opportunity to save the agency's properties before exiting the system.

Agency Viewer

The Agency Viewer tool is used to run a set of agents belonging to an agency. The **Agency Viewer** provides a pane for viewing the icons that represent the agents. You are able to drag the icons to a new position, or change the agent's icon. Once the agency is running, you can view agent communications in two ways. First, notice that when two agents are communicating, a line is drawn between the two agents with a small ball moving along this line. The line represents the connection being made between agents, and the ball represents the message being delivered to the agent. You can also examine agent messages in a text area at the bottom of the panel. The message text area displays the sender and receiver of the message, as well as for the content of the message. Figure 32 shows the **Agency Viewer**.

Overview

The Agency Viewer contains a menu bar with six items: **File**, **Edit**, **Exec**, **Options**, **Windows**, and **Help**. The **File** menu allows you to

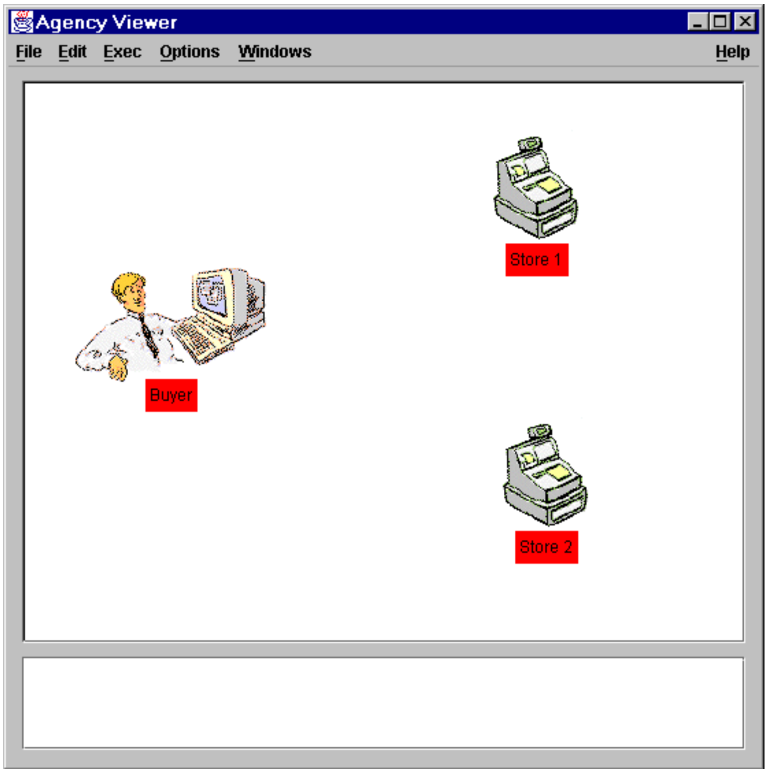


Figure 32. Agency Viewer

save an agency, create a message log, and save/open runtime messages. The **Edit** menu gives you access to the agency properties and the agent properties dialogs. The **Exec** menu allows you to start, pause, stop, and reset the agents. The **Options** menu allows you to turn on or off message text area, display the agent status window, and specify the message buffer size.

During each run, agents switch to different states: *Stop*, *Registering*, *Registered*, *Running*, and *Paused*. The agent's icon label changes colors according to the state it is in. The red label signifies the agent is stopped. The pink label means the agent is in the process of registering with the agency. The yellow label means the agent successfully registered with the agency. The green label means the agent is currently running. The cyan label indicates that the agent is paused.

Operation: Running an Agency

Setting Agency in Register Mode

In order to run agents in the Agency Viewer, the agents must first register with the agency. In order for the agency to register agents, the Agency Viewer needs to be set to **Register Mode**. You can do this by selecting **Exec → Register Mode** menu item from the Agency Viewer's menu bar.

Registering the Agents

Once the Agency Viewer is in register mode, you must start the agents in agency-mode. There are several ways to start running an agent. The agents can be started individually by right clicking on an agent's icon and selecting **Run** from the resulting pop-up menu. You can also start all agents at once using the **Agency Viewer** by selecting the **Run All** menu item from the **Exec** menu bar. If you select the **Run All** menu item, the Agency Viewer will run only the agents that are set to run in the current host. If an agent has been set to run on a different host, the Agency Viewer will not run it and will display a warning message. You can start this agent from the Agent Engine; make sure the **-agency-mode** flag is turned on so that it registers with the agency. (see “Run-Time System” on page 251.). If any JVM groups are specified for the agency, it will use it to run multiple agents in the same Java Virtual Manchine. This will only occur when you select the **Run All** menu item from the Agency Viewer. If the agents are started within the Agency Viewer tool, the **agency-mode** flag is set automatically. When the agent's label changes to the color yellow, it means it has registered successfully.

Running and Resetting agents

Once the agents have registered, you can start running the agents from the Agency Viewer by selecting the **Exec→ Begin** menu item. The Agency Viewer will then start the engines for each registered agent. The agency can still register agents when it is running (i.e., in run mode). The agent's label will change to green when the agent starts running.

Agents can only be reset if they are either *running* or *paused*. To reset an agent, right-click on the agent's icon and select the **Reset** menu item. If you want to reset the entire agency, select the **Exec → Reset** menu item from the Agency Viewer's menu bar.

Pausing and Unpausing the agents

Whenever the agents are running in agency-mode, you can *pause* and *unpause* them. There are two ways of pausing the agents. You can pause an individual agents by right-clicking on the agent and selecting **Pause** from the pop-up menu. You can also select the **Pause** menu item from the **Exec** menu bar. If you select **Pause** from the menu bar, it will cause all running agents to pause. You can use the same method to unpause the agents. Selecting **Unpause** from the menu bar will unpause any agents that have been paused. Any agent with a cyan color label is in pause mode.

Displaying the Agent's Message History Dialog

You can view the messages that each agent has sent or received by displaying the **Message History** dialog (Figure 33). To view the **Message History** dialog, you need to right-click on an agent and select the **Message History** menu item. With this dialog, you can see two lists and a description area. The top list displays the name of the agents that have sent messages to this agent. The bottom list displays the names of the agents that this agent has sent messages. Clicking on any of the agent names will display the message in the description area. You can display the **Message History** dialog for more than one agent. To close the dialog, simply click on the **OK** button.

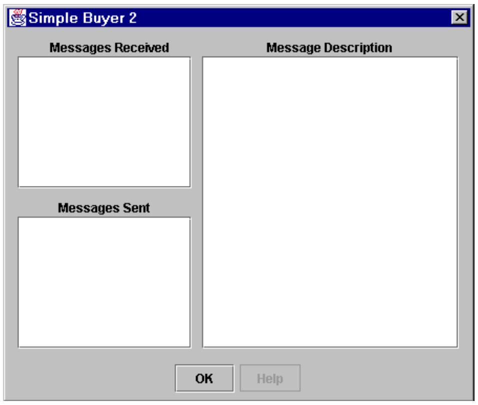
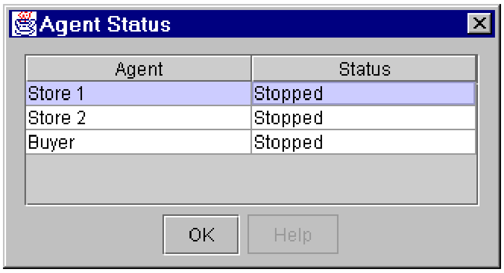


Figure 33. Message History Dialog

Displaying the Agent's Status Window

The agent status window provides you with a list of agents along with their current state. You can access the window by selecting **Options → Agent Status** from the **Agency Viewer** menu bar. Figure 34 shows the **Agent Status** dialog.



Agent	Status
Store 1	Stopped
Store 2	Stopped
Buyer	Stopped

Figure 34. Agent Status Dialog

Opening and Saving Runtime Messages

At the end of each run, you can save the runtime messages for later examination. There are two ways to save these messages. You can select **File → Save Run** and save the message to a file. (A default name has been provided with the following format: `date AgencyName RunNumber`. e.g. `11_11_98 NewAgency Run0`). You can also specify the file name by selecting **File → Save Run As** from the **Agency Viewer** menu bar.

Specifying Message Buffer Size

You can set the number of messages that are saved when running an agency by selecting the **Options →Message Buffer Size** menu item from the **Agency Viewer** menu bar. You can enter any number between 0 and 100,000. The default value for all the agencies is 1000. When the buffer size has reached its limit and a new message arrives, the oldest message will be discarded and the new one will be kept.

Creating a Runtime Message log

If you wish to log all messages sent to a text file when running the agency, select the **Message Log** menu item from the **File** menu bar. This will bring up the **Save File** dialog where you can enter the location and filename of the log file. Once you have selected the name of the file, click on the **Save** button. A check mark will appear in the **Message Log** menu item showing that all messages are currently being logged.

Handling High Volumes of Communication

If the agency you are building is creating a high volume of messages you may wish to speed up the AgencyViewer. There are several changes you can make to speed up the rate at which the tool handles messages. The first is to set the message buffer size to 0, see previous section for details. The second requires turning off the message trace at the bottom of the AgencyViewer window. To accomplish this, toggle the **Show Messages** menu item on the **Options** menu. You may also want to consider turning off all logging since logging generates a large amount of overhead that should be avoided when handling high message volumes.

Viewing and Altering the Properties of the Agency

You can view and modify the properties of the agency by selecting the **Properties** menu item in the **Edit** menu bar. The only restriction is that the agency name cannot be modified from the **Agency Viewer** tool, but the description, author, company, and communications can be modified. The only tool that allows you to change the agency name is the **Project Manager**.

Viewing and Altering the Properties of the Agent

To display the agent's properties, you can either click on the agent and select **Agent Properties** menu item from the **Edit** menu bar, or simply double-click on the agent's icon. The **Agent Properties** dialog will be displayed along with the current information for the selected agent. In order for your changes to be applied, you must click on the **OK** button. Clicking on the **Cancel** button will cause the system to ignore any changes that have been made. The agency will be saved if the name of an agent is modified.

Changing the Agent's Icon

To change the agent's icon, you will need to use the agent's **Properties Dialog** (See previous section for a description of opening the **Agent Properties** dialog). From the **Agent Properties** dialog, click on the **Browse** button to display the **Icon Dialog**. The **Icon Dialog** allows you to select an icon for the agent by clicking on an icon. The selected icon will be highlighted with a red border. Once you have selected the desired icon for the agent, click on the **OK** button. The **Cancel** button will close the Icon dialog without changing the agent's icon file. Figure 35 shows the Icon Dialog.

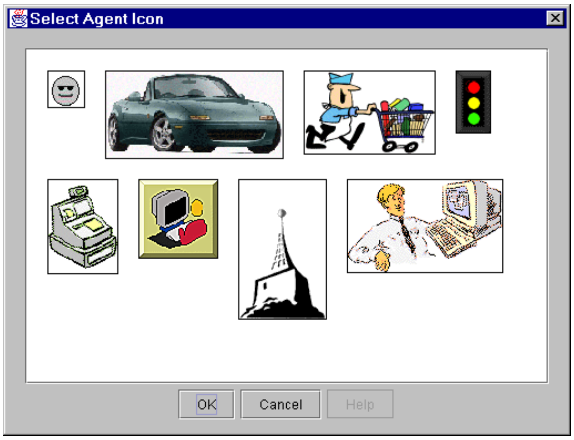


Figure 35. Icon Dialog

Saving Agency Properties

You can save the agency by selecting **Save** in the **File** menu. The save action will save the agents properties and agency properties, which include the current location in the panel and the message buffer size.

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are currently open. This menu facilitates switching between the tools when multiple tools are active. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be brought to the top of the window stack.

Accessing Help

The help system con be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item displays an index of the help system's contents. The **Tutorial** menu item displays the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item displays the AgentBuilder logo along with the version number and copyright information. Selecting the **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the Agency Viewer

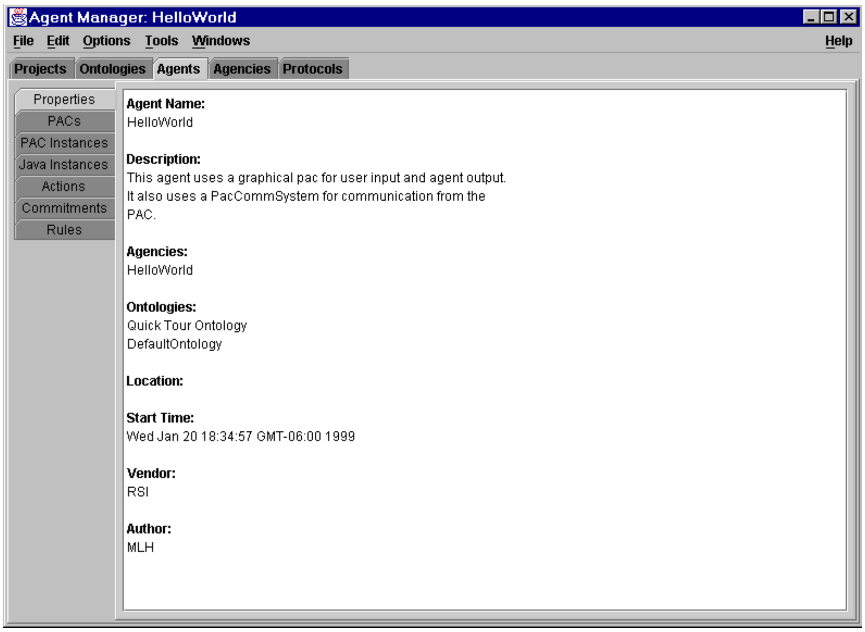
You close the **Agency Viewer** by selecting the **Close** item in the **File** menu. If necessary, AgentBuilder will query whether to save the changes made to the agency.

Exiting AgentBuilder

To exit AgentBuilder, select the **Exit** in the **File** menu. AgentBuilder will display a dialog asking whether you are sure you want to exit. If so, then click on **Yes** button; if you do not wish to exit, click on the **No** button.

E. Agent Manager

The Agent Manager allows you to view, edit, create, and run agents. Using the Agent Manager, you can quickly view an agent's properties, actions, beliefs, commitments, PACs and rules. Once an agent has been completely defined, the Agent Manager allows you to generate an agent's definition file and classes. Thereafter, you can run the agent from the Agent Manager. If you are starting from scratch, the Agent Manager allows you to create a new agent. Figure 36 shows the **Agent Manager** panel.



Agent Name:
HelloWorld

Description:
This agent uses a graphical pac for user input and agent output.
It also uses a PacCommSystem for communication from the
PAC.

Agencies:
HelloWorld

Ontologies:
Quick Tour Ontology
DefaultOntology

Location:

Start Time:
Wed Jan 20 18:34:57 GMT-06:00 1999

Vendor:
RSI

Author:
MLH

Figure 36. The Agent Manager

Overview

The **File** menu allows you to create a new agent, open an existing agent, save the current agent and generate a text file describing the agent. The **File** menu also allows you to exit the system. The **Edit** menu allows you to cut, copy, paste, delete or modify certain agent properties. The **Options** menu provides functions for executing the current agent and displays options for the Agent Manager. The **Tools** menu gives you access to the other agent tools. The **Windows** menu lets you switch between other AgentBuilder tools that are open. The **Help** menu gives you access to the AgentBuilder help system.

The tabbed pane in the main window allows you to quickly view the current agent's properties, PACs, PAC instances, Java instances, actions, commitments, and rules. Selecting a tab will change the panel accordingly.

Operation

Using the Tabbed Pane

The tabbed pane allows you to see information about the current agent. For instance, selecting the **Actions** tab will change the tabbed pane to display a list of defined actions and a description area. The description area is used to display information about the selected action. The **Properties** panel is the only exception to the list-description paradigm, since its panel only contains a description area.

You can also modify the placement of the tabs. The **Options** menu contains a **Tab Placement** menu. The **Tab Placement** menu allows you to change the tab placement to be on the top, bottom, right, or left side of the main window.

You can delete items in the list by using the **Edit** menu's **Delete** function. The **Delete** function will delete the selected list item after user confirmation. You can also launch an agent tool for a specific list item. For example, double-clicking on a list item in the **Commitments** panel will cause the commitment editor to be loaded for the current agent.

List items can also be cut, copied and pasted in the same agent and between different agents. To do this, first select a list item from one of the tabbed panels. Then select **Cut** or **Copy** from the **Edit** menu. Before pasting the item from the clipboard, you must decide if you also want the clipboard item to be pasted into another agent. The tool automatically ensures that actions are only pasted into action lists, rules are only pasted into rule lists, etc. When you are ready to paste the clipboard item, select the **Paste** menu item from the **Edit** menu. If valid, the clipboard item will be added to the list in the tabbed panel.

Creating a New Agent

To create a new agent, select the **New** function from the **File** menu. You will be presented the **Agent Properties** dialog shown in Figure 37. When creating a new agent, the **Creation Date** field is automatically set and is therefore a read-only text field. The other text fields are editable and include a **Name** field, a **Description** area, an **Author** field, and a **Company** field.

In addition to the text fields and text area, the agent properties dialog also includes a combo box for the **Engine Cycle Time**, and buttons for **Agencies…**, **Ontologies…**, **Communications…**, and **Security…**. The **Security…** button is currently not implemented. If no agencies or ontologies exists then the **Agencies…** and/or **Ontologies…** button will be disabled.

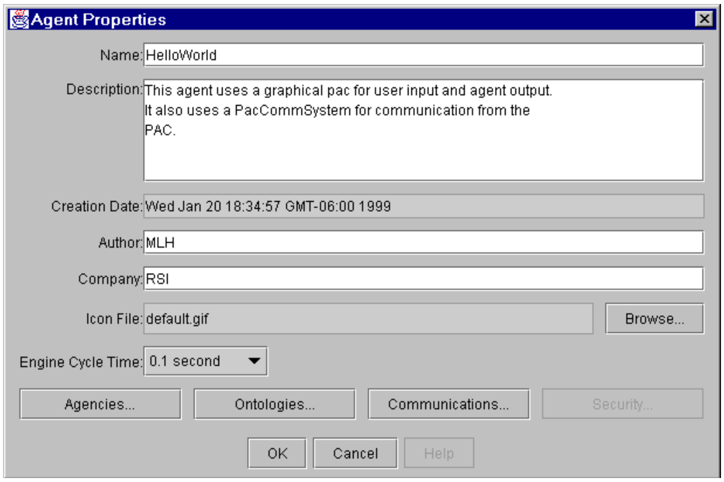

Agent Properties
Name: HelloWorld
Description: This agent uses a graphical pac for user input and agent output.
It also uses a PacCommSystem for communication from the
PAC.
Creation Date: Wed Jan 20 18:34:57 GMT-06:00 1999
Author: MLH
Company: RSI
Icon File: default.gif
Browse...
Engine Cycle Time: 0.1 second
Agencies...
Ontologies...
Communications...
Security...
OK
Cancel
Help

Figure 37. Agent Properties Dialog

The **Engine Cycle Time** combo box allows you to modify the cycle time when the agent is running in the runtime agent engine (See "Run-Time System" on page 251). The **Browse** button allows you to change the agent's icon file. The **Icon Dialog** allows you to select an icon for the agent by clicking on an icon. The selected icon will be highlighted with a red border. Once you have selected the icon, click on the **OK** button to accept the new icon. The **Cancel** button will close the **Icon Dialog** without changing the agent's icon file. Figure 38 shows the **Icon Dialog**.

The **Agencies Dialog** allow you to associate a set of agencies with a particular agent. The dialog consists of a two column table. The left column lists all of the available agencies. To associate an agent with a particular agency, select the checkbox in the right column. A check indicates that the agent is associated with a particular agency. When finished with the Agencies dialog, you must select the **OK** button for the selections to be registered. Selecting the **Cancel** button will ignore any selections that have been made. Figure 39 shows the **Agencies Dialog**.

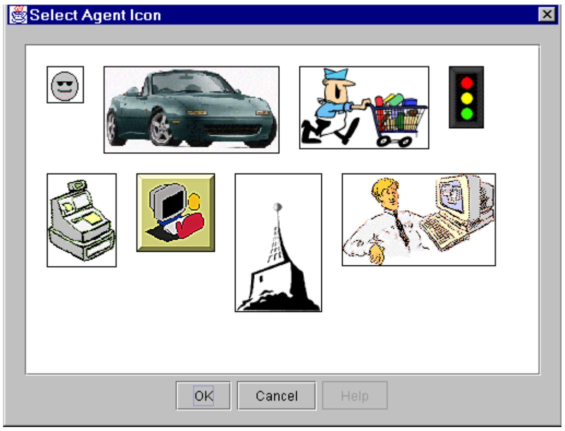
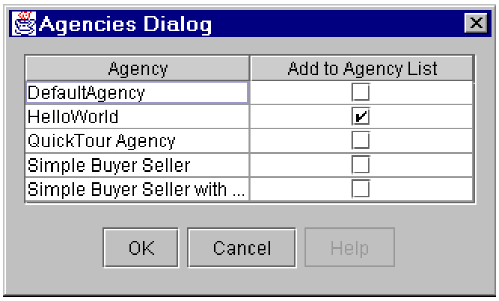


Figure 38. Icon Dialog

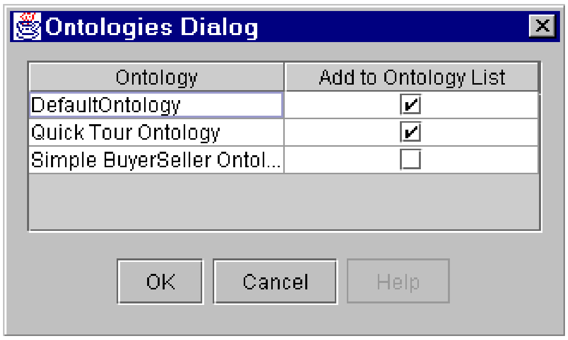


Agency	Add to Agency List
DefaultAgency	☐
HelloWorld	✓
QuickTour Agency	☐
Simple Buyer Seller	☐
Simple Buyer Seller with ...	☐

Figure 39. Agencies Dialog

The **Ontologies Dialog**, is similar to that of the **Agencies Dialog**, except that it is read only. Depending on which PACs the agent imports, the proper ontologies will be added automatically. The **Ontologies Dialog** is shown in Figure 40.

The **Communications Dialog** allows you to specify the communication types (e.g., RMI, sockets) available to the agent, and specify



Ontology	Add to Ontology List
DefaultOntology	✓
Quick Tour Ontology	✓
Simple BuyerSeller Ontol...	☐

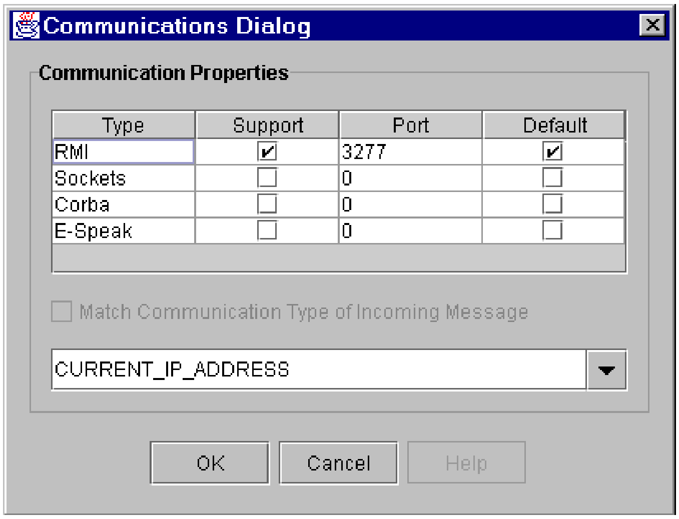
Figure 40. Ontologies Dialog

communication parameters such as IP address. You can enter the port number and IP address for the agent, or accept the default values provided. The default value for the IP address is set to the keyword CURRENT_IP_ADDRESS, this means the agent's IP address will be set to the current machine at the time the agent runs. The port number is set to a random number between 1,000 and 6,000. If you change the default values for the **Communications Dialog**, you must select the **OK** button in order for the changes to be registered. Selecting the **Cancel** button will cancel any changes that have been made. Figure 41 shows the **Communications Dialog**.

In order for you to successfully create a new agent, you must supply a name for the new agent and select at least one agency. When you are done entering information for the new agent, you must click on the **OK** button. Clicking on the **Cancel** button will cancel the creation of a new agent.

Opening an Agent

To open an existing agent, select the **Open** menu item from the **File** menu. You will be given a list of agents that exist for the current repository being used. To select an agent from the list, you can either double-click on the agent, or select an agent from the list and



Type	Support	Port	Default
RMI	✓	3277	✓
Sockets	☐	0	☐
Corba	☐	0	☐
E-Speak	☐	0	☐

CURRENT_IP_ADDRESS

Figure 41. Communications Dialog

click on the **OK** button. Clicking on the **Cancel** button will cancel the opening of another agent.

Saving the Current Agent

If the agent has been modified in any way, you must save the current agent information in order to make the changes permanent. Changes that require saving include modifying the agent's properties in the **Agent Properties** dialog and deleting list items in any of the tabbed panes. The **File** menu's **Save** item will save the current agent's information into the current repository's agent store.

Saving the Agent to a File

You can generate a text file describing the current agent. To do this, select **File → Generate Printable**. This will display a file dialog for saving the agent to a file. By default, the directory is set to the current working directory, and the filename is set to **agent-name.txt.** "Agent Description (Printable)" on page 327 shows the contents of the file generated from the Hello World agent. The file contains a text description of the agent's properties, including every PAC, PAC instance, JAVA instance, action, commitment and rule.

Viewing/Editing Agent Properties

You can easily view the agent's properties by clicking on the **Properties** tab in the tabbed pane. If you want to modify any information that is displayed in the properties panel, you must select the **Edit** menu's **Properties** menu item. You will be given a **Properties** dialog, from which you can modify the agent's properties. This is the same dialog used to create a new agent. In order for your changes to be applied, you must click on the **OK** button. Clicking on the **Cancel** button will cause the system to ignore any changes that have been made.

Generating the Agent Definition

When you have finished constructing an agent, you must generate an agent definition. Do this by selecting the **Generate Agent Definition** menu item from the **Options** menu. You will then be presented with the file dialog **Choose RADL File Location.** The **Choose RADL File Location** dialog is shown in Figure 42

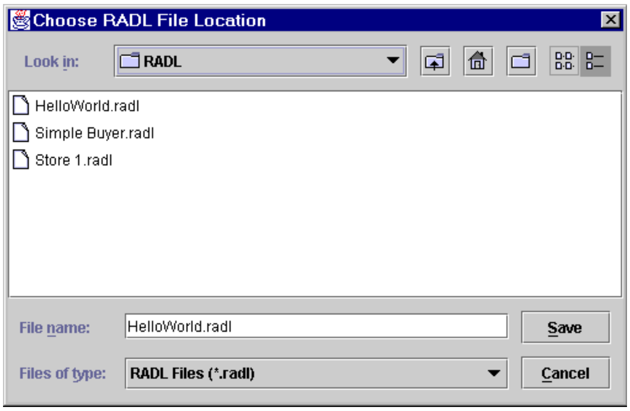
The file dialog allows you to select the name and location of your RADL file. The file dialog has several features worth noting. At the top of the dialog, there is a combo box that displays the current directory. The combo box allows you to go up in the directory hierarchy. The dialog also contains icons for going up to the next directory, returning to the home directory, and creating a new directory, respectively. In the center of the dialog is a pane for displaying the contents of the current directory. In this pane, you can double click directories to open them. At the bottom of the dialog, there is and editable text field where you can modify the default name of the RADL file.

Once you determine the name and location of the RADL file, you have three different ways in which you can make your file selection. The first way is to select a file in the file list which will overwrite the selected file. The second way is to press the **Enter** key on your keyboard while your cursor is in the text field displaying your current selection. The third way to register the file selection is to press the **Save** button when a file is specified in the **File** text field.

Once a RADL file is selected, the RADL file is generated for the current agent. The agent definition is then used by the Agent Manager to generate Java source files for the agent's PACs.

Running the Agent

With the agent definition and class files generated, you are now ready to run the agent. If you select the **Run Agent** menu item from the **Options** menu, you are first presented with a file dialog for selecting a RADL file to use. The **File Dialog** is the same dialog described above. Once a valid RADL file has been selected, the **Agent Engine Options** dialog shown in Figure 43 will be displayed.



HelloWorld.radl
Simple Buyer.radl
Store 1.radl

File name: HelloWorld.radl

Files of type: RADL Files (*.radl)

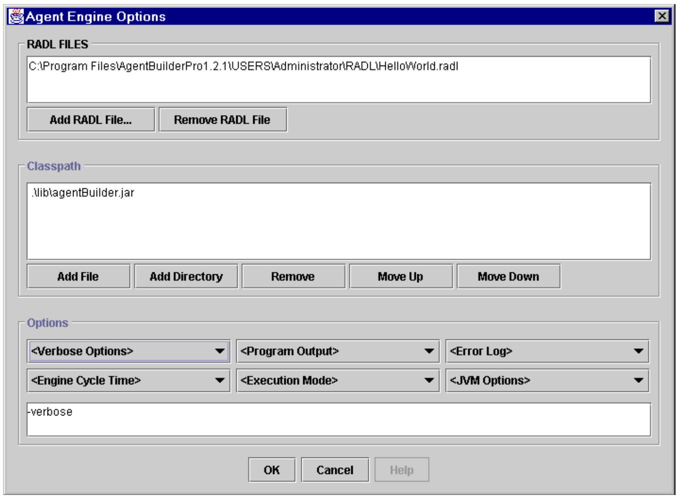
Figure 42. File Dialog

The **Agent Engine Options** dialog allows the user to specify all of the parameters for running the agent. Once all of the parameters have been specified, you must select the **OK** button. Selecting the **Cancel** button will cancel running the agent. If you select the **OK** button, you will be presented with the engine console, shown in Figure 44.

The engine console allows you to monitor the progress of the running agent. Any output from the running agent will be shown in the output description area at the top of the console. Any errors that occur during the execution of the agent will be shown in the error description area at the bottom of the console. The engine console allows you to save, clear, and freeze the output from the running agent as well as save and clear the errors that are generated.

For a more complete description of the **Agent Engine Options** dialog and the engine console, see “Run-Time System” on page 251.

The agent engine that runs the agent is always launched on a separate Java virtual machine. Since the agent engine is run separately


Agent Engine Options

RADL FILES

C:\Program Files\AgentBuilderPro1.2.1\USERS\Administrator\RADL\HelloWorld.radl

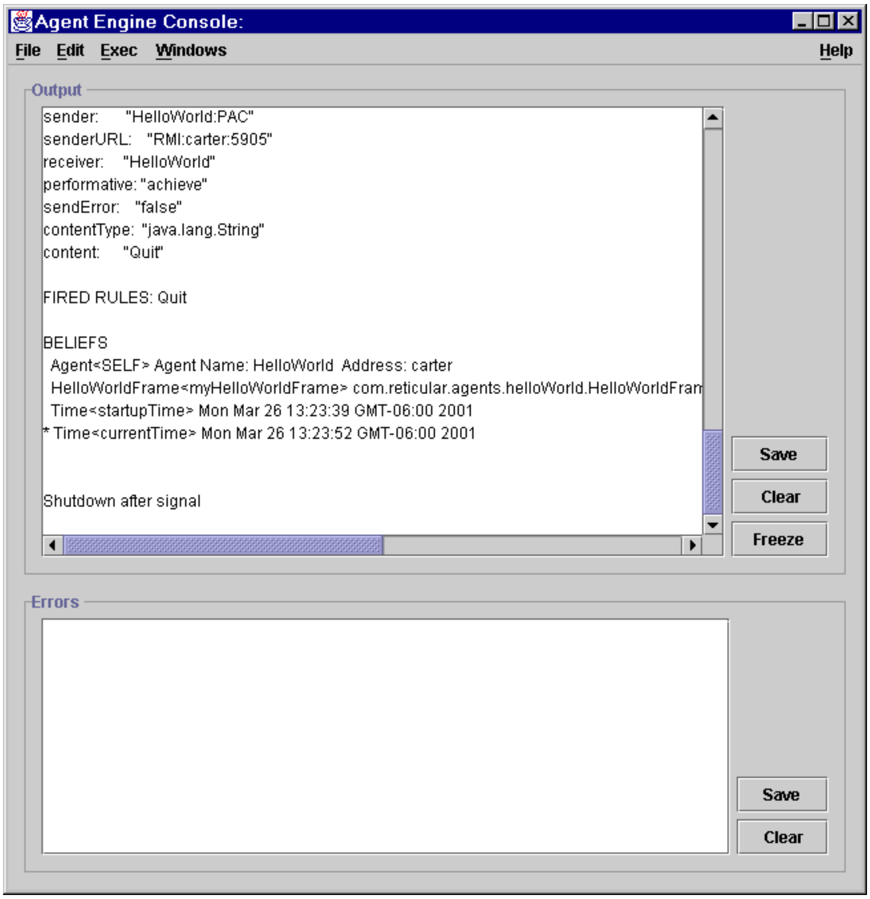
Classpath

.\lib\agentBuilder.jar

Options

-verbose

Figure 43. The Agent Engine Options Dialog



```
sender:      "HelloWorld:PAC"
senderURL:   "RMI:carter:5905"
receiver:    "HelloWorld"
performative: "achieve"
sendError:   "false"
contentType: "java.lang.String"
content:     "Quit"

FIRED RULES: Quit

BELIEFS
 Agent<SELF> Agent Name: HelloWorld  Address: carter
 HelloWorldFrame<myHelloWorldFrame> com.reticular.agents.helloWorld.HelloWorldFram
 Time<startupTime> Mon Mar 26 13:23:39 GMT-06:00 2001
* Time<currentTime> Mon Mar 26 13:23:52 GMT-06:00 2001


Shutdown after signal
```

Figure 44. Engine Console

from the AgentBuilder tools, you can shutdown AgentBuilder and still have the agent engine running. Running the engine on a separate virtual machine also allows the user to modify their classpath settings in the **Agent Engine Options** dialog. Changes to the classpath can only occur with each new execution of the agent engine. You cannot select the **Edit → Set Engine Options** menu item and dynamically change the classpath while the engine console is running.

Running Multiple Agents on Different Machines

You can run multiple agents on different machines and still have them communicate, cooperate, and negotiate with each other. However, there are a couple of criteria that must be met in order to run the agents on different machines. First, make sure that each agent, including the agency, has the IP address of the machine it will run on. The agency and agents cannot have the keyword CURRENT_IP_ADDRESS assigned to it. Next, you will need to regenerate the RADL files for all the agents in the agency. This will ensure that each agent in the agency will know what machine the other agents are running on. You can now run the agents on the machines they were assigned (see "Run-Time System" on page 251. for information on how to run agents outside of AgentBuilder).

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tools are open. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be brought to the top of the window stack.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Exiting AgentBuilder

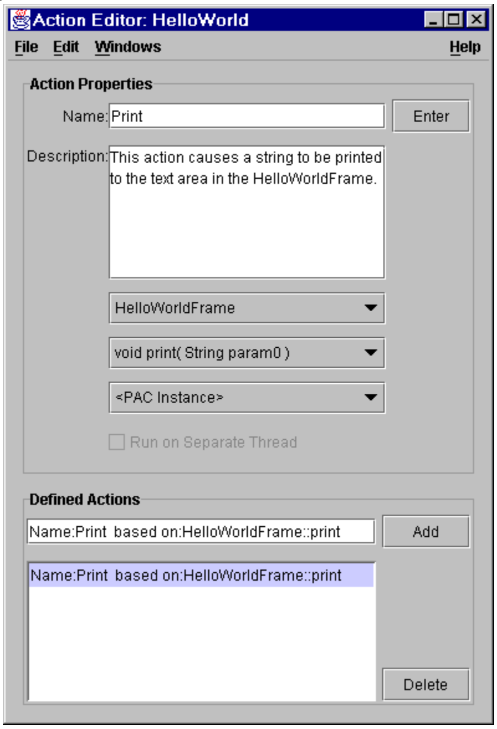
The **File** menu's **Exit** menu item allows you to exit AgentBuilder. Selecting this will close all of the tools that are currently open.

Action Editor

The Action Editor allows you to view, edit and create actions. An action is an association between an action name and a method on a PAC. The action name may be the same as the method name or it may be different. An action also contains lists of preconditions and effects which will automatically be added to any rule containing the action. The actions are optional, you can use direct method invocation instead. The Actions will be needed when you create Commitments that need to invoke methods on PACs.

Overview

You can open the Action Editor by selecting **Action Editor** from the **Tools** menu in the Agent Manager or selecting the **Action** tab in the Agent Manager. You can also double-click on an action in the **Agent Manager Actions Panel** to open the Action Editor. After opening the Action Editor, you can use the **File** menu to create a new action, save the current list of actions, close the Action Editor, and exit the system. The **Edit** menu allows you to delete actions from the defined actions list. The **Windows** menu lets you switch between other AgentBuilder tools that are open. The **Help** menu provides access to the AgentBuilder help system. The Action Editor is shown in Figure 45.



Name: Print

Description: This action causes a string to be printed to the text area in the HelloWorldFrame.

HelloWorldFrame

void print(String param0)

<PAC Instance>

☐ Run on Separate Thread

Name:Print based on:HelloWorldFrame::print

Name:Print based on:HelloWorldFrame::print

Figure 45. Action Editor

The **Action Properties** panel allows you to either create a new action or view a defined action. The **Action Properties** panel allows you to specify an action's name, description, PAC, PAC Instance and PAC method.

The **Defined Actions** panel allows you to add new actions to the list, or delete defined actions from the list.

Operation

Creating an Action

If you want to create a new action, the **Action Properties** panel must be cleared. If an action is currently being worked on, you can select the **New** menu item from the **File** menu. Selecting the **New** menu item will prompt you to confirm that you want to create a new action.

To create an action, fill out the **Action Properties** panel. The action's name must be entered into the **Name** text field. You can either press the **Enter** key on your keyboard while your cursor is in the **Name** text field, or click on the **Enter** button on the panel. Once an action name has been entered, the new action name will appear in the **Defined Action** text field. If the text in the **Defined Action** text field has scrolled out of view, you can click on the text field, and use the ←, →, **Home** and **End** keyboard keys.

You can select a PAC from the **<PAC>** combo-box. PACs that have been defined for the currently loaded agent will be loaded into the **<PAC>** combo-box. The selected PAC will be shown in the **Defined Action** text field.

The **Method** combo-box is dependent on the selection in the **PAC** combo-box. Once a PAC has been selected, the PAC's methods will be loaded into the **<Method>** combo-box. To complete the action definition, you must select a method. The selected method will then be shown in the **Defined Action** text field.

You can also specify a PAC Instance. If there are any PAC Instances that have been built from the currently selected PAC, they will show up in the **PAC Instance** combo-box.

You can also enter a description for the action in the **Description** text area.

Adding an Action

Once an action has been completely defined, the **Defined Action Add** button will become enabled. At the minimum, an action must have a name, PAC, and method selected. To add the action shown in the **Defined Action** text field, the user must press the **Add** button.

Pressing the **Add** button will add the action in the **Defined Action** text field to the list of defined actions. The **Action Properties** panel will be reset so a new action can be defined, or so an existing action can be displayed.

Viewing Defined Actions

You can view actions that are in the defined actions list. Selecting an action will cause the selected action to be loaded into the **Action Properties** panel.

Editing a Defined Action

Once you load a defined action into the **Action Properties** panel, you can edit the action. You can repeat any of the steps outlined in the section on “Creating an Action” on page 81. If you want to add the edited action to the list, then add the action as outlined in the “Adding an Action” on page 82. If the edited action has the same name as the old action, the edited action will overwrite the old action. Otherwise, it will be added to the list of defined actions, along with the old action.

Deleting a Defined Action

If you wish to delete a defined action, you have two options. You can first select the action in the **Defined Actions** list, then click on the **Delete** button. You can also select the action in the list, then select the **Delete** menu item from the **Edit** menu.

Saving the Actions

After you're finished adding and deleting actions, you need to save the action list to the current agent. To do this, select the **Save** menu item from the **File** menu.

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tools are open. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be brought to the front on your screen.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the Action Editor

To close the Action Editor and leave all other tools open, select **Close** from the **File** menu. If you have modified the list of defined actions, you will be given a chance to save the action list before closing the Action Editor.

Exiting AgentBuilder

The **File** menu's **Exit** menu can be used to exit AgentBuilder. This will close all of the tools that are currently open. If you have modified the list of defined actions, you will be given a chance to save the action list before exiting the system.

Commitment Editor

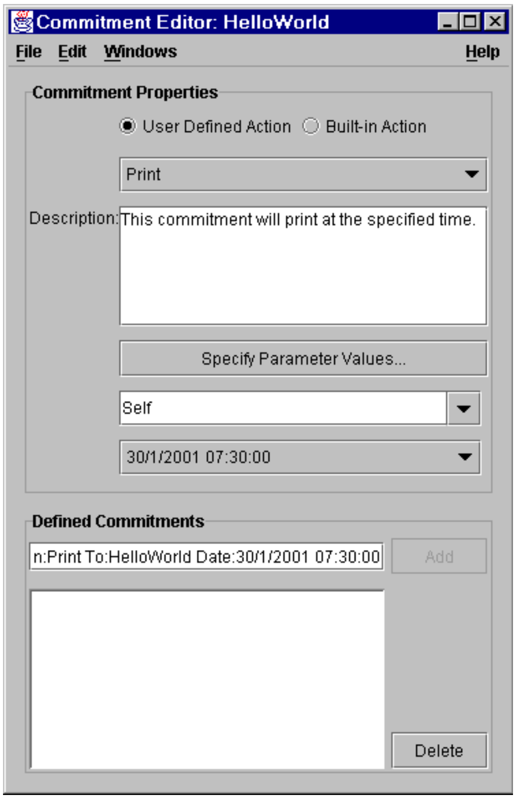
The Commitment Editor allows you to view, edit, and create commitments. Commitments are based on a particular action; they specify the time an action will be committed and the agent to which the action will be committed. The Commitment Editor is shown in Figure 46.

Overview

The **File** menu allows you to create a new commitment, save the current list of commitments, close the Commitment Editor, and exit the system. The Edit menu allows you to delete commitments from the defined commitments list. The **Windows** menu lets you switch between AgentBuilder tools that are open. The **Help** menu gives you access to the AgentBuilder Help system.

The **Commitment Properties** panel allows you to create a new commitment or view a defined commitment. The **Commitment Properties** panel allows you to specify a commitment's action, description, parameter values for the action, agent to commit the action to, and the time the action should be executed.

The **Defined Commitments** panel allows you to add new commitments to the list or delete defined commitments from the list.



Print

Description: This commitment will print at the specified time.

Self

30/1/2001 07:30:00

n:Print To:HelloWorld Date:30/1/2001 07:30:00

Figure 46. Commitment Editor

Operation

Creating a Commitment

Before creating a new commitment, the **Commitment Properties** panel must be cleared. If a commitment is currently being edited, then you can select **New** from the **File** menu. Selecting the **New** menu item will cause the system to ask you to confirm that you wish to start editing a new commitment.

To create a commitment, you must enter the required information into the **Commitment Properties** panel. You must select an action for which this commitment is defined. The **Commitment Properties** panel contains two radio buttons for selecting **User Defined Actions** or **Built-In Actions**. The **Actions** combo box displays the appropriate actions depending on which radio button is selected. Currently, actions that are tied to commitments must be connected to a PAC instance. Once an action has been selected two things will happen. The selected action's method will be queried and, if the action's method contains any parameters, the **Specify Parameter Values** button will be enabled; if the selected action's method does not contain any parameters, the **Specify Parameter Values** button will be disabled. If the **Specify Parameter Values** button is enabled, then you must specify all parameter values. This process is explained in the following section. After selecting an action, the text field in the **Defined Commitment's** panel will show the name of the selected action.

After an action has been selected, you must specify the agent to whom the commitment will be made. The **Committed To** combo-box is located below the **Specify Parameter Values** button. This combo-box is an editable combo-box, and contains a list of available agents. The list of agents is constructed using the agencies to which the current agent belongs. To specify an agent, you can either select from the list of agents provided or manually enter an agent's name in the combo-box. If you manually enter an agent's name, you must press the **Enter** key in the combo-box so that the agent's name can be entered. Once an agent has been specified, the agent's name will be displayed in the **Defined Commitments** text field.

You must also specify a time to execute the action. The combo-box at the bottom of the **Commitment Properties** panel is the **Time** pull-down menu. The **Time** pull-down menu provides a number of options including **StartupTime**, **ShutdownTime** and a user-defined time. If you select the user-defined item, the system will display the **Time Dialog** described below. If a user-defined time is entered, the time will be shown in the combo-box. Once a time is specified for the commitment, this time will be shown in the **Defined Commitments** text field.

You can also enter a description for the commitment in the **Description** text area.

Specifying Parameter Values

The **Parameters Dialog** allows you to easily specify parameter values for simple and complex objects. The dialog consists of two sections: a **Parameter Tree** panel and a panel that can switch between a **Parameter** panel and a **Complex Parameter** panel. The **Parameters Dialog** is shown in Figure 47.

The **Parameter Tree** panel contains a hierarchical tree of the parameters. The tree nodes use the following convention for labeling name(type). The root node is labeled with the action name and the action's PAC type. The children of the root node are the parameters for the action's method. Any child nodes with a folder icon indicate that the parameter is a complex object whose constructor must be specified.

The color of the nodes is also significant. Nodes whose values have not been specified have a red label; nodes whose values have been specified have a black label. An unspecified node recursively sets its parent to be unspecified so that you can easily see which nodes have not been specified when the tree is fully collapsed.

To specify a parameter value, you must first select the parameter to be specified. Based on the type of node selected, the panel below the **Parameter Tree** will display either a **Parameter** panel or a **Complex Parameter** panel. Both the **Parameter** panel and the **Complex**

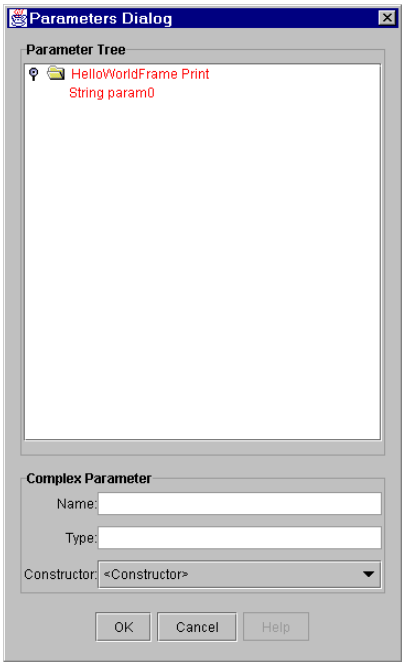


Figure 47. Parameters Dialog

Parameter panel contain a read-only **Name** and **Type** text field. The name and type of the selected node is shown in the read-only text fields. The primary difference between the two panels is that the **Parameter** panel allows you to specify a value for the selected parameter. To enter a value for the selected parameter, enter a value into the **Value** text field in the **Parameter** panel. To enter a null value for the parameter, you must type *`null`* into the value field. An empty string is considered to be a valid value and can also be entered into the value field. Once a value has been entered, you must press the **Enter** key on your keyboard while the cursor is positioned in the **Value** text field or click on the **Enter** button in order to enter the value in the selected parameter. The new value will be shown in the tree after it has been entered.

The **Complex Parameter** panel contains a pull-down menu for specifying a constructor for the selected parameter. This is necessary because the selected parameter's type is a complex object. You must now select a constructor from the **Constructor** pull-down menu.

If no constructors are listed, you must create a constructor for the complex object. To do this, go back to the Object Modeler and open the ontology that contains the complex object. You can then modify the object's properties to include one or more constructors. The object model must then be saved, and the PAC Editor must be opened from the Agent Manager. From the PAC Editor, update the object model that contains the updated PAC. (Please refer to “PAC Editor” on page 92 for information on updating PACs.)

Once a constructor is selected for the complex object, the tree node for the complex object will change based on the selected constructor. Any parameters that are in the selected constructor become children of the tree node. If the children's types are all Java types, then you can assign values to the parameters as described above. Otherwise, if the children's parameter types contain complex objects, the process of selecting a constructor and specifying values must be repeated. This process must be repeated until all leaves either contain Java types or the selected constructors for complex objects have no parameters.

Once you have fully specified the parameter tree, you can click on the **OK** button to register the new parameter values. Clicking on the **Cancel** button will cancel any changes that were made to the parameter tree.

Specifying a User-Defined Time

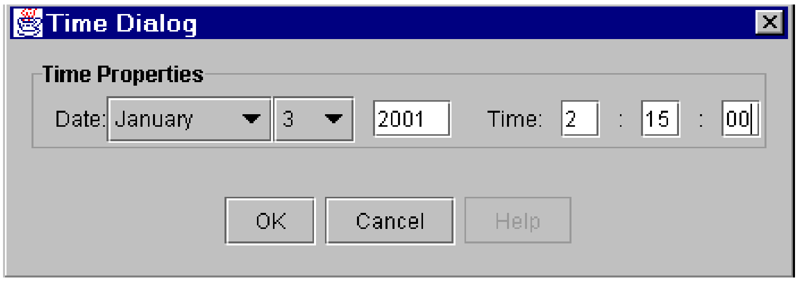
The **Time Dialog** allows you to specify a specific date and time that an action will be committed. The date is specified by selecting a month and day from the pull-down menu. The year must be entered (typed) in a four digit, numerical format. The time is specified using the format `hh:mm:ss`. Time uses three text fields with only numerical entries allowed. This dialog is shown in Figure 48.

Once you have specified a date and time, you must select the **OK** button in order for the time to be set for the commitment. Any invalid entries will cause an error dialog to be displayed. You will be given an opportunity to correct any errors. If you wish to cancel operations using the **Time Dialog** then select the **Cancel** button. Selecting the **Cancel** button will ensure that the **Time** pull-down menu remains unmodified.

Adding a Commitment

Once a commitment has been completely defined, the **Defined Commitments** panel **Add** button is enabled. In order to construct a completely defined commitment, you must specify a selected action, any parameter values specified through the **Parameters Dialog**, a selected agent to commit to, and a time to perform the action.

Pressing the **Add** button will add the commitment in the **Defined Commitments** text field in the list of defined commitments. Notice that the **Commitment Properties** panel is reset, readying it for defining a new commitment or viewing defined commitments



Date: January 3 2001 Time: 2 : 15 : 00

Figure 48. Time Dialog

You can view commitments that are in the **Defined Commitments** list. Selecting a commitment will cause the selected commitment to be displayed in the **Commitment Properties** panel.

Editing a Defined Commitment

Once you load a defined commitment into the **Commitment Properties** panel, you can edit the commitment using any of the steps outlined in “Creating a Commitment” on page 85. If you want to add the edited commitment to the list, then add the commitment as outlined in “Adding a Commitment” on page 90. If the edited commitment has the same action and agent, the edited commitment will overwrite the old commitment. Otherwise, it will be added to the list of defined commitments.

Deleting a Defined Commitment

If you want to delete a defined commitment, you have two options. You can first select the commitment in the **Defined Commitments** list and then click on the **Delete** button to the right of the list. You can also select the commitment in the list, then select the **Delete** menu item in the **Edit** menu.

Saving Commitments

Once you have completed adding and deleting commitments, you must save the commitment list for the current agent. To do this, select the **Save** menu item from the **File** menu.

Switching Windows

The **Windows** menu is a dynamic menu that displays a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tools are open. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be brought to the front on your display.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the Commitment Editor

To close the Commitment Editor and leave all of the other tools open, you should select **Close** from the **File** menu. If you have modified the list of defined commitments, you will be given an opportunity to save the commitment list before closing the Commitment Editor.

Exiting AgentBuilder

The **File** menu's **Exit** menu item should be selected to exit AgentBuilder. This will close all of the tools that are currently open. If you have modified the list of defined commitments, you will be given an opportunity to save the commitment list before exiting the system.

PAC Editor

The PAC Editor allows you to view, edit and create PACs, PAC Instances, and Java Instances. The data created by this editor is used extensively by the Rule Editor to construct various patterns. You cannot directly use classes defined in the Object Modeler in your rules. The PAC Editor is necessary for creating PACs from the classes so they can be made a part of your rules.

Overview

The **File** menu can be used to perform various operations on the PACs and instances. From the **File** menu, you can create new PACs, PAC Instances and Java instances. You can also save the list of defined PACs, PAC Instances, and Java Instances. You can also import and update the list of defined PACs. As with all of the other AgentBuilder tools, the **File** menu provides the **Close** and **Exit** menu items for closing the editor and exiting AgentBuilder. The PAC Editor is shown in Figure 49.

The **Edit** menu allows you to delete any of the items in the defined PACs, PAC Instances, or Java Instances list. The **Windows** menu lets you switch between other open AgentBuilder tools. The **Help** menu provides you with access to the AgentBuilder help system.

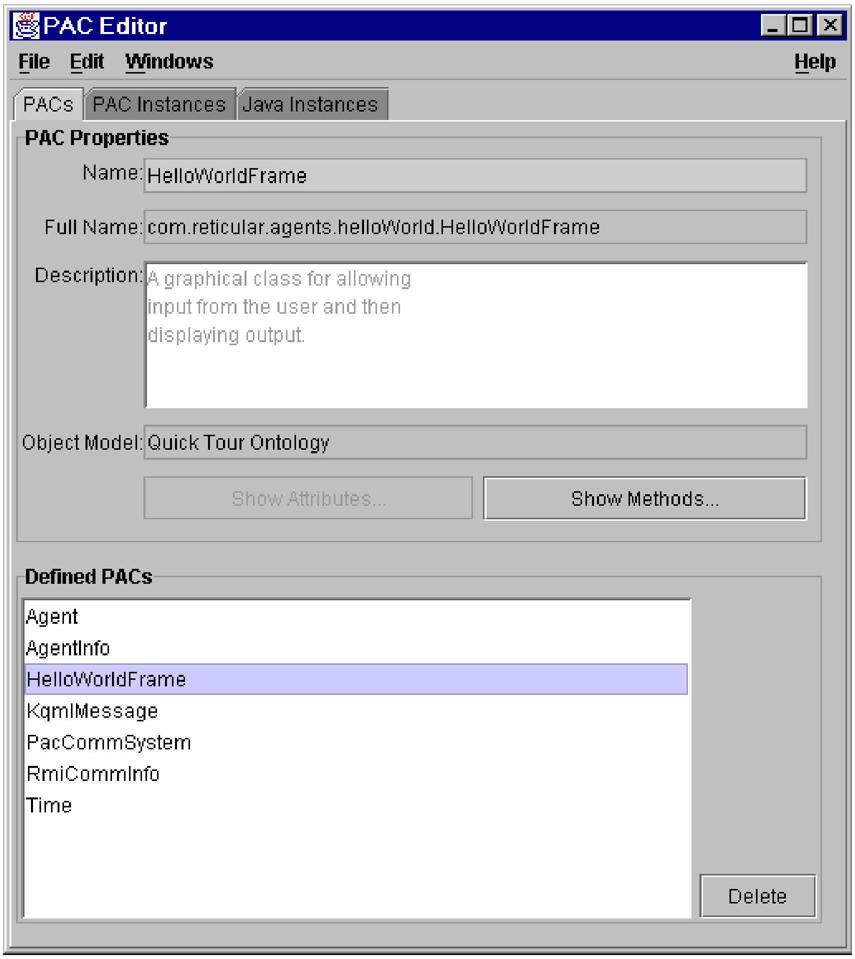
The main panel of the PAC Editor contains a **Panel Options** panel. The **Panel Options** panel contains tabs that allow you to switch between the PACs, PAC Instances, or Java Instances panel. The contents of each of the panels is explained in the following sections.

Operation

Importing PACs

The current version of AgentBuilder does not allow you to directly create PACs. Instead, you must import the PACs from classes defined in object models. To import PACs, select the **Import** menu item from the **File** menu. Figure 50 shows the **Import Dialog**.

The **Import Dialog** consists of two sections: the **Available Classes** panel and the **Selected Classes** panel. To import PACs, you must first select an object model in the **Available Classes** panel. Once an object model is selected, the dialog will display the names of the classes that have been defined in the selected object model.



Name: HelloWorldFrame

Full Name: com.reticular.agents.helloWorld.HelloWorldFrame

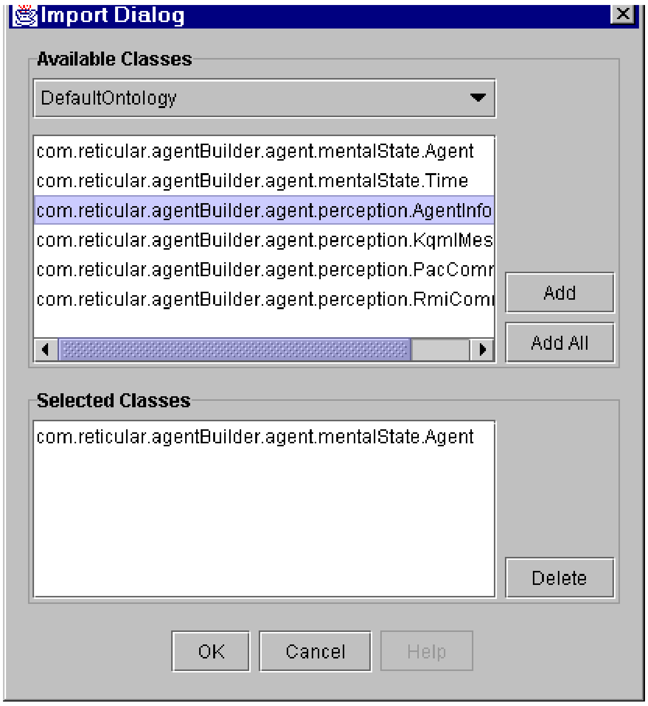
Description: A graphical class for allowing input from the user and then displaying output.

Object Model: Quick Tour Ontology

Agent
AgentInfo
HelloWorldFrame
KqmlMessage
PacCommSystem
RmiCommInfo
Time

Figure 49. PAC Editor

The contents of the **Object Model** pull-down menu is a complete listing of all defined ontologies in the user's repository. You have several ways to select classes to be added to the list in the **Selected Classes** panel. The first method is to select a class in the list and then click on the **Add** button. If you need to select multiple classes, you must hold down the **Control** key on your keyboard while



com.reticular.agentBuilder.agent.mentalState.Agent
com.reticular.agentBuilder.agent.mentalState.Time
com.reticular.agentBuilder.agent.perception.AgentInfo
com.reticular.agentBuilder.agent.perception.KqmlMes
com.reticular.agentBuilder.agent.perception.PacComr
com.reticular.agentBuilder.agent.perception.RmiComi

com.reticular.agentBuilder.agent.mentalState.Agent

Figure 50. Import Dialog

selecting the classes. Once all of the classes have been selected, click on the **Add** button. If you want to add all of the classes listed, you can click on the **Add All** button. You can now select a different object model and continue to add classes as described above.

You can delete classes listed in the **Selected Classes** panel at any time. To do this, first select the classes to be deleted. Then, click on the **Delete** button. The selected class will then be deleted from the list.

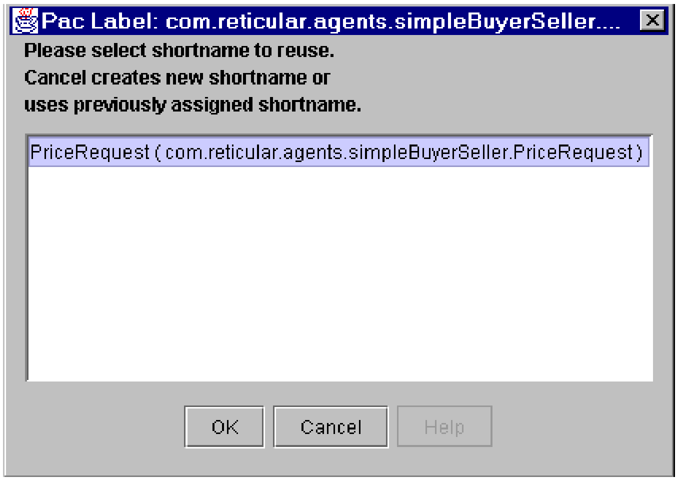
When you are finished adding classes to the list, you must click on the **OK** button. Clicking on the **Cancel** button will cancel the import operation. If the **OK** button has been pressed and there were classes listed in the **Selected Classes** panel, then the selected classes will be converted into PACs. The new PACs will be displayed in the **Defined PACs** panel. If you cannot see the **Defined PACs** panel, make sure that the **PACs** radio button in the **Panel Options** panel of the PAC editor is selected.

Packages and Short Names

The Pac Editor supports java package naming conventions, if you are unfamiliar with Java package naming, see Sun Microsystem's documentation. Each class that is imported as a PAC is given a unique short name by the PAC Editor. This means when importing a class with a base class name of another class, it is assigned a name other than its class name, typically the PAC Editor adds a %2 after the class name.

For example, if you import the class `com.company.Container` and issue a **Save** it will be given the short name of `Container`. This means that everywhere in the agency manager where `Container` appears, it is actually referring to `com.company.Container`. If a class `tool.util.Container` is imported and saved, the PAC Editor gives the class a new short name of `Container%2` (with some input from the user as described below). Consequently the user is able to determine the difference between the two using the percent sign. If you delete the `com.company.Container` PAC, the short name will NOT be released. All short name assignments are permanent, with the exception described below. Therefore, unless a remapping is done, the old references to the `Container` short name are still referencing the `com.company.Container` class.

When the user has imported a new PAC with a potential short name collision and then issues a **Save**, a dialog is displayed to inform and query the user. This dialog, shown in Figure 51, allows the user to choose whether to map the class to an existing short name or allow the tool to create a new short name. By selected an existing short name the user is remapping all usages in the rules, commitments, instances and actions to this new class. This remapping is normally only appropriate when a class has been renamed or the package has changed.


Pac Label: com.reticular.agents.simpleBuyerSeller....

Please select shortname to reuse.
Cancel creates new shortname or
uses previously assigned shortname.

PriceRequest (com.reticular.agents.simpleBuyerSeller.PriceRequest)

Figure 51. Dialog for Handling Short Names

Viewing Defined PACs

To view the properties of a defined PAC, select the PAC in the **Defined PACs** list. Selecting a defined PAC from this list will cause the PAC Properties panel to display information about the selected PAC. The selected PAC's name, description, package, and ontology will be displayed. To view attributes or methods simply click the appropriate button and an attribute or methods list will appear in a dialog box.

Deleting a Defined PAC

You can use the **Defined PACs** panel to delete any of the defined PACs. To do this, you must first select the PAC to be deleted. Then, click on the **Delete** button. Clicking on the **Delete** button will remove the selected PAC from the list.

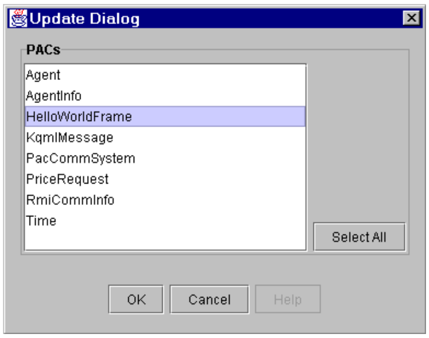


Figure 52. PAC Update Dialog

Updating PACs

On occasion you will need to update a PAC from an ontology. The PAC Editor provides a way to update the list of defined PACs. Figure 52 shows the **Update Dialog** used to update defined PACs.

The **Update Dialog** is accessed from the **File** menu's **Update** menu item. The dialog contains a PACs panel, which lists all the PACs associated with the agent. To update the defined PACs in the **Update Dialog**, select the PACs you wish to update. Holding the **Ctrl** key while clicking on the PACs, will allow you to make multiple PAC selections. If you want to update all of the PACs listed, click the **Select All** button. Once the PACs that need to be updated are selected, you need to click on the **OK** button.

Any of the defined PAC's whose object models were updated will now display the updated information.

Defining PAC Instances

Once you have defined a PAC, you can define PAC Instances. To define a PAC Instance, select the PAC you want to instantiate. All other items in the **Instance Properties** panel are optional for defining a PAC Instance. To specify a name for the PAC Instance, type the name into the **Name** text field. To add the name to the PAC

Instance, you must either press the **Enter** key on your keyboard while the cursor is in the **Name** text field, or click on the **Enter** button next to the **Name** text field. You can also type the description of the PAC Instance in the **Description** text area.

Once a PAC is selected, you are given the option of defining the PAC Instance as an Initial PAC Instance. This is done using the **Initial PAC Instance** checkbox. If the **Initial PAC Instance** checkbox is selected, the **Specify Constructor** button will be enabled. A necessary condition for defining an initial PAC Instance is that its constructor be selected and that all of its constructor parameters be specified.

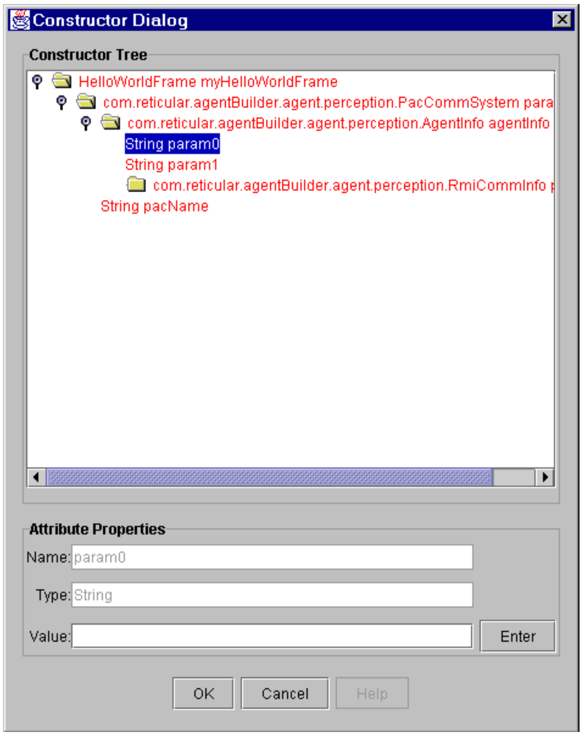
Specifying Constructors for PAC Instances

To specify constructors for a PAC Instance click on the **Specify Constructor** button. Figure 53 shows the dialog that will appear.

The **Constructor Dialog** allows you to specify the constructor and its corresponding parameter values for various kinds of objects. The dialog consists of two sections: a **Constructor Tree** panel and a panel that switches between a **Parameter** panel and a **Complex Parameter** panel.

The **Constructor Tree** panel contains a hierarchical tree of the parameters. The tree nodes use the `name(type)` convention for labeling. The root node is labeled with the PAC Instance name and the PAC Instance's PAC type. The children of the root node are the parameters for the PAC type's constructors. Any nodes with a folder icon indicate that the parameter is a complex object whose constructor needs to be specified.

The color of the nodes is also significant. Nodes whose values have *not* been specified have a red label; nodes whose value *have* been specified have a black label. An unspecified node recursively sets



HelloWorldFrame myHelloWorldFrame
com.reticular.agentBuilder.agent.perception.PacCommSystem para
com.reticular.agentBuilder.agent.perception.AgentInfo agentInfo
String param0
String param1
com.reticular.agentBuilder.agent.perception.RmiCommInfo
String pacName

Name: param0
Type: String
Value:

Figure 53. PAC Instance Constructor Dialog

its parent to be unspecified, so you can easily see which nodes have not been specified even when the tree is fully collapsed.

To specify a parameter value, you must first select the parameter to be specified. The panel below the **Constructor Tree** will change to either a **Parameter** panel or a **Complex Parameter** panel based on the type of node selected. Both the **Parameter** panel and the **Complex Parameter** panel contain a read-only **Name** and **Type** text field. The name and type of the selected node are shown in the read-only text fields. The difference between the two panels is that the **Parameter** panel allows you to specify a value for the selected parameter. To enter a value for the selected parameter, enter a value into the **Value** text field in the **Parameter** panel. To enter a null value for the parameter, type `null` into the value field. An empty string is considered to be a valid value and can also be entered into the value field. Once a value has been entered, you can press the **Enter** key on your keyboard while the cursor is in the **Value** text field, or the **Enter** button can be clicked. This will cause the value to be registered with the selected parameter. The entered value is shown in the tree once it has been registered.

The **Complex Parameter** panel contains a pull-down menu for specifying a constructor for the selected parameter. This is because the selected node type is a complex object. You must select a constructor from the **Constructor** pull-down menu.

If no constructors are listed, you can create a constructor for the complex object. To do this, go back to the Object Modeler and open the ontology that contains the complex object. You can then modify the object's properties and include one or more constructors. The object model must then be saved and the defined PACs updated.

Once a constructor is selected for the complex object, the tree node for the complex object will change. Any parameters that are in the selected constructor become children of the tree node. If the children's types are all Java types, then you can assign values to the parameters as described above. Otherwise, if the children's parameter types contain complex objects, the process of selecting a constructor and specifying values must be repeated. This process must be repeated until all leaves either contain Java types, or the selected constructors for complex objects have no parameters.

Once you have fully specified the constructor tree, you can click on the **OK** button to register the new parameter values. Clicking on the **Cancel** button will cancel any changes that were made to the constructor tree.

Adding a PAC Instance

Once a PAC Instance has been defined, the PAC Instance can be added to the list of Defined Instances. If the **Add** button in the **Defined Instances** panel is disabled, then the PAC Instance has not been completely defined. Check to make sure that a PAC has been selected. If the PAC Instance is checked to indicate that it is an Initial PAC Instance, make sure the constructor and its parameters have been fully specified.

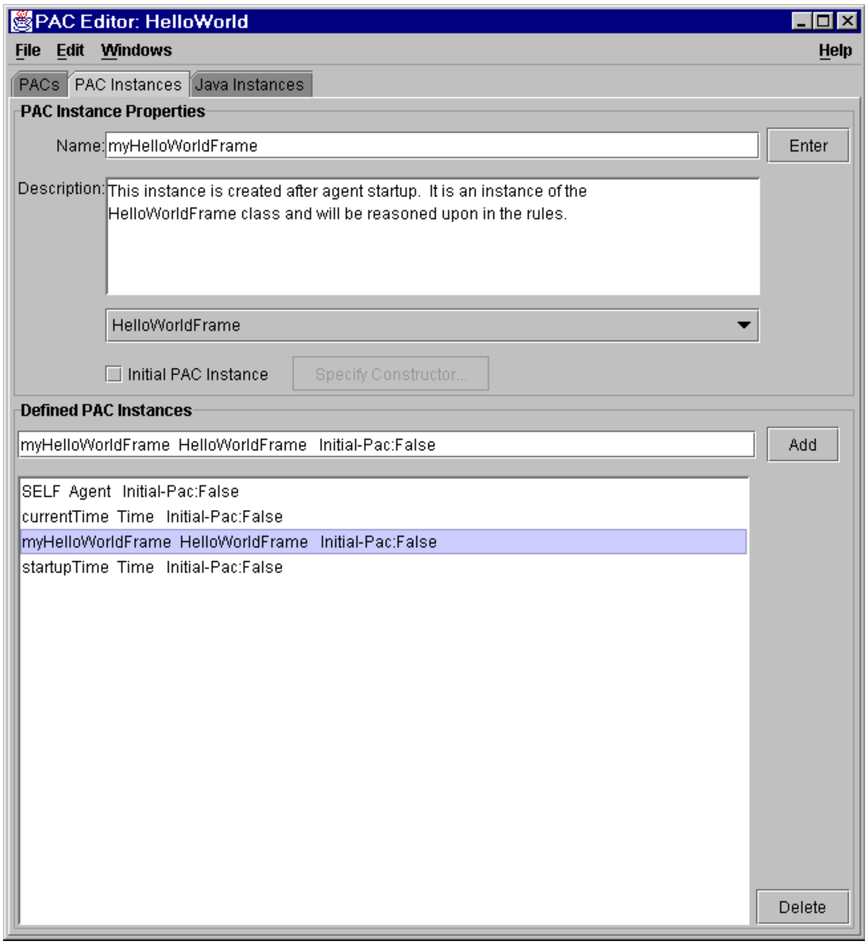
Once the **Add** button is enabled, you can add the PAC Instance to the list of defined PACs. Clicking on the **Add** button will add the PAC Instance to the list. Notice that the **Instance Properties** panel is cleared and the text field in the **Defined Instance** panel updated to reflect that a new PAC Instance has been defined.

Viewing Defined PAC Instances

To view the properties of a defined PAC Instance, select the PAC Instance in the **Defined Instances** list. Selecting a defined PAC Instance will cause the **Instance Properties** panel to display information about the selected PAC Instance. The selected PAC Instance's name, description, PAC and Initial PAC Instance status will all be displayed. If the PAC Instance is an Initial PAC Instance, you can view the specified constructor by clicking on the **Specify Constructor** button (see Figure 54).

Editing a PAC Instance

To edit a PAC Instance, follow the above instructions for viewing a defined PAC Instance. Once the PAC Instance is loaded into the **Instance Properties** panel, you can modify the PAC Instance. When finished modifying the PAC Instance, the PAC Instance can be added to the list of Defined Instances as described above. Note that if the modified PAC Instance has the same name as the original PAC Instance, the modified PAC Instance will overwrite the original PAC Instance. If the modified PAC Instance has a different



PAC Instance Properties

Name: myHelloWorldFrame

Description: This instance is created after agent startup. It is an instance of the HelloWorldFrame class and will be reasoned upon in the rules.

HelloWorldFrame

☐ Initial PAC Instance

Defined PAC Instances

myHelloWorldFrame HelloWorldFrame Initial-Pac:False

SELF Agent Initial-Pac:False
currentTime Time Initial-Pac:False
myHelloWorldFrame HelloWorldFrame Initial-Pac:False
startupTime Time Initial-Pac:False

Figure 54. Viewing PAC Instances

name, the PAC Instance will be added to the list in the same manner as a new instance.

Deleting a Defined PAC Instance

From the **Defined Instances** panel, you can delete any of the defined PAC Instances. To do this, first select the PAC Instance to be deleted. Then, click on the **Delete** button. Clicking on the delete button will delete the selected PAC Instance from the list.

Creating Java Instances

The PAC Editor also supports the construction of Java Instances. Java Instances are used in the Rule Editor in the construction of patterns. To specify a Java Instance, you must select a Java type for the instance. Like PAC Instances, everything else in the **Java Instance Properties** panel is optional. Java types include some non-primitive types, which are Arrays, Enumeration, Hashtable, Object, and Vector. To specify a name for the Java Instance, type the name into the **Name** text field. To add the name to the Java Instance, you must either press the **Enter** key on your keyboard while the cursor is in the **Name** text field, or click on the **Enter** button next to the **Name** text field. You can also type in a description of the Java Instance in the **Description** text area.

Once a Java type has been selected, you are given the option to make the Java Instance an Initial Java Instance. This is done using the **Initial Java Instance** checkbox. The **Initial Java Instance** checkbox will be disabled for non-primitive java types. If the **Initial Java Instance** checkbox is selected, the text field and the **Add** button adjacent to the checkbox are enabled. The text field is used to enter a value for the Java Instance. Once a value has been entered in the text field, you can either press the **Enter** button while the cursor is in the text field or click on the **Add** button.

Adding a Java Instance

Once a Java Instance has been defined, the Java Instance can be added to the list of Defined Java Instances. If the **Add** button in the **Defined Java Instances** panel is disabled, then the Java Instance has not been completely defined. Check to make sure that a Java type has been selected. If the Java Instance is checked as an Initial Java Instance, ensure that a value has been specified.

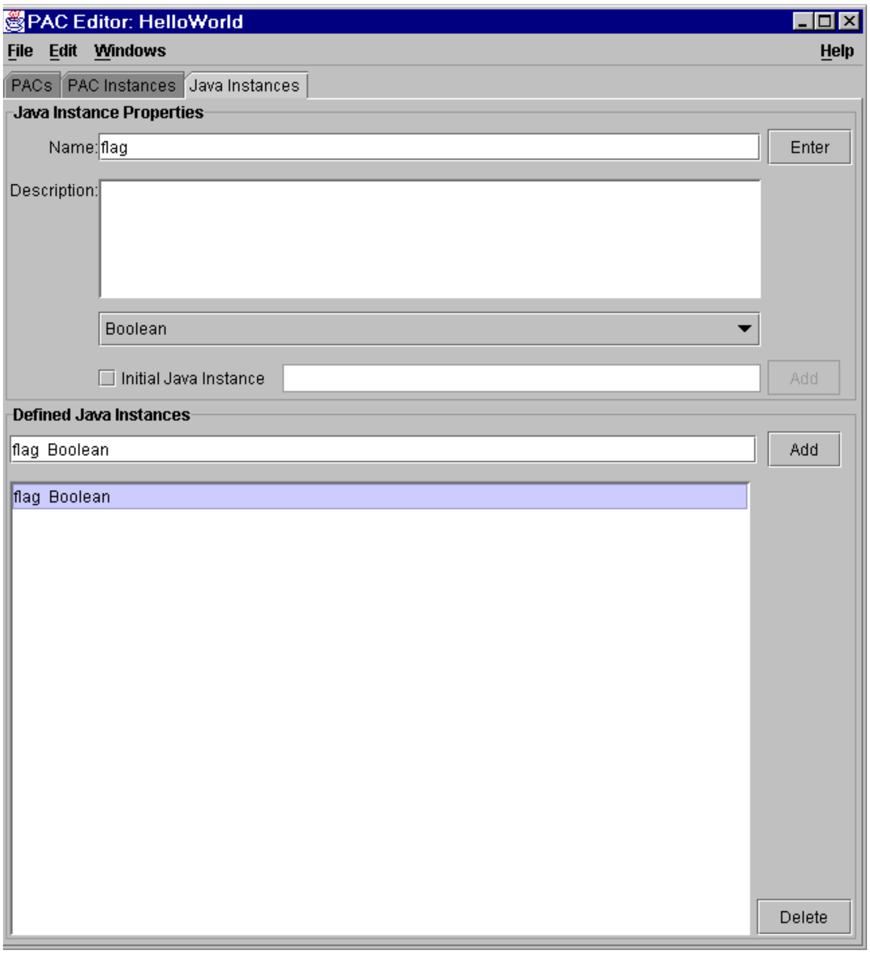
Once the **Add** button is enabled, you can add the Java Instance to the list of Defined Java Instances. Clicking on the **Add** button will add the Java Instance to the list. Note that the **Java Instance Properties** panel is cleared and the text field in the **Defined Java Instance** panel updated to show that a new Java Instance has been defined.

Viewing Defined Java Instances

To view the properties of a defined Java Instance, you must select the Java Instance in the **Defined Java Instances** list. Selecting a defined Java Instance will cause the **Java Instance Properties** panel to display information about the selected Java Instance. The selected Java Instance's name, description, Java type and Initial Java Instance status will be displayed. If the Java Instance is an initial Java instance, the text field will display the value assigned to the Java Instance (see Figure 55).

Editing a Java Instance

To edit a Java Instance, follow the instructions for viewing a defined Java Instance. When the Java Instance is loaded into the **Java Instance Properties** panel, you can modify the Java Instance. When finished modifying the Java Instance, the Java Instance can be added to the list of Defined Java Instances as described above. Note that if the modified Java Instance has the same name as the original Java Instance, the modified Java Instance will overwrite the original Java Instance. If the modified Java Instance has a different name, the Java Instance will be added to the list.



Java Instance Properties

Name: flag

Description:

Boolean

☐ Initial Java Instance

Defined Java Instances

flag Boolean

flag Boolean

Figure 55. Viewing Java Instances

Deleting a Defined Java Instance

Using the **Defined Java Instances** panel, you can delete any of the defined Java Instances. To do this, you must first select the Java Instance to be deleted, then click on the **Delete** button. Clicking on the **Delete** button will delete the selected Java Instance from the list.

Saving

If you have modified the contents of the Defined PACs, PAC Instances or Java Instances, you must save the new or modified data to the currently loaded agent. To do this, select the **Save** menu item in the **File** menu. The **Save** menu item selection will save all PACs, PAC Instances, and Java Instances to the currently loaded agent.

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools. To switch a different tool, select the tool you wish to switch to from the **Windows** menu list. The selected tool will then be brought to the front on your display.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the PAC Editor

If you want to close only the PAC Editor and leave all of the other tools open, select **Close** from the **File** menu. If you have modified the list of defined PACs, PAC Instances, or Java Instances, you will be prompted to save everything before closing the **PAC Editor**.

Exiting AgentBuilder

The **File** menu's **Exit** menu item allows the user to exit Agent-Builder. This will close all of the tools that are currently open. If you have modified the list of defined PACs, you will be given an opportunity to save the modified PACs list before exiting the system.

Rule Editor

The Rule Editor tool is used to specify the behavioral rules used by the agent. (Note: Throughout this section *behavioral rules* are simply called *rules*). Rules are the basic control mechanism for Agent-Builder agents; they determine how the agent responds to external and internal stimuli. Each rule consists of a set of conditions and the desired actions or mental changes that will occur when those conditions are met. The set of conditions is sometimes referred to as the rule's the *left-hand side*, or *LHS*; the actions and mental changes are the rule's *right-hand side*, or *RHS*. The Rule Editor has separate panels for constructing the two sides of a rule.

The Rule Editor allows you to create two different types of conditions for the left-hand side of a rule. One type of condition is based upon external events and is termed a *message condition*. A message condition is a test performed on any new messages arriving in the agent's input buffers (e.g., comparing the name of the message sender against an expected sender name). The second type of condition is based upon the agent's internal state and is termed a *mental* *condition*. A mental condition is a test performed on beliefs in the mental model of the agent.

Patterns on the left-hand side of a rule generally consist of a combination of operators, variables, constant values, and references to named instances in the mental model. Most variables will bind to all instances of a specified type in the mental model. For example, assume `?i` is the name of an `Integer` variable. In the pattern `(?i >= 4)` the variable `?i` will bind to every `Integer` instance in the mental model and the value of each binding will be used in the comparison. In contrast, a *named instance variable* is a special type of variable which will bind only to a named instance with the specified name and type. For example, the pattern `( currentCount >= 4 )` contains the named instance variable `currentCount`, which can bind only to the named instance `currentCount` in the mental model. A named instance variable is similar to a global variable in that it can be used in any of the agent's rules and will always bind to the same instance in the mental model.

The following example illustrates the difference between the two types of variables, using a trivial mental model with three `Integer` beliefs.

Mental Model

```
Integer<width>  8
Integer<length>  3
Integer<currentCount> 5
```

Rule 1

```
IF ( ?i >= 4)
THEN ( DO SystemOutPrintln( "Rule 1 Fired." )
```

Rule 2

```
IF ( currentCount >= 4)
```

```
THEN ( DO SystemOutPrintln( "Rule 2 Fired." )
```

In Rule 1 the `Integer` variable `?i` will bind to each of the `Integer` values in the mental model and each value will be compared to 4. Rule 1 will fire twice: once with `?i` bound to 8, then once with `?i` bound to 5. The binding of `?i` to 3 does not satisfy the pattern in Rule 1 so the rule will not fire with that binding. In Rule 2 the named instance variable `currentCount` will bind to the value 5 from the `currentCount` named instance (which satisfies the pattern) and so Rule 2 fires once.

Technically there's a difference between a *named instance* and a *named instance variable*: a named instance is part of the mental model (i.e., it's a belief), but a named instance variable is a component of a pattern in a rule. In some situations it's important to understand this distinction. It's possible (and fairly common) for a pattern in a rule to refer to a named instance which does not always exist in the mental model. Adding a named instance variable to a pattern in a rule does **not** guarantee that the associated named instance will be present in the mental model.

For example, consider the Print rule of Example Agent 3 in the AgentBuilder User's Guide. The second `IF` pattern in that rule contains a `Boolean` named instance variable named `Ready_to_print`:

```
( Ready_to_print EQUALS true )
```

The `Ready_to_print` named instance variable refers to an instance that does not exist in the agent's mental model until it gets asserted during execution, by the `Connect` rule. Even though at the start there is no instance named `Ready_to_print` in the mental model, the `Print` rule *does* contain a `Ready_to_print` named instance variable. This pattern evaluates to true only if the `Ready_to_print` instance exists in the mental model and has the value `true`. The pattern evaluates to false if the mental model does not contain a named instance that matches the named instance variable in the pattern. In this example, the `Print` rule is prevented from firing until the `Ready_to_print` instance is found in the mental model.

When the `Connect` rule is fired it performs the mental change:

```
( ASSERT ("Ready_to_print" true) )
```

This is the mental change that actually creates the `Ready_to_print` instance and assigns it a name and a value. After this instance has been asserted into the mental model the pattern for the `Print` rule (shown above) will evaluate to true and the `Print` rule will be activated.

A named instance variable is similar to a global variable in that it can be used in any of the agent's rules and will always bind to the same instance in the mental state. In the previous example with the `currentCount` named instance, several rules could have patterns that refer to `currentCount` and all of them would access the same value each cycle.

For the right-hand side of a rule, the Rule Editor allows you to specify the actions that occur when the rule's conditions are satisfied. Actions can be created so that they affect the agent's mental model of the external world.

Overview

The Rule Editor is divided into two major panels; one panel for editing the left-hand side of a rule and one panel for editing the right-hand side. The editor for the left-hand side of the rule is referred to as the *LHS editor*; the editor for the right-hand side is referred to as the *RHS editor*.

The Rule Editor contains a menu bar with four items: **File**, **Edit**, **Windows** and **Help**. The **File** menu allows you to open (i.e., load and start editing) previous rules as well as save the current one. Selecting **File → New → New Rule** creates a completely new rule, or selecting other items under **File → New** will clear selected sections in the existing rule. For example, choosing **File → New Condition** causes the condition accumulator line to be cleared.

The main features of the Rule Editor are the *accumulator text field*s and the *pattern lists.* Each editor has accumulator text fields; these are single-line text fields with a **New** and **Add** button directly below them. The accumulators are used to accumulate the components of arbitrarily complex conditions or action expressions. The pattern lists are at the middle and bottom of the LHS editor; the action/mental change list is at the bottom of the RHS editor.

A rule is made up of individual patterns which are built up one at a time in the accumulator line and then added to the lists. By utilizing the pull-down menus as well as context sensitive pop-up menus, you can construct complex expressions as desired. Several of the dialogs used in constructing the rule use a tree paradigm. In addition, the **Up** and **Down** buttons at the right side allow you to change the ordering of individual lines. Thus it's easy for you to rearrange ordering of patterns (or actions or mental changes) within the rule.[1]

Rule Properties

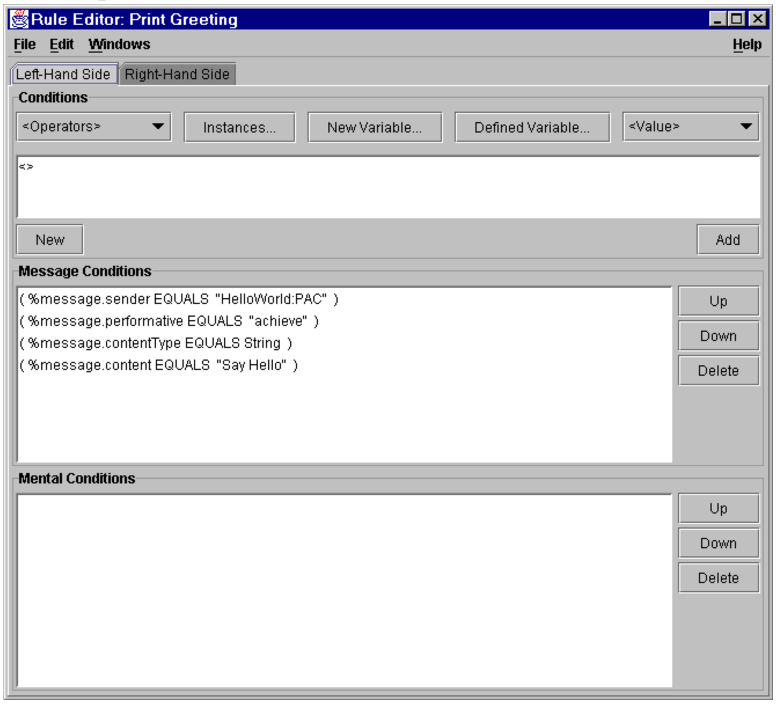
Edit→Properties allows the user to access the rule properties. The two basic parts of the **Rule Properties** dialog are the name of the rule and the rule description. There are few constraints on the name of a rule; only that it be unique within the rule set and that it not use hidden characters. You're encouraged to choose descriptive rule names; you may even find that descriptive phrases work well as rule names. The description window allows you to add a descrip-

1.A complete description of building complex patterns using the accumulator is described in “Building Complex Expressions” on page 8.

tion to accompany the rule. No size limitations are imposed, but generally only a few sentences are required.

LHS Editor

Figure 56 shows the LHS Editor. There are three main panels: the **Conditions** panel, the message conditions pattern list and the mental conditions pattern list.



```
( %message.sender EQUALS "HelloWorld:PAC" )
( %message.performative EQUALS "achieve" )
( %message.contentType EQUALS String )
( %message.content EQUALS "Say Hello" )
```

Figure 56. Rule Editor LHS

The **Conditions** panel is composed of two pull-down menus and three pop-up dialogs, an accumulator, a **New** button, and **Add** button, a **Message Conditions** panel, and a **Mental Conditions** panel. The **Operators** menu is common to all panels in both the LHS and RHS editors, but the items available in each **Operators** menu will differ depending on the panel. In the **Conditions** panel, the **Operators** menu contains various relations and pattern modifiers that are useful in patterns. The operator elements in this menu fall into several different categories shown below. See "Operators and Patterns" on page 301 for a detailed description of the operations.

- equality relations (`EQUALS, NOT_EQUALS`)
- boolean relations (`AND, OR, NOT`)
- binding relation (`BIND`)
- quantified relations (`FOR_ALL, EXISTS`)
- numerical relations (`<=,<, !=, =,` ...)
- string functions (`concat, substring`, ...)
- arithmetic functions (`+, -, *, /`)
- mathematical functions (`tan, arctan, cos, arccos, sqrt`, ...)
- miscellaneous functions (SET_TEMPORARY)

The **Values** menu allows you to construct an instance of any `Java` type, a `KQMLMessage` type, a `Time` type, or a `Class` type and insert it into the current pattern. This is useful when you want to specify a comparison against a constant value. For example, if you want to test whether a field in an object is less than 4, you could specify the constant value (i.e., 4, probably as an `integer` or `float`) in the **Values** menu.

For the `Java` types, you will be presented with a dialog for entering a literal value. For the `KQMLMessage`, `TIME` and `Class` type, the user is present with a specific dialog for entering each of the aforementioned types.

There are three different pop-up dialogs. The first is the **New Variable** dialog. This dialog allows you to construct new variables that will be available for use in patterns.Figure 57 shows the **New Variable** dialog. To create a new variable, you would first select the

PACs or **Java Types** radio button. Once a radio button has been selected, a list of available variables types will be displayed. You can then select a PAC or Java type. Once a variable type has been selected, the focus will be transferred to the **Variable Name** text field. Type in a name for the new variable and select the **Add** button. Selecting the Add button will add the new variable to the list at the bottom of the dialog.

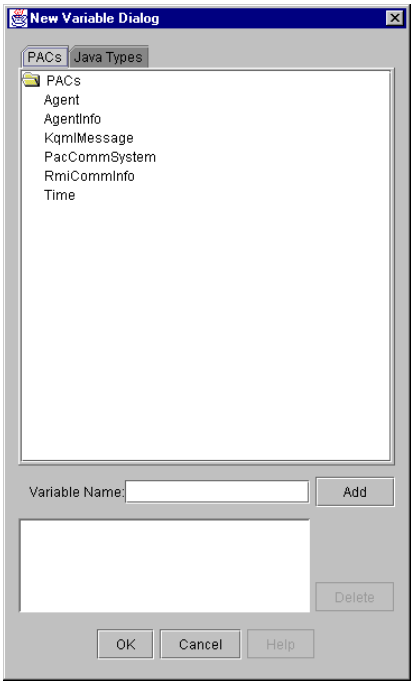


Figure 57. New Variable Dialog

The creation of a `KqmlMessage` variable differs in that a different pop-up dialog is displayed when you click on the **Add** button. You will be shown a **Binding Dialog** for the `KqmlMessage` variable, as shown in Figure 58. You can specify the KQML message binding to be an incoming or mental model message by using the pull-down menu. Once selected, you can click on the **OK** button so that the `KqmlMesage` variable can be added to the list of new variables and the dialog can be dismissed.

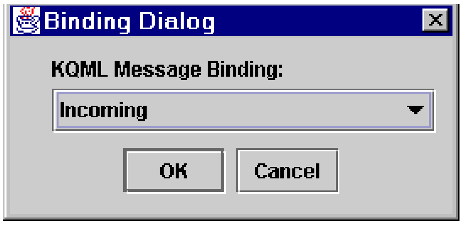


Figure 58. KQML Message Binding Dialog

The Rule Editor supports the creation of an `Array` variable. The `Array` type is listed in the **Java Types** panel. Like the creation of the `KqmlMessage`, a different pop-up dialog is displayed when you click on the **Add** button. You will be shown an **Array Dialog** for the `Array` variable, as shown in Figure 59. You can specify the array type by selecting a type from the list. Notice that you will be able to create an array of Java types, Vector, Hashtable, Enumeration, and PAC types. Once selected, you can click on the **OK** button so that the array variable can be added to the list of new variables and the dialog can be dismissed.

The **Defined Variables** dialog allows you to select previously defined variables for use in a message or mental pattern. Figure 60 shows the **Defined Variables** dialog. For any array types in the **Java Types** panel, the array variable will only have attributes for the component type and length. This dialog also allows you to cast the selected variable to a different type. To cast a selected variable, select from the **Casting Type** combo box. The combo box contains a list of all available PACs and Java types. Once a casting types has been selected, the type of the selected variable will be updated in the dialog. You can no click on the **OK** button to add the selected variable to the **Condition** panel's accumulator text field.

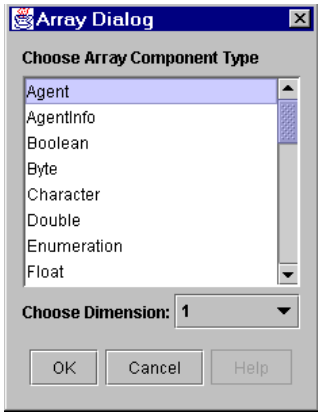


Figure 59. Array Dialog

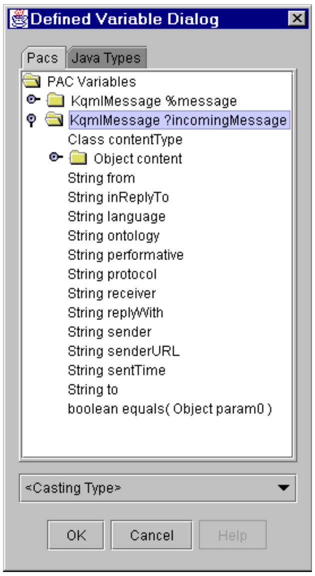


Figure 60. Defined Variable Dialog

The **Instances** pop-up dialog allows you to specify conditions based on named instances in the mental state. Figure 61 shows the **Instances** dialog.

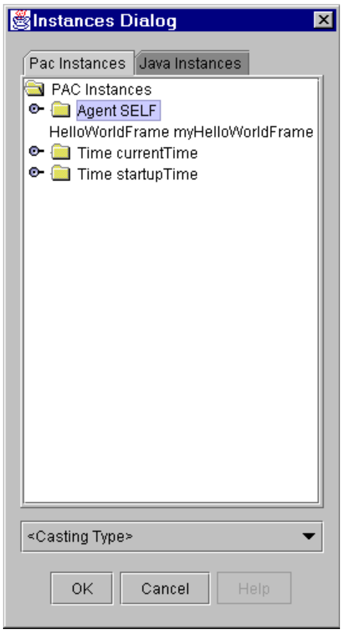


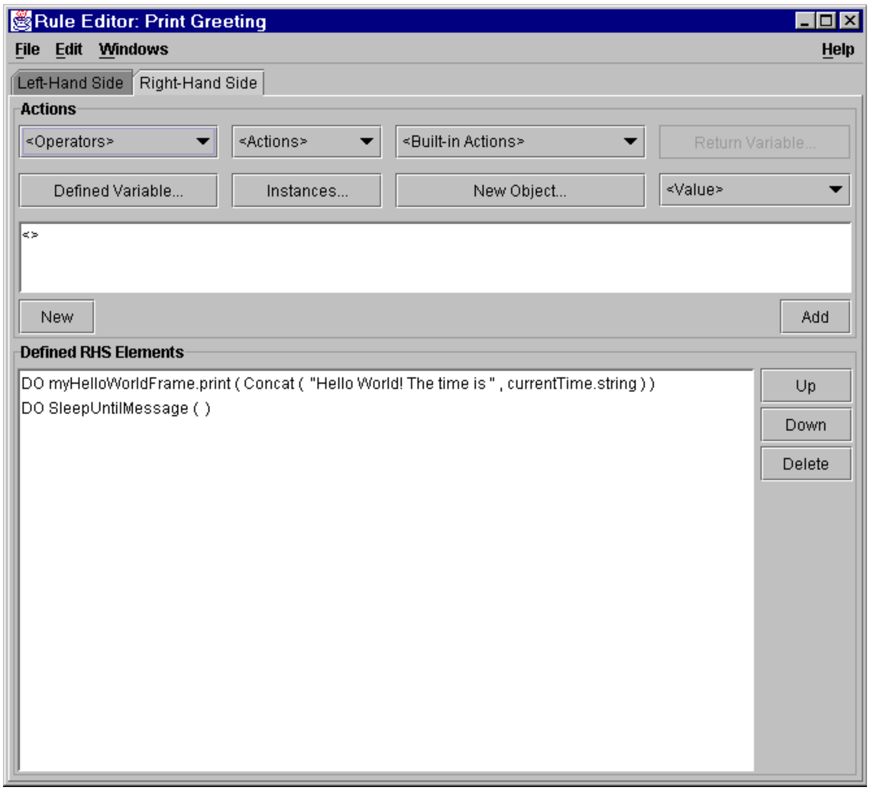
Figure 61. Instances Dialog

Like the **Defined Variables** dialog, the **Instances** dialog allows you to cast named instances to an available PAC or Java type. To cast a named instance, use the same method as was described in the **Defined Variables** dialog.

The **New** button allows the user to create a new message or mental condition. The same can also be done by selecting **File → New → New Condition**. Pressing the **Add** button will cause AgentBuilder to check to see if the pattern in the accumulator is valid. If the pattern is valid, it will be added to either the **Message Conditions** list or the **Mental Conditions** list.

RHS Editor

Figure 62 shows the RHS Editor. There are two main panels: the **Actions** panel and the **Defined RHS Elements** panel. The created patterns from the actions panels supply the **Defined RHS Elements** list with action statements.



```
DO myHelloWorldFrame.print ( Concat ( "Hello World! The time is " , currentTime.string ) )
DO SleepUntilMessage ( )
```

Figure 62. Rule Editor (RHS)

Action Panel

The **Actions** panel is composed of four pull-down menu, four pop-up dialogs, an accumulator line, a **New** button and an **Add** button. By using these dialogs and menus, you can create arbitrarily complex arguments and mental change statements.

The **Operators** pull-down menu allows access to various functional operators. This menu supplies a subset of the operators in the LHS operator lists. These operators (along with their operands) can be used as arguments for actions. These operators include:

- string functions
- arithmetic functions
- mathematical functions
- miscellaneous functions (ASSERT, RETRACT, SET_TEMPORARY)

The **Actions** pull-down menu allows access to the user-defined actions. The user-defined actions must first be defined in the Action Editor before they can be used in the Rule Editor. Please see the Action Editor section for more information about defined actions.

The built-in **Actions** pull-down allows access to the system-defined actions supported by the Run-Time System. For a more detailed description of the built-in actions please see “Intrinsics” on page 287.

There are three key components which are identical to their counterparts in the **Conditions** panel in the LHS editor: the **Values** pull-down menu, the **Defined Variables** dialog, and the **Instances** dialog.

The **New Object** dialog is used when defining new instances of PACs or Java types. Figure 63 shows the **New Object** dialog. To specify a new object, you must first select the type of object that you want to create. Once you have selected a type, you must then select the constructor to use for that type. The new object will then be inserted into the action pattern. Any parameters for the new object will need to be specified in the action pattern. The New Object dialog is used when specifying a parameter to an action that needs to be built at rule execution time.

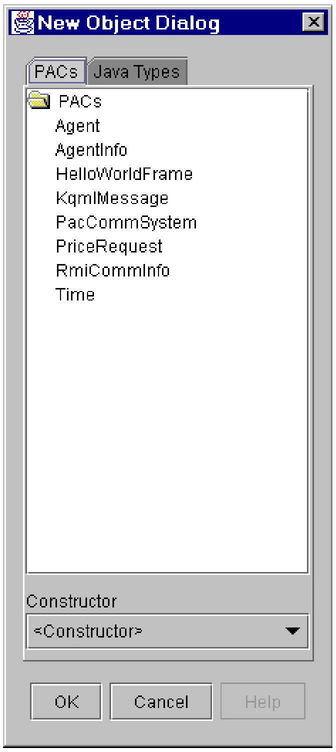


Figure 63. New Object Dialog

The **Return Variable** dialog is used to rename any automatically generated return variable. Figure 64 shows the **Return Variable** dialog. Return variables are automatically created for any private action, built-in action, or method that returns something. To change the name of a return variable, you first select the return variable in the Action pattern's accumulator. Then click on the **Return Variable** button to pop-up the dialog. Once the dialog is displayed, you select the type of the return variable that you are replacing. Once the type is selected, you can enter a variable name. Clicking on the **OK** button will then rename the selected return variable.

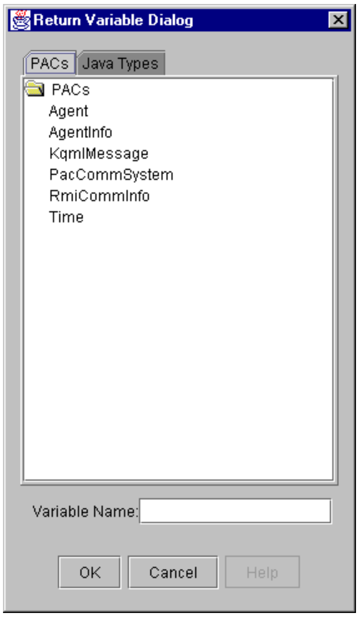


Figure 64. Return Variable Dialog

The **New** button allows you to create an action. You can also do this by selecting **File → New → New Action**. The **Add** button will check to see if the pattern in the accumulator is valid. If the pattern is valid, it will be added to the **Defined RHS Elements** list.

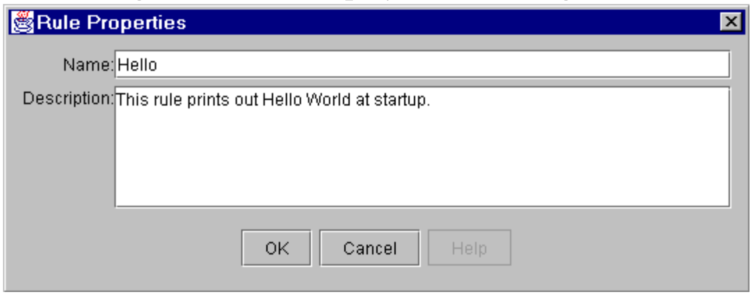
Defined RHS Elements Panel

The last panel in the RHS editor is the **Defined RHS Elements** panel which contains the list of private actions and mental changes on the RHS of the rule. The **Up** and **Down** buttons on the right of the panel can be used to rearrange the elements in the list, and the **Delete** button can be used to delete a selected element. It is crucial to maintain the integrity of the ordering within this list. Actions, except for the built-in `SendKqmlMessage`, will always be executed before the mental changes are performed. You may change the ordering within the list of actions, but if any action uses the returned value from another action, it must follow that action in the list. For example, if action `Foo` returns an `Integer` which is to be used as an argument for action `Bar`, then `Foo` must be listed ahead of `Bar` in the **Defined RHS Elements** list; this will cause Foo to be executed first and the return value will then be available for use in `Bar`.

Rule Editor Operations

Creating a New Rule

Constructing a new rule is relatively simple. Select the **File →New → New Rule** menu item. This clears all elements from the Rule Editor. If a rule was currently loaded, and it was modified, a dialog will be displayed asking if the current rule is to be saved. Figure 65 shows the dialog that will be displayed in creating a new rule.



Name: Hello

Description: This rule prints out Hello World at startup.

Figure 65. Rule Properties Dialog

The **Rule Properties Dialog** allows you to provide a name and description for the new rule. This information can later be changed by selecting the **Properties** menu item from the **Edit** menu. Once the information for the rule has been entered, select the **OK** button to apply the changes. Selecting **Cancel** button will cancel any changes to the rule's name or description. If this dialog was displayed as a result of selecting the **New Rule** menu item, selecting **Cancel** will defer naming the new rule.

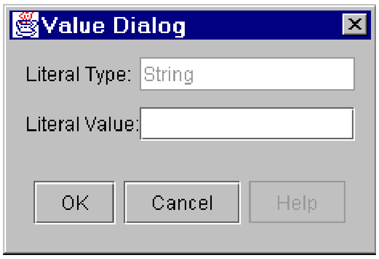


Figure 66. String Value Dialog

Loading an Existing Rule

To load an existing rule, select **File → Open**. A list of the rules previously defined for the agent will appear. Load the existing rule by either double clicking on its name or by clicking on its name and clicking on the **OK** button. Either selection method causes the rule to be loaded into the editor.

Constructing a Simple Mental Condition

The first step in constructing a simple mental condition is to choose the operator to be used, usually the `EQUAL` or `NOT_EQUAL` operators. Next, specify the operands that are to be compared, starting with the operand to the left of the operator. For example, a mental condition might test for equality between an attribute in a PAC variable and a string constant. Each operand can be an instance, a defined variable, a constant value, or the result returned from a function (e.g., the concatenation of two strings). To specify an operand, choose the correct dialog and select the desired element. Finally, specify the operand on the right side of the operator. Figure 66 shows the **Values** dialog where the string constant is defined. Enter the desired string and click on **OK**. This causes the string value to be entered into the current pattern. When the pattern is complete, click on **Add** to add the current pattern to the list of mental patterns.

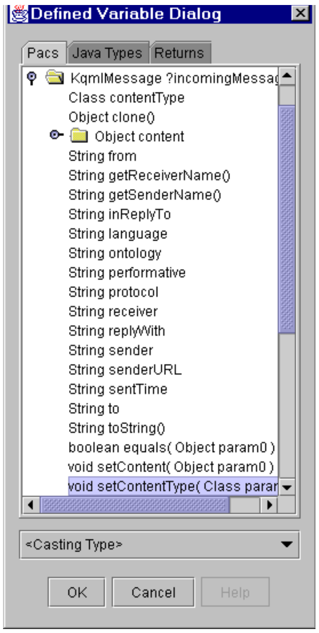


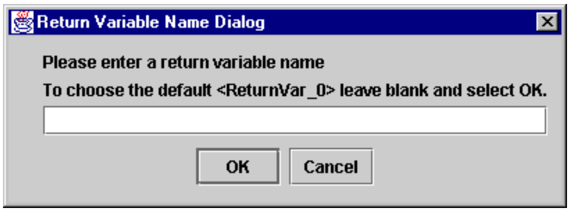
Figure 67. Direct Method Dialog

Direct Method Invocation

Directly invoking methods allows you to bypass creating a private action for a method you want to use in an action pattern. To use direct method invocation, you can either pop-up the **Defined Variable** or **Instances** dialog. In both dialogs, each non-primitive type is represented with a folder icon, and will contain zero or more child nodes. The child nodes represent field attributes of the type as well as any methods, with attributes listed first. To specify a method to be invoked, you must select the desired method and click on the **OK** button. The method invocation will then be inserted into the action pattern being built. Now, actions need only be created when you want to use the private action to create an initial commitment. Figure 67 shows this process.

Return Variable Naming

Anytime a method invocation or a built-in action that has a return variable is chosen, a **Return Variable Name Dialog** is displayed, as shown in Figure 68. You can either name the variable or accept a default name provided by AgentBuilder. Choosing **OK**, accepts the default name. There is no way to rename this variable after this operation, although you can easily recreate a pattern. The alternative is to type in the name you want the variable to have. If you enter a duplicate variable name, then an error dialog will be displayed and you must re-enter the name.


Return Variable Name Dialog

Please enter a return variable name

To choose the default <ReturnVar_0> leave blank and select OK.

Figure 68. Return Variable Name Dialog

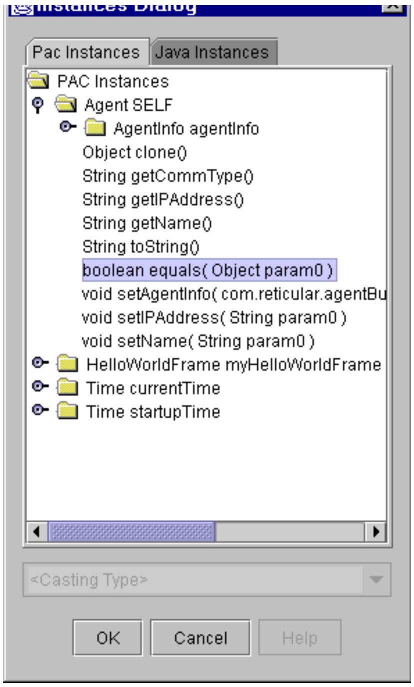
Using a Predicate Method

Predicate methods (i.e., methods that return a Boolean value) defined in the PACs may be used to construct mental conditions. The available predicate methods are displayed in the **Defined Variables** and **Instances** dialogs in the same way that the PAC attributes are displayed. Click on the method name to select it, then click on **OK** to insert the predicate method into the current pattern. After selecting the predicate method you'll need to specify values for any parameters required by the method. Figure 69 shows the format for the predicate methods on an example PAC in the **Defined Variables** dialog.

Constructing an Action Statement

Constructing an action statement is straightforward. Select the desired action from the **Actions** or **Built-in Actions** lists. Next, specify all parameters for the selected action. The Rule Editor automatically creates a variable to hold the value returned by each action statement so that the returned value can be used as a parameter in other actions or in mental changes. The tool adds these variables to the Defined Variables list so they can be selected for use in other RHS patterns.

Note: If an action has not been associated with an instance of a PAC, the `ConnectAction` built-in action must be executed first. For example, if an action named `Print` has been defined to use the `print(String)` method on a PAC instance named `myControlPanel-Pac` and the action has not been associated with the instance, `ConnectAction(Print, myControlPanelPAC)` must be executed before



```
PAC Instances
  Agent SELF
    AgentInfo agentInfo
    Object clone()
    String getCommType()
    String getIPAddress()
    String getName()
    String toString()
    boolean equals( Object param0 )
    void setAgentInfo( com.reticular.agentBu
    void setIPAddress( String param0 )
    void setName( String param0 )
  HelloWorldFrame myHelloWorldFrame
  Time currentTime
  Time startupTime
```

Figure 69. Predicate Methods in the Defined Variable Dialog

`Print` is executed. The section, “PAC Editor” on page 92, provides more information about actions and PACs.

Building an Assertion with a New Object

As previously mentioned, an assertion adds a new instance of a class to the agent's mental model. To specify an assertion, first select the **ASSERT** item in the **Operators** menu from the **Actions** panel. This causes a dialog to appear with a text field and a prompt. If you want to associate a name with the instance that will be asserted (e.g., you may want to assert a new `Location` instance named `currentLocation`), type the name into the text field and click **OK**. If you're not interested in providing a name for the instance, just leave the text field blank and click **OK** to close the dialog. After the dialog is closed the assertion template, **ASSERT(<>)** will be added to the mental change accumulator.

Next, click on the **New Object** button, which will cause the **New Object** dialog to appear, as shown in Figure 70. This allows you to select the class type for the instance. Click on the name of a class to select it. After you have selected a class type, select a constructor from the pull-down menu at the bottom of the dialog, then click on **OK**. This inserts the new object into the mental change accumulator. At this point you must specify the parameters for the constructor. The parameters can be existing objects found in the **Instances** or **Defined Variables** dialogs, or they can be other new objects created in the **New Objects** dialog. It is possible to use new objects as parameters to constructors, actions or other functions.

Closing the Rule Editor

You can close the Rule Editor by selecting the **Close** item in the **File** menu. If necessary, AgentBuilder will ask you whether you want to save your changes or not. By selecting **Yes** you can save

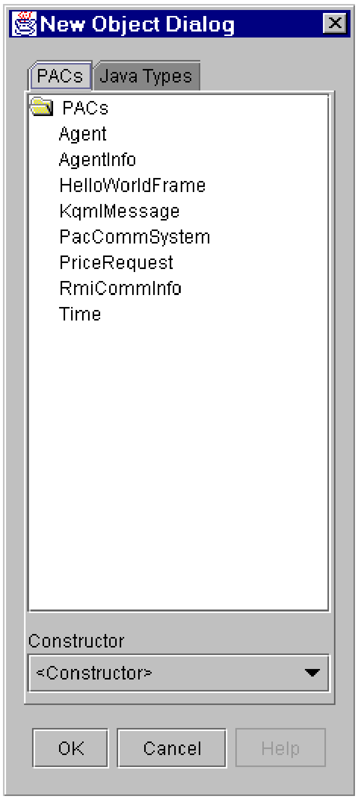
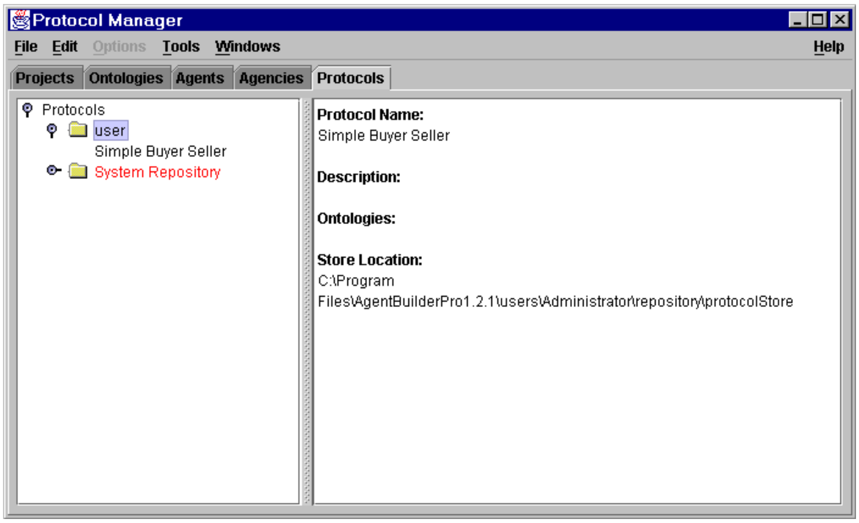


Figure 70. The New Object Dialog

all of your changes since the previous save. If you select **No** you'll revert to the rule configuration last saved.

F. Protocol Manager.

The Protocol Manager is shown in Figure 71. This tool allows you to create and view the set of protocols to be used with various agencies. This tool gives you a high-level view of protocols. The Protocol Editor is used to define all of the properties needed to use a protocol with a specified agency.


Protocol Manager
File Edit Options Tools Windows Help
Projects Ontologies Agents Agencies Protocols
Protocols
user
Simple Buyer Seller
System Repository

Protocol Name:
Simple Buyer Seller

Description:

Ontologies:

Store Location:
C:\Program Files\AgentBuilderPro1.2.1\users\Administrator\repository\protocolStore

Figure 71. The Protocol Manager

Overview

The **Protocol Manager** has five menus: **File**, **Edit**, **Tools**, **Windows** and **Help**. The **File** menu allows you to create new protocols, close the **Protocol Manager** and shut down AgentBuilder. The **Edit** menu allows you to **Cut**, **Copy**, **Paste** and **Delete** a protocol. You can use the **Tools** menu to open the **Protocol Editor** for a selected protocol.

The **Windows** menu allows you to quickly and easily switch between the various open AgentBuilder tools.

The protocol tree view allows you to view user-defined and system protocols. You can create and view any number of protocols. The system protocols in the protocol tree are displayed with red text labels. The red text labels signifies that the protocols are read-only and cannot be altered. A user's personal protocols are displayed with black text labels. When a protocol is selected in the left panel, the right panel displays general information about the protocol and includes a short textual description of the protocol as well as information about where the protocol is located. Note that the divider between the protocol tree structure and the properties window can be moved horizontally to provide more viewing space for the protocols.

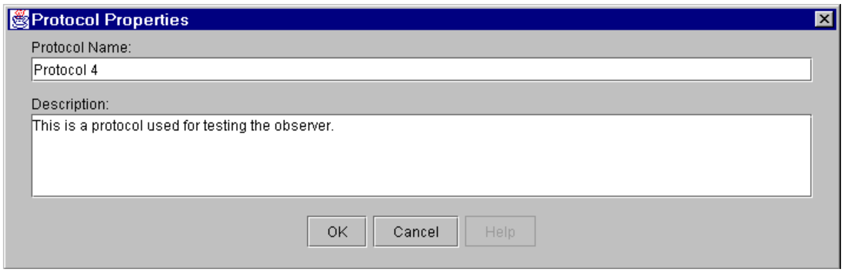
Operation

Using the Protocol Tree

The protocol manager uses the same tree structures found in other AgentBuilder tools The protocol manager has three levels in its tree structure. The highest level is the Protocols level, which contains repository folders, which in turn, contains the defined protocols. The repository folders are represented in the tree by a folder icon. The protocols are represented without an icon. Any protocols that read-only are displayed with red text labels.

Creating a New Protocol

A new protocol can be added to your user's repository folder by first selecting the user's repository folder. You would then select the **New Protocol** menu item from either the **File** menu or the repository folder's pop-up menu. This will bring up the dialog shown in Figure 72. The properties of the protocol can be entered into this dialog.



Protocol Name:
Protocol 4

Description:
This is a protocol used for testing the observer.

Figure 72. Protocol Properties Dialog

Cutting, Copying and Pasting a Protocol

Protocols can be cut, copied, or pasted. There are two ways to use the clipboard functions for a protocol. The first way is to use the **Edit** menu's **Cut**, **Copy** and **Paste** menu items. The other method is to use the protocol's pop-up menu for cut and copy, and the repository folder's pop-up menu for paste. Whichever method is used, a protocol must first be selected before a cut or copy operation. For the paste operation, a user or system repository folder must be selected. Invalid selections will be ignored. If the folder being pasted into already contains the name of the protocol being pasted, the protocol to be pasted will recursively have **CopyOf** prepended to its name.

Modifying Protocol Properties

The general protocol properties can be modified by right-clicking on the appropriate protocol in the tree structure and selecting the **Properties…** menu item from the pop-up menu.

Deleting Protocols

To delete a protocol from the protocol tree structure, simple select the desired protocol and choose **Delete** from the **Edit** menu or from the pop-up menu. This will display a confirmation dialog before deleting the selected protocol.

Launching Protocol Tools

You can launch the **Protocol Editor** tool from the **Tools** menu. It is necessary to first select the desired protocol before launching the **Protocol Editor**. If you launch the Protocol Editor without first selecting a protocol, AgentBuilder will remind you by displaying a dialog asking you to select a protocol.

Switching AgentBuilder Windows

To switch between different AgentBuilder windows, select the desired window in the **Windows** menu. This will bring the selected window to the foreground.

Closing Protocol Manager

To close the protocol manager, select the **Close** item under the **File** menu. This will close the protocol manager window and the **Protocol Editor**, if it is still open. If you have any unsaved changes in the **Protocol Editor**, you will be given a chance to save them.

Exiting AgentBuilder

To exit from AgentBuilder, select **Exit** from the **File** menu. AgentBuilder will then display a confirmation dialog before actually exiting the system.

Protocol Editor

The **Protocol Editor** tool is used to modify the state diagram and roles that are associated with a particular protocol. The **Protocol Editor** provides a a dialog for creating and modifying roles, a drawing canvas for graphically defining states and transitions and a dialog for viewing the protocol's state table.

Overview

The **Protocol Editor** contains a menu bar with file items: **File**, **Edit**, **Diagram**, **Windows** and **Help**. The **File** menu allows you to open protocols and save the current protocol. You can also save the protocol description to a file. The **File** menu also allows you to close the **Protocol Editor** and shut down AgentBuilder. The **Edit** menu allows you to cut, copy, and paste states. Using the **Delete** menu item you can delete states and transitions. The **Diagram** menu allows you to modify the current state diagram by adding new states or transitions. The dialogs for viewing the state table and the roles can also be accessed using the **Map** menu items. The **Map** menu also provides menu items for clearing the state diagram and refreshing the display. A sample state diagram is shown in Figure 73.

Operation

Creating a New State

You can create a new state by right-clicking on an unoccupied region of the state diagram and selecting **New State** from the pop-up menu. This will display the state properties dialog that will allow you to enter a name, description, and state type. The state properties dialog is shown in Figure 74. The state type provides a graphical way of indicating which state is the initial state, which states are final states, and which states are normal states. Only one state is allowed to be an initial state. You will receive an error dialog if you attempt to set more than one state as an initial state. The tool doesn't require you specify an initial and final state. The state types provide a graphical enhancement of the state diagram view. Clicking the **OK** button will create a new state on the state diagram. This state is represented by a circular node with the state's name as its label. Note that the point on the state diagram where you right-click is the location where the state is placed. If you decide not to create a new

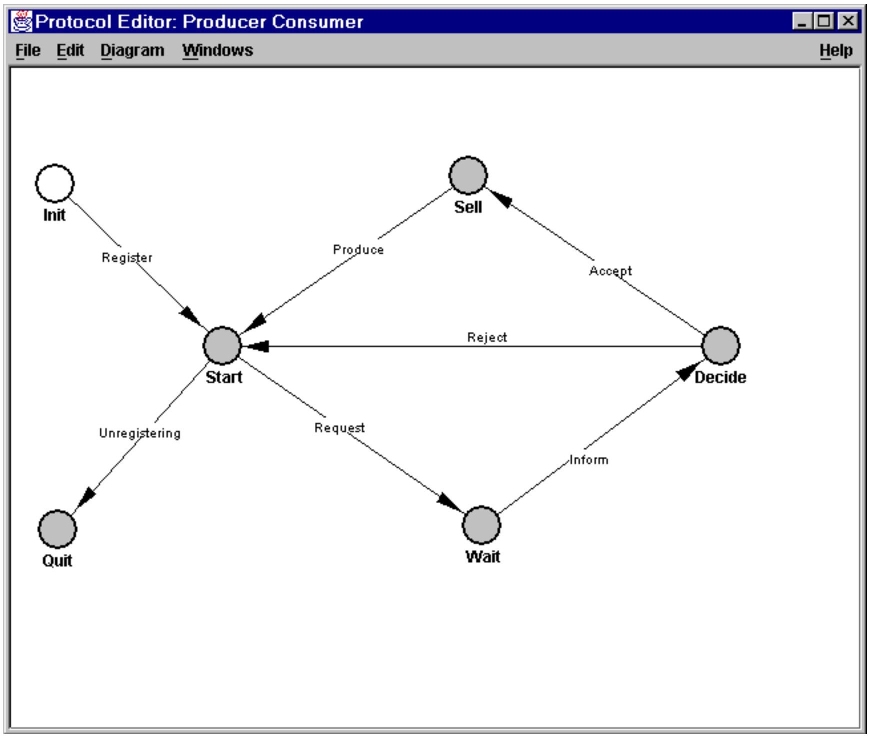


Figure 73. State Diagram

state, you can click on the **Cancel** button and the state diagram will remain unchanged. You can also use the **Map** menu to create a new state in a similar manner.

Cutting, Copying and Pasting a State

States can be cut, copied, or pasted. There are two ways to use the clipboard functions for a concept. The first way is to use the **Edit** menu's **Cut**, **Copy** and **Paste** menu items. The other method is to use the state node’s pop-up menus for cut and copy, and the Map's pop-up menu for paste. Whichever method is used, a state must first be selected before a cut or copy operation. If the map being pasted

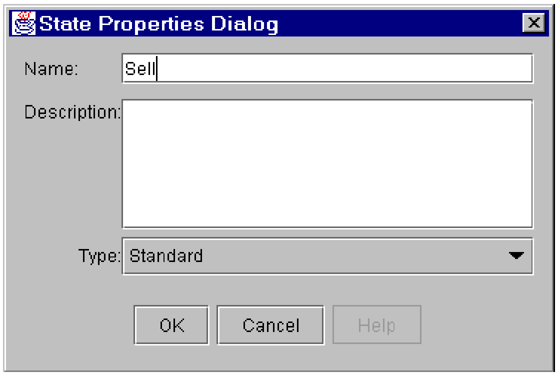


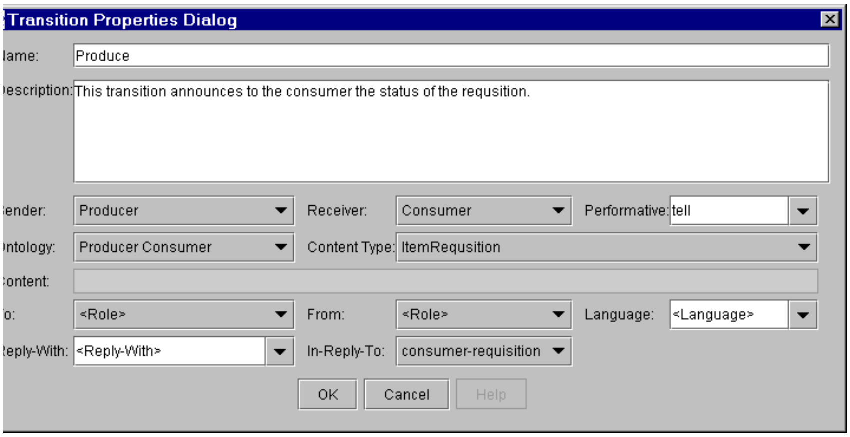
Figure 74. State Properties Dialog

into already contains the name of the state being pasted, the state being pasted will recursively have **CopyOf** prepended to its name.

Creating a New Transition

You can create a new transition by right-clicking on any unoccupied region of the state diagram and selecting the **New Transition** menu item. Once selected, the cursor will change to a cross-hair cursor and you can then click on the starting state and drag the cursor to the ending state. Once a valid starting state and ending state have been selected, the `Transition Properties` dialog will be displayed as shown in Figure 75. The **Properties Dialog** allows you to specify a name for the transition and an optional description.

The rest of the dialog allows you to specify the `KqmlMessage` that is used in the transition. The required KQML fields are the `sender`, `receiver`, `performative`, `ontology`, and `content type`. Since the `sender` and `receiver` fields are required, you must first have some roles defined before you can create any transitions. If no roles exists, you will see an error when trying to create a new transition (See Creating Roles below). Selecting an ontology will also load the selected ontology's objects into the **Content Type** combo box. For the content field, you can only enter a value if the selected con-


Transition Properties Dialog

lame: Produce

escription: This transition announces to the consumer the status of the requsition.

ender: Producer
Receiver: Consumer
Performative: tell

ntology: Producer Consumer
Content Type: ItemRequsition

ontent:

o: <Role>
From: <Role>
Language: <Language>

eply-With: <Reply-With>
In-Reply-To: consumer-requisition

Figure 75. Transition Properties Dialog

tent type is a Java primitive type. Entering a new values into the **Reply-With** combo box will automatically add the new entry to the **In-Reply-To** combo box. All entries will then be saved when the protocol is saved so that for any new transitions created, they will have the list of all **Reply-With** entries added.

Clicking on **OK** will add the transition between the selected states. An arrowhead will be drawn with the arrow pointing towards the ending node. Clicking on **Cancel** will cancel the creation of the new transition.

Moving a State

You can freely move a state node anywhere on the state diagram. This can be done by clicking on the desired state node and dragging the node to a new location. Transitions will adjust themselves automatically to maintain the connection with the relocated state.

Moving Multiple States

You can move multiple states at a time. First, select the states you wish to move. To do this, hold the **Control** key down and select the states using the mouse. Once you have a group of states selected, hold the **Control** key down and drag one of the selected states to a new location. You will notice that all the selected states will move in relation to the mouse cursor. To deselect the states, simply click on the canvas.

Deleting a State

You can delete an existing state node by clicking on that node and selecting **Delete** from the pop-up menu. Likewise, a selected state can be deleted using the **Delete** item in the **Edit** menu. Note that when a state is deleted, all links to that state are also deleted.

Deleting a Transition

You can delete an existing transition by selecting the transition and selecting **Delete** from the pop-up menu. You can also select **Delete** from the **Edit** menu with a selected transition.

Viewing and Altering State Properties

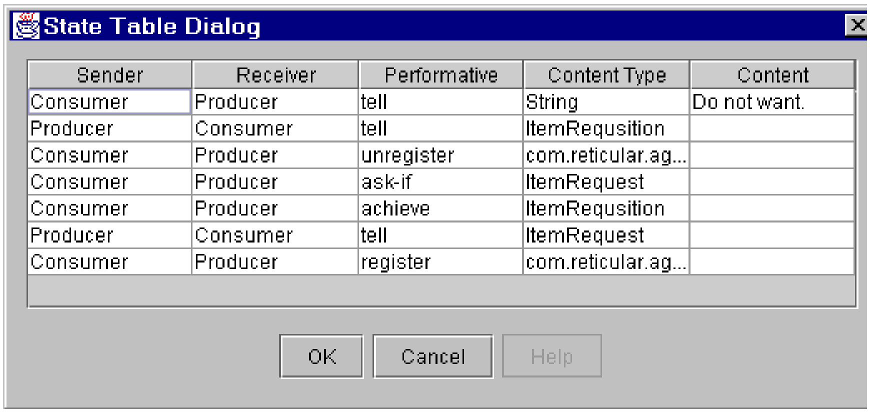
You can view and modify the name, description and type of an existing state by selecting **Properties…** from the state's pop-up menu. The state properties dialog will then be shown, and you can modify any or all of the state's properties. Clicking on **OK** will commit any changes that you have made to the state. Clicking on **Cancel** will revert the state to its original properties.

Viewing and Altering Transition Properties

You can view and modify the name, description and `KqmlMessage` fields of an existing transition by selecting **Properties…** from the state's pop-up menu. The **Transition Properties** dialog will then be shown, and you can modify any or all of the transition's properties. Clicking on **OK** will commit any changes that you have made to the state. Clicking on **Cancel** will revert the state to its original properties.

Viewing the State Table

The state table is generated from the transitions in the state diagram. If you currently have no transitions created, you will be given an error dialog and the state table dialog will be shown. Otherwise, the state table dialog will appear as shown in Figure 76. The state table dialog lets you see the important KQML message fields for each transition: `sender`, `receiver`, `performative`, `content type` and `content`. There is a combo box for selecting which role to view the state table. Currently, only the **All Roles** selection is supported.

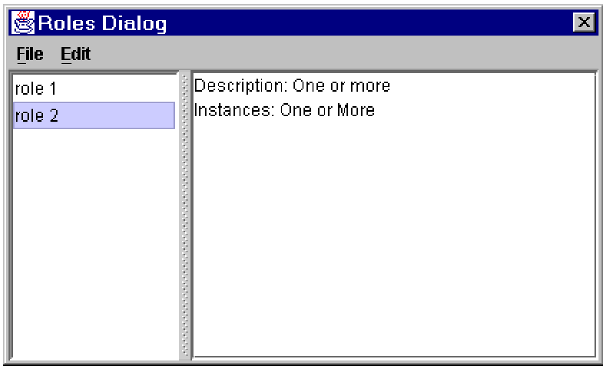


Sender	Receiver	Performative	Content Type	Content
Consumer	Producer	tell	String	Do not want.
Producer	Consumer	tell	ItemRequsition	
Consumer	Producer	unregister	com.reticular.ag...	
Consumer	Producer	ask-if	ItemRequest	
Consumer	Producer	achieve	ItemRequsition	
Producer	Consumer	tell	ItemRequest	
Consumer	Producer	register	com.reticular.ag...	

Figure 76. State Table Dialog

Viewing Roles

To view the roles that this protocol is using, select the **Diagram → Roles…** menu item. This will display the **Roles Dialog** as shown in Figure 77. The **Roles Dialog** has two panels: the left panel lists the defined roles, and the right panel gives a description of a selected role in the left panel. This dialog also contains a menu bar with a **File** and **Edit** menu. The **File** menu allows you to create a new role, and to close the dialog. The **Edit** menu gives you cut, copy, paste, and delete functions for the role. The **Edit** menu also allows you to bring up the properties dialog for a selected role.



Description: One or more
Instances: One or More

Figure 77. Roles Dialog

Creating Roles

Roles need to be created so that you can create transitions. To create a new Role, select the File->New menu item. The Role Properties dialog will be displayed and will allow you to specify the new role's properties. The Role Properties dialog is shown in Figure rolePropertiesDialog. The dialog will allow you to specify a name, description and instance number for the role. The instance number is specified by selecting the **Instance** combo box. The combo box is an editable combo box so that you can enter an integer number to specify the maximum number of instances allowed for the role.

Cutting, Copying and Pasting Roles

The **Edit** menu items will allow you to cut, copy and paste roles. To cut or copy a role, you must first select the role from the left panel, then select **Copy** or **Paste** from the **Edit** menu. To paste a role that has been cut or copied to the clipboard, select the **Paste** menu item from the **Edit** menu. The role will then be added to the list of roles in the left panel. If a role in the list already has the name of the role being pasted, the pasted role will recursively have **CopyOf** prepended to its name until it is unique.

Deleting Roles

To delete a role from the list, you must first select the role, then select **Edit → Delete**. A confirmation dialog will be shown before the role is actually deleted. Once deleted, the role is removed from the list.

Viewing and Modifying Role Properties

To view a role's properties, you must first select the role in the list, then select **Edit → Properties**. The **Role Properties** dialog will then be displayed showing the selected role's properties. Once the dialog is displayed, you can modify any or all of the roles properties. Clicking on **OK** will commit changes made to the role's properties. Clicking on **Cancel** will cancel any changes that have been made to the role.

Saving a Protocol

You can save a state diagram by selecting **Save** under the **File** menu. You can then close the **Protocol Editor** and return to it at a later time. The save action will save the states and transitions as well as their locations relative to each other.

Saving the Protocol to a File

The state diagram can be saved to a text file by selecting **File → Generate Printable**. Selecting this menu item will bring up a file dialog for saving a state diagram to a file. By default, the directory is set to the current working directory, and the filename is set to `protocol-name.txt`. The text file that is generated will contains a text description of the protocol's name and description as well as a description of every state and transition that make up the protocol.

Clearing the Protocol

If you wish to delete everything on the current state diagram and start again, you can do so by selecting **Clear** under the **Diagram** menu item or from the state diagram's pop-up menu. Note that this function will remove all states and transitions that have been entered. You will be presented with a confirmation dialog before actually clearing the state diagram. If you later decide that you liked the original state diagram and you have not yet saved the changes, you can just close the **Protocol Editor** or reload the **Protocol Editor** using the **File → Open** menu item.

Closing the Protocol Editor

You can close the **Protocol Editor** by selecting the **Close** item in the **File** menu. If you have any unsaved changes, you will be given a chance to save them before the tool is closed.

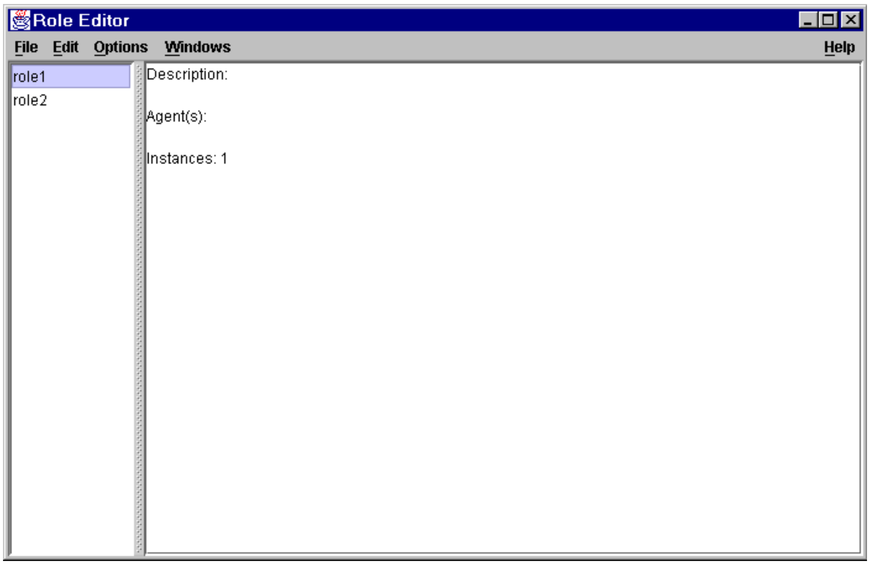
Role Editor

The Role Editor is used to assign an agency's agents to roles in a particular protocol. Using the **Role Editor,** you can update any or all agents so that necessary rules can be generated for their assigned roles.

Overview

The **Role Editor** contains a menu bar with five menu items: **File**, **Edit**, **Options**, **Windows** and **Help**. The **File** menu allows you to save the roles. From the **File** menu, you can also close the **Role Editor** and shut down AgentBuilder. The **Edit** menu allows you to view the properties of a selected role. The **Options** menu allows you to assign agents to a selected role, update a particular agent, or update all agents that belong to the currently loaded agency.

The **Role Editor** is made up of two panels: the left panel is a listing of the roles for a particular protocol, and the right panel displays a description of a selected role. Figure 78 shows the Role Editor.



Description:

Agent(s):

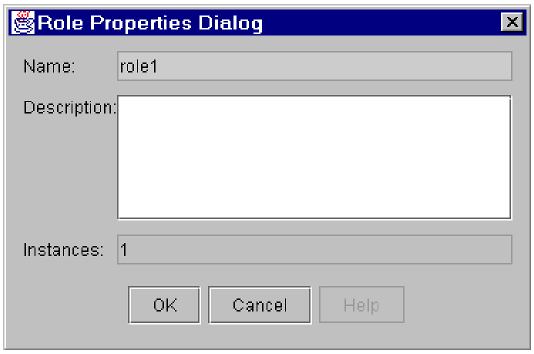
Instances: 1

Figure 78. Role Editor

Operation

Viewing and Modifying Role Properties

To view a role's properties, select the role in the left panel. The role's properties will then be displayed in the right panel. Alternatively, you can also select the role, then select the **Edit → Properties…** menu item. Either way, you will see the **Role Properties** dialog as illustrated in Figure 79. The **Role Properties** dialog will display the role's name, description and maximum number of instances allowed. Of these properties, only the role description can be modified.



Name: role1

Description:

Instances: 1

Figure 79. Role Properties Dialog

The role's name and instance number cannot be modified in the **Role Editor**. The roles are created in the **Protocol Editor.** If changes need to be made to a particular role, the changes will have be made in the **Protocol Editor**, and the protocol will then have to be re-imported into the **Agency Manager**.

Assigning Agents to Roles

In order for a protocol to be successfully applied to an agency, each role must be assigned to one or more agents. The only exception is when a particular role is specified to have zero or more agents, in which case, assigning an agent is optional. To assign one or more agents to a role, you must first select the role from the left panel. Then, click on the **Options → Assign Agent(s)**, and the **Assign Agent(s)** dialog will be displayed as shown in Figure 80.

The role being assigned to an agent is at the top of the dialog. There are two main panels in the dialog: the **Available Agents** panel lists the agents that belong to the currently loaded agency. Also, buttons are provided in this panel for adding a selected agent from the list,

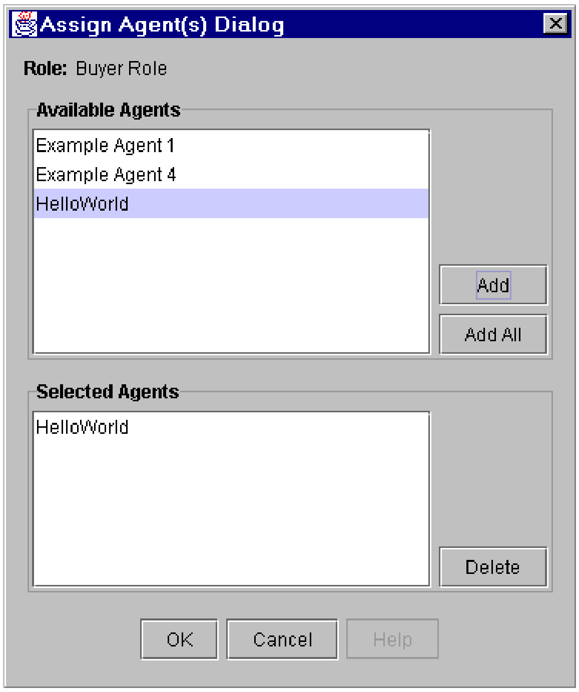


Figure 80. Assign Agent Dialog

or adding all agents from the list. To add a single agent, first select an agent, then click on the **Add** button. The selected agent will then be added to the list in the **Selected Agents** panel. Clicking on the **Add All** button does not require the user to select the entire list.

The **Selected Agents** panel will first list any agents that have already been assigned to the selected role. Agents can be removed from the role by selecting the agent in the list, then clicking on the **Delete** button. Clicking on the **OK** button will set the changes made to the role. Also, a check will be done to make sure that the number of agents assigned to the role does not exceed the maximum number of instances allowed for the role. Clicking on the **Cancel** button will cancel any changes made to the role's assignment.

Updating an Agent

Updating an agent will generate the necessary rules needed for the agent to fulfill its assigned role(s). An agent will need to be updated whenever the agent has been assigned to a role. To update an agent, select the **Options → Update Agent…** menu item. This will display the **Update Agent** dialog as shown in Figure 81. The dialog consists of a combo box that contains the agents for the currently loaded agency. Select the agent that you want updated, and click on the **OK** button. Clicking on the **Cancel** button will cancel the agent update.

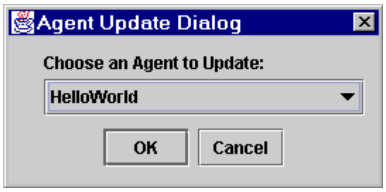


Figure 81. Agent Update Dialog

Updating All Agents

As a convenience, the **Role Editor** also allows you to update all agents for the currently loaded agency. Selecting the **Options → Update All Agents** menu item will update all agents. Selecting the menu item will then generate the rules needed for each agent to fully implement each of its assigned roles.

Saving the Roles

The roles can be saved by selecting **Save** under the **File** menu. You can then close the **Role Editor** and return to it at a later time. The save action will save the role assignments to each of the agents.

Switching Windows

To switch between different AgentBuilder windows, select the desired window in the **Windows** menu. This will bring the selected window to the foreground.

Closing the Role Editor

To close the role editor, select the **Close** item under the **File** menu. This will close the role editor window. If you have any unsaved changes in the role editor, you will be given a chance to save them.

Exiting AgentBuilder

To exit from AgentBuilder, select **Exit** from the **File** menu. AgentBuilder will then display a confirmation dialog before actually exiting the system.

Action Editor

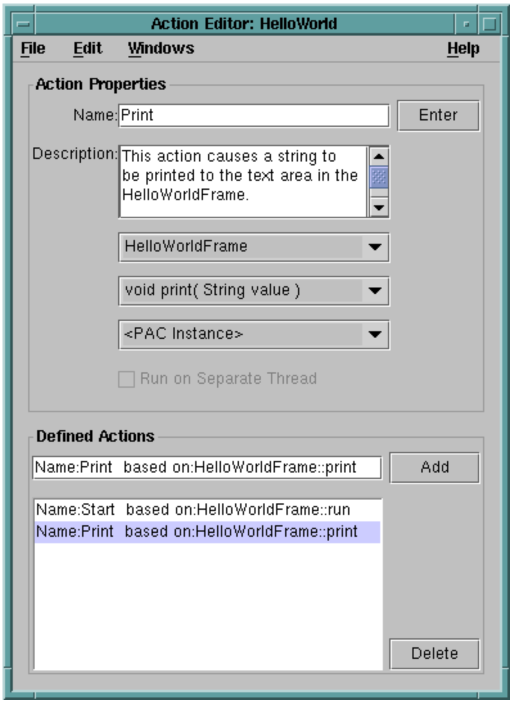
The Action Editor allows you to view, edit and create actions. An action is an association between an action name and a method on a PAC. The action name may be the same as the method name or it may be different. An action also contains lists of preconditions and effects which will automatically be added to any rule containing the action. The actions are optional, you can use direct method invocation instead. The Actions will be needed when you create Commitments that need to invoke methods on PACs.

Overview

You can open the Action Editor by selecting **Action Editor** from the **Tools** menu in the Agent Manager or selecting the **Action** tab in the Agent Manager. You can also double-click on an action in the **Agent Manager Actions Panel** to open the Action Editor. After opening the Action Editor, you can use the **File** menu to create a new action, save the current list of actions, close the Action Editor, and exit the system. The **Edit** menu allows you to delete actions from the defined actions list. The **Windows** menu lets you switch between other AgentBuilder tools that are open. The **Help** menu provides access to the AgentBuilder help system. The Action Editor is shown in Figure 1.

The **Action Properties** panel allows you to either create a new action or view a defined action. The **Action Properties** panel allows you to specify an action's name, description, PAC, PAC Instance and PAC method.

The **Defined Actions** panel allows you to add new actions to the list, or delete defined actions from the list.



Name: Print

Description: This action causes a string to be printed to the text area in the HelloWorldFrame.

HelloWorldFrame

void print(String value)

<PAC Instance>

Run on Separate Thread

Name:Print based on:HelloWorldFrame::print

Name:Start based on:HelloWorldFrame::run

Name:Print based on:HelloWorldFrame::print

Figure 1. Action Editor

Operation

Creating an Action

If you want to create a new action, the **Action Properties** panel must be cleared. If an action is currently being worked on, you can select the **New** menu item from the **File** menu. Selecting the **New** menu item will prompt you to confirm that you want to create a new action.

To create an action, fill out the **Action Properties** panel. The action's name must be entered into the **Name** text field. You can either press the **Enter** key on your keyboard while your cursor is in the **Name** text field, or click on the **Enter** button on the panel. Once an action name has been entered, the new action name will appear in the **Defined Action** text field. If the text in the **Defined Action** text field has scrolled out of view, you can click on the text field, and use the ←, →, **Home** and **End** keyboard keys.

You can select a PAC from the **<PAC>** combo-box. PACs that have been defined for the currently loaded agent will be loaded into the **<PAC>** combo-box. The selected PAC will be shown in the **Defined Action** text field.

The **Method** combo-box is dependent on the selection in the **PAC** combo-box. Once a PAC has been selected, the PAC's methods will be loaded into the **<Method>** combo-box. To complete the action definition, you must select a method. The selected method will then be shown in the **Defined Action** text field.

You can also specify a PAC Instance. If there are any PAC Instances that have been built from the currently selected PAC, they will show up in the **PAC Instance** combo-box.

You can also enter a description for the action in the **Description** text area.

Adding an Action

Once an action has been completely defined, the **Defined Action Add** button will become enabled. At the minimum, an action must have a name, PAC, and method selected. To add the action shown in the **Defined Action** text field, the user must press the **Add** button.

Pressing the **Add** button will add the action in the **Defined Action** text field to the list of defined actions. The **Action Properties** panel will be reset so a new action can be defined, or so an existing action can be displayed.

Viewing Defined Actions

You can view actions that are in the defined actions list. Selecting an action will cause the selected action to be loaded into the **Action Properties** panel.

Editing a Defined Action

Once you load a defined action into the **Action Properties** panel, you can edit the action. You can repeat any of the steps outlined in the section on "Creating an Action" on page 150. If you want to add the edited action to the list, then add the action as outlined in the "Adding an Action" on page 151. If the edited action has the same name as the old action, the edited action will overwrite the old action. Otherwise, it will be added to the list of defined actions, along with the old action.

Deleting a Defined Action

If you wish to delete a defined action, you have two options. You can first select the action in the **Defined Actions** list, then click on the **Delete** button. You can also select the action in the list, then select the **Delete** menu item from the **Edit** menu.

Saving the Actions

After you're finished adding and deleting actions, you need to save the action list to the current agent. To do this, select the **Save** menu item from the **File** menu.

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tools are open. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be brought to the front on your screen.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the Action Editor

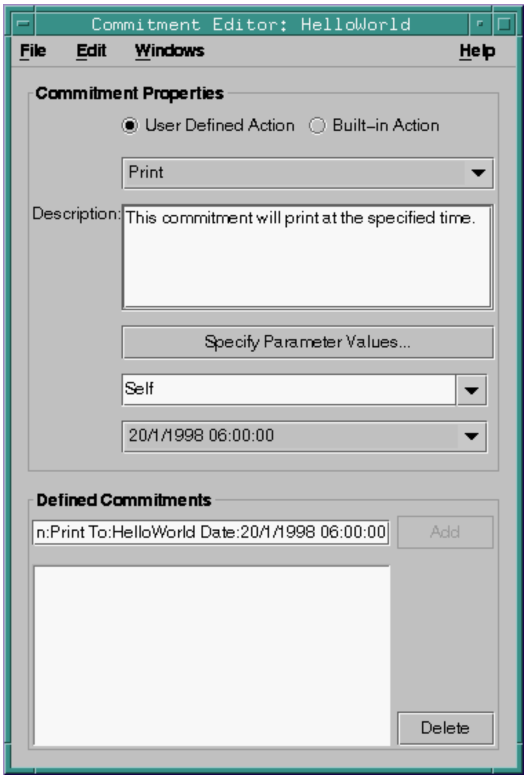
To close the Action Editor and leave all other tools open, select **Close** from the **File** menu. If you have modified the list of defined actions, you will be given a chance to save the action list before closing the Action Editor.

Exiting AgentBuilder

The **File** menu's **Exit** menu can be used to exit AgentBuilder. This will close all of the tools that are currently open. If you have modified the list of defined actions, you will be given a chance to save the action list before exiting the system.

Commitment Editor

The Commitment Editor allows you to view, edit, and create commitments. Commitments are based on a particular action; they specify the time an action will be committed and the agent to which the action will be committed. The Commitment Editor is shown in Figure 2.



(●) User Defined Action ( ) Built-in Action

Print

Description: This commitment will print at the specified time.

Self

20/1/1998 06:00:00

n:Print To:HelloWorld Date:20/1/1998 06:00:00

Figure 2. Commitment Editor

Overview

The **File** menu allows you to create a new commitment, save the current list of commitments, close the Commitment Editor, and exit the system. The Edit menu allows you to delete commitments from the defined commitments list. The **Windows** menu lets you switch between AgentBuilder tools that are open. The **Help** menu gives you access to the AgentBuilder Help system.

The **Commitment Properties** panel allows you to create a new commitment or view a defined commitment. The **Commitment Proper-**

ties panel allows you to specify a commitment's action, description, parameter values for the action, agent to commit the action to, and the time the action should be executed.

The **Defined Commitments** panel allows you to add new commitments to the list or delete defined commitments from the list.

Operation

Creating a Commitment

Before creating a new commitment, the **Commitment Properties** panel must be cleared. If a commitment is currently being edited, then you can select **New** from the **File** menu. Selecting the **New** menu item will cause the system to ask you to confirm that you wish to start editing a new commitment.

To create a commitment, you must enter the required information into the **Commitment Properties** panel. You must select an action for which this commitment is defined. The **Commitment Properties** panel contains two radio buttons for selecting **User Defined Actions** or **Built-In Actions**. The **Actions** combo box displays the appropriate actions depending on which radio button is selected. Currently, actions that are tied to commitments must be connected to a PAC instance. Once an action has been selected two things will happen. The selected action's method will be queried and, if the action's method contains any parameters, the **Specify Parameter Values** button will be enabled; if the selected action's method does not contain any parameters, the **Specify Parameter Values** button will be disabled. If the **Specify Parameter Values** button is enabled, then you must specify all parameter values. This process is explained in the following section. After selecting an action, the text field in the **Defined Commitment's** panel will show the name of the selected action.

After an action has been selected, you must specify the agent to whom the commitment will be made. The **Committed To** combo-box is located below the **Specify Parameter Values** button. This combo-box is an editable combo-box, and contains a list of available agents. The list of agents is constructed using the agencies to which the current agent belongs. To specify an agent, you can either select from the list of agents provided or manually enter an agent's name in the combo-box. If you manually enter an agent's name, you must press the **Enter** key in the combo-box so that the agent's name can be entered. Once an agent has been specified, the agent's name will be displayed in the **Defined Commitments** text field.

You must also specify a time to execute the action. The combo-box at the bottom of the **Commitment Properties** panel is the **Time** pull-down menu. The **Time** pull-down menu provides a number of options including **StartupTime**, **ShutdownTime** and a user-defined time. If you select the user-defined item, the system will display the **Time Dialog** described below. If a user-defined time is entered, the time will be shown in the combo-box. Once a time is specified for the commitment, this time will be shown in the **Defined Commitments** text field.

You can also enter a description for the commitment in the **Description** text area.

Specifying Parameter Values

The **Parameters Dialog** allows you to easily specify parameter values for simple and complex objects. The dialog consists of two sections: a **Parameter Tree** panel and a panel that can switch between a **Parameter** panel and a **Complex Parameter** panel. The **Parameters Dialog** is shown in Figure 3.

The **Parameter Tree** panel contains a hierarchical tree of the parameters. The tree nodes use the following convention for labeling

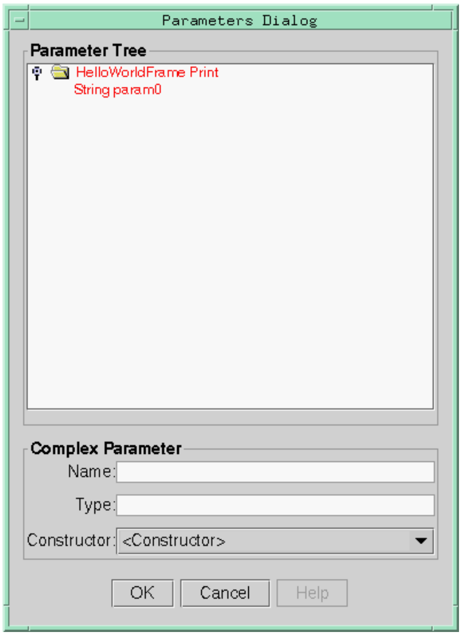


Figure 3. Parameters Dialog

name(type). The root node is labeled with the action name and the action's PAC type. The children of the root node are the parameters for the action's method. Any child nodes with a folder icon indicate that the parameter is a complex object whose constructor must be specified.

The color of the nodes is also significant. Nodes whose values have not been specified have a red label; nodes whose values have been specified have a black label. An unspecified node recursively sets its parent to be unspecified so that you can easily see which nodes have not been specified when the tree is fully collapsed.

To specify a parameter value, you must first select the parameter to be specified. Based on the type of node selected, the panel below the **Parameter Tree** will display either a **Parameter** panel or a **Complex Parameter** panel. Both the **Parameter** panel and the **Complex Parameter** panel contain a read-only **Name** and **Type** text field. The name and type of the selected node is shown in the read-only text fields. The primary difference between the two panels is that the **Parameter** panel allows you to specify a value for the selected parameter. To enter a value for the selected parameter, enter a value into the **Value** text field in the **Parameter** panel. To enter a null value for the parameter, you must type *`null`* into the value field. An empty string is considered to be a valid value and can also be entered into the value field. Once a value has been entered, you must press the **Enter** key on your keyboard while the cursor is positioned in the **Value** text field or click on the **Enter** button in order to enter the value in the selected parameter. The new value will be shown in the tree after it has been entered.

The **Complex Parameter** panel contains a pull-down menu for specifying a constructor for the selected parameter. This is necessary because the selected parameter's type is a complex object. You must now select a constructor from the **Constructor** pull-down menu.

If no constructors are listed, you must create a constructor for the complex object. To do this, go back to the Object Modeler and open the ontology that contains the complex object. You can then modify the object's properties to include one or more constructors. The object model must then be saved, and the PAC Editor must be opened from the Agent Manager. From the PAC Editor, update the object model that contains the updated PAC. (Please refer to “PAC Editor” on page 162 for information on updating PACs.)

Once a constructor is selected for the complex object, the tree node for the complex object will change based on the selected constructor. Any parameters that are in the selected constructor become children of the tree node. If the children's types are all Java types, then you can assign values to the parameters as described above. Otherwise, if the children's parameter types contain complex objects, the process of selecting a constructor and specifying values must be repeated. This process must be repeated until all leaves either contain Java types or the selected constructors for complex objects have no parameters.

Once you have fully specified the parameter tree, you can click on the **OK** button to register the new parameter values. Clicking on the **Cancel** button will cancel any changes that were made to the parameter tree.

Specifying a User-Defined Time

The **Time Dialog** allows you to specify a specific date and time that an action will be committed. The date is specified by selecting a month and day from the pull-down menu. The year must be entered (typed) in a four digit, numerical format. The time is specified using the format `hh:mm:ss`. Time uses three text fields with only numerical entries allowed. This dialog is shown in Figure 4.

Once you have specified a date and time, you must select the **OK** button in order for the time to be set for the commitment. Any invalid entries will cause an error dialog to be displayed. You will be given an opportunity to correct any errors. If you wish to cancel operations using the **Time Dialog** then select the **Cancel** button.

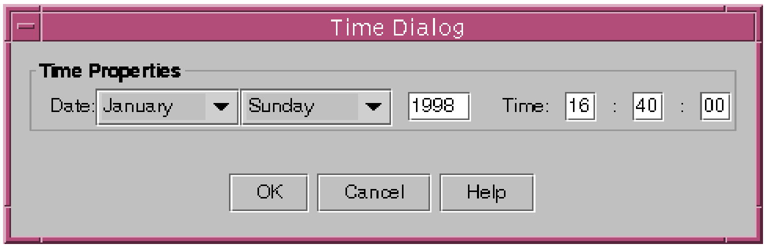


Figure 4. Time Dialog

Selecting the **Cancel** button will ensure that the **Time** pull-down menu remains unmodified.

Adding a Commitment

Once a commitment has been completely defined, the **Defined Commitments** panel **Add** button is enabled. In order to construct a completely defined commitment, you must specify a selected action, any parameter values specified through the **Parameters Dialog**, a selected agent to commit to, and a time to perform the action.

Pressing the **Add** button will add the commitment in the **Defined Commitments** text field in the list of defined commitments. Notice that the **Commitment Properties** panel is reset, readying it for defining a new commitment or viewing defined commitments

You can view commitments that are in the **Defined Commitments** list. Selecting a commitment will cause the selected commitment to be displayed in the **Commitment Properties** panel.

Editing a Defined Commitment

Once you load a defined commitment into the **Commitment Properties** panel, you can edit the commitment using any of the steps outlined in “Creating a Commitment” on page 155. If you want to add the edited commitment to the list, then add the commitment as outlined in “Adding a Commitment” on page 160. If the edited commitment has the same action and agent, the edited commitment will overwrite the old commitment. Otherwise, it will be added to the list of defined commitments.

Deleting a Defined Commitment

If you want to delete a defined commitment, you have two options. You can first select the commitment in the **Defined Commitments** list and then click on the **Delete** button to the right of the list. You can also select the commitment in the list, then select the **Delete** menu item in the **Edit** menu.

Saving Commitments

Once you have completed adding and deleting commitments, you must save the commitment list for the current agent. To do this, select the **Save** menu item from the **File** menu.

Switching Windows

The **Windows** menu is a dynamic menu that displays a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools when multiple tools are open. To switch to the desired tool, select the tool from the **Windows** menu. The selected tool will then be brought to the front on your display.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the Commitment Editor

To close the Commitment Editor and leave all of the other tools open, you should select **Close** from the **File** menu. If you have modified the list of defined commitments, you will be given an opportunity to save the commitment list before closing the Commitment Editor.

Exiting AgentBuilder

The **File** menu's **Exit** menu item should be selected to exit Agent-Builder. This will close all of the tools that are currently open. If you have modified the list of defined commitments, you will be given an opportunity to save the commitment list before exiting the system.

PAC Editor

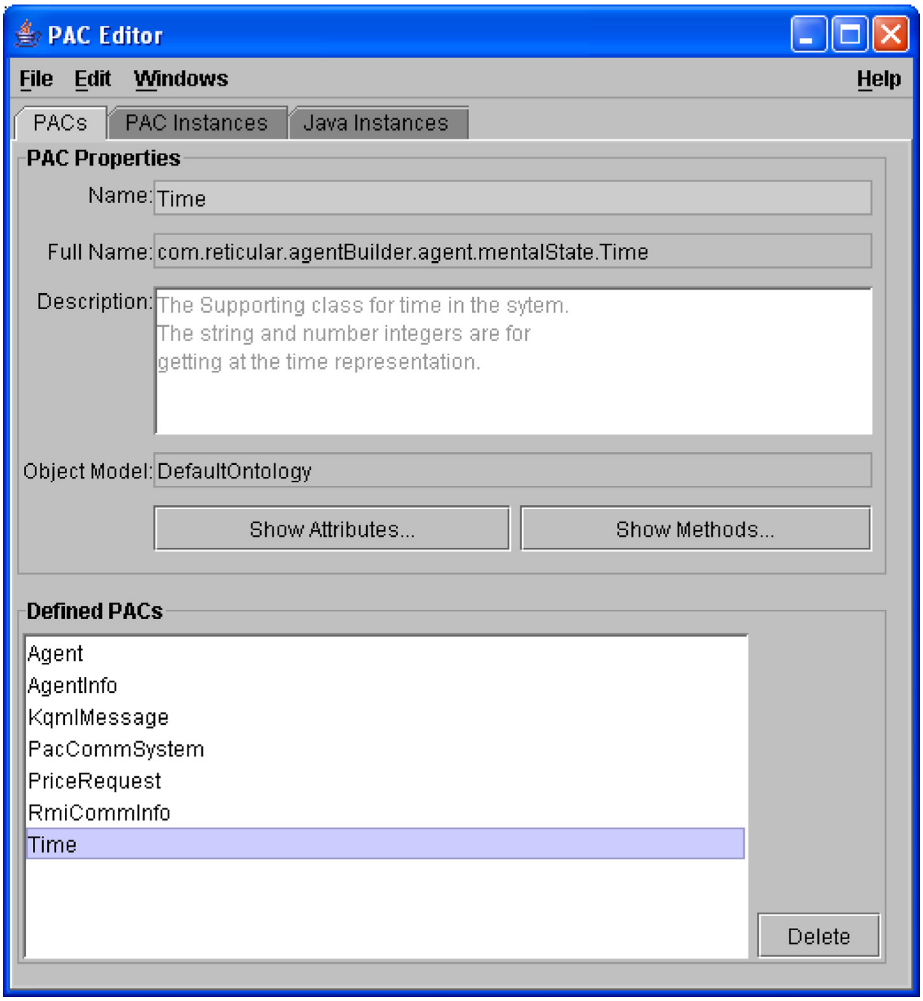
The PAC Editor allows you to view, edit and create PACs, PAC Instances, and Java Instances. The data created by this editor is used extensively by the Rule Editor to construct various patterns. You cannot directly use classes defined in the Object Modeler in your rules. The PAC Editor is necessary for creating PACs from the classes so they can be made a part of your rules.

Overview

The **File** menu can be used to perform various operations on the PACs and instances. From the **File** menu, you can create new PACs, PAC Instances and Java instances. You can also save the list of defined PACs, PAC Instances, and Java Instances. You can also import and update the list of defined PACs. As with all of the other AgentBuilder tools, the **File** menu provides the **Close** and **Exit** menu items for closing the editor and exiting AgentBuilder. The PAC Editor is shown in Figure 5.

The **Edit** menu allows you to delete any of the items in the defined PACs, PAC Instances, or Java Instances list. The **Windows** menu lets you switch between other open AgentBuilder tools. The **Help** menu provides you with access to the AgentBuilder help system.

The main panel of the PAC Editor contains a **Panel Options** panel. The **Panel Options** panel contains tabs that allow you to switch between the PACs, PAC Instances, or Java Instances panel. The contents of each of the panels is explained in the following sections.



Name: Time

Full Name: com.reticular.agentBuilder.agent.mentalState.Time

Description: The Supporting class for time in the sytem. The string and number integers are for getting at the time representation.

Object Model: DefaultOntology

Agent
AgentInfo
KqmlMessage
PacCommSystem
PriceRequest
RmiCommInfo
Time

Figure 5. PAC Editor

Operation

Importing PACs

The current version of AgentBuilder does not allow you to directly

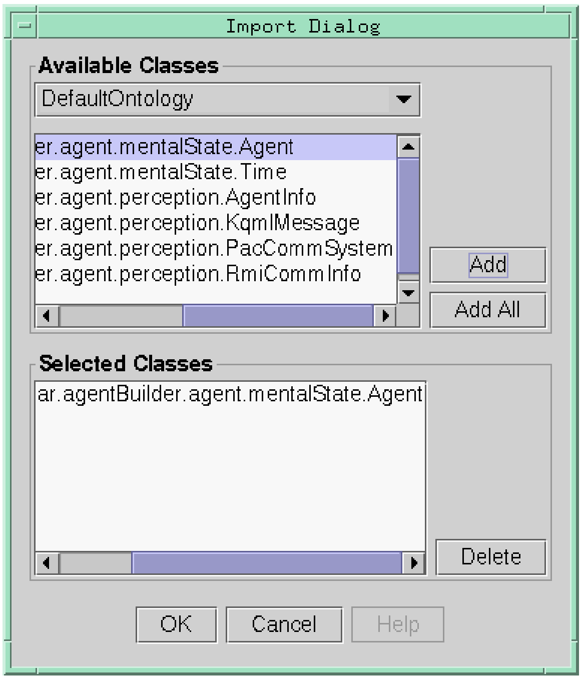

Import Dialog
Available Classes
DefaultOntology
er.agent.mentalState.Agent
er.agent.mentalState.Time
er.agent.perception.AgentInfo
er.agent.perception.KqmlMessage
er.agent.perception.PacCommSystem
er.agent.perception.RmiCommInfo
Add
Add All
Selected Classes
ar.agentBuilder.agent.mentalState.Agent
Delete
OK
Cancel
Help

Figure 6. Import Dialog

create PACs. Instead, you must import the PACs from classes defined in object models. To import PACs, select the **Import** menu item from the **File** menu. Figure 6 shows the **Import Dialog**.

The **Import Dialog** consists of two sections: the **Available Classes** panel and the **Selected Classes** panel. To import PACs, you must first select an object model in the **Available Classes** panel. Once an object model is selected, the dialog will display the names of the classes that have been defined in the selected object model.

The contents of the **Object Model** pull-down menu is a complete listing of all defined ontologies in the user's repository. You have several ways to select classes to be added to the list in the **Selected**

Classes panel. The first method is to select a class in the list and then click on the **Add** button. If you need to select multiple classes, you must hold down the **Control** key on your keyboard while selecting the classes. Once all of the classes have been selected, click on the **Add** button. If you want to add all of the classes listed, you can click on the **Add All** button. You can now select a different object model and continue to add classes as described above.

You can delete classes listed in the **Selected Classes** panel at any time. To do this, first select the classes to be deleted. Then, click on the **Delete** button. The selected class will then be deleted from the list.

When you are finished adding classes to the list, you must click on the **OK** button. Clicking on the **Cancel** button will cancel the import operation. If the **OK** button has been pressed and there were classes listed in the **Selected Classes** panel, then the selected classes will be converted into PACs. The new PACs will be displayed in the **Defined PACs** panel. If you cannot see the **Defined PACs** panel, make sure that the **PACs** radio button in the **Panel Options** panel of the PAC editor is selected.

Viewing Defined PACs

To view the properties of a defined PAC, select the PAC in the **Defined PACs** list. Selecting a defined PAC from this list will cause the PAC Properties panel to display information about the selected PAC. The selected PAC's name, description, package, ontology, and sendable status will be displayed. To view attributes or methods simply click the appropriate button and an attribute or methods list will appear in a dialog box.

Deleting a Defined PAC

You can use the **Defined PACs** panel to delete any of the defined PACs. To do this, you must first select the PAC to be deleted. Then,

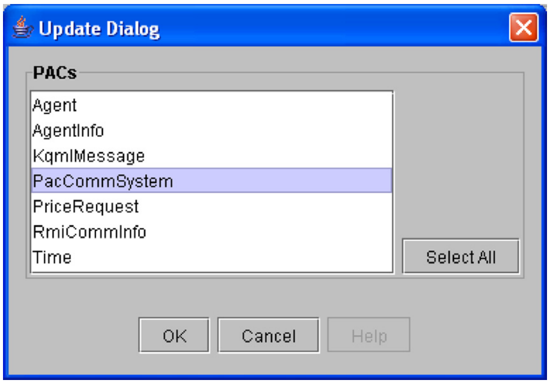


Figure 7. PAC Update Dialog

click on the **Delete** button. Clicking on the **Delete** button will remove the selected PAC from the list.

Updating PACs

On occasion you will need to update a PAC from an ontology. The PAC Editor provides a way to update the list of defined PACs. Figure 7 shows the **Update Dialog** used to update defined PACs.

The **Update Dialog** is accessed from the **File** menu's **Update** menu item. The dialog contains a PACs panel, which lists all the PACs associated with the agent. To update the defined PACs in the **Update Dialog**, select the PACs you wish to update. Holding the **Ctrl** key while clicking on the PACs, will allow you to make multiple PAC selections. If you want to update all of the PACs listed, click the **Select All** button. Once the PACs that need to be updated are selected, you need to click on the **OK** button.

Any of the defined PAC's whose object models were updated will now display the updated information.

Defining PAC Instances

Once you have defined a PAC, you can define PAC Instances. To define a PAC Instance, select the PAC you want to instantiate. All other items in the **PAC Instance Properties** panel are optional for defining a PAC Instance. To specify a name for the PAC Instance, type the name into the **Name** text field. To add the name to the PAC Instance, you must either press the **Enter** key on your keyboard while the cursor is in the **Name** text field, or click on the **Enter** button next to the **Name** text field. You can also type the description of the PAC Instance in the **Description** text area.

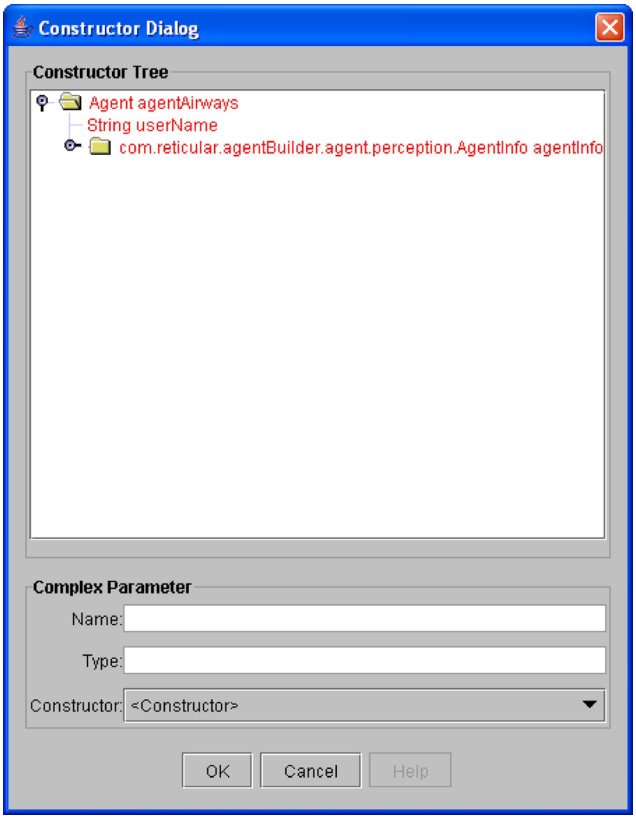
Once a PAC is selected, you are given the option of defining the PAC Instance as an Initial PAC Instance. This is done using the **Initial PAC Instance** checkbox. If the **Initial PAC Instance** checkbox is selected, the **Specify Constructor** button will be enabled. A necessary condition for defining an initial PAC Instance is that its constructor be selected and that all of its constructor parameters be specified.

Specifying Constructors for PAC Instances

To specify constructors for a PAC Instance click on the **Specify Constructor** button. Figure 8 shows the dialog that will appear.

The **Constructor Dialog** allows you to specify the constructor and its corresponding parameter values for various kinds of objects. The dialog consists of two sections: a **Constructor Tree** panel and a panel that switches between a **Parameter** panel and a **Complex Parameter** panel.

The **Constructor Tree** panel contains a hierarchical tree of the parameters. The tree nodes use the `name(type)` convention for labeling. The root node is labeled with the PAC Instance name and the PAC Instance's PAC type. The children of the root node are the parameters for the PAC type's constructors. Any nodes with a



Agent agentAirways
String userName
com.reticular.agentBuilder.agent.perception.AgentInfo agentInfo

Name:
Type:
Constructor: <Constructor>

Figure 8. PAC Instance Constructor Dialog

folder icon indicate that the parameter is a complex object whose constructor needs to be specified.

The color of the nodes is also significant. Nodes whose values have *not* been specified have a red label; nodes whose value *have* been specified have a black label. An unspecified node recursively sets its parent to be unspecified, so you can easily see which nodes have not been specified even when the tree is fully collapsed.

To specify a parameter value, you must first select the parameter to be specified. The panel below the **Constructor Tree** will change to either a **Parameter** panel or a **Complex Parameter** panel based on the type of node selected. Both the **Parameter** panel and the **Complex Parameter** panel contain a read-only **Name** and **Type** text field. The name and type of the selected node are shown in the read-only text fields. The difference between the two panels is that the **Parameter** panel allows you to specify a value for the selected parameter. To enter a value for the selected parameter, enter a value into the **Value** text field in the **Parameter** panel. To enter a null value for the parameter, type *`null`* into the value field. An empty string is considered to be a valid value and can also be entered into the value field. Once a value has been entered, you can press the **Enter** key on your keyboard while the cursor is in the **Value** text field, or the **Enter** button can be clicked. This will cause the value to be registered with the selected parameter. The entered value is shown in the tree once it has been registered.

The **Complex Parameter** panel contains a pull-down menu for specifying a constructor for the selected parameter. This is because the selected node type is a complex object. You must select a constructor from the **Constructor** pull-down menu.

If no constructors are listed, you can create a constructor for the complex object. To do this, go back to the Object Modeler and open the ontology that contains the complex object. You can then modify the object's properties and include one or more constructors. The object model must then be saved and the defined PACs updated.

Once a constructor is selected for the complex object, the tree node for the complex object will change. Any parameters that are in the selected constructor become children of the tree node. If the children's types are all Java types, then you can assign values to the parameters as described above. Otherwise, if the children's parameter types contain complex objects, the process of selecting a constructor and specifying values must be repeated. This process must be repeated until all leaves either contain Java types, or the selected constructors for complex objects have no parameters.

Once you have fully specified the constructor tree, you can click on the **OK** button to register the new parameter values. Clicking on the **Cancel** button will cancel any changes that were made to the constructor tree.

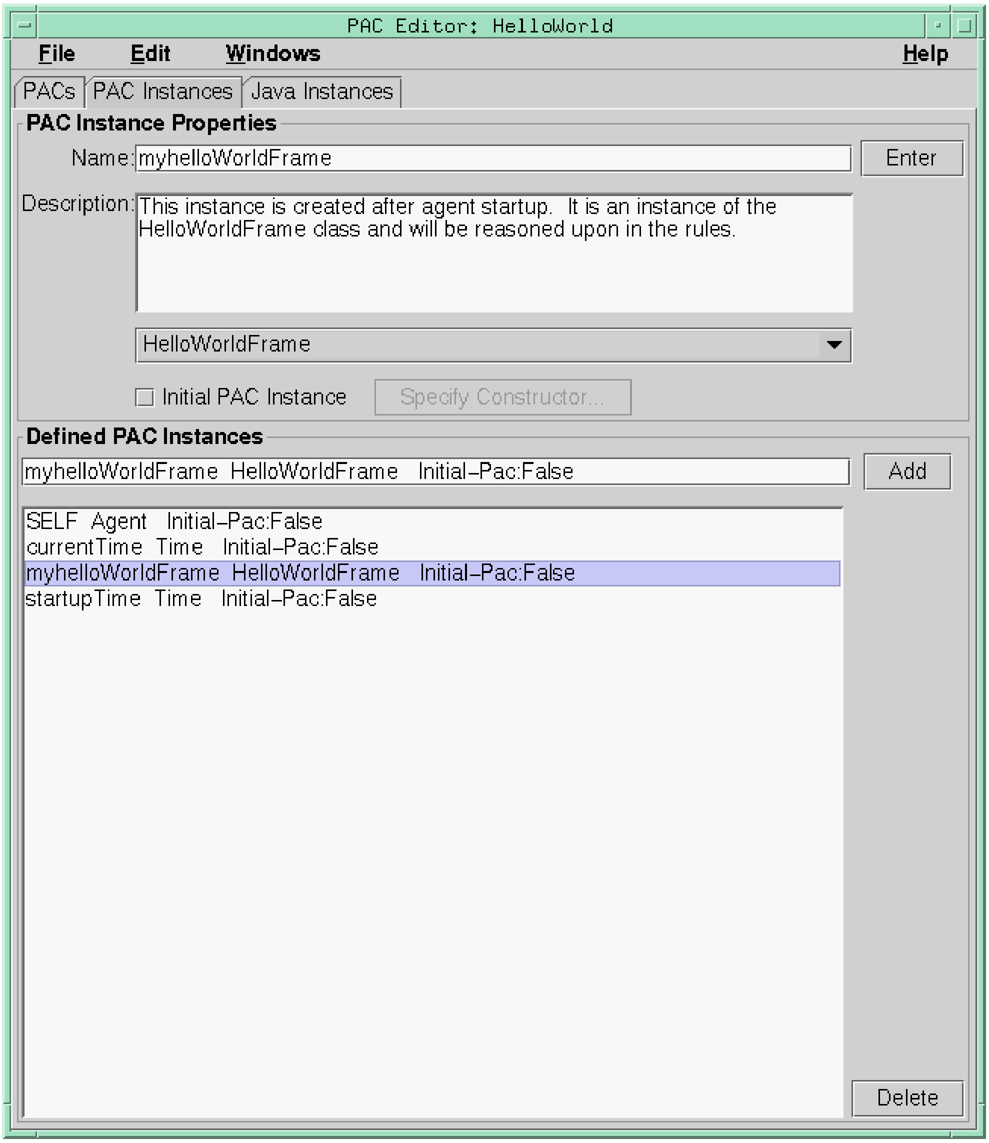
Adding a PAC Instance

Once a PAC Instance has been defined, the PAC Instance can be added to the list of Defined Instances. If the **Add** button in the **Defined Instances** panel is disabled, then the PAC Instance has not been completely defined. Check to make sure that a PAC has been selected. If the PAC Instance is checked to indicate that it is an Initial PAC Instance, make sure the constructor and its parameters have been fully specified.

Once the **Add** button is enabled, you can add the PAC Instance to the list of defined PACs. Clicking on the **Add** button will add the PAC Instance to the list. Notice that the **Instance Properties** panel is cleared and the text field in the **Defined Instance** panel updated to reflect that a new PAC Instance has been defined.

Viewing Defined PAC Instances

To view the properties of a defined PAC Instance, select the PAC Instance in the **Defined Instances** list. Selecting a defined PAC Instance will cause the **Instance Properties** panel to display information about the selected PAC Instance. The selected PAC Instance's name, description, PAC and Initial PAC Instance status will all be displayed. If the PAC Instance is an Initial PAC Instance, you can view the specified constructor by clicking on the **Specify Constructor** button (see Figure 9).



PAC Instance Properties

Name: myhelloWorldFrame

Description: This instance is created after agent startup. It is an instance of the HelloWorldFrame class and will be reasoned upon in the rules.

HelloWorldFrame

☐ Initial PAC Instance

Defined PAC Instances

myhelloWorldFrame HelloWorldFrame Initial–Pac:False

```
SELF  Agent   Initial–Pac:False
currentTime  Time   Initial–Pac:False
myhelloWorldFrame  HelloWorldFrame   Initial–Pac:False
startupTime  Time   Initial–Pac:False
```

Figure 9. Viewing PAC Instances

Editing a PAC Instance

To edit a PAC Instance, follow the above instructions for viewing a defined PAC Instance. Once the PAC Instance is loaded into the **Instance Properties** panel, you can modify the PAC Instance. When finished modifying the PAC Instance, the PAC Instance can be added to the list of Defined Instances as described above. Note that if the modified PAC Instance has the same name as the original PAC Instance, the modified PAC Instance will overwrite the original PAC Instance. If the modified PAC Instance has a different name, the PAC Instance will be added to the list in the same manner as a new instance.

Deleting a Defined PAC Instance

From the **Defined Instances** panel, you can delete any of the defined PAC Instances. To do this, first select the PAC Instance to be deleted. Then, click on the **Delete** button. Clicking on the delete button will delete the selected PAC Instance from the list.

Creating Java Instances

The PAC Editor also supports the construction of Java Instances. Java Instances are used in the Rule Editor in the construction of patterns. To specify a Java Instance, you must select a Java type for the instance. Like PAC Instances, everything else in the **Java Instance Properties** panel is optional. Java types include some non-primitive types, which are Arrays, Enumeration, Hashtable, Object, and Vector. To specify a name for the Java Instance, type the name into the **Name** text field. To add the name to the Java Instance, you must either press the **Enter** key on your keyboard while the cursor is in the **Name** text field, or click on the **Enter** button next to the **Name** text field. You can also type in a description of the Java Instance in the **Description** text area.

Once a Java type has been selected, you are given the option to make the Java Instance an Initial Java Instance. This is done using the **Initial Java Instance** checkbox. The **Initial Java Instance** checkbox will be disabled for non-primitive java types. If the **Initial Java Instance** checkbox is selected, the text field and the **Add** button adjacent to the checkbox are enabled. The text field is used to enter a value for the Java Instance. Once a value has been entered in the text field, you can either press the **Enter** button while the cursor is in the text field or click on the **Add** button.

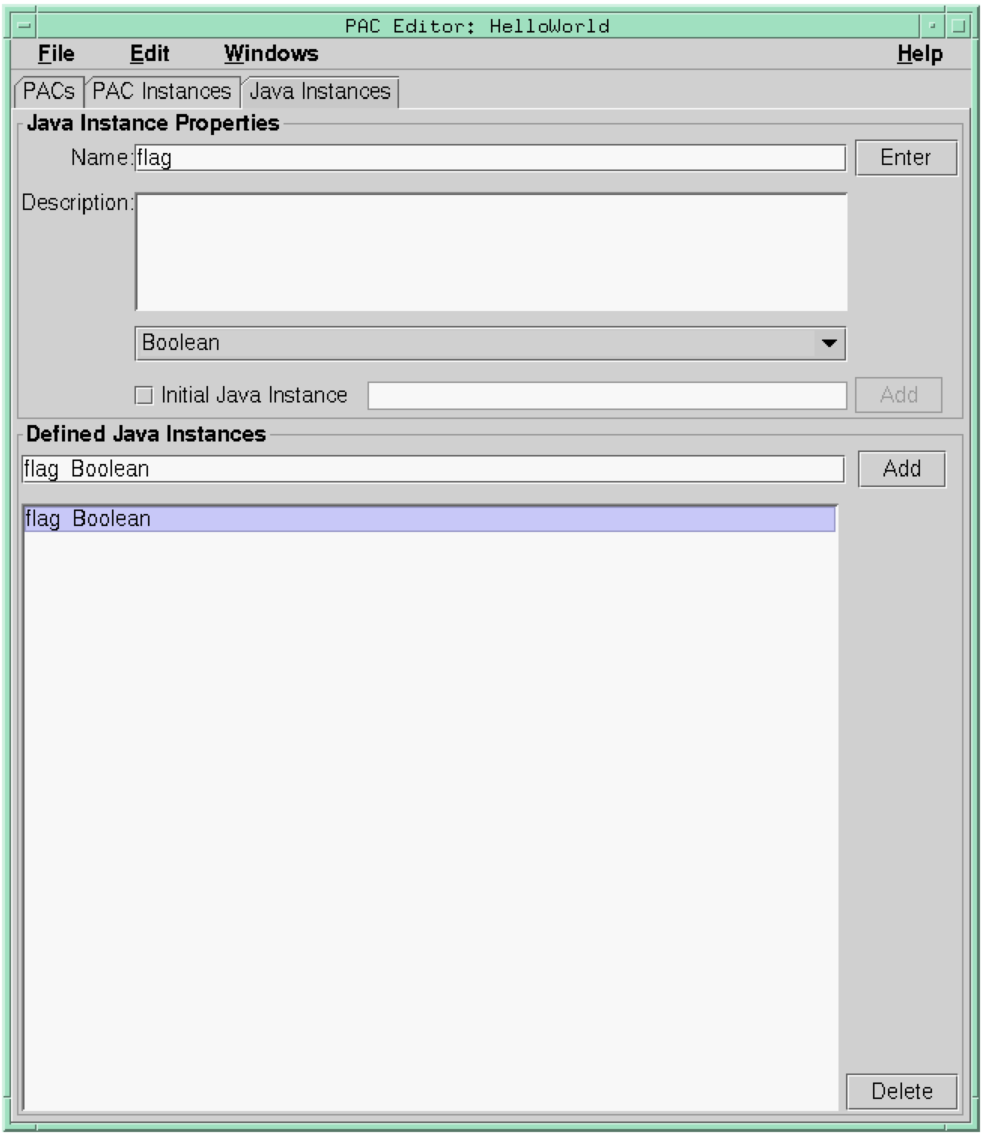
Adding a Java Instance

Once a Java Instance has been defined, the Java Instance can be added to the list of Defined Java Instances. If the **Add** button in the **Defined Java Instances** panel is disabled, then the Java Instance has not been completely defined. Check to make sure that a Java type has been selected. If the Java Instance is checked as an Initial Java Instance, ensure that a value has been specified.

Once the **Add** button is enabled, you can add the Java Instance to the list of Defined Java Instances. Clicking on the **Add** button will add the Java Instance to the list. Note that the **Java Instance Properties** panel is cleared and the text field in the **Defined Java Instance** panel updated to show that a new Java Instance has been defined.

Viewing Defined Java Instances

To view the properties of a defined Java Instance, you must select the Java Instance in the **Defined Java Instances** list. Selecting a defined Java Instance will cause the **Java Instance Properties** panel to display information about the selected Java Instance. The selected Java Instance's name, description, Java type and Initial Java Instance status will be displayed. If the Java Instance is an initial Java instance, the text field will display the value assigned to the Java Instance (see Figure 10).



Java Instance Properties

Name: flag

Description:

Boolean

☐ Initial Java Instance

Defined Java Instances

flag Boolean

flag Boolean

Figure 10. Viewing Java Instances

Editing a Java Instance

To edit a Java Instance, follow the instructions for viewing a defined Java Instance. When the Java Instance is loaded into the **Java Instance Properties** panel, you can modify the Java Instance. When finished modifying the Java Instance, the Java Instance can be added to the list of Defined Java Instances as described above. Note that if the modified Java Instance has the same name as the original Java Instance, the modified Java Instance will overwrite the original Java Instance. If the modified Java Instance has a different name, the Java Instance will be added to the list.

Deleting a Defined Java Instance

Using the **Defined Java Instances** panel, you can delete any of the defined Java Instances. To do this, you must first select the Java Instance to be deleted, then click on the **Delete** button. Clicking on the **Delete** button will delete the selected Java Instance from the list.

Saving

If you have modified the contents of the Defined PACs, PAC Instances or Java Instances, you must save the new or modified data to the currently loaded agent. To do this, select the **Save** menu item in the **File** menu. The **Save** menu item selection will save all PACs, PAC Instances, and Java Instances to the currently loaded agent.

Switching Windows

The **Windows** menu is a dynamic menu that contains a list of all AgentBuilder tools that are open. This menu facilitates switching between the tools. To switch a different tool, select the tool you wish to switch to from the **Windows** menu list. The selected tool will then be brought to the front on your display.

Accessing Help

The help system can be accessed using the **Help** menu item. The **About** menu item allows you to read the help information for the current tool. The **Index** menu item will display an index to the help system's contents. The **Tutorial** menu item will display the Quick Tour of the AgentBuilder toolkit. The **About AgentBuilder** menu item will display the AgentBuilder logo along with the version number and copyright information. The **AgentBuilder Home Page** menu item will display the home page for the AgentBuilder product.

Closing the PAC Editor

If you want to close only the PAC Editor and leave all of the other tools open, select **Close** from the **File** menu. If you have modified the list of defined PACs, PAC Instances, or Java Instances, you will be prompted to save everything before closing the **PAC Editor**.

Exiting AgentBuilder

The **File** menu's **Exit** menu item allows the user to exit Agent-Builder. This will close all of the tools that are currently open. If you have modified the list of defined PACs, you will be given an opportunity to save the modified PACs list before exiting the system.

Rule Editor

The Rule Editor tool is used to specify the behavioral rules used by the agent. (Note: Throughout this section *behavioral rules* are simply called *rules*). Rules are the basic control mechanism for Agent-Builder agents; they determine how the agent responds to external and internal stimuli. Each rule consists of a set of conditions and the desired actions or mental changes that will occur when those conditions are met. The set of conditions is sometimes referred to as the rule's the *left-hand side*, or *LHS*; the actions and mental changes are the rule's *right-hand side*, or *RHS*. The Rule Editor has separate panels for constructing the two sides of a rule.

The Rule Editor allows you to create two different types of conditions for the left-hand side of a rule. One type of condition is based upon external events and is termed a *message condition*. A message condition is a test performed on any new messages arriving in the agent's input buffers (e.g., comparing the name of the message sender against an expected sender name). The second type of condition is based upon the agent's internal state and is termed a *mental condition*. A mental condition is a test performed on beliefs in the mental model of the agent.

Patterns on the left-hand side of a rule generally consist of a combination of operators, variables, constant values, and references to named instances in the mental model. Most variables will bind to all instances of a specified type in the mental model. For example, assume `?i` is the name of an `Integer` variable. In the pattern `(?i >= 4)` the variable `?i` will bind to every `Integer` instance in the mental model and the value of each binding will be used in the comparison. In contrast, a *named instance variable* is a special type of variable which will bind only to a named instance with the specified name and type. For example, the pattern `( currentCount >= 4 )` contains the named instance variable `currentCount`, which can bind only to the named instance `currentCount` in the mental model. A named instance variable is similar to a global variable in that it can be used in any of the agent's rules and will always bind to the same instance in the mental model.

The following example illustrates the difference between the two types of variables, using a trivial mental model with three `Integer` beliefs.

Mental Model

```
Integer<width>  8
```

```
   Integer<length>  3
   Integer<currentCount> 5
```

Rule 1

```
IF ( ?i >= 4)
THEN ( DO SystemOutPrintln( "Rule 1 Fired." )
```

Rule 2

```
IF ( currentCount >= 4)
THEN ( DO SystemOutPrintln( "Rule 2 Fired." )
```

In Rule 1 the `Integer` variable `?i` will bind to each of the `Integer` values in the mental model and each value will be compared to 4. Rule 1 will fire twice: once with `?i` bound to 8, then once with `?i` bound to 5. The binding of `?i` to 3 does not satisfy the pattern in Rule 1 so the rule will not fire with that binding. In Rule 2 the named instance variable `currentCount` will bind to the value 5 from the `currentCount` named instance (which satisfies the pattern) and so Rule 2 fires once.

Technically there's a difference between a *named instance* and a *named instance variable*: a named instance is part of the mental model (i.e., it's a belief), but a named instance variable is a component of a pattern in a rule. In some situations it's important to understand this distinction. It's possible (and fairly common) for a pattern in a rule to refer to a named instance which does not always exist in the mental model. Adding a named instance variable to a pattern in a rule does **not** guarantee that the associated named instance will be present in the mental model.

For example, consider the Print rule of ExampleAgent3 in the AgentBuilder User’s Guide. The second `IF` pattern in that rule contains a `Boolean` named instance variable named `Ready_to_print`:

```
( Ready_to_print EQUALS true )
```

The `Ready_to_print` named instance variable refers to an instance that does not exist in the agent's mental model until it gets asserted during execution, by the `Connect` rule. Even though at the start there is no instance named `Ready_to_print` in the mental model, the `Print` rule *does* contain a `Ready_to_print` named instance variable. This pattern evaluates to true only if the `Ready_to_print` instance exists in the mental model and has the value `true`. The pattern evaluates to false if the mental model does not contain a named instance that matches the named instance variable in the pattern. In this example, the `Print` rule is prevented from firing until the `Ready_to_print` instance is found in the mental model.

When the `Connect` rule is fired it performs the mental change:

```
( ASSERT ("Ready_to_print" true) )
```

This is the mental change that actually creates the `Ready_to_print` instance and assigns it a name and a value. After this instance has been asserted into the mental model the pattern for the `Print` rule (shown above) will evaluate to true and the `Print` rule will be activated.

A named instance variable is similar to a global variable in that it can be used in any of the agent's rules and will always bind to the same instance in the mental state. In the previous example with the `currentCount` named instance, several rules could have patterns that refer to `currentCount` and all of them would access the same value each cycle.

For the right-hand side of a rule, the Rule Editor allows you to specify the actions that occur when the rule's conditions are satisfied. Actions can be created so that they affect the agent's mental model of the external world.

Overview

The Rule Editor is divided into two major panels; one panel for editing the left-hand side of a rule and one panel for editing the right-hand side. The editor for the left-hand side of the rule is referred to as the *LHS editor*; the editor for the right-hand side is referred to as the *RHS editor*.

The Rule Editor contains a menu bar with four items: **File**, **Edit**, **Windows** and **Help**. The **File** menu allows you to open (i.e., load and start editing) previous rules as well as save the current one. Selecting **File → New → New Rule** creates a completely new rule, or selecting other items under **File → New** will clear selected sections in the existing rule. For example, choosing **File → New Condition** causes the condition accumulator line to be cleared.

The main features of the Rule Editor are the *accumulator text field*s and the *pattern lists.* Each editor has accumulator text fields; these are single-line text fields with a **New** and **Add** button directly below them. The accumulators are used to accumulate the components of arbitrarily complex conditions or action expressions. The pattern lists are at the middle and bottom of the LHS editor; the action/mental change list is at the bottom of the RHS editor.

A rule is made up of individual patterns which are built up one at a time in the accumulator line and then added to the lists. By utilizing the pull-down menus as well as context sensitive pop-up menus, you can construct complex expressions as desired. Several of the dialogs used in constructing the rule use a tree paradigm. In addition, the **Up** and **Down** buttons at the right side allow you to change the ordering of individual lines. Thus it's easy for you to rearrange ordering of patterns (or actions or mental changes) within the rule.[1]

Rule Properties

Edit→Properties allows the user to access the rule properties. The two basic parts of the **Rule Properties** dialog are the name of the rule and the rule description. There are few constraints on the name of a rule; only that it be unique within the rule set and that it not use hidden characters. You're encouraged to choose descriptive rule names; you may even find that descriptive phrases work well as rule names. The description window allows you to add a description to accompany the rule. No size limitations are imposed, but generally only a few sentences are required.

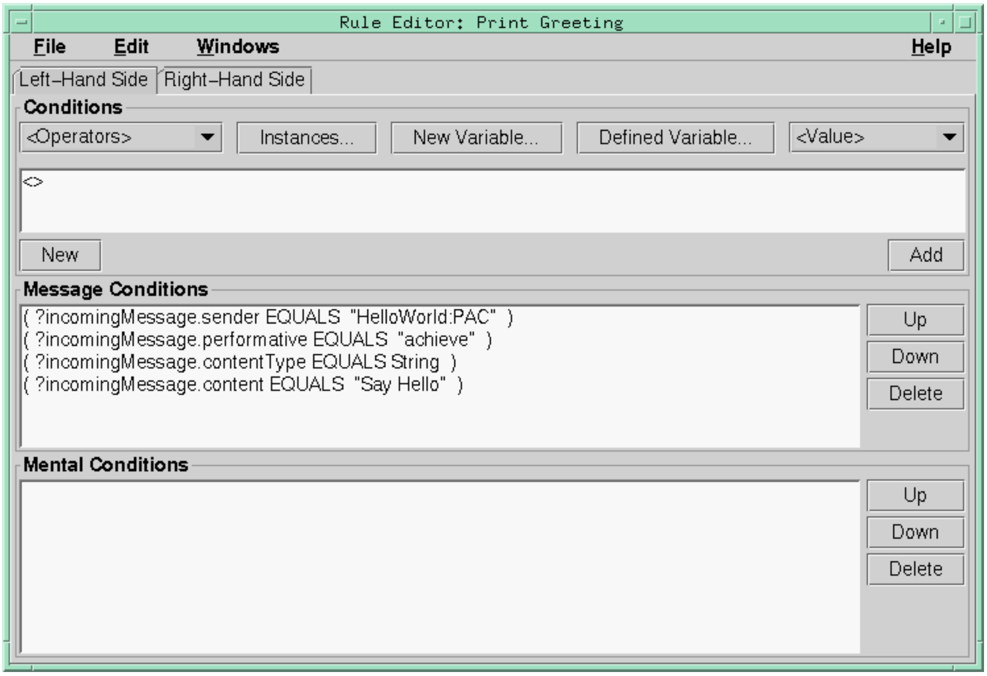
LHS Editor

Figure 11 shows the LHS Editor. There are three main panels: the **Conditions** panel, the message conditions pattern list and the mental conditions pattern list.

The **Conditions** panel is composed of two pull-down menus and three pop-up dialogs, an accumulator, a **New** button, and **Add** button, a **Message Conditions** panel, and a **Mental Conditions** panel. The **Operators** menu is common to all panels in both the LHS and RHS editors, but the items available in each **Operators** menu will differ depending on the panel. In the **Conditions** panel, the **Operators** menu contains various relations and pattern modifiers that are usefule in patterns. The operator elements in this menu fall into several different categories shown below. See "Operators and Patterns" on page 301 for a detailed description of the operations.

- equality relations (`EQUALS, NOT_EQUALS`)
- boolean relations (`AND, OR, NOT`)
- binding relation (`BIND`)

1.A complete description of building complex patterns using the accumulator is described in "Building Complex Expressions" on page 8.



```
( ?incomingMessage.sender EQUALS "HelloWorld:PAC" )
( ?incomingMessage.performative EQUALS "achieve" )
( ?incomingMessage.contentType EQUALS String )
( ?incomingMessage.content EQUALS "Say Hello" )
```

Figure 11. Rule Editor LHS

- quantified relations (`FOR_ALL, EXISTS`)
- numerical relations (`<=,<, !=, =,` ...)
- string functions (`concat, substring`, ...)
- arithmetic functions (`+, -, *, /`)
- mathematical functions (`tan, arctan, cos, arccos, sqrt`, ...)
- miscellaneous functions (SET_TEMPORARY)

The **Values** menu allows you to construct an instance of any `Java` type, a `KQMLMessage` type, a `Time` type, or a `Class` type and insert it into the current pattern. This is useful when you want to specify a comparison against a constant value. For example, if you want to test whether a field in an object is less than 4 you could specify the constant value (i.e., 4, probably as an `integer` or `float`) in the **Values** menu.

For the `Java` types, you will be presented with a dialog for entering a literal value. For the `KQMLMessage`, `TIME` and `Class` type, the user is present with a specific dialog for entering each of the aforementioned types.

There are three different pop-up dialogs. The first is the **New Variable** dialog. This dialog allows you to construct new variables that will be available for use in patterns.Figure 12 shows the **New Variable** dialog. To create a new variable, you would first select the **PACs** or **Java Types** tab panel. Once a panel has been selected, a list of available variables types will be displayed. You can then select a PAC or Java type. Once a variable type has been selected, the focus will be transferred to the **Variable Name** text field. Type in a name for the new variable and select the **Add** button. Selecting the **Add** button will add the new variable to the list at the bottom of the dialog.

The creation of a `KqmlMessag`e variable differs in that a different pop-up dialog is displyaed when you click on the **Add** button. You will be shown a **Binding Dialog** for the `KqmlMessage` variable, as shown in Figure 13. You can specify the Kqml message binding to be an incoming or mental model message by using the pull-down menu. Once selected, you can click on the **OK** button so that the `KqmlMesage` variable can be added to the list of new variables and the dialog can be dismissed.

The Rule Editor supports the creation of an `Array` variable. The `Array` type is listed in the **Java Types** panel. Like the creation of the `KqmlMessage`, a different pop-up dialog is displayed when you click on the **Add** button. You will be shown an **Array Dialog** for the `Array` variable, as shown in Figure 14. You can specify the array type by selecting a type from the list. Notice that you will be able to create

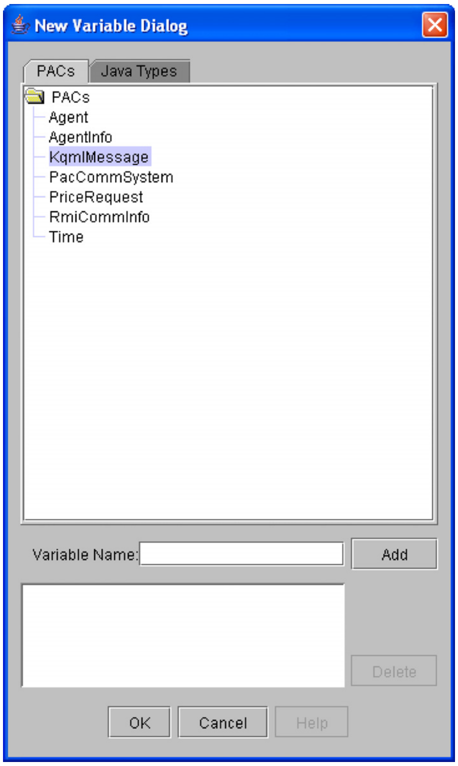


Figure 12. New Variable Dialog

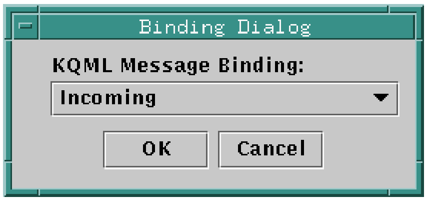


Figure 13. KQML Message Binding Dialog

an array of Java types, Vector, Hashtable, Enumeration, and PAC types. Once selected, you can click on the **OK** button so that the array variable can be added to the list of new variables and the dialog can be dismissed.

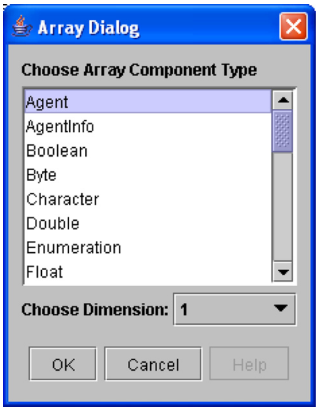
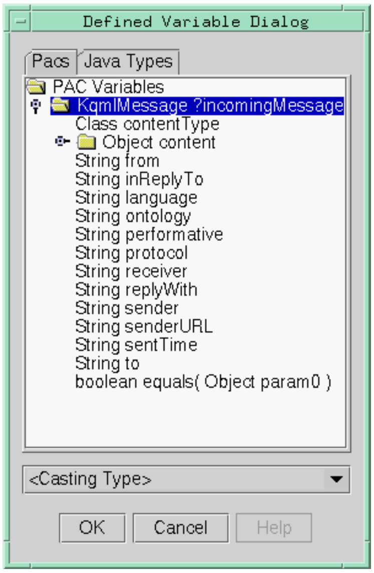


Figure 14. Array Dialog

The **Defined Variables** dialog allows you to select previously defined variables for use in a message or mental pattern. Figure 15 shows the **Defined Variables** dialog. For any array types in the **Java Types** panel, the array variable will only have attributes for the component type and length. This dialog also allows you to cast the selected variable to a different type. To cast a selected variable, select from the **Casting Type** combo box. The combo box contains a list of all avilable PACs and Java types. Once a casting types has been selected, the type of the selected variable will be updated in the dialog. You can no click on the **OK** button to add the selected variable to the **Condition** panel's accumulator text field.

The **Instances** pop-up dialog allows you to specify conditions based on named instances in the mental state. Figure 16 shows the **Instances** dialog.



PAC Variables
KqmlMessage ?incomingMessage
Class contentType
Object content
String from
String inReplyTo
String language
String ontology
String performative
String protocol
String receiver
String replyWith
String sender
String senderURL
String sentTime
String to
boolean equals(Object param0)

Figure 15. Defined Variable Dialog

Like the **Defined Variables** dialog, the **Instances** dialog allows you to cast named instances to an available PAC or Java type. To cast a named instance, use the same method as was described in the **Defined Variables** dialog.

The **Intentions**, **Actions** and **Commitments** pull-down menus are currently disabled. They will be enabled in later versions of Agent-Builder.

The **New** button allows the user to create a new message or mental condition. The same can also be done by selecting **File → New → New Condition**. Pressing the **Add** button will cause AgentBuilder to check to see if the pattern in the accumulator is valid. If the pattern is valid, it will be added to either the **Message Conditions** list or the **Mental Conditions** list.

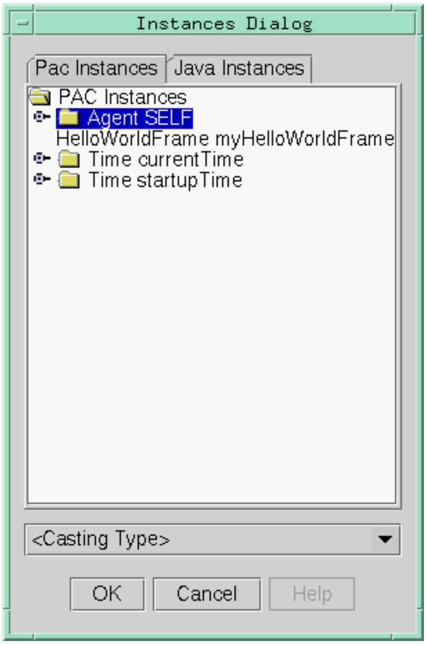


Figure 16. Instances Dialog

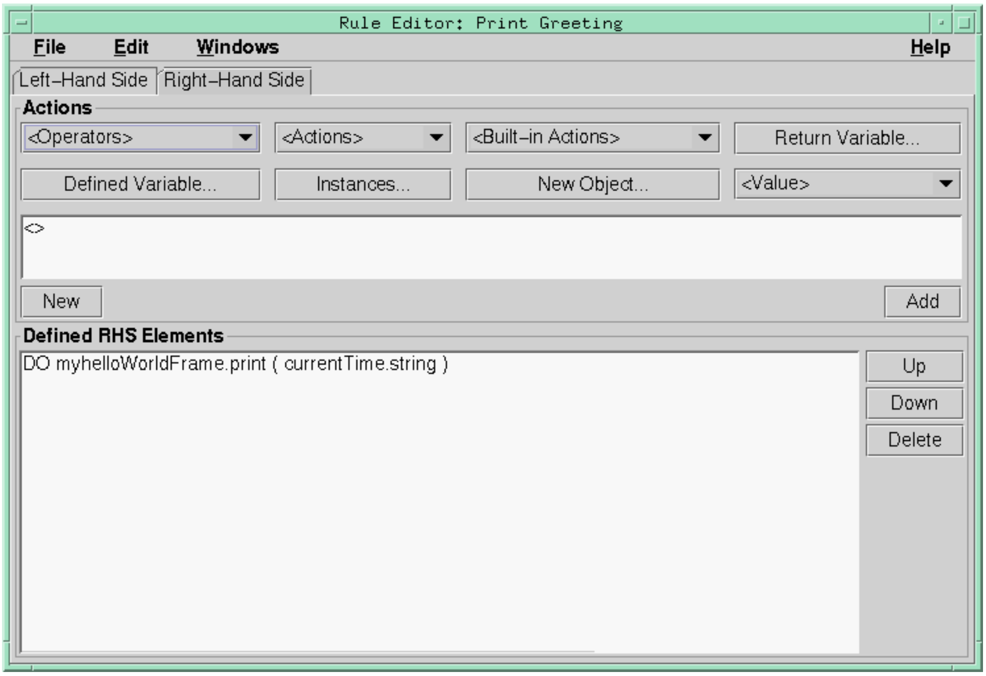
RHS Editor

Figure 17 shows the RHS Editor. There are two main panels: the **Actions** panel and the **Defined RHS Elements** panel. The created patterns from the actions panels supply the **Defined RHS Elements** list with action statements.

Action Panel

The **Actions** panel is composed of four pull-down menu, four pop-up dialogs, an accumulator line, a **New** button and an **Add** button. By using these dialogs and menus, you can create arbitrarily complex arguments and mental change statements.

The **Operators** pull-down menu allows access to various functional operators. This menu supplies a subset of the operators in the LHS



DO myhelloWorldFrame.print (currentTime.string)

Figure 17. Rule Editor (RHS)

operator lists. These operators (along with their operands) can be used as arguments for actions. These operators include:

- string functions
- arithmetic functions
- mathematical functions
- miscellaneous functions (ASSERT, RETRACT, SET_TEMPORARY)

The **Actions** pull-down menu allows access to the user-defined actions. The user-defined actions must first be defined in the Action

Editor before they can be used in the Rule Editor. Please see the Action Editor section for more information about defined actions.

The built-in **Actions** pull-down allows access to the system-defined actions supported by the Run-Time System. For a more detailed description of the built-in actions please see “Intrinsics” on page 287.

There are three key components which are identical to their counterparts in the **Conditions** panel in the LHS editor: the **Values** pull-down menu, the **Defined Variables** dialog, and the **Instances** dialog.

The **New Object** dialog is used when defining new instances of PACs or Java types. Figure 18 shows the **New Object** dialog. To specify a new object, you must first select the type of object that you want to create. Once you have selected a type, you must then select the constructor to use for that type. The new object will then be inserted into the action pattern. Any parameters for the new object will need to be specified in the action pattern. The New Object dialog is used when specifying a parameter to an action that needs to be built at rule execution time.

The **Return Variable** dialog is used to rename any automatically generated return variable. Figure 19 shows the **Return Variable** dialog. Return variables are automatically created for any private action, built-in action, or method that returns something. To change the name of a return variable, you first select the return variable in the Action pattern's accumulator. Then click on the **Return Variable** button to pop-up the dialog. Once the dialog is displayed, you select the type of the return variable that you are replacing. Once the type is selected, you can enter a variable name. Clicking on the **OK** button will then rename the selected return variable.

The **New** button allows you to create an action. You can also do this by selecting **File → New → New Action**. The **Add** button will

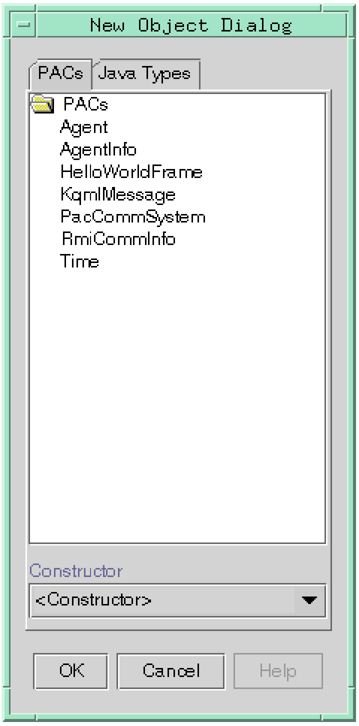


Figure 18. New Object Dialog

check to see if the pattern in the accumulator is valid. If the pattern is valid, it will be added to the **Defined RHS Elements** list.

Defined RHS Elements Panel

The last panel in the RHS editor is the **Defined RHS Elements** panel which contains the list of private actions and mental changes on the RHS of the rule. The **Up** and **Down** buttons on the right of the panel can be used to rearrange the elements in the list, and the **Delete** button can be used to delete a selected element. It is crucial to maintain the integrity of the ordering within this list. Actions, except for the built-in `SendKqmlMessage`, will always be executed before the

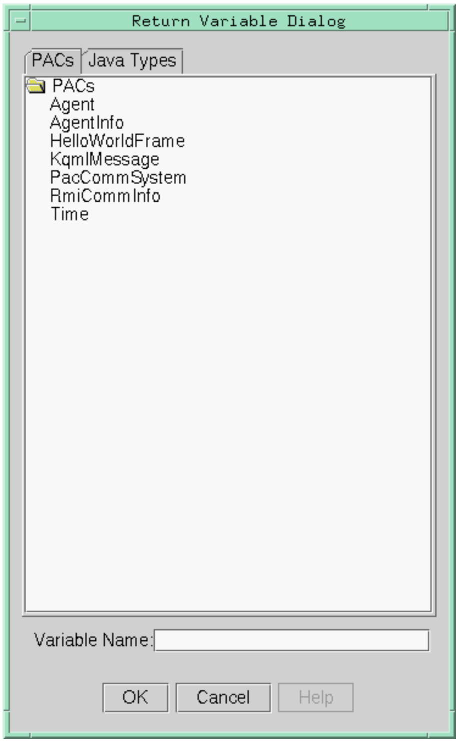


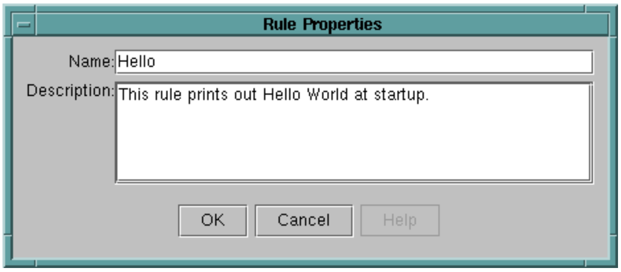
Figure 19. Return Variable Dialog

mental changes are performed. You may change the ordering within the list of actions, but if any action uses the returned value from another action, it must follow that action in the list. For example, if action `Foo` returns an `Integer` which is to be used as an argument for action `Bar`, then `Foo` must be listed ahead of `Bar` in the **Defined RHS Elements** list; this will cause Foo to be executed first and the return value will then be available for use in `Bar`.

Rule Editor Operations

Creating a New Rule

Constructing a new rule is relatively simple. Select the **File → New → New Rule** menu item. This clears all elements from the Rule Editor. If a rule was currently loaded, and it was modified, a dialog will be displayed asking if the current rule is to be saved. Figure 20 shows the dialog that will be displayed in creating a new rule.



Name: Hello

Description: This rule prints out Hello World at startup.

Figure 20. Rule Properties Dialog

The **Rule Properties Dialog** allows you to provide a name and description for the new rule. This information can later be changed by selecting the **Properties** menu item from the **Edit** menu. Once the information for the rule has been entered, select the **OK** button to apply the changes. Selecting **Cancel** button will cancel any changes to the rule's name or description. If this dialog was displayed as a result of selecting the **New Rule** menu item, selecting **Cancel** will defer naming the new rule.

Loading an Existing Rule

To load an existing rule, select **File → Open**. A list of the rules previously defined for the agent will appear. Load the existing rule by either double clicking on its name or by clicking on its name and

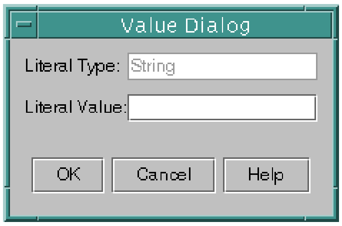


Figure 21. String Value Dialog

clicking on the **OK** button. Either selection method causes the rule to be loaded into the editor.

Constructing a Simple Mental Condition

The first step in constructing a simple mental condition is to choose the operator to be used, usually the `EQUAL` or `NOT_EQUAL` operators. Next, specify the operands that are to be compared, starting with the operand to the left of the operator. For example, a mental condition might test for equality between an attribute in a PAC variable and a string constant. Each operand can be an instance, a defined variable, a constant value, or the result returned from a function (e.g., the concatenation of two strings). To specify an operand, choose the correct dialog and select the desired element. Finally, specify the operand on the right side of the operator. Figure 21 shows the **Values** dialog where the string constant is defined. Enter the desired string and click on **OK**. This causes the string value to be entered into the current pattern. When the pattern is complete, click on **Add** to add the current pattern to the list of mental patterns.

Direct Method Invocation

Directly invoking methods allows you to bypass creating a private action for a method you want to use in an action pattern. To use direct method invocation, you can either pop-up the **Defined Variable** or **Instances** dialog. In both dialogs, each non-primitive type is represented with a folder icon, and will contain zero or more child

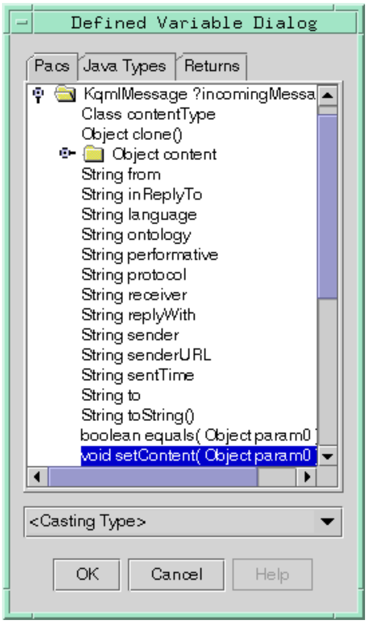


Figure 22. Direct Method Dialog

nodes. The child nodes represent field attributes of the type as well as any methods, with attributes listed first. To specify a method to be invoked, you must select the desired method and click on the **OK** button. The method invocation will then be inserted into the action pattern being built. Now, actions need only be created when you want to use the private action to create an initial commitment. Figure 22 shows this process.

Using a Predicate Method

Predicate methods (i.e., methods that return a Boolean value) defined in the PACs may be used to construct mental conditions. The available predicate methods are displayed in the **Defined Variables** and **Instances** dialogs in the same way that the PAC attributes are displayed. Click on the method name to select it, then click on

OK to insert the predicate method into the current pattern. After selecting the predicate method you'll need to specify values for any parameters required by the method. Figure 23 shows the format for the predicate methods on an example PAC in the **Defined Variables** dialog.

Constructing an Action Statement

Constructing an action statement is straightforward. Select the desired action from the **Actions** or **Built-in Actions** lists. Next, specify all parameters for the selected action. The Rule Editor automatically creates a variable to hold the value returned by each action statement so that the returned value can be used as a parameter in other actions or in mental changes. The tool adds these variables to

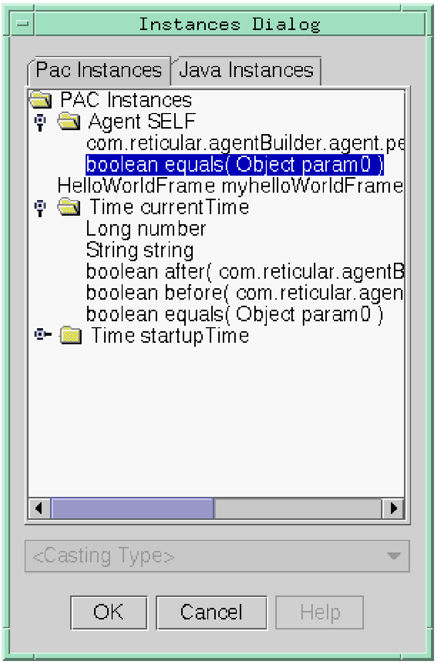

Instances Dialog
Pac Instances
Java Instances
PAC Instances
Agent SELF
com.reticular.agentBuilder.agent.pe
boolean equals(Object param0)
HelloWorldFrame myhelloWorldFrame
Time currentTime
Long number
String string
boolean after(com.reticular.agentB
boolean before(com.reticular.agen
boolean equals(Object param0)
Time startupTime
<Casting Type>
OK
Cancel
Help

Figure 23. Predicate Methods in the Defined Variable Dialog

the Defined Variables list so they can be selected for use in other RHS patterns.

Note: If an action has not been associated with an instance of a PAC, the `ConnectAction` built-in action must be executed first. For example, if an action named `Print` has been defined to use the `print(String)` method on a PAC instance named `myControlPanel-Pac` and the action has not been associated with the instance, `Con-nectAction(Print, myControlPanelPAC)` must be executed before `Print` is executed. The section, "PAC Editor" on page 162, provides more information about actions and PACs.

Building an Assertion with a New Object

As previously mentioned, an assertion adds a new instance of a class to the agent's mental model. To specify an assertion, first select the **ASSERT** item in the **Operators** menu from the **Actions** panel. This causes a dialog to appear with a text field and a prompt. If you want to associate a name with the instance that will be asserted (e.g., you may want to assert a new `Location` instance named `currentLocation`), type the name into the text field and click **OK**. If you're not interested in providing a name for the instance, just leave the text field blank and click **OK** to close the dialog. After the dialog is closed the assertion template, **ASSERT(<>)** will be added to the mental change accumulator.

Next, click on the **New Object** button, which will cause the **New Object** dialog to appear, as shown in Figure 24. This allows you to select the class type for the instance. Click on the name of a class to select it. After you have selected a class type, select a constructor from the pull-down menu at the bottom of the dialog, then click on **OK**. This inserts the new object into the mental change accumulator. At this point you must specify the parameters for the constructor. The parameters can be existing objects found in the **Instances** or **Defined Variables** dialogs, or they can be other new objects cre-

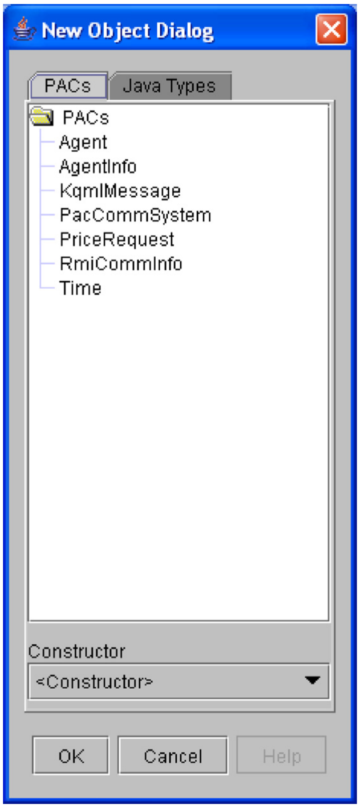


Figure 24. The New Object Dialog

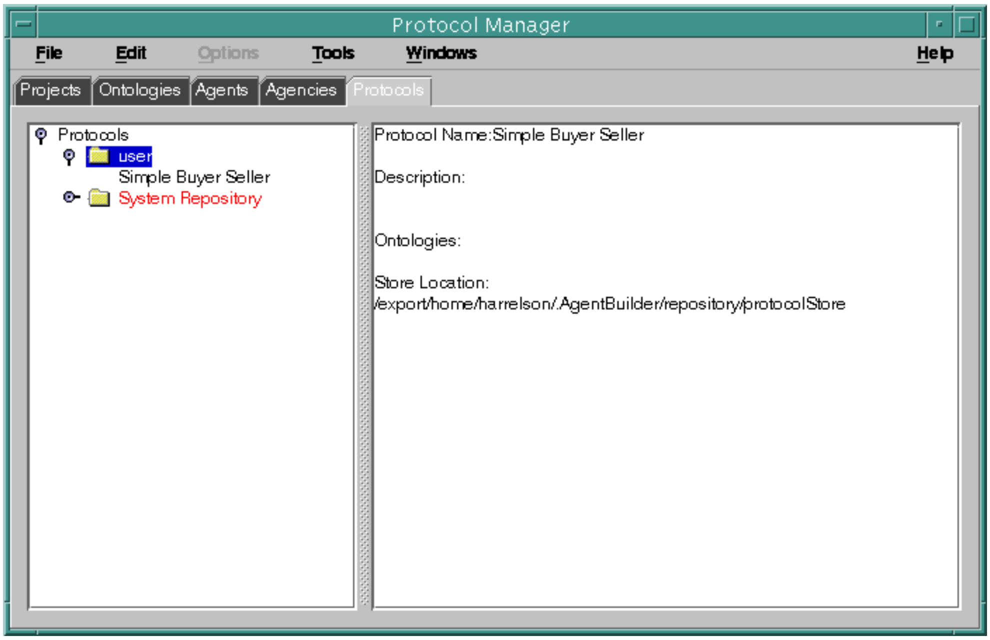
ated in the **New Objects** dialog. It is possible to use new objects as parameters to constructors, actions or other functions.

Closing the Rule Editor

You can close the Rule Editor by selecting the **Close** item in the **File** menu. If necessary, AgentBuilder will ask you whether you want to save your changes or not. By selecting **Yes** you can save all of your changes since the previous save. If you select **No** you'll revert to the rule configuration last saved.

A. Protocol Manager.

The Protocol Manager is shown in Figure 25. This tool allows you to create and view the set of protocols to be used with various agencies. This tool gives you a high-level view of protocols. The Protocol Editor is used to define all of the properties needed to use a protocol with a specified agency.



Protocol Name:Simple Buyer Seller

Description:

Ontologies:

Store Location:
/export/home/harrelson/.AgentBuilder/repository/protocolStore

Figure 25. The Protocol Manager

Overview

The **Protocol Manager** has five menus: **File**, **Edit**, **Tools**, **Windows** and **Help**. The **File** menu allows you to create new protocols, close the **Protocol Manager** and shut down AgentBuilder. The **Edit** menu allows you to **Cut**, **Copy**, **Paste** and **Delete** a protocol. You can use the **Tools** menu to open the **Protocol Editor** for a selected protocol. The **Windows** menu allows you to quickly and easily switch between the various open AgentBuilder tools.

The protocol tree view allows you to view user-defined and system protocols. You can create and view any number of protocols. The system protocols in the protocol tree are displayed with red text labels. The red text labels signifies that the protocols are read-only and cannot be altered. A user's personal protocols are displayed with black text labels. When a protocol is selected in the left panel, the right panel displays general information about the protocol and includes a short textual description of the protocol as well as information about where the protocol is located. Note that the divider between the protocol tree structure and the properties window can be moved horizontally to provide more viewing space for the protocols.

Operation

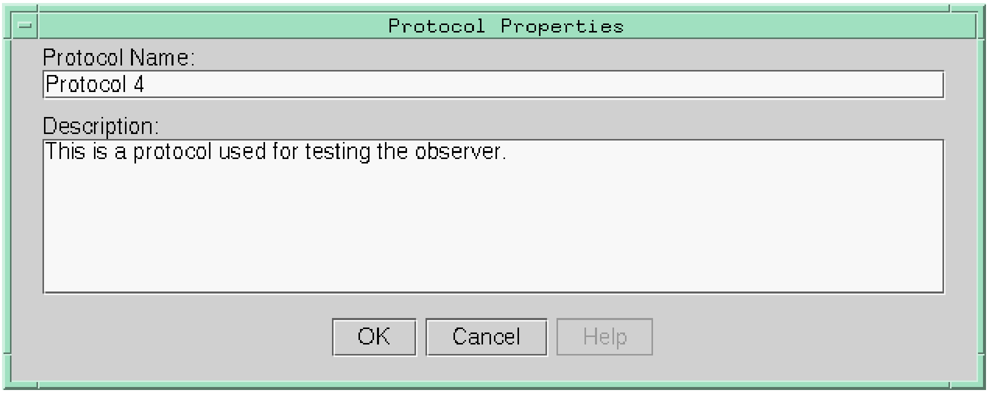
Using the Protocol Tree

The protocol manager uses the same tree structures found in other AgentBuilder tools The protocol manager has three levels in its tree structure. The highest level is the Protocols level, which contains repository folders, which in turn, contains the defined protocols. The repository folders are represented in the tree by a folder icon. The protocols are represented without an icon. Any protocols that read-only are displayed with red text labels.

Creating a New Protocol

A new protocol can be added to your user's repository folder by first selecting the user's repository folder. You would then select the **New Protocol** menu item from either the **File** menu or the repository folder's pop-up menu. This will bring up the dialog shown in

Figure 26. The properties of the protocol can be entered into this dialog.



Protocol 4

This is a protocol used for testing the observer.

Figure 26. Protocol Properties Dialog

Cutting, Copying and Pasting a Protocol

Protocols can be cut, copied, or pasted. There are two ways to use the clipboard functions for a protocol. The first way is to use the **Edit** menu's **Cut**, **Copy** and **Paste** menu items. The other method is to use the protocol's pop-up menu for cut and copy, and the repository folder's pop-up menu for paste. Whichever method is used, a protocol must first be selected before a cut or copy operation. For the paste operation, a user or system repository folder must be selected. Invalid selections will be ignored. If the folder being pasted into already contains the name of the protocol being pasted, the protocol to be pasted will resursively have **CopyOf** prepended to its name.

Modifiying Protocol Properties

The general protocol properties can be modified by right-clicking on the appropriate protocol in the tree structure and selecting the **Properties…** menu item from the pop-up menu.

Deleting Protocols

To delete a protocol from the protocol tree structure, simple select the desired protocol and choose **Delete** from the **Edit** menu or from the pop-up menu. This will display a confirmation dialog before deleting the selected protocol.

Launching Protocol Tools

You can launch the **Protocol Editor** tool from the **Tools** menu. It is necessary to first select the desired protocol before launching the **Protocol Editor**. If you launch the Protocol Editor without first selecting a protocol, AgentBuilder will remind you by displaying a dialog asking you to select a protocol.

Switching AgentBuilder Windows

To switch between different AgentBuilder windows, select the desired window in the **Windows** menu. This will bring the selected window to the foreground.

Closing Protocol Manager

To close the protocol manager, select the **Close** item under the **File** menu. This will close the protocol manager window and the **Protocol Editor**, if it is still open. If you have any unsaved changes in the **Protocol Editor**, you will be given a chance to save them.

Exiting AgentBuilder

To exit from AgentBuilder, select **Exit** from the **File** menu. AgentBuilder will then display a confirmation dialog before actually exiting the system.

Protocol Editor

The **Protocol Editor** tool is used to modify the state diagram and roles that are associated with a particular protocol. The **Protocol Editor** provides a a dialog for creating and modifying roles, a drawing canvas for graphically defining states and transitions and a dialog for viewing the protocol's state table.

Overview

The **Protocol Editor** contains a menu bar with file items: **File**, **Edit**, **Diagram**, **Windows** and **Help**. The **File** menu allows you to open protocols and save the current protocol. You can also save the protocol description to a file. The **File** menu also allows you to close the **Protocol Editor** and shut down AgentBuilder. The **Edit** menu allows you to cut, copy, and paste states. The **Delete** menu item allows you to delete states and transitions. The **Diagram** menu allow syou to modify the current state diagram by adding new states or transitions. The dialogs for viewing the state table and the roles can also be accessed using the **Map** menu items. The **Map** menu also provides menu items for clearing the state diagram and refreshing the display. A sample state diagram is shown in Figure 27.

Operation

Creating a New State

You can create a new state by right-clicking on an unoccupied region of the state diagram and selecting **New State** from the pop-up menu. This will display the state properties dialog that will allow you to enter a name, description, and state type. The state properties dialog is shown in Figure 28. The state type provides a graphical way of indicating which state is the initial state, which states are final states, and which states are normal states. Only one state is allowed to be an initial state. You will receive an error dialog if you attempt to set more than one state as an initial state. The tool doesn't require you specify an initial and final state. The state types provide a graphical enhancement of the state diagram view. Clicking the **OK** button will create a new state on the state diagram. This state is

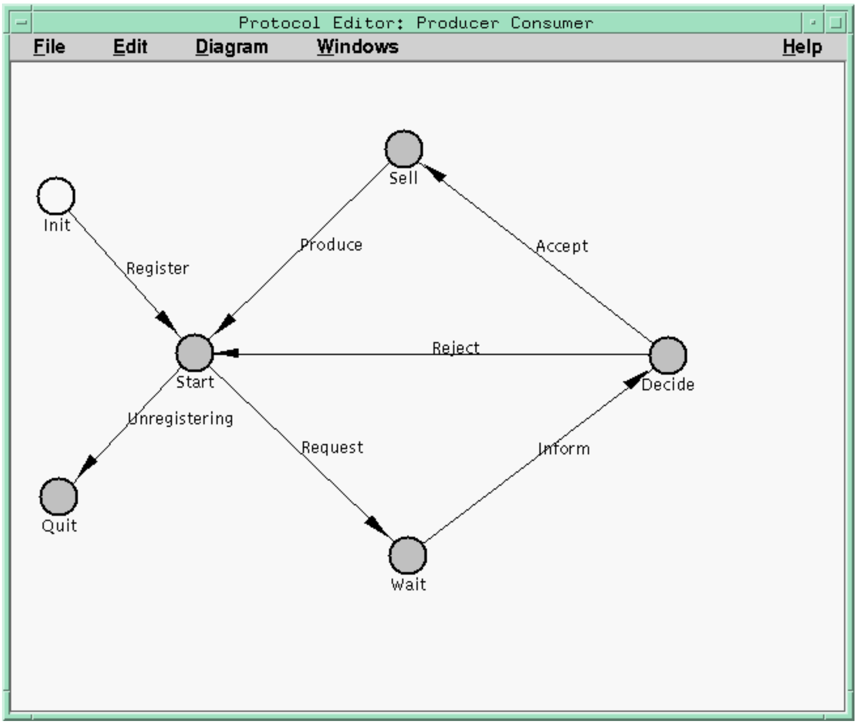
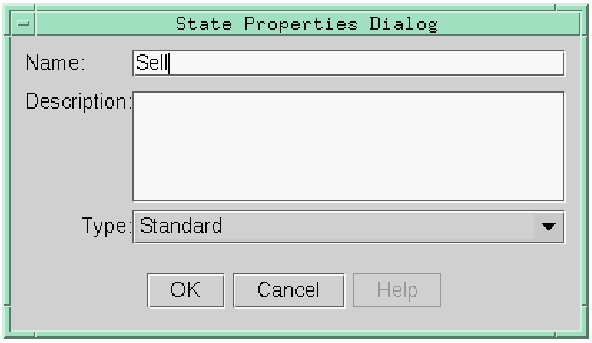


Figure 27. State Diagram

represented by a circular node with the state's name as its label. Note that the point on the state diagram where you right-click is the location where the state is placed. If you decide not to create a new state, you can click on the **Cancel** button and the state diagram will remain unchanged. You can also use the **Map** menu to create a new state in a similar manner.

Cutting, Copying and Pasting a State

States can be cut, copied, or pasted. There are two ways to use the clipboard functions for a concept. The first way is to use the **Edit** menu's **Cut**, **Copy** and **Paste** menu items. The other method is to



Name: Sell

Type: Standard

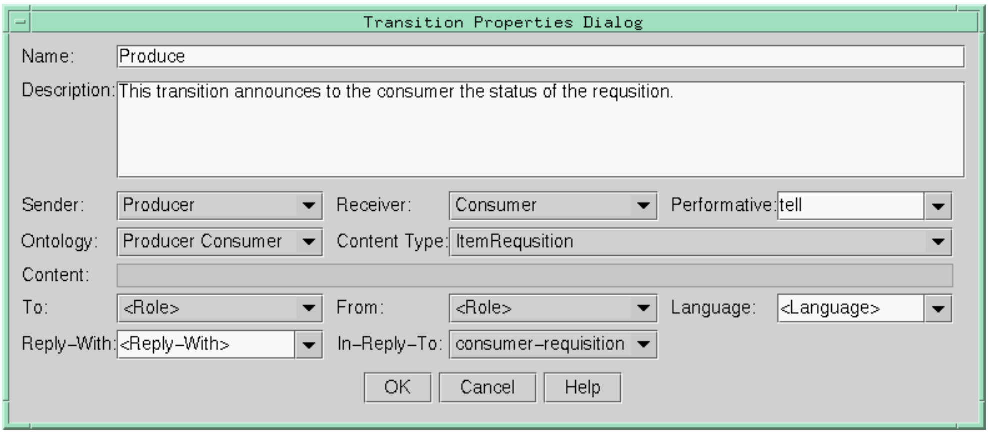
Figure 28. State Properties Dialog

use the state node’s pop-up menus for cut and copy, and the Map's pop-up menu for paste. Whichever method is used, a state must first be selected before a cut or copy operation. If the map being pasted into already contains the name of the state being pasted, the state being pasted will recursively have **CopyOf** prepended to its name.

Creating a New Transition

You can create a new transition by right-clicking on any unoccupied region of the state diagram and selecting the **New Transition** menu item. Once selected, the cursor will change to a cross-hair cursor and you can then click on the starting state and drag the cursor to the ending state. Once a valid starting state and ending state have been selected, the `Transition Properties` dialog will be displayed as shown in Figure 29. The **Properties Dialog** allows you to specify a name for the transition and an optional description.

The rest of the dialog allows you to specify the `KqmlMessage` that is used in the transition. The required Kqml fields are the `sender`, `receiver`, `performative`, `ontology`, and `content type`. Since the `sender` and `receiver` fields are required, you must first have some roles defined before you can create any transitions. If no roles exists, you will see an error when trying to create a new transition


Transition Properties Dialog

Name: Produce

Description: This transition announces to the consumer the status of the requsition.

Sender: Producer
Receiver: Consumer
Performative: tell
Ontology: Producer Consumer
Content Type: ItemRequsition
Content:
To: <Role>
From: <Role>
Language: <Language>
Reply–With: <Reply–With>
In–Reply–To: consumer–requisition

Figure 29. Transition Properties Dialog

(See Creating Roles below). Selecting an ontology will also load the selected ontology's objects into the **Content Type** combo box. For the content field, you can only enter a value if the selected content type is a Java primitive type. Entering a new values into the **Reply-With** combo box will automatically add the new entry to the **In-Reply-To** combo box. All entries will then be saved when the protocol is saved so that for any new transitions created, they will have the list of all **Reply-With** entries added.

Clicking on **OK** will add the transition between the selected states. An arrowhead will be drawn with the arrow pointing towards the ending node. Clicking on **Cancel** will cancel the creation of the new transition.

Moving a State

You can freely move a state node anywhere on the state diagram. This can be done by clicking on the desired state node and dragging the node to a new location. Transitions will adjust themselves automatically to maintain the connection with the relocated state.

Deleting a State

You can delete an existing state node by clicking on that node and selecting **Delete** from the pop-up menu. Likewise, a selected state can be deleted using the **Delete** item in the **Edit** menu. Note that when a state is deleted, all links to that state are also deleted.

Deleting a Transition

You can delete an existing transtion by selecting the transition and selecting **Delete** from the pop-up menu. You can also select **Delete** from the **Edit** menu with a selected transition.

Viewing and Altering State Properties

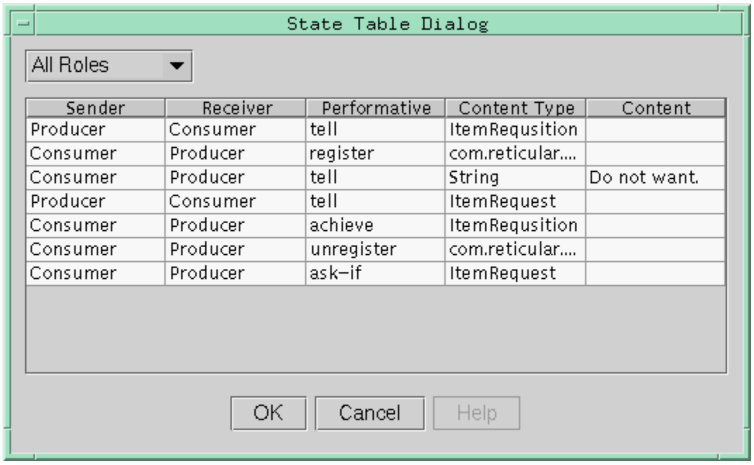
You can view and modify the name, description and type of an existing state by selecting **Properties…** from the state's pop-up menu. The state properties dialog will then be shown, and you can modify any or all of the state's properties. Clicking on **OK** will commit any changes that you have made to the state. Clicking on **Cancel** will revert the state to its original properties.

Viewing and Altering Transition Properties

You can view and modify the name, description and `KqmlMessage` fields of an existing transition by selecting **Properties…** from the state's pop-up menu. The TransitionProperties dialog will then be shown, and you can modify any or all of the transition's properties. Clicking on **OK** will commit any changes that you have made to the state. Clicking on **Cancel** will revert the state to its original properties.

Viewing the State Table

The state table is generated from the transitions in the state diagram. If you currently have no transitions created, you will be given an error dialog and the state table dialog will be shown. Otherwise, the state table dialog will appear as shown in Figure 30. The state table dialog lets you see the important Kqml message fields for each transition: `sender`, `receiver`, `performative`, `content type` and `content`. There is a combo box for selecting which role to view the state table. Currently, only the **All Roles** selection is supported.



Sender	Receiver	Performative	Content Type	Content
Producer	Consumer	tell	ItemRequsition	
Consumer	Producer	register	com.reticular....	
Consumer	Producer	tell	String	Do not want.
Producer	Consumer	tell	ItemRequest	
Consumer	Producer	achieve	ItemRequsition	
Consumer	Producer	unregister	com.reticular....	
Consumer	Producer	ask-if	ItemRequest	

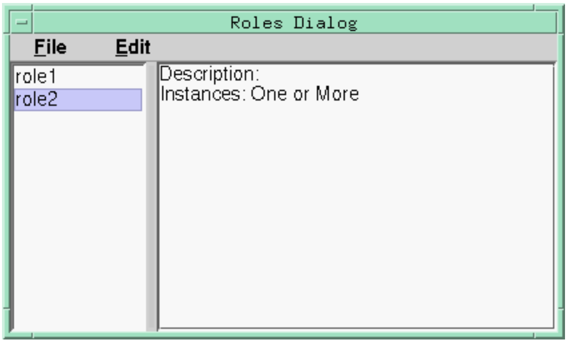
Figure 30. State Table Dialog

Viewing Roles

To view the roles that this protocol is using, select the **Diagram → Roles…** menu item. This will display the **Roles Dialog** as shown in Figure 31. The **Roles Dialog** has two panels: the left panel lists the defined roles, and the right panel gives a description of a selected role in the left panel. This dialog also contains a menubar with a **File** and **Edit** menu. The **File** menu allows you to create a new role, and to close the dialog. The **Edit** menu gives you cut, copy, paste, and delete functions for the role. The **Edit** menu also allows you to bring up the properties dialog for a selected role.

Creating Roles

Roles need to be created so that you can create transitions. To create a new Role, select the **File →New** menu item. The Role Proper-



role1
role2
Description:
Instances: One or More

Figure 31. Roles Dialog

ties dialog will be displayed and will allow you to specify the new role's properties. The Role Properties dialog is shown in Figure rolePropertiesDialog. The dialog will allow you to specify a name, description and instance number for the role. The instance number is specified by selecting the **Instance** combo box. The combo box is an editable combo box so that you can enter an integer number to specify the maximum number of instances allowed for the role.

Cutting, Copying and Pasting Roles

The **Edit** menu items will allow you to cut, copy and paste roles. To cut or copy a role, you must first select the role from the left panel, then select **Copy** or **Paste** from the **Edit** menu. To paste a role that has been cut or copied to the clipboard, select the **Paste** menu item from the **Edit** menu. The role will then be added to the list of roles in the left panel. If a role in the list already has the name of the role being pasted, the pasted role will recursively have **CopyOf** prepended to its name until it is unique.

Deleting Roles

To delete a role from the list, you must first select the role, then select **Edit** → **Delete**. A confirmation dialog will be shown before the role is actually deleted. Once deleted, the role is removed from the list.

Viewing and Modifying Role Properties

To view a role's properties, you must first select the role in the list, then select **Edit → Properties**. The **Role Properties** dialog will then be displayed showing the selected role's properties. Once the dialog is displayed, you can modify any or all of the roles properties. Clicking on **OK** will commit changes made to the role's properteis. Clicking on **Cancel** will cancel any changes that have been made to the role.

Saving a Protocol

You can save a state diagram by selecting **Save** under the **File** menu. You can then close the **Protocol Editor** and return to it at a later time. The save action will save the states and transitions as well as their locations relative to each other.

Saving the Protocol to a File

The state diagram can be saved to a text file by selecting **File → Generate Printable**. Selecting this menu item will bring up a file dialog for saving a state diagram toa file. By default, the directory is set to the current working directory, and the filename is set to `protocol-name.txt`. The text file that is generated will contains a text description of the protocol's name and description as well as a description of every state and transition that make up the protocol.

Clearing the Protocol

If you wish to delete everything on the current state diagram and start again, you can do so by selecting **Clear** under the **Diagram** menu item or from the state diagram's pop-up menu. Note that this function will remove all states and transitions that have been entered. You will be presented with a confirmation dialog before actually clearing the state diagram. If you later decide that you liked the original state diagram and you have not yet saved the changes, you can just close the **Protocol Editor** or reload the **Protocol Editor** using the **File → Open** menu item.

Closing the Protocol Editor

You can close the **Protocol Editor** by selecting the **Close** item in the **File** menu. If you have any unsaved changes, you will be given a chance to save them before the tool is closed.

Role Editor

The Role Editor is used to assign an agency's agents to roles in a particular protocol. Using the **Role Editor,** you can update any or all agents so that necessary rules can be generated for their assigned roles.

Overview

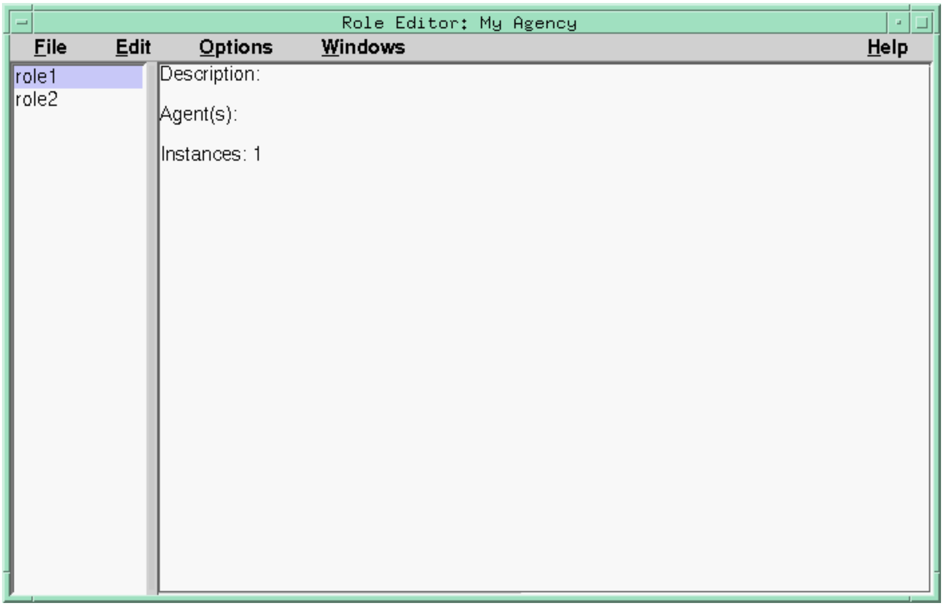
The **Role Editor** contains a menubar with five menu items: **File**, **Edit**, **Options**, **Windows** and **Help**. The **File** menu allows you to save the roles. From the **File** menu, you can also close the **Role Editor** and shut down AgentBuilder. The **Edit** menu allos you to view the properties of a selected role. The **Options** menu allows you to assign agents to a selected role, update a particular agent, or update all agents that belong to the currently loaded agency.

The **Role Editor** is made up of two panels: the left panel is a listing of the roles for a particular protocol, and the right panel displays a description of a selected role. Figure 32 shows the Role Editor.

Operation

Viewing and Modifying Role Properties

To view a role's properties, select the role in the left panel. The role's properties will then be displayed in the right panel. Alterna-



Description:

Agent(s):

Instances: 1

Figure 32. Role Editor

tively, you can also select the role, then select the **Edit → Proper-ties…** menu item. Either way, you will see the **Role Properties** dialog as illustrated in Figure 33. The **Role Properties** dialog will display the role's name, description and maximum number of instances allowed. Of these propeties, only the role description can be modified.

The role's name and instance number cannot be modified in the **Role Editor**. The roles are created in the **Protocol Editor.** If changes need to be made to a particular role, the changes will have be made in the **Protocol Editor**, and the protocol will then have to be re-imported into the **Agency Manager**.

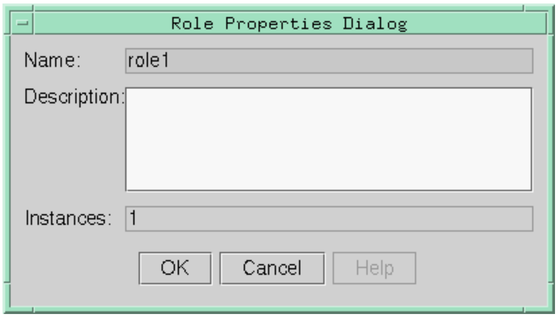


Figure 33. Role Properties Dialog

Assigning Agents to Roles

In order for a protocol to be successfully applied to an agency, each role must be assigned to one or more agents. The only exception is when a particular role is specified to have zero or more agents, in which case, assigning an agent is optional. To assign one or more agents to a role, you must first select the role from the left panel. Then, click on the **Options → Assign Agent(s)**, and the **Assign Agent(s)** dialog will be displayed as shown in Figure 34.

The role being assigned agentsis at the top of the dialog. There are two main panels in the dialog: the **Available Agents** panel lists the agents that belong to the currently loaded agency. Also, buttons are provided in this panel for adding a selected agent from the list, or adding all agents from the list. To add a single agent, first select an agent, then click on the **Add** button. The selected agent will then be added to the list in the **Selected Agents** panel. Clicking on the **Add All** button does not require the user to select the entire list.

The **Selected Agents** panel will first list any agents that have already been assigned to the selected role. Agents can be removed from the role by selecting the agent in the list, then clicking on the **Delete** button. Clicking on the **OK** button will set the changes made

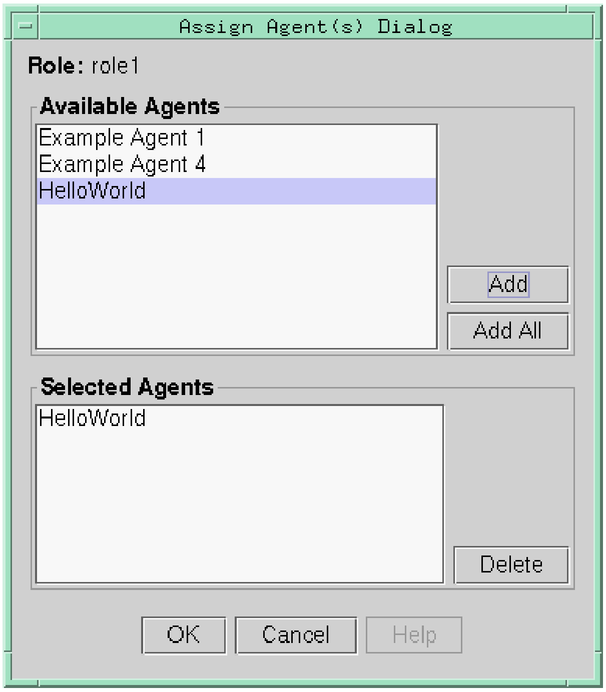


Figure 34. Assign Agent Dialog

to the role. Also, a check will be done to make sure that the number of agents assigned to the role does not exceed the maximum number of instances allowed for the role. Clicking on the **Cancel** button will cancel any changes made to the role's assignment.

Updating an Agent

Updating an agent will generate the necessary rules needed for the agent to fulfill its assigned role(s). An agent will need to be updated whenever the agent has been assigned to a role. To update an agent, select the **Options → Update Agent…** menu item. This will display the **Update Agent** dialog as shown in Figure 35. The dialog consists of a combo box that contains the agents for the currently loaded agency. Select the agent that you want updated, and click on the **OK** button. Clicking on the **Cancel** button will cancel the agent update.

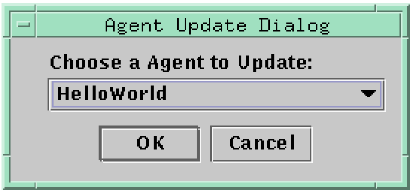


Figure 35. Agent Update Dialog

Updating All Agents

As a convenience, the **Role Editor** also allows you to update all agents for the currently loaded agency. Selecting the **Options → Update All Agents** menu item will update all agents. Selecting the menu item will then generate the rules needed for each agent to fulfilly each of its assigned roles.

Saving the Roles

The roles can be saved by selecting **Save** under the **File** menu. You can then close the **Role Editor** and return to it at a later time. The save action will save the role assignments to each of the agents.

Switching Windows

To switch between different AgentBuilder windows, select the desired window in the **Windows** menu. This will bring the selected window to the foreground.

Closing the Role Editor

To close the role editor, select the **Close** item under the **File** menu. This will close the role editor window. If you have any unsaved changes in the role editor, you will be given a chance to save them.

Exiting AgentBuilder

To exit from AgentBuilder, select **Exit** from the **File** menu. Agent-Builder will then display a confirmation dialog before actually exiting the system.

Chapter 1

Project Accessories Class Library

Chapter Overview

You can find the following information in this chapter:

- Project Accessory Classes
- Input and Output
- External Processing
- Threading
- Arguments and Return Values
- PAC Interfaces
- Control Panel Design

A. Project Accessory Classes

A Project Accessory Class (PAC) is a Java class with methods which can be invoked by an agent to modify or interact with the agent’s environment. PACs are used to transfer information from a user into the agent’s mental model, to display information from the agent’s mental model, and for executing actions.

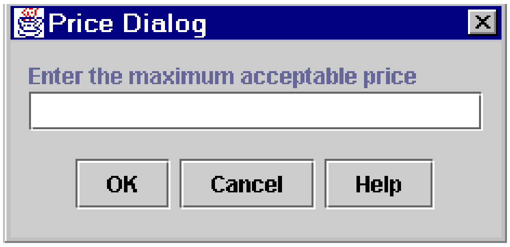
PACs may come from a variety of sources: PACs can be written by the developer, may be obtained from commercial off-the-shelf packages, or may be freeware downloaded from the Internet. Although you can write PACs entirely in Java, there are situations where Java alone does not meet the needs of an application. You can use the Java Native Interface (JNI) to write non-Java methods to handle those situations when an action cannot be implemented in Java; C/C++ functions that conform to the JNI can be used to implement private actions. You are free to create new methods and/or integrate existing code (Java, C or C++) into a PAC.

PACs are assembled into an agent’s Project Accessory Class (PAC) library, which contains all of the domain-specific code the agent requires for operation in its domain.

Input and Output

Sometimes PACs are used to implement a control panel for an agent. We describe a BuyerSeller example where a PAC is used as the control panel for the Buyer agent; it allows the user to choose a product and amount to be purchased, and it displays the buyer’s inventory and account information. Several questions to consider when designing a control panel are discussed in the section entitled “Control Panel Design” on page 224.

AgentBuilder provides a simple built-in PAC for input, the Input-Dialog, which can be easily customized and launched from an

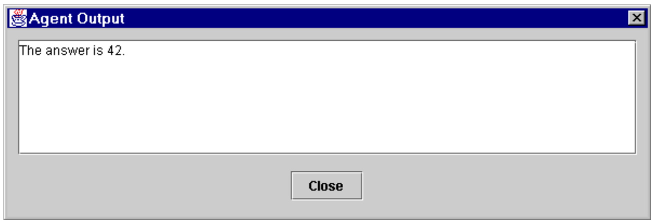


Enter the maximum acceptable price

Figure 36. Input Dialog

agent, and which can be used to get input of any simple data type from the user. This is adequate in many situations, such as getting string or integer values from the user. Complex data types with more than a single field, however, will require the developer to create a PAC which allows input of the fields of the complex data type, and which performs any domain-specific testing of the inputs. Figure 36 shows the built-in Input Dialog, customized for use in a travel agent application.

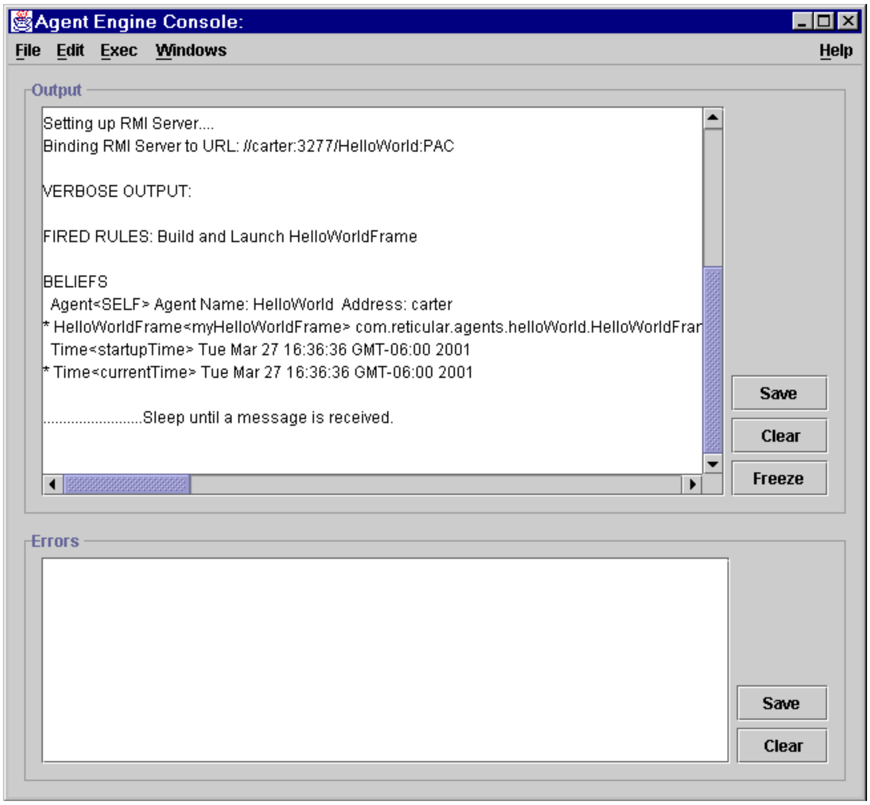
AgentBuilder provides a simple built-in PAC for output, the **Output Dialog**, which will display a message from the agent. shows the built-in Output Dialog.



The answer is 42.

Figure 37. Output Dialog

AgentBuilder also provides a built-in console which displays program output and error output, and allows the user to freeze the display and save output to a file. For many projects, this built-in



```
Setting up RMI Server....
Binding RMI Server to URL: //carter:3277/HelloWorld:PAC

VERBOSE OUTPUT:

FIRED RULES: Build and Launch HelloWorldFrame

BELIEFS
 Agent<SELF> Agent Name: HelloWorld  Address: carter
* HelloWorldFrame<myHelloWorldFrame> com.reticular.agents.helloWorld.HelloWorldFrar
 Time<startupTime> Tue Mar 27 16:36:36 GMT-06:00 2001
* Time<currentTime> Tue Mar 27 16:36:36 GMT-06:00 2001

........................Sleep until a message is received.
```

Figure 38. The Console

console will be adequate for monitoring the agent and displaying results to the user. Figure 38 shows the built-in console.

Other projects, however, will require the developer to create a PAC to display the agent’s output in a domain-specific manner. In the BuyerSeller example PACs are used for the display of the Buyer agent’s and Seller agent’s output. Although it would be possible to display their outputs in the built-in consoles, the use of custom PACs allows the different quantities of interest (e.g., account balance) to be displayed in separate text areas for easier viewing.

External Processing

PACs are also used to modify an agent's environment, perform domain-specific processing, and facilitate information retrieval outside of an agent's mental model. PACs can be used to access a file and return a piece of information, or make a database query and return the result to the agent's mental model. An agent may use a PAC to perform data analysis, e.g., a PAC could analyze a volume of data using a neural network and report the results back to the agent for use in the agent's reasoning. An agent's mental model is based on symbolic reasoning; using PACs, they can also perform various types of sub-symbolic reasoning when that is appropriate to the problem domain.

For example, an agent designed for a loan approval program could have several PACs for input, output, and processing. An input PAC could be used to extract all the data for the next loan applicant from a file containing loan applications, or perhaps from a graphical interface used by the applicant. The input PAC would provide the agent's mental model with all the parameters of interest in the loan application domain (applicant's name, social security number, income, etc.). The agent could perform some preliminary processing such as immediately rejecting the application if the income level is below some threshold, for example. This could trigger the display of another interface (another PAC, or another method on the input PAC) to notify the applicant of the decision or ask for confirmation of the income level.

The agent could have a PAC that interacts with a database of credit histories. The agent would supply the PAC with the applicant's parameters and the PAC would perform a database query at one of the credit history services and return a rating to the agent. Or the agent could supply a neural network PAC with the applicant's parameters and receive a rating of the applicant. Perhaps both methods would be used and the results would be further analyzed by the agent, using reasoning developed from the experience of human loan-approval officers. Finally, the agent would use a PAC to display its recommendation and its justifications to a human loan officer or directly to the applicant.

For another example, an agent could be used to control a piece of machinery in an industrial plant. Such an agent might have PACs to measure physical quantities such as temperature or RPM, and other PACs to perform physical activities such as moving an actuator or halting the machine. This type of agent would probably also have a PAC for a control panel to allow a human user to set parameters in the agent or override the agent's normal processing. Java PACs used to control physical devices might be provided by the manufacturer of the device, or the developer may need to create a Java PAC to encapsulate a C-language device driver provided by the manufacturer.

Threading

Not all private actions are suitable for running on the agent's execution thread, i.e., as part of the sequence of instructions within a computational process. Actions running on the same thread as the agent must execute relatively quickly, because the agent is blocked until the private action finishes execution. To prevent blocking, it may be necessary to run a private action on its own thread. Running actions on their own threads allows long-running private actions to execute without interfering with the basic agent cycle. For example, control panels require execution on a separate thread because of their long-running nature.

PACs are able to communicate information back to the agent in two ways. For non-threaded actions, information is returned to the agent via a return value. For threaded actions, information can only be returned to the agent in a message, such as a KQML message. For example, a control panel can send messages back to an agent as the user enters information.

It is possible, but not generally recommended, for a non-threaded action to return results to an agent via a message. It is also possible for an agent to send a message to a PAC, instead of invoking a method on the PAC. Thus it's possible to use messages for all agent/PAC communication, but this is not generally recommended. It's much more efficient to use method invocations and return values for communication between an agent and a non-threaded PAC, and use method invocations to start a threaded PAC and messages only for returning results from a threaded PAC.

Arguments and Return Values

There are no restrictions on the types used for communication of data between an agent and a PAC, except that messages cannot contain primitive values. Any Java object type or primitive type may be passed from the agent to a PAC method in a method invocation. For a non-threaded action any Java object type or primitive type may be returned as the return value of a method by a PAC to an agent. For a threaded action, which must return a value to the agent via a message, any Java object type may be returned to the agent as the content of a message.

PAC Interfaces

There are two AgentBuilder interfaces that may be implemented by PACs; both are optional and only need to be used in situations where a particular functionality is desired.

The `AgentBuilderDestroyablePac` interface defines the API that will be used by the agent engine when the engine is destroyed via the built-in action `ShutdownEngine` or via a signal from the console. This interface defines a destroy method which takes no arguments.

Before the engine is destroyed it will invoke the destroy method on all PAC objects which declare they implement the `AgentBuilderDestroyablePac` interface.

If a PAC does not implement the `AgentBuilderDestroyablePac` interface and the PAC is descended from java.awt.Window, then the java.awt.Window dispose method will be invoked on the PAC object. This will dispose of PAC control panels when the engine is destroyed. If you want a PAC interface object to remain visible after the engine is destroyed, the PAC must implement the `AgentBuilderDestroyablePac` interface and use an empty destroy method.

The `AgentBuilderRestorablePac` defines the API that will be used by the agent engine when saving/restoring a PAC object. The format of the string used to represent a PAC can be chosen by the developer. The methods are:

```
String toPacString()
void fromPacString(String pacString)
```

To create a saveable/restorable PAC these methods must be implemented and the PAC must have a constructor which takes no arguments. The save/restore functionality is currently not available. More information will be provided when this functionality becomes available.

Control Panel Design

We now look at some of the issues in designing a control panel for an agent. Questions to consider when building a control panel for an agent include:

(1) What information will be displayed to the user?

(2) What inputs will the user be allowed to make?

(3) What information will be stored in the control panel?

(4) What information will be stored in the mental model of the agent?

(5) What information will flow from the agent to the control panel?

(6) What information will flow from the control panel to the agent?

Questions (1) and (2) are standard interface design issues and will not be discussed here. Questions (3) through (6) are discussed in turn below.

What information will be stored in the control panel?

Ideally you should store all interface information (e.g., button color) in the control panel, and you should store all domain information in the mental state of the agent. This is only a guideline, however, and many practical systems will not follow this guideline completely. For example, in some projects it may be convenient to perform some basic processing of inputs, such as range-checking of numerical inputs, in the control panel. This requires that some domain information (e.g., the limits of the acceptable range) be stored in the control panel. In other projects it may be more appropriate to do all processing of inputs in the agent's mental model.

What information will be stored in the mental model of the agent?

All domain information should be stored in the mental model of the agent so the information can be used in reasoning. In some situations it may be necessary to store some of the domain information both in the agent's mental model and in the control panel. This may be necessary in order to reduce the amount of information exchanged between the agent and the control panel or for efficient display of the information. Note that this can lead to problems if the information in the agent's mental model and the information in the control panel get out-of-sync.

What information will flow from the agent to the control panel?

In general, information will be transmitted from the agent to the control panel via method calls on the control panel object. In the HelloWorld example the agent sends a String to the control panel by invoking `HelloWorldFrame::print(String)`, which appends the string to the text area in the frame.

Any Java type or user-defined type can be used as the argument of a method call on the control panel object. In the BuyerSeller example the buyer agent sends the accepted `PriceQuote` object to its control panel for display to the user. The `PriceQuote` class is a PAC defined by the user; it contains information important to the agents in that domain.

The control panel class will need a method which can be called by the agent for every distinct information type or activity. Assume, for the sake of an example, that the HelloWorld agent needs to display integer values in addition to the "HelloWorld!" strings. A second print method could be added to the `HelloWorldFrame` class, perhaps `print(int)` or `print(Integer)`. This new print method would be something like:

```
public void print( Integer i )
  {
    _textArea.append( "Current integer = " + i.toString() );
  }
```

The methods invoked by the agent are not restricted to displaying values; if the agent needs to modify the internal state or behavior of the control panel, a method could be added to the control panel class to modify its state. For example, the agent could hide the control panel and make it reappear at a later time, perhaps in response to some event detected by the agent. To hide the window a `hide()` method could be added to the control panel class, something like:

```
public hide()
  {
     this.setVisible( false );
  }
```

What information will flow from the control panel to the agent?

Information will be transmitted from the control panel to the agent via KQML messages. Information can be transmitted via KQML performatives, e.g., ‘tell’ or ‘achieve’; via other built-in KQML fields such as Reply-With or In-Reply-To; or contained in the Content field of the KQML message. The Content field can contain any Java object; the agent receiving the message can invoke methods on the content object to extract information.

The choice of KQML field(s) to use when transmitting information depends on the complexity of the information and the overall design of the system. A simple system might allow all information from the control panel to be transmitted via the KQML performatives without any need for the Content field. Other systems will require message strings to be sent in the Content field, and more-complex systems might require PACs to be sent as the message content.

Whatever mechanism is used it is imperative that the sender and receiver agree on the message formats, otherwise the messages will not have the desired effects. If the control panel sends a message with the string “quit” as the content object, but the agent expects the string “Quit”, the message will not trigger the desired behavior in the agent. Similarly, if the control panel sends a message with Performative = ‘tell’ but the agent expects a message with Performative = ‘achieve’, the message will have no effect. It is the responsibility of the system designer to ensure that the control panel and the agent agree on message formats.

Information flow from the control panel will usually be triggered by a user event (e.g., a button push), so messages will usually be sent from within callback methods. In the HelloWorld example, all messages are sent from the `HelloWorldFrame::actionPerformed` method, which is registered as the callback method for events in the `HelloWorldFrame`. This example is simple enough that only a single callback method is needed and the two button-push events can be differentiated by action commands (in this example, button labels are used). Complex control panels may require several callback methods or even separate callback classes, but their construction and operation will be similar to the `actionPerformed` method in this example.

B. Building a Control Panel: The HelloWorld Example

This section shows the step-by-step development of a simple control panel for the HelloWorld agent. Steps (1) through (4) describe the code that is needed in the PAC, HelloWorldFrame.java. Steps (5) through (8) describe the rules in the RADL file HelloWorld.radl that interact with the PAC (in this simple example, all rules interact with the PAC). Development of the system will not always proceed from (1) through (8) in that order; it may be better to build the rules first and later develop the PACs, or build the rule and PACs together, incrementally.

(1) Define a constructor.

This will require at least one argument, a `PacCommSystem`, which will be supplied by the agent. Some control panels may require other arguments as well. For example, a control panel constructor may also use a String argument to get the agent's name for the frame title. The constructor in HelloWorldFrame.java is the simplest possible constructor:

```
public HelloWorldFrame( PacCommSystem pacCommSystem )
  {
     _communicationSystem = pacCommSystem;
  }
```

This constructor stores the `PacCommSystem` object into the data member `_communicationSystem`, so it will be available for use later. It would be possible to build the control panel in the constructor, but that's not done here. Building the control panel can be time consuming, so we wait and do it on the separate thread used for displaying the interface.

(2) Write the run() method, define the control panel.

Any class intended for use as a control panel should implement the `Runnable` interface so that it can be run on a separate thread, hence the need for a `run()` method. Writing the code for the control panel is a standard Java programming task, involving all the usual thrills and frustrations of GUI development. The `HelloWorldFrame` features a very simple interface with a text area to display output from the agent, and two buttons to send messages to the agent. As mentioned in step (1), it's usually better to build the control panel in the `run()` method than in the constructor because building the control panel can be slow and may cause the engine cycle to be delayed if it's done on the main execution thread. All code in the `run()` method can execute on a separate thread so the engine cycle is not delayed.

(3) Write an event handler method(s) to handle the events generated by the control panel.

This may require an `actionPerformed` method, and/or `itemStateChanged` method, and/or some internal classes or separate callback classes. The event handler code will depend on the type of components used in the interface, but will usually be similar to the code in `HelloWorldFrame::actionPerformed`. Events generated by the control panel will cause messages to be sent to the agent. The information required by the agent may be contained in the message attributes (e.g., performative) or in the content object contained in the message.

The control panel's event handler code in the Java file must be in agreement with the message handling rules in the RADL file. As mentioned in the previous section, the same message protocol must be used by the sender (the control panel) and receiver (the agent), otherwise the message will not be interpreted correctly.

In the current example, when the control panel was built in the `run()` method, the action command “Print” was associated with a button-push of the `printButton` object, in the statement:

```
printButton.setActionCommand( PRINT_COMMAND );
```

Then the `HelloWorldFrame` object itself was registered as the callback object for the button-push, in the statement:

```
printButton.addActionListener( this );
```

After the user pushes the print button the sequence of events leading to a print request from the control panel to the agent are as follows:

- The Java event dispatcher invokes `actionPerformed(ActionEvent)` on the registered callback object, which is the `HelloWorldFrame` object itself.
- The callback method `HelloWorldFrame::actionPerformed(ActionEvent)` instantiates a new `KqmlMessage` object, which gets initialized with all fields empty.

```
          KqmlMessage message = new KqmlMessage();
```

The callback method tests the action command and determines that the current invocation is a print request, so the callback method sets the performative to “achieve” and the content to the string “Say Hello”.

```
          message.setPerformative(“achieve”);
          message.setContent(“Say Hello”);
```

- The callback method invokes `sendKqmlMessageToAgent(KqmlMessage)` on its `PacCommSystem` object, `_communicationSystem`. This method will set the sender and receiver fields in the message and send the message to the agent.

```
_communicationSystem.sendKqmlMessageToAgent(message);
```

The agent in this system will have a rule which recognizes an inbound message with Performative = “achieve” and the string “Say Hello” for the message content. The message will cause the agent to assemble a string with “HelloWorld!” and its belief about the current time, which it will then transmit to the control panel via the Print private action.

(4) Write the method(s) to display output from the agent.

The type of output to display will be determined by the type of data being output from the agent’s mental model. In this example the agent will use the PAC to print a string in a text area, so the method will be named `print` and will take a String argument. The choice of method name and argument list is up to the designer of the system, but must agree with the data types in the agent’s mental model. Here the print method appends the string sent in by the agent to the text area in the control panel, which displays the string to the user.

```
public void print( String stringFromAgent )
   {
      _textArea.append( stringFromAgent + "\n" );
   }
```

In a more complex control panel there may be several `print` methods for different types of data output from the agent.

(5) Write a rule to initialize the control panel object.

Initialization of a control panel can be performed at agent start-up or during execution as the result of the firing of a rule. The main difficulty in constructing a control panel is the initialization of a `PacCommSystem` object with the agent’s Internet address and the control panel name. In the HelloWorld example the `PacCommSystem` object is constructed as an argument to the `HelloWorldFrame` constructor. This occurs in the first rule in HelloWorld.radl in Figure 39.

```
WHEN:
IF:
1. ( BIND startupTime )
THEN:
1. SET_TEMPORARY $helloWorldFrameVar TO
                HelloWorldFrame( PacCommSystem(
                SELF.agentInfo, "HelloWorldFrame:PAC" ))
2. DO $helloWorldFrameVar.run()
3. ASSERT( "myHelloWorldFrame" $helloWorldFrameVar )
4. DO SleepUntilMessage()
```

Figure 39. Build HelloWorldFrame Rule

The Build and Launch HelloWorldFrame rule gets activated by an Agent belief with instance name startupTime, which is automatically created by the agent engine when the RADL file contains a startup-Time belief in the initial agency beliefs section. When this rule fires a `PacCommSystem` object gets constructed from the agent's `AgentInfo` object (which contains the Internet address and related information) and a name assigned to the PAC by the agent (here, "`Hel-loWorld:PAC`"). The new `PacCommSystem` object then becomes an argument to the `HelloWorldFrame` constructor, which creates a PAC instance. The new `HelloWorldFrame` instance is given the instance name `myHelloWorldFrame` and is asserted into the agent's mental model.

The name stored in the `PacCommSystem` by the agent is used for communication between the PAC and the agent. Messages sent from the PAC to the agent will have the sender name set to the name in the `PacCommSystem`, and that is the value the agent will look for when expecting a message from the PAC. The name stored into the `PacCommSystem` **does not** need to be the same as the instance name of the PAC, although the same name can be used for both. The PAC instance name is used when testing for the existence of the

PAC instance, e.g., (`BIND myHelloWorldFrame`). The name stored into the `PacCommSystem` is used when testing an incoming message to see whether it was sent by the PAC, e.g., (`?message.sender EQUALS "HelloWorld:PAC"`).

(6) Write rules to handle the messages sent by the control panel.

Using the protocol developed when designing the event handler for the PAC, write rules that accept the messages sent by the PAC. The name of the PAC was stored into the `PacCommSystem` in the Build HelloWorldFrame rule, and that same PAC name will be used as the `Sender` for all messages sent by the PAC. The Print Greeting rule tests the `Sender` field of the message to ensure that a received message came from the PAC. As mentioned in step (3) above, a print request message will have performative = "achieve" and the message content will be the string "Say Hello", which is exactly what is tested here. See the code segment in Figure 40.

```
WHEN
1. ( %message.sender EQUALS "HelloWorld:PAC" )
2. ( %message.performative EQUALS "achieve" )
3. ( %message.contentType EQUALS String )
4. ( %message.content EQUALS "Say Hello" )
IF:
THEN:
1. DO myHelloWorldFrame.print ( Concat( "Hello World! The time is ",
                    currentTime.string ))
2. DO SleepUntilMessage()
```

Figure 40. Print Greeting Rule

This rule then gets the current time from a built-in currentTime belief that is always present in the agent's mental model, and concatenates the time string onto its greeting and invokes the Print action. The

Print action will invoke the `print(String)` method on the `myHel-loWorldFrame` object, which will display the string from the agent in the text area on the control panel.

C. Building a Control Panel: A BuyerSeller PAC Example

This section shows the step-by-step development of a control panel for the buyer agent in the Buyer/Seller example. This example is considerably more complicated than the HelloWorld agent example; you may wish to read through Chapter 9 in the User's Guide before reading this section.

The buyer agent uses a control panel to allow the user to select the product and quantity that the buyer agent will attempt to buy from store agents, and to display the buyer's inventory and account balance. The seller agents use display panels to display their inventories and their account balances. Both the buyer's and seller's interfaces also display information printed by the agents to indicate which messages have been received and what actions have been taken. Figure 41 and Figure 42 show these control panels.

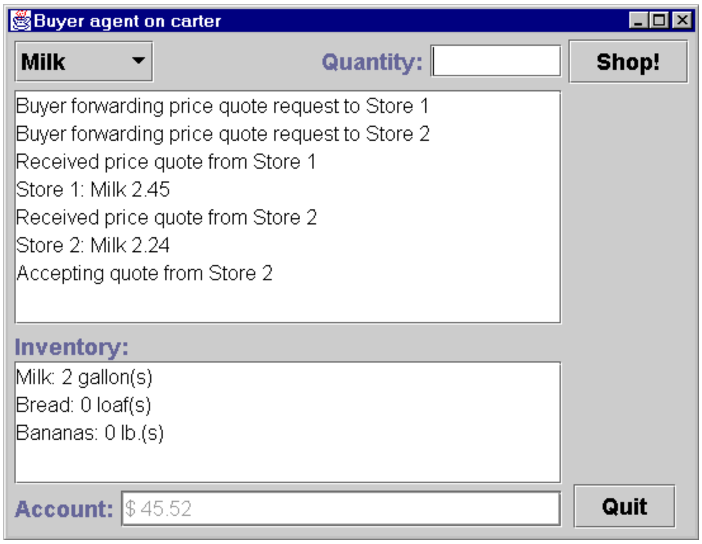
In addition to the control panel PACs, the buyer and seller agents also use PACs to represent entities in the problem domain and to store and transfer information. In this example a PriceQuote PAC is used to record product name, quantity, store name, and the quoted price. A `PriceQuote` object is sent from the BuyerFrame control panel to the buyer agent in a KQML message and then forwarded to each store agent known to the buyer agent. The store agents fill in the `PriceQuote` objects with their store names and quoted amounts and return the completed `PriceQuote` objects to the buyer agent. The buyer agent uses the information in the `PriceQuote` objects to determine the best deal and invokes methods on the `BuyerFrame` control panel with the `PriceQuote` objects as arguments to display the quotes to the user. The construction of data-storage PACs such as `PriceQuote` will not be discussed in this section. Here we'll assume they've already been built in the Object

Modeling Tool and are available for use when designing the control panel and building the rule base.

Steps (1) through (6) describe the code that is needed in the PAC, `BuyerFrame.java`. Steps (7) through (12) describe the rules in the RADL file buyer.radl that interact with the `BuyerFrame` PAC. As was mentioned in the previous example, development of the system will not always proceed from (1) through (12) in that order. The ideal order of construction is: data-storage PACs, then the interface PACs, then the rules. It will usually be necessary to iterate through the process several times, adding or modifying PACs, adjusting the rules, etc.

(1) Decide what data must be stored in the control panel object.

This will depend on the type of information that will be displayed to the user and what kinds of input will be allowed. For the `Buyer-Frame` PAC, the user will be allowed to specify a product from a list

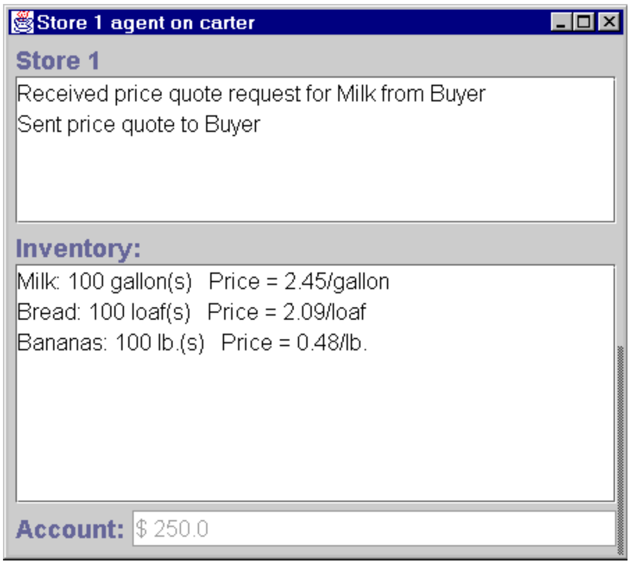


```
Buyer forwarding price quote request to Store 1
Buyer forwarding price quote request to Store 2
Received price quote from Store 1
Store 1: Milk 2.45
Received price quote from Store 2
Store 2: Milk 2.24
Accepting quote from Store 2
```

```
Milk: 2 gallon(s)
Bread: 0 loaf(s)
Bananas: 0 lb.(s)
```

$ 45.52

Figure 41. BuyerFrame PAC Buyer



Store 1 agent on carter

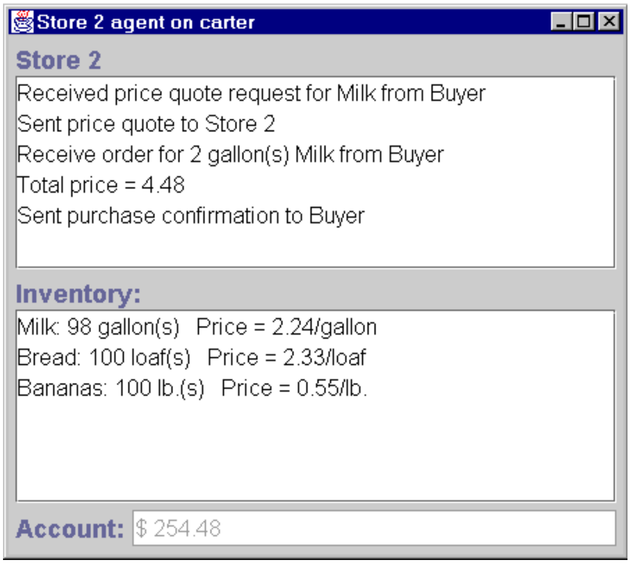
Store 1

```
Received price quote request for Milk from Buyer
Sent price quote to Buyer
```

Inventory:

```
Milk: 100 gallon(s)   Price = 2.45/gallon
Bread: 100 loaf(s)   Price = 2.09/loaf
Bananas: 100 lb.(s)   Price = 0.48/lb.
```

Account: $ 250.0



Store 2 agent on carter

Store 2

```
Received price quote request for Milk from Buyer
Sent price quote to Store 2
Receive order for 2 gallon(s) Milk from Buyer
Total price = 4.48
Sent purchase confirmation to Buyer
```

Inventory:

```
Milk: 98 gallon(s)   Price = 2.24/gallon
Bread: 100 loaf(s)   Price = 2.33/loaf
Bananas: 100 lb.(s)   Price = 0.55/lb.
```

Account: $ 254.48

Figure 42. SellerFrame PACs

```
private Choice _productChoice;
private TextField _quantityTextField;

private TextArea _messageTextArea;
private TextArea _inventoryTextArea;
private TextField _accountTextField;

private PacCommSystem _pacCommSystem;
private String _buyerName;

private Hashtable _productTable;
private Hashtable _inventoryTable;
```

Figure 43. Java Code Segment

of known products, so a `java.awt.`Choice will be used. To specify the quantity to shop for, the user will be allowed to type a number into a `java.awt.`TextField, or accept a default quantity. The `_messageTextArea`, `_inventoryTextArea`, and `_accountTextField` are java.awt components that will display information to the user. The `_pacCommSystem` is a built-in AgentBuilder component that enables communication between a PAC object and its parent agent; this must be supplied by the parent agent. The `_buyerName` string must also be supplied by the parent agent. Finally, two hashtables provide easy lookups of data-storage PAC objects that are used by the `BuyerFrame`. Figure 43 shows the data members of the `Buyer-Frame` PAC.

(2) Define a constructor for the control panel PAC.

This will require at least one argument, a `PacCommSystem` which will be supplied by the agent. The constructor in `BuyerFrame.java` stores data into the object and initializes its hashtables. Figure 44 shows the constructor for the `BuyerFrame` PAC.

It would be possible to build the interface in the constructor, but that's not done here. Building the interface can be time consuming, so it’s best to wait and do it in the `run()` method. This allows us to build the interface on the separate thread used for displaying the interface.

Note, however, that we need to initialize several of the components here in the constructor because it's possible that they'll be used before the initialization of the interface is complete. Although long-running actions such as building an interface should usually be done on a separate thread, during initial development it may be easier to build the interface in the constructor. Then, when everything is working well, move the bulk of the interface initialization code to the `run()` method. Finally, determine which components could

```
public BuyerFrame( PacCommSystem pacCommSystem, String buyerName )
{
    // Store the pac comm system and buyer name into this object.
    _pacCommSystem = pacCommSystem;
    _buyerName = buyerName;

    // Initialize the product and inventory tables to empty hashtables.
    _productTable = new Hashtable();
    _inventoryTable = new Hashtable();

    // Initialize the components which may be needed before the interface
    /// is fully built.
    _productChoice = new Choice();
    _productChoice.add( PRODUCT_CHOICE_LABEL );

    _inventoryTextArea = new TextArea( EMPTY_STRING,
                                       INV_TEXT_AREA_ROWS,
                                       INV_TEXT_AREA_COLS,
                                       TextArea.SCROLLBARS_VERTICAL_ONLY );

    _accountTextField = new TextField( EMPTY_STRING );
}
```

Figure 44. Java Code Segment

possibly be accessed before the interface is complete, and either adjust the rules or move the initialization of some components back into the constructor.

(3) Write the run() method, build the interface.

Any class intended for use as a control panel should implement the runnable interface so that it can be run on a separate thread, hence the need for a `run()` method. Writing the code for the control panel is a standard Java programming task involving all the usual thrills and frustrations of GUI development. The `BuyerFrame` features several `java.awt` components for input and output. As mentioned in step (2), it's usually better to build the control panel in the `run()` method than in the constructor because building the control panel can be slow and may cause the engine cycle to be delayed if it's done on the main execution thread. All code in the `run()` method can execute on a separate thread, so the engine cycle is not delayed.

(4) Write any methods that are needed to initialize the interface PAC with data from the agent.

In some situations all data can be transferred via the PAC constructor arguments, but in other situations extra methods may be needed. `BuyerFrame` has an `addProduct(Product)` method which the buyer agent invokes once for every Product object (a data-storage PAC defined for this project) in the initial objects section of the RADL file. These Product objects are used to populate the choice box in the interface; the product choices offered to the user are exactly the same as those known to the buyer agent.

(5) Write an event handler method(s) to handle the events generated by the control panel.

This may require an `actionPerformed` method, and/or `itemState-Changed` method, and/or some internal classes or separate callback classes. The event handler code will depend on the type of components used in the interface, but will always be similar to the code in

`BuyerFrame::actionPerformed`. Events generated by the control panel may cause changes in the interface and/or messages to be sent to the agent. The information required by the agent may be contained in the message attributes (e.g., performative) or in the content object contained in the message.

The `actionPerformed` method for the `BuyerFrame` PAC will handle the events generated when the user clicks on the **Shop!** or **Quit** buttons. Events generated from other activities, such as selecting a product or typing a number for the quantity, will not be handled, but those other activities will change the state of the `_productChoice` or `_quantityTextField`. Then when the **Shop!** button event is handled the latest values for product and quantity will be extracted from the interface components.

The control panel's event handler code in the Java file must be in agreement with the message handling rules in the RADL file. As mentioned in the previous section, the same message protocol must be used by the sender (the control panel) and receiver (the agent), otherwise the message will not be interpreted correctly.

In the current example, when the control panel was built in the `run()` method, the **Shop!** button label was associated with a button-push of the `shopButton` object in the statement:

```
shopButton.setActionCommand(SHOP_BUTTON_LABEL);
```

Any string could be used for this command string, as long as the comparison done in `actionPerformed` tests for the same string. In this example the button labels are each used only once so the labels can also be used as command strings to distinguish between buttons. In an interface containing several buttons with the same label, unique command strings will need to be defined for each button.

Next, the `BuyerFrame` object itself was registered as the callback object for the button-push in the statement:

```
shopButton.addActionListener(this);
```

After the user pushes the **Shop!** button the following will occur:

- The Java event dispatcher invokes `actionPerformed(Action-Event)` on the registered callback object, which is the `Buyer-Frame` object itself.
- The callback method `BuyerFrame::actionPerformed(Action-Event)` instantiates a new `KqmlMessage` object, which gets initialized with all fields empty.

  ```
  KqmlMessage message = new KqmlMessage();
  ```

- The `actionPerformed` method extracts the name of the selected product from the `_productChoice` data member, and looks up Product object associated with the selected product name in the `_productTable` hashtable. The Product object is used to get the appropriate unit name, such as *gallon* of milk or *loaf* of bread. Then the numeral in the `_quantityTextField` is converted to a number. A new `PriceQuote` object is built using the values extracted from the interface components. This `PriceQuote` object (a data-storage PAC defined for this project) will be the content of the message sent to the buyer agent. At this point it's not really a price quote, it's an object which will be interpreted as a request to shop for the named product.

  ```
  PriceQuote priceQuote = new PriceQuote(productName,
          quantity, unitName);
  ```

- The `actionPerformed` method sets the KQML message performative to “forward” and the content to the new `PriceQuote` object. The performative “forward” is used so the buyer agent will forward the message on to the store agents.

  ```
  message.setPerformative(“forward”);
  message.setContent(priceQuote);
  ```

- The `actionPerformed` method invokes `sendKqmlMessageToAg-ent(KqmlMessage)` on its `PacCommSystem` object, `_pacCommSystem`. This method will set the sender and receiver fields in the message and send the message to the agent.

```
    _pacCommSystem.sendKqmlMessageToAgent(message);
```

The agent in this system will have a rule which recognizes an inbound message with Performative = “forward” and a `PriceQuote` object for the message content. The message will cause the agent to forward the message on to all known store agents and assert a belief about its current goal (purchase the specified quantity of the specified product).

(6) Write the method(s) to display output from the agent.

The type of output to display will be determined by the type of data being output from the agent's mental model. In this example the agent will use the interface PAC to print status strings in a text area, print the buyer's current inventory in another text area, and print the buyer's current account balance in a text field.

The printing of status strings will be done by the methods `displayMessage(String)` and `displayPriceQuote(PriceQuote)`. Both of these methods will write output to the same text area, but different methods are used so that the extraction of values from the price quote doesn't need to be done in the buyer agent's rules. It's much easier to use the `displayPriceQuote` method to extract the store name, product name, and quoted price, then format and print them. This simplifies the work of the buyer agent and leaves the formatting details to the Java code in the `BuyerFrame` PAC.

The `displayInventory(InventoryRecord)` method updates a list of `InventoryRecord` objects (another data-storage PAC defined for this project) then prints the list to the inventory text area. The `displayAccount(float)` method updates the display of the account balance in the text field at the bottom of the interface.

Steps (7) through (12) describe some of the rules that interact with the `BuyerFrame` PAC.

(7) Write a rule to initialize the control panel object.

Initialization of a control panel can be performed at agent start-up, or during execution as the result of the firing of a rule. The main difficulty in constructing a control panel is the initialization of a `PacCommSystem` object with the agent's Internet address and the control panel name. In this example the `PacCommSystem` object is constructed as an argument to the `BuyerFrame` constructor. This occurs in the Build BuyerFrame rule in buyer.radl shown in Figure 45.

```
WHEN:
IF:
1. ( BIND SELF )
THEN:
1. SET_TEMPORARY $pacName TO
                    Concat( SELF.agentInfo.name, ":ControlPanel" )
2. ASSERT( "pacName" $pacName )
3. ASSERT( "buyerFrame" BuyerFrame( PacCommSystem
                    ( SELF.agentInfo, $pacName ), SELF.agentInfo.name )
DO RemoveRule (“Build BuyerFrame”)
```

Figure 45. Build BuyerFrame Rule

This rule will be activated by an Agent belief with instance name “SELF”, which is automatically created by the agent engine when the RADL file contains a SELF belief in the initial agency beliefs section. A `PacCommSystem` object gets constructed from the agent's `AgentInfo` object (which contains Internet address and related information) and the PAC's name, which is in the temporary variable ?pacName. `BuyerFrame` requires two arguments in its constructor: a `PacCommSystem` object and a `String` with the buyer agent's name.

(8) Write the rule(s) to transfer initial data from the agent to the interface.

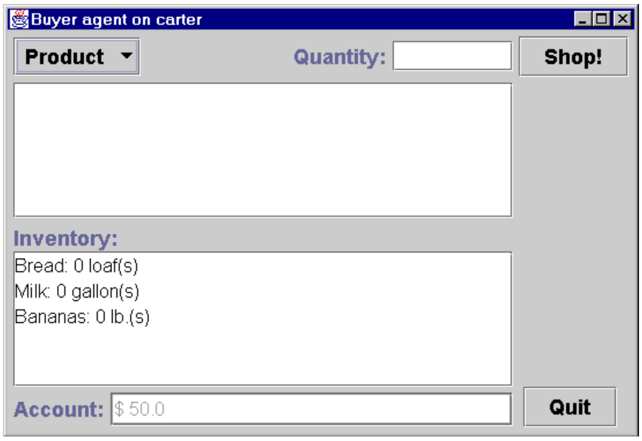
In this example the buyer agent's mental model will be initialized with several Product objects; the control panel must be initialized with the same objects. The Add Product rule will be activated once for each `Product` object in the agent's mental model, and will cause the control panel to add the product to the product choice box that is displayed to the user. See the code segment in Figure 46.

```
WHEN:
IF:
1. ( BIND buyerFrame )
2. ( BIND ?product )
THEN:
1. DO buyerFrame.addProduct( ?product )
```

Figure 46. Add Product Rule

(9) Write a rule to display the control panel.

After the initial data has been loaded into the control panel object, the interface can be displayed. The Start Control Panel rule invokes the `run()` method on the `buyerFrame` object, which will display the interface. Construction of the interface will occur on a separate thread, so the agent's main thread will not be delayed. Note that in many applications there will not be a need for a separate rule to add initial data (e.g., the Add Product rule) and so the interface display rule can be combined into the interface initialization rule. In this example the separate step is required because of the varying number. Figure 47 shows the `BuyerFrame` interface after it has been initialized. The code is shown in Figure 48.



Bread: 0 loaf(s)
Milk: 0 gallon(s)
Bananas: 0 lb.(s)

$ 50.0

Figure 47. BuyerFrame PAC

```
WHEN:
IF:
1. ( BIND buyerFrame )
2. ( BIND accountBalance )
THEN:
1. buyerFrame.run()
2. buyerFrame.displayAccountBalance( accountBalance )
3. DO RemoveRule( "Add Product" )
4. DO RemoveRule( "Start Control Panel" )
```

Figure 48. Start Control Panel Rule

(10) Write rules to handle the messages sent by the control panel.

Using the protocol developed when designing the event handler for the PAC, write rules that accept the messages sent by the PAC. The name of the PAC was stored into the `PacCommSystem` in the Build BuyerFrame rule, and that same PAC name will be used as the sender name for all messages sent by the PAC. The following rule tests the Sender field of the message to ensure that a received message came from the PAC, the performative is “forward”, and the content is a `PriceQuote` object. This rule then stores the message temporarily into the agent's mental model and initializes a `Purchase` object to represent the current situation. Another rule will forward this newly-received message on to the store agents. Figure 49 shows the rule which handles messages sent by the control panel.

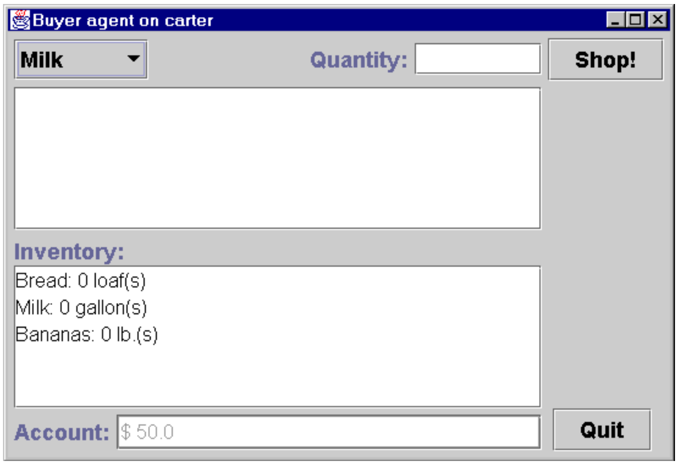
```
WHEN
1. ( %message.sender EQUALS pacName )
2. ( %message.performative EQUALS "forward" )
3. ( %message.contentType EQUALS PriceQuote )
IF
THEN
1. ASSERT( "currentMessage" %message )
2. ASSERT( "currentPriceQuote" %message.content )
3. ASSERT( Purchase(%message.content.productName,%message.content.quantity,
           %message.content.unitName, "New Request From User" ) )
```

Figure 49. Receive Message from Control Panel Rule

(11) Write rules to generate output to the control panel.

Sometimes output to the control panel will be triggered by messages received from other agents or by combinations of values in the mental model of the agent. Private actions can be used to display output to the user in response to events or situations detected by the agent. Figure 50 shows the `BuyerFrame` interface after a message has been received from the control panel and forwarded to the store agent.

Figure 51 shows the rule which handles messages from store agents. This rule will be activated by a message containing a completed `PriceQuote` object. The user will be notified that the mes-



Bread: 0 loaf(s)
Milk: 0 gallon(s)
Bananas: 0 lb.(s)

$ 50.0

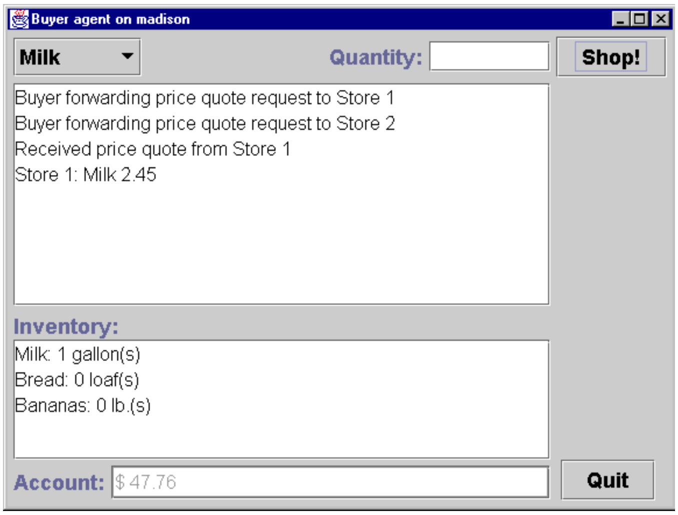
Figure 50. BuyerFrame PAC

sage has been received and the contents of the `PriceQuote` will be displayed.

Figure 52 shows the `BuyerFrame` interface after a `PriceQuote` message has been received from one of the store agents.

```
WHEN:
1. ( %message.sender  EQUALS ?agent.agentInfo.name )
2. ( %message.performative  EQUALS "tell" )
3. ( %message.contentType  EQUALS PriceQuote )
4. ( %message.content.productName  EQUALS ?product.productName )
IF:
1. ( BIND buyerFrame )
2. ( ?product.status EQUALS "Requested Bids" )
THEN:
1. DO buyerFrame.displayMessage( Concat( "Received price quote from
",
                    %message.sender ) )
2. DO buyerFrame.displayPriceQuote( %message.content )
3. ASSERT( %message.content )
```

Figure 51. Receive Message from Store Agent Rule



```
Buyer forwarding price quote request to Store 1
Buyer forwarding price quote request to Store 2
Received price quote from Store 1
Store 1: Milk 2.45
```

```
Milk: 1 gallon(s)
Bread: 0 loaf(s)
Bananas: 0 lb.(s)
```

$ 47.76

Figure 52. BuyerFrame PAC

Chapter 2

Run-Time System

Chapter Overview

You can find the following information in this chapter:

- Starting the Agent Engine
- Engine Options
- Engine Launcher
- Engine Console
- Built-In Actions
- Agent Engine Cycle
- Engine Threads

A. Run-Time System

Run-Time Agent Engine

The Run-Time System consists of the Agent Program and the run-time agent engine. The Agent Program is a combination of the agent definition in the RADL file and the Project Accessories Library (PAL). The Agent Program is executed by the run-time agent engine; the combination of the Agent Program and the agent engine produces an executable agent.

At start-up, the Run-Time System initializes the agent engine using information stored in the RADL file and links the required components from the Project Accessories Library. Both the agent definition and the project accessory classes (PACs) are needed: the agent definition supplies the agent with a reasoning capability and an initial mental model; the PACs are the objects used to represent the problem domain and provide the agent the ability to interact with its environment.

The Run-Time System allows an agent to be created in the development environment and then be deployed as a stand-alone entity executing in the run-time environment. It is the agent executing in the Run-Time System that performs useful work. The Run-Time System is distinct from the AgentBuilder Toolkit. After the agent development process is completed, an agent can be executed on any platform with a Java Virtual Machine (version 1.1 or later). The Run-Time System does not require access to the Java Development Environment.

The agent engine is a proprietary inferencing engine implemented in Java. This engine utilizes an efficient and robust inferencing procedure to match the agent's behavioral rules with the agent's beliefs and incoming messages. The agent engine performs the reasoning

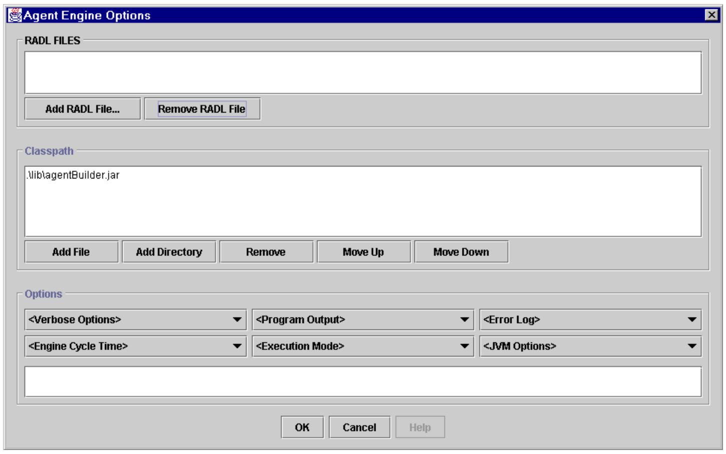


Figure 53. Agent Engine Options

defined in the RADL file and executes the specified actions. The available actions are the built-in actions provided by the Run-Time System and the actions based on methods from project accessory classes. The agent engine monitors the execution of the actions and returns execution results to the agent.

Starting the Agent Engine

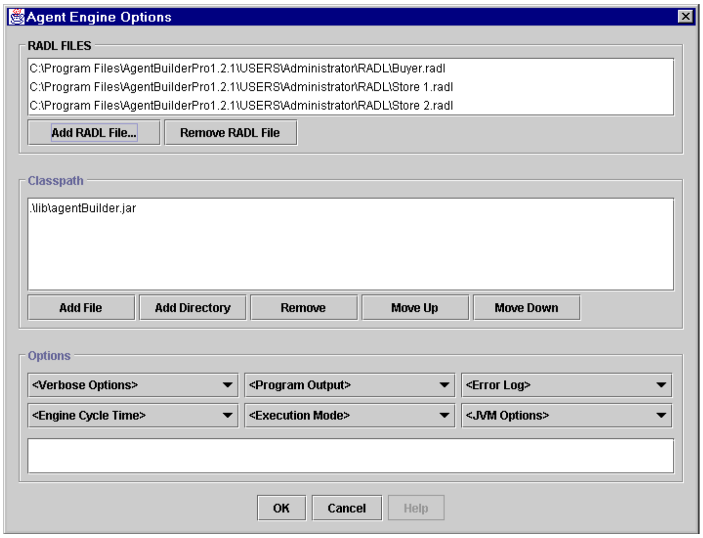
There are several ways to launch the agent engine and begin executing an agent program. The **Agent Engine Options** dialog can be started from the agent tool. The **Agent Engine Options** dialog can also be started from a shell script, or the agent engine can be started directly from a command line by invoking the engine shell script.

Agent Engine Options

The **Agent Engine Options** dialog provides a graphical interface for specifying agent engine options. Listed below are the options that can be specified. As the options are specified they are displayed in the text area at the bottom of the dialog. Figure 53 shows the interface.

RADL File

The RADL file can be specified by clicking on the **Add RADL File...** button. This will bring up the File dialog in which you can select the RADL file. Multiple RADL files can be specified to run in the same Java Virtual Machine by continuing to click on the **Add Radl File...** button. To remove a RADL file from the list, select the RADL file name and then click on the **Remove RADL File** button. Figure 54 shows the **Agent Engine Options** panel with a RADL file specified.



C:\Program Files\AgentBuilderPro1.2.1\USERS\Administrator\RADL\Buyer.radl
C:\Program Files\AgentBuilderPro1.2.1\USERS\Administrator\RADL\Store 1.radl
C:\Program Files\AgentBuilderPro1.2.1\USERS\Administrator\RADL\Store 2.radl

.\lib\agentBuilder.jar

Figure 54. Agent Engine Options

Classpath

The CLASSPATH is displayed in a list. It is initialized with the CLASSPATH which was set at the time the **Agent Engine Options** dialog was opened. You can modify the classpath by using the buttons located below the classpath list. The **Add File** button will display the **File** dialog, which will allow you to select either a jar or zip file. The **Add Directory** button will display the **Directory** dialog which only allows you to select directories. The **Remove** button will remove the current selection under the classpath list. The **Move Up** and **Move Down** buttons allow you to specify the order the classpath is read. When the Agent Engine is launched, a new Java Virtual Machine will be instantiated using the specified classpath. Note: The classpath field is not editable when the **Agent Engine Options** dialog is started from the console because the agent will run in the existing Java Virtual Machine and must use the existing classpath.

Verbose Options

The **Option**s choice box contains the following items: **Messages**, **Changed Beliefs**, **All Beliefs**, **Fired Rules**, **Everything**, **Trace File…** and **Clear Verbose Options**

Almost any combination of the first five options may be selected. The **Changed Beliefs** and **All Beliefs** options are mutually exclusive, so selecting one of them will cause the other to be de-selected. Selecting the **Everything** option will replace any other options that have been selected. Verbose output will be sent to the output stream used for regular program output (e.g., console or screen) as well as to any trace files that have been specified by the user. Figure 55 shows the **Agent Engine Options** panel with several verbose options specified.

Shown below are examples of the run-time output for each of the verbose options. These were taken from the BuyerSeller example application. This output will appear in the console or on the screen at run-time and not in the **Agent Engine Options** dialog. These examples are included here to show the effects of the options selected in the **Agent Engine Options** dialog.

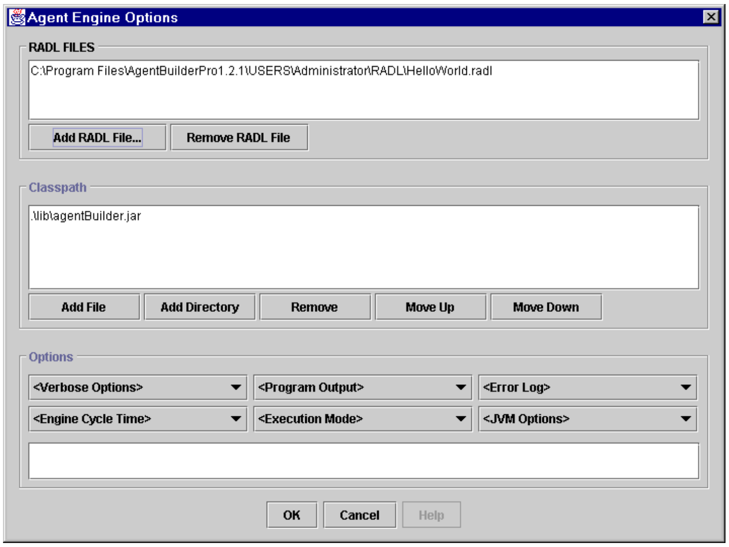


Figure 55. Agent Engine Options

The **Verbose Messages** option will cause the engine to print all incoming messages received by the agent during each cycle. All non-empty KQML fields in messages are printed; the content object is printed via the `toString` method. Figure 56 shows verbose output after the buyer agent has received a message from the Store 2 agent and the message contains a price quote.

The **Changed Beliefs** option will cause the engine to print the beliefs which changed during the cycle. Below is an example of the verbose output taken from the BuyerSeller application. Figure 57 shows the changes in the buyer agent's mental model due to a purchase of 2 gallons of milk: the amount the buyer believes is in the account has decreased, and the quantity of milk the buyer believes is in his inventory has increased.

```
 INBOUND MESSAGES
sender:      "Store 2"
receiver:    "Buyer"
performative: "tell"
replyWith:    "Price quote"
inReplyTo:    "Price quote"
contentType:  com.reticular.agents.buyerSeller.PriceQuote
content:    "{Product Name = Milk, Quantity = 2 gallon, Price = 2.24,
                Store Name = Store 2}"
```

Figure 56. Run-Time Output (Verbose Option)

```
CHANGED BELIEFS
Changed: Float<accountBalance> 50.0 To: 45.52
Changed: InventoryRecord<> Name = Milk, Quantity = 0 gallon To:
                           Name = Milk,       Quantity = 2 gallon
Retracted: Purchase<> {Product Name = Milk, Quantity = 2 gallon,
                           Unit   Price = 0.0, Total Price = 4.48,
                           Status = Accepted quote}
```

Figure 57. Run-Time Output (Changed Beliefs)

The **All Beliefs** option will cause the engine to print all beliefs in the agent's mental model at end of cycle. New beliefs or beliefs which changed during the cycle are marked with a *. Figure 58 is an example of the verbose output taken from the Buyer-Seller application at the same cycle used in the previous example. The inventory record for milk and the account balance are marked as new beliefs because they changed during the cycle. The `current-Time` belief is also marked as a new belief; it's an instance of the built-in Time class which is automatically updated every cycle by the agent engine.

```
BELIEFS
* InventoryRecord<> Name = Milk, Quantity = 2 gallon
  InventoryRecord<> Name = Bread, Quantity = 0 loaf
  InventoryRecord<> Name = Bananas, Quantity = 0 lb.
  Agent<SELF> Agent Name: Buyer Address: quincy.reticular.com
  Agent<Store 1> Agent Name: Store 1 Address: harding.reticular.com
  Agent<Store 2> Agent Name: Store 2 Address: harding.reticular.com
  String<selfName> Buyer
  String<pacName> Buyer:ControlPanel
  Boolean<controlPanelIsReady> true
  BuyerFrame<buyerFrame>
            com.reticular.agents.buyerSeller.BuyerFrame[frame0,      0,
            3,549x379,layout=java.awt.GridBagLayout,resizable,title=Bu
            er agent on quincy.reticular.com]
* Float<accountBalance> 45.52
* Time<currentTime> Thu Apr 23 11:45:02 PDT 1998
  Time<startupTime> Thu Apr 23 11:44:36 PDT 1998
```

Figure 58. Agent's Beliefs at End of Cycle

The **Fired Rules** option will cause the engine to print the names of the rules which fired in the cycle. As you can see in this example, rules may fire more than once per cycle. Here the "Cleanup old quotes" rule fired twice; once per each old price quote that was found in the mental model.

FIRED RULES: Cleanup old quotes, Cleanup old quotes, Receive purchase confirmation, Update account

Selecting the **Trace File** item will display a File Dialog and a trace file can be selected. Verbose output will also be printed to whatever output stream is being used for program output. A trace file, if one is specified, will contain only the verbose output from a program.

Selecting the **Clear Verbose Options** item will clear any verbose options that were previously specified. Figure 59 shows the **Agent Engine Options** dialog after the verbose options have been cleared.

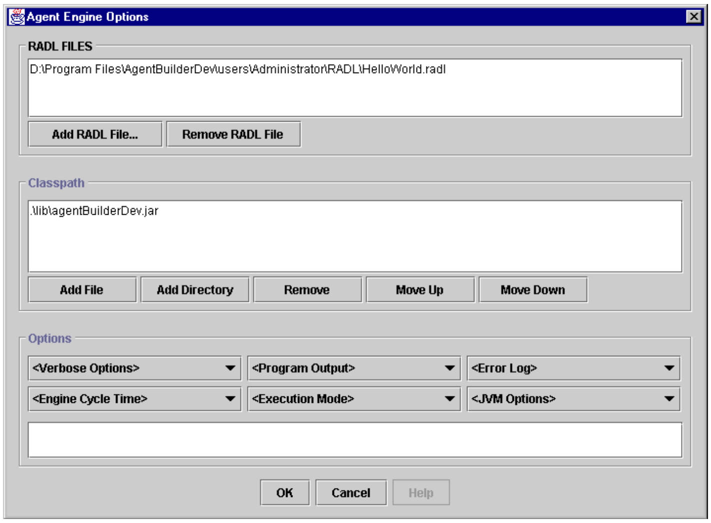
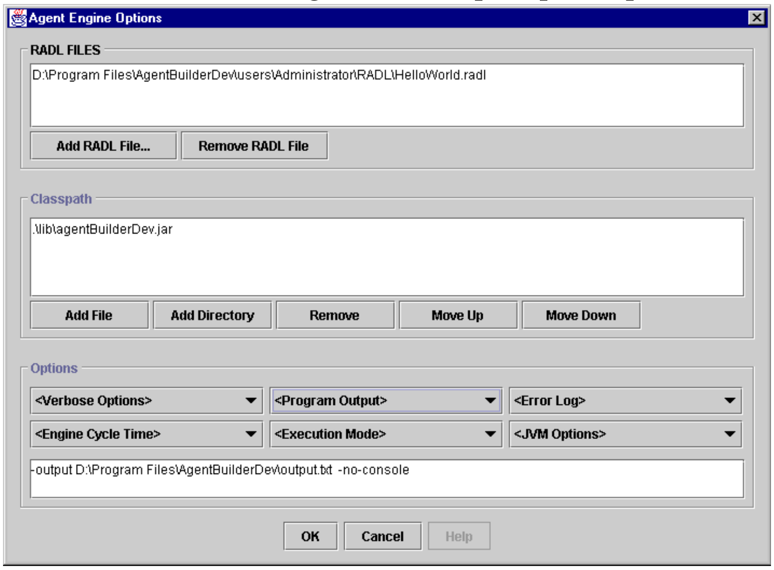


Figure 59. Agent Engine Options

Program Output

The **Program Output** choice box contains the following items: **Output File...**, **No Console**, **No System.out**, and **Clear Output Options**

Program output is all the output generated from the agent engine or from the built-in action `SystemOutPrintln`, and all output generated by calls to `System.out.println` in PAC code. Figure 60 shows the **Agent Engine Options** dialog with an output option specified.



RADL FILES: D:\Program Files\AgentBuilderDev\users\Administrator\RADL\HelloWorld.radl

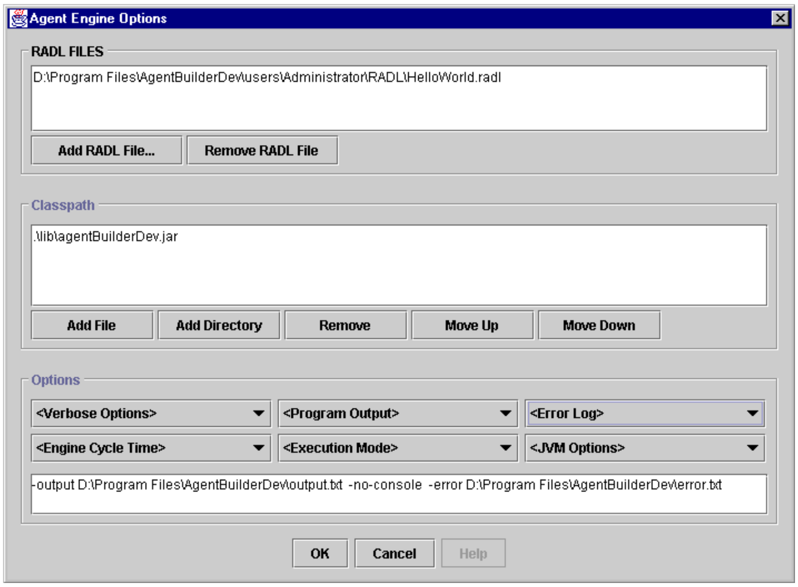
Classpath: .\lib\agentBuilderDev.jar

Options: -output D:\Program Files\AgentBuilderDev\output.txt -no-console

Figure 60. Agent Engine Options

Selecting the **Output File** item will display a File Dialog and an output file can be selected. Specifying an output file will not affect the standard screen or console output; all output will automatically be printed to the screen or console and the output file(s). Any number of output files can be used. Note that there may be a noticeable slowdown in the agent engine when printing large volumes of data to output files.

Selecting the **No Console** option will cause the engine to operate without the default console. If you have not also selected the **No System.out** option, program output will be printed to the standard output stream for your system and to any output files you've specified. If you've selected **No Console** *and* **No System.out**, program



```
D:\Program Files\AgentBuilderDev\users\Administrator\RADL\HelloWorld.radl
```

```
.\lib\agentBuilderDev.jar
```

```
-output D:\Program Files\AgentBuilderDev\output.txt  -no-console  -error D:\Program Files\AgentBuilderDev\error.txt
```

Figure 61. Agent Engine Options

output will be printed only to files you've specified. Selecting **No System.out** without selecting **No Console** has no effect.

Error Log

The **Error Log Options** choice box contains the following items: **Error File…**, **No System.err**, and **Clear Error.**

Error output is all the error messages generated from the agent engine and all output generated by calls to `System.err.println` in PAC code. Figure 61 shows the **Agent Engine Options** dialog with an error log option specified.

Selecting the **Error File** item will display a File Dialog in which an error log file can be selected. Specifying an error log file will not affect the standard screen or console error display; all error messages will automatically be printed to the screen or console and the error log file(s). Any number of error log files can be used.

The **No System.err** option is similar to the **No System.out** option: it only has an effect if **No Console** is also specified. If you've selected **No Console** *and* **No System.err**, error messages will be printed only to any error log files you've specified. Selecting **No System.err** without selecting **No Console** has no effect.

Starting the Agent Engine from a Command Line

Figure 62 shows the options available when starting the agent engine via the shell script. (This is the output that is displayed when invoking the shell script with the help option, i.e., `engine -h`.) For more explanation, please see the descriptions of the options in "Agent Engine Options" on page 253.

Java options

The options that can be specified for the Java Virtual Machine are listed in Table 1. For a complete description of these options please see the Java documentation. Numerical option arguments, such as *`-ss`* or *`-mx`*, should be given with a space between the option name and the number, e.g., *`-ss 1024.`*

Table 1. AgentEngine Command Line Options

Option	Use
-verbosegc	Causes the garbage collector to print out messages whenever it frees memory.
-noasyncgc	Turns off asynchronous garbage collection.
-noclassgc	Turns off garbage collection of Java classes.
-verify	Performs byte-code verification on the class file.

Table 1. AgentEngine Command Line Options

Option	Use
-verifyremote	Runs the verifier on all code that is loaded into the system via a class loader.
-noverify	Turns verification off.
-ss<number>	Sets the maximum stack size that can be used by C code in a thread.
-oss <number>	Sets the maximum stack size that can be used by Java code in a thread.
-ms <number>	Sets the start-up size of the memory allocation pool.
-mx <number>	Sets the maximum size of the memory allocation pool.
-prof	Starts the Java Runtime with Java profiling enabled.
-checksource	Compares the modification time of the class byte code file to the modification time of the class source file, and automatically recompiles and reloads if needed.
-cs	Same as -checksource.
-classpath <path-list>	Specifies the path java uses to look up classes.

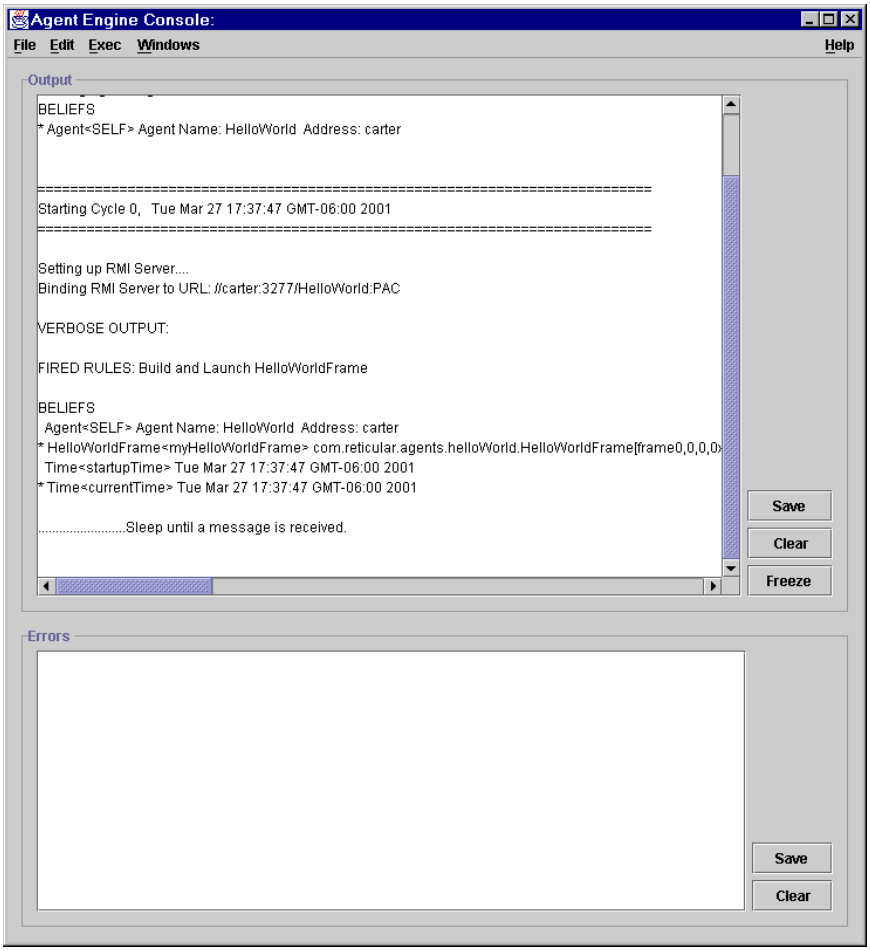
Agent Engine Console

The agent engine console provides a standard way to view program output and error messages, as well as control some aspects of agent execution. For some agent applications the standard console will be sufficient for monitoring agent execution; for other applications specialized display panels will be needed. The console can be turned off by specifying the `-no-console` option on the command

```
Usage: /common/agentBuilder/bin/engine <RADL filename> [<options>*]
Example: /common/agentBuilder/bin/engine test2.radl -vc -o
test2results.txt
Options:
    -h  or  -help
    -i  or  -interface
    -v<level> [trace=<filename(s)>]  or
    -verbose <level> [trace=<filename(s)>]
          m  print inbound messages
          c  print changed beliefs each cycle
          b  print all beliefs each cycle
          r  print fired rule names
          e  print everything each cycle
          Note: -v and -verbose are equivalent to -ve
          Examples: -v, -vm, -vmbra, -verbose r
                -vcr trace=myFile.txt,otherFile.txt

    -o <filename(s)>  or  -output <filename(s)>
    Example: -o myFile.txt   -o file1.txt,file2.txt

    -e <filename(s)>  or  -error <filename(s)>
    Example: -e myErrorFile.txt

    -no-console     Don't use the console, write to the screen.
    -no-system-out  Suppress printing of System.out to the screen.
    -no-system-err  Suppress printing of System.err to the screen.
    -d<parameter>=<value> or -define <parameter>=<value>
        Example: -dCycleTime=4

Java options: The following options can be specified for the Java
virtual machine.  For a description of these options please see
the Java documentation.  Numerical option arguments, such as for
-ss or -mx, should be given with a space between the option name
and the number, e.g., -ss 1024.

  -verbosegc, -noasyncgc, -noclassgc, -verify, -verifyremote, -
noverify,   -ss <number>, -oss <number>, -ms <number>, -mx
<number>,   -prof, -cs  or  -checksource

-classpath <path-list>       Use the specified classpath.
Example: -classpath .:../..:/myPath/myPackage
```

Figure 62. Agent Engine Command Line Options

line or in the Agent Engine Options **Program Output** choice box. If the console is not used, all program output and error messages will go to the standard output and error streams (e.g., the screen) and to any files specified by the user. Figure 63 shows the console.



```
BELIEFS
* Agent<SELF> Agent Name: HelloWorld  Address: carter


==============================================================================
Starting Cycle 0,   Tue Mar 27 17:37:47 GMT-06:00 2001
==============================================================================

Setting up RMI Server....
Binding RMI Server to URL: //carter:3277/HelloWorld:PAC

VERBOSE OUTPUT:

FIRED RULES: Build and Launch HelloWorldFrame

BELIEFS
  Agent<SELF> Agent Name: HelloWorld  Address: carter
* HelloWorldFrame<myHelloWorldFrame> com.reticular.agents.helloWorld.HelloWorldFrame[frame0,0,0,0x
  Time<startupTime> Tue Mar 27 17:37:47 GMT-06:00 2001
* Time<currentTime> Tue Mar 27 17:37:47 GMT-06:00 2001

........................Sleep until a message is received.
```

Figure 63. Engine Console

When the console is used it will capture all program output and error messages. The agent engine redirects system output and system errors to the console (and any other files that the user specifies), so all calls to `System.out.println` or `System.err.println` in PAC code will be sent to the appropriate console text area. Also, any output generated from the built-in action `SystemOutPrintln` or other output or error messages from the agent engine will be sent to the console. Capturing PAC output using the console should be acceptable in most situations. However, if you're using the console and need to print some PAC output to the screen instead of the console, code such as the following can be used:

```
PrintWriter standardOutput =
  new PrintWriter(new FileWriter(FileDescriptor.out),
  true);
standardOutput.println("yada yada yada");
```

This will print to the standard output stream on your system, NOT to the console. Similarly, building a `PrintWriter` from `FileDescriptor.err` will allow you to print to the standard error stream on your system. Of course the name of the `PrintWriter` object (here, “`standardOutput`”) can be any name you choose. The second argument in the constructor, “`true`”, tells the system to flush the output buffer automatically after every `println` statement; use this value if you want output to appear immediately after it's printed. To use this code your PAC will need the import statement `import java.io.*;` (or you could import each of the classes separately or use the full package.class names such as `java.io.PrintWriter`).

The main purpose of the console is to display program output. The console features a scrollable text area and three buttons that affect the output display: **Save**, **Clear**, and **Freeze**. Clicking on the **Save** button will cause a File Dialog to appear; all the output currently in the output text area (including any text that has scrolled out of view) will be saved into the selected file. Clicking on the **Clear**

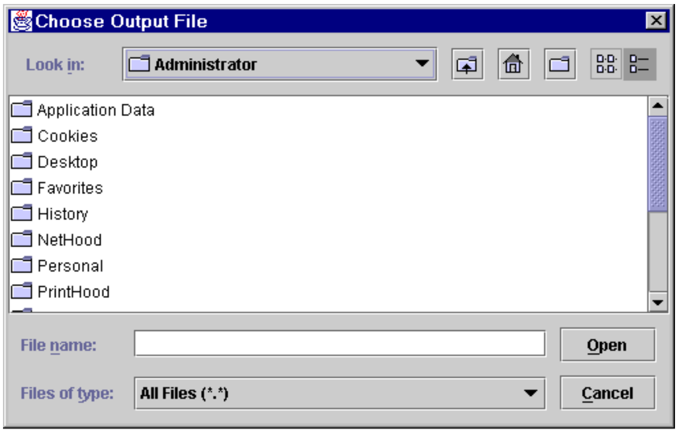


Figure 64. Save File Dialog

button will clear the output text area. Clicking on the **Freeze** button will change the button label to **Resume** and will freeze the output but will NOT halt the agent engine. The engine will continue executing normally and all output will be stored in the console until the **Resume** button is clicked, then all the stored output will be written to the output text area. The **Freeze** button is provided to make it easier to view a running agent without distraction from frequent updates in the output text area. Figure 64 shows the **Save Output File** dialog used to select a file for the console's program output.

The console has a scrollable error text area located below the output text area. This area includes two buttons—**Save** and **Clear—**that affect the error display. These buttons have the same effect as the buttons for the output text area, except that they save or clear the error text area. There is no **Freeze** button for the error display since error messages will be infrequent and freezing the display is unnecessary.

The **File** menu in the console offers the following items and their keyboard shortcuts: **Load RADL File…**,**Close Console**, **Exit Run-**

Time System. The control key equivalents for each of these File menu selections are shown in Table 2.

Table 2. File Menu Keyboard Equivalents

Menu Item	Key
Load RADL File…	Ctrl + L
Close Console	Ctrl + H
Exit Run-Time System	Ctrl + E

Selecting the **Load RADL File…** item will cause a File Dialog to be displayed. If a file is selected, the current agent engine will be destroyed and a new agent engine will be started using the newly selected RADL file and whatever options were used in the previous agent engine.

Selecting the **Close Console** item will close the console but **leaves the agent engine running**. Output and error messages from the agent engine and PACs will continue to be printed to files if any files were specified by the user.

Selecting the **Exit Run-Time System** item will terminate the agent engine and close the console.

The **Edit** menu in the console provides the following commands: **Cut**, **Copy**, and **Paste**. The keyboard shortcuts for these menu items are shown in Table 3.

Table 3. Edit Menu Keyboard Equivalents

Menu Item	Key
Cut	Ctrl + X
Copy	Ctrl + C

Table 3. Edit Menu Keyboard Equivalents

Menu Item	Key
Paste	Ctrl + P

All Edit options are applied to a highlighted block of text in either the output or error text areas. (Currently not functional)

The **Exec** menu provides the following commands: **Set Engine Options…**, **Restart Engine**, **Terminate Engine**. The control key equivalents for each of these **Exec** menu selections are shown in Table 4.

Table 4. File Menu Keyboard Equivalents

Menu Item	Key
Set Engine Options…	Ctrl + O
Restart Engine	Ctrl + R
Terminate Engine	Ctrl + T

Selecting the **Set Engine Options…** item will cause the **Agent Engine Options** dialog to be displayed. This allows the user to select a new RADL file and/or new options for the agent engine. If the user clicks the **OK** button the current agent engine will be destroyed and a new agent engine will be started using the RADL file and options specified in the **Agent Engine Options** panel.

Selecting the **Restart Engine** item will destroy the current agent engine and start a new agent engine using the current RADL file and options. Any PAC control panels subclassed from `java.awt.Window` or its subclasses (e.g., `java.awt.Frame`, `java.awt.Dialog`) will be destroyed.

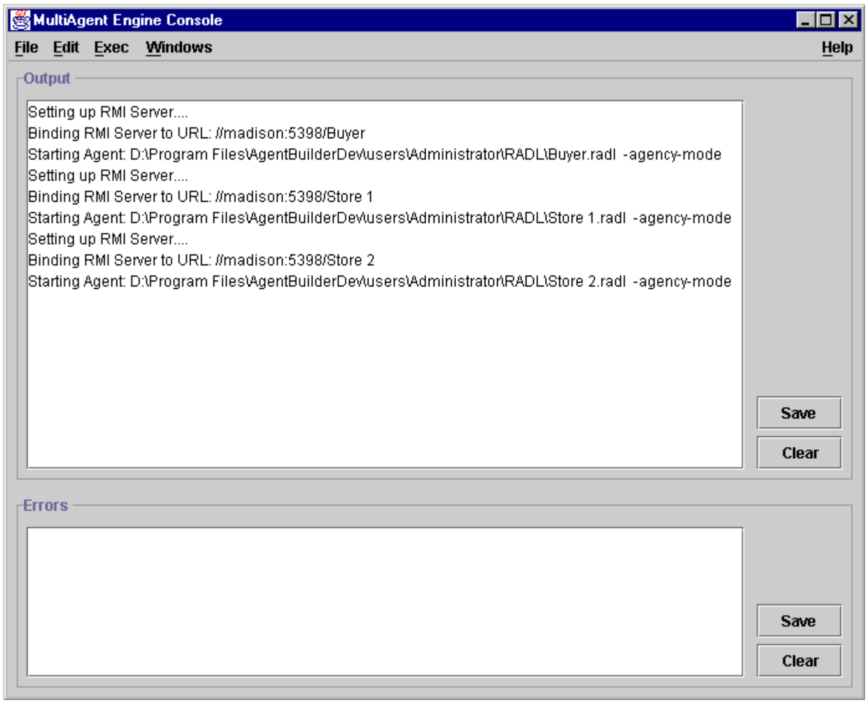
Selecting the **Terminate Engine** item will destroy the current agent engine. Any PAC control panels subclassed from `java.awt.Window` or its subclasses (e.g., `java.awt.Frame, java.awt.Dialog`) will be destroyed. Terminating the agent engine will **not** close the console.

MultiAgent Engine Console

When more than one agent is running in the same Java Virtual Machine, the **MultiAgent** console will be used, see Figure 65. In addition to the regular Agent Engine Console you get for each agent running, you will also see a MultiAgent Engine Console. The MultiAgent Engine Console has a similar interface to the Agent Engine Console, but behaves differently. The output you see on the output text area of the MultiAgent Engine Console includes the RADL files and options used to start the engines for the agents. Also, if any output/error messages cannot be redirected to the proper Agent Engine Console, it will be displayed on the Multi-Agent Engine Console.

The console provides a scrollable text area and two buttons that affect the ouput display: **Save** and **Clear**. These buttons perform the same functions as the **Save** and **Clear** buttons for the Agent Engine Console. The **Save** button saves the output to a file, and the **Clear** button clears the output text area. The second scrollable text area is for error output. This area also includes two buttons: **Save** and **Clear**. These buttons have the same effect as the buttons for the output text area, except that they save and clear the error text area.

The **File** menu in the console offers the following items: **Close** and **Exit**. The **Close** button will close the MultiAgent Engine display and leave running any agent engines that are currently executing. If the MultiAgent Engine Console is the only window open, this will



```
Setting up RMI Server....
Binding RMI Server to URL: //madison:5398/Buyer
Starting Agent: D:\Program Files\AgentBuilderDev\users\Administrator\RADL\Buyer.radl  -agency-mode
Setting up RMI Server....
Binding RMI Server to URL: //madison:5398/Store 1
Starting Agent: D:\Program Files\AgentBuilderDev\users\Administrator\RADL\Store 1.radl  -agency-mode
Setting up RMI Server....
Binding RMI Server to URL: //madison:5398/Store 2
Starting Agent: D:\Program Files\AgentBuilderDev\users\Administrator\RADL\Store 2.radl  -agency-mode
```

Figure 65. Multi-Agent Engine Console

exit the virtual machine. If the **Exit** menu item is selected, the console will terminate all agent engines and exit the virtual machine.

The **Exec** menu provides commands to **Restart All Engines** and **Terminate All Engines**. Selecting **Restart All Engines** will terminate all agent engines running in this Java Virtual Machine and restart them using the current RADL file and options. Selecting the **Terminate All Engines** menu item will terminate all agent engines running in this Java Virtual Machine.

Built-in Actions

The agent engine provides a suite of built-in actions which can be called from rules. This section describes the actions and the required arguments.

`ConnectAction`

```
void ConnectAction( String actionName, Object pacObject )

RADL example:
(DO ConnectAction( [VAL String "action1"], [INST MyPac<pac1>] ))
```

`ConnectAction` provides a way to connect a named action to its underlying PAC object. This is not required in AgentBuilder 1.1 and later and exists only for compatibility with older RADL code.

`GetCycleTime`

```
Float GetCycleTime()

RADL example:
(DO [RVAR Float<currentCycleTime>] = GetCycleTime())
```

`GetCycleTime` returns the current value for the agent engine cycle length, in seconds. The `Float` value returned by the action can be stored into a variable and then asserted into the agent's mental model or used elsewhere on the RHS of the enclosing rule.

`GetHostAddress`

```
String GetHostAddress())

RADL example:
(DO [RVAR String<hostAddress>] = GetHostAddress())
```

`GetHostAddress` returns the IP address (as a `String`, e.g., “199.106.0.42”) of the physical machine on which the agent engine is running. The `String` value returned by the action can be stored into a variable and then asserted into the agent's mental model or used elsewhere on the RHS of the enclosing rule.

GetHostName

```
String GetHostName())

RADL example:
(DO [RVAR String<hostName>] = GetHostName())
```

`GetHostName` returns the host name of the physical machine on which the agent engine is running. The `String` value returned by the action can be stored into a variable and then asserted into the agent's mental model or used elsewhere on the RHS of the enclosing rule.

GetSystemProperty

```
String GetSystemProperty( String keyString )

RADL example:
(DO [RVAR String<cwd>] = GetSystemProperty( [VAL String "user.dir"] ))
```

`GetSystemProperty` provides access to the standard system properties that are available to any Java program. `GetSystemProperty` is implemented via the standard `System.getProperty(String)` method and uses the same key strings, e.g., “`user.dir`”. The `String` value returned by the action can be stored into a variable and then asserted into the agent's mental model or used elsewhere on the RHS of the enclosing rule.

OpenConsole

```
void OpenConsole()

RADL example:
(DO OpenConsole())
```

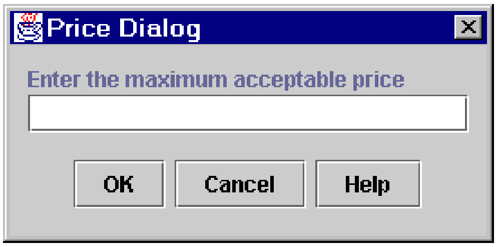
The `OpenConsole` built-in action is intended for use in environments where the engine is running without a console and a situation is detected which should be brought to the attention of the user. If there is no console at the time `OpenConsole` is invoked it will open a console and begin printing program output and error messages to the console (in addition to any output or error log files that may be open). If `OpenConsole` is invoked when a console is already open then the action will have no effect.

OpenInputDialog

```
void OpenInputDialog( String dialogName,
       String promptString, Class expectedType )

RADL example:
(DO OpenInputDialog( [VAL String "Price Dialog"]
                [VAL String "Enter the maximum acceptable price"]
                [VAL Class Float] ))
```

The `OpenInputDialog` built-in action provides easy access to the `InputDialog` built-in PAC. Although it is possible to build and use the `InputDialog` directly, this built-in action simplifies the construction. The `InputDialog` can be used to obtain input of class `String`, `Character`, `Boolean`, or any of the numerical classes (e.g., `Integer`). For each dialog the user must provide a reference to the agent's `AgentInfo` object, a name for the input dialog, a prompt string, and the expected type (as a `java.lang.Class` object). The dialog will only accept input that can be converted to the specified type, so for example an `InputDialog` that expects a `Float` input will not accept an input string containing non-numeric characters. Any range checking or further testing of the input must be done by the agent. Figure 66 shows the `InputDialog` built by the `OpenInputDialog` action shown above.

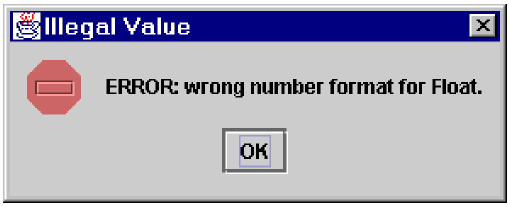
In the example shown above the dialog will be named **Price Dialog**, the prompt string will be *`Enter the maximum acceptable price`*, and the dialog will only accept a `Float` input. The `InputDialog` will be a modal dialog which will be displayed until the user types a valid input and hits the **Enter** key on the keyboard or clicks on the **OK** button or the **Cancel** button. If the input cannot be con-



Enter the maximum acceptable price

Figure 66. Built-In Input Dialog

verted into the specified type an error dialog will be displayed. Figure 67 shows the error dialog for this example.



ERROR: wrong number format for Float.

Figure 67. Error Dialog

If the user enters a valid input, the input object (in this case a `Float` object) will be sent to the agent as the content of a KQML message with the performative “tell”. The following patterns could be used in a rule to detect a message from the `InputDialog` built in the `OpenInputDialog` example above:

(OBJ [VAR KqmlMessage<%m>.Sender] EQUALS [VAL String “Price Dialog”]) (OBJ [VAR KqmlMessage<%m>.Performative] EQUALS [VAL String “tell”]) (OBJ [VAR KqmlMessage<%m>.ContentType] EQUALS [VAL Class Float])

If a KQML message satisfies these patterns then the message will be accepted as an input message from the `InputDialog` and the value can be extracted from the content object and stored into the agent's mental model.

`OpenOutputDialog`

```
void OpenOutputDialog( String outputString )

RADL example:
```
(DO OpenOutputDialog([VAL String "The answer is 42."])

`OpenOutputDialog` creates a simple output dialog with a text area and displays the given output string in the text area.

`RegisterAgent`

```
void RegisterAgent( AgentInfo otherAgentInfo )

RADL example:
```
(DO RegisterAgent([VAR AgentInfo<?info>]))

The `RegisterAgent` built-in action is used to register a new agent with the communication system.

`RemoveRule`

```
void RemoveRule( String ruleName )

RADL example:
```
(DO RemoveRule([VAL String "Rule 1"]))

`RemoveRule` removes the specified rule from the agent's rule base.

`SendKqmlMessage`

```
void SendKqmlMessage( KqmlMessage outputMessage,
             String receiverName,
             String performative,
             Object content,
             String replyWith,
             String inReplyTo,
             String language,
             String ontology,
             String protocol,
             String toAgentName,
             String fromAgentName )

 RADL example:
 (DO SendKqmlMessage( [NEW KqmlMessage],
             RECEIVER[ VAR Store<?s>.Name ],
             PERFORMATIVE[ VAL String "achieve" ],
             CONTENT[ VAR Purchase<?p> ],
             REPLY_WITH[ VAL String "Purchase Confirmation" ],
             [], [], [], [], [], [] ))
```

`SendKqmlMessage` fills in the fields of the `outboundMessage` argument with any values specified in the remainder of the argument list then queues the message for sending at the end of the cycle. This action will not modify the fields in the outbound message if the corresponding argument is unspecified. For example, if the content argument is not specified, the content field in the outbound message will not be changed. This allows for easy re-use of message objects without requiring a complete rebuilding of the message. The `outboundMessage` argument may be a reference to a new or existing KQML message object.

In the example shown (from buyer.radl) a new KQML message object is used as the first argument. The message will be initialized with the agent's name as the sender, a store's name as the receiver, the performative “achieve” and “Purchase confirmation” as the string with which to reply. These values are determined by a communications protocol developed when the system was designed; the buyer agent and the seller agents have rules that were written according to this protocol. For the fourth argument a variable of type `Purchase` will be installed as the content object in the outbound message. The other fields (i`nReplyTo`, `language`, `ontology`, `protocol`, `toAgentName`, and `fromAgentName`) will be left as-is in the outbound message, which in this case means they will be null.

`SetCycleTime`

`void SetCycleTime( float cycleTimeInSeconds )`

`RADL example:`

(DO SetCycleTime([VAL Float 3.0]))

`SetCycleTime` changes the length of the agent engine cycle to the specified value. The input value must be given in seconds.

`Sleep`

`void Sleep( int numberOfSeconds )`

`RADL example:`

(DO Sleep([VAL Integer 30]))

The `Sleep` built-in action will put the agent to sleep for a specified amount of time, starting at the end of the cycle in which the action is invoked. The agent engine will monitor its incoming message queue and wake the agent when a message has been received, even if the specified sleep time has not elapsed. If no messages are received, the engine will wake the agent when the specified sleep time has elapsed.

`SleepUntilMessage`

`void SleepUntilMessage()`

`RADL example:`

(DO SleepUntilMessage())

The `SleepUntilMessage` built-in action will put the agent to sleep for an unspecified amount of time, starting at the end of the cycle in which the action is invoked. The agent engine will monitor its incoming message queue and wake the agent when a message has been received.

`SleepWhenIdle`

`void SleepWhenIdle()`

```
RADL example:
```
(DO SleepWhenIdle())

The `SleepWhenIdle` built-in action will put the agent to sleep for an unspecified amount of time, starting at the end of the next cycle in which no rules are fired. The sleep may start as soon as the next cycle following the cycle in which `SleepWhenIdle` was invoked, or the start of the sleep may be delayed indefinitely. The agent engine will monitor its incoming message queue and wake the agent when a message has been received.

This action is useful in situations where the agent must complete some processing, possibly over several cycles, before going to sleep.

`SystemOutPrintln`

```
void SystemOutPrintln()
```

```
RADL example:
```
(DO SystemOutPrintln([VAL String "yada yada yada"]))

`SystemOutPrintln` provides a mechanism for printing output from within a rule. Note that the argument must be a string. Printing any other data type requires that it be converted into a string before `SystemOutPrintln` is invoked.

Kqml Message Failure Handling

There are several reasons why a KQML message might not be delivered. We'll explain some of these and provide a method for detecting and handling them.

The first reason that a message might fail to be delivered is that the sending agent doesn't have communication information about the receiving agent; i.e., the receiving agent is not registered with the sending agent. When this occurs, an error message will be printed and the KQML message will be asserted into the agent's mental state. The following error message results from attempting to send a message to an agent that is not registered with the sending agent.

ERROR Unknown receiver agent "TestAgent"

If the communication information is unknown, there are two alternative methods for adding the agent communication information. The easiest solution is to include all of the agents (that need to communicate) in the same agency. The second solution is to construct an `AgentInfo` object containing the receiving agent's communication information and use the built-in action, `RegisterAgent,` to register the receiving agent with the sending agent.

A message might also fail to be delivered if the receiving agent is not currently executing (i.e. the agent is off-line). This kind of event generates a warning message and, like the previous failure, the KQML message is asserted into the agent's mental state. The following warning message is generated when sending a message to an agent that is off-line.

WARNING: Unable to send message to agent "KqmlMessageAgent"

In either case described above, the user is responsible for handling these types of message failures. Whenever a message fails to be delivered a flag in the KQML message is set. This flag can be obtained by calling the message's `getSendError` method. The following rule is an example of how to handle message send failures:

```
("Receive Undelivered Messages"

WHEN

IF

( OBJ [ VAR KqmlMessage<%message>]  getSendError ( ) )

THEN

(  DO SystemOutPrintln ( [ SFUNC Concat(
      [ VAL String "Unable to Deliver Message To: "  ]
      [ VAR KqmlMessage<%message>.receiver] )
                       ] ) )
( RETRACT ( [ VAR KqmlMessage<%message>]  ) )
)
```

The mental conditions of this rule will be met when a KQML Message's `getSendError` method returns true. When this is the case, the engine will execute the RHS elements of this rule. First, it will print a message indicating that a KQML message was undelivered. Next, it will remove the KQML message from the agent's mental state.

You need to keep in mind that in most cases, the KQML Message send routine will not return in the next cycle of the engine. That is, it could take the engine from 2 to 4 engine cycles to determine that a KQML message is not deliverable.

Agent Engine Cycle

The agent engine operates on a fixed-time-length cycle. During each cycle the agent engine:

- Matches received messages and the agent's current beliefs against the agent's behavioral rules to determine which rules should be activated.
- Executes the actions specified in the activated rules.
- Updates the agent's mental model based on the assertions and retractions specified in the activated rules and the values returned from the executed actions.

If the engine finishes processing before the time allotted for the cycle the engine automatically goes to sleep until the start of the next cycle. Approximately one-half of the engine cycle is intended for pattern matching and updating the mental model, and the other half is intended for executing actions.

On the other hand, if the engine does not finish processing before the end of the allowed time, the cycle simply continues past its allowed time. Currently there is no mechanism in place for warning the user about over-long processing, but later versions will notify the user when the pattern matching or action execution cannot be completed in the allotted time. Overrunning the cycle boundary is not usually a problem, except that it may shift the starting time of a later cycle and delay the execution of a committed action.

Agent Engine Threads

It's expected that separate threads will be used for all long-running actions. Any invocation of a `run()` method will automatically be executed on its own thread. All regular actions, i.e., the actions that will execute on the common execution thread, are expected to execute quickly (in comparison to the cycle length). If long-running actions are executed on the regular execution thread they may cause a delay of several agent cycles.

Thread priorities. Currently the agent engine main thread and regular action execution threads run at priority 4; this value is used because it is 1 less than the Java Event-handler thread. Any separately-threaded actions run at priority 3 so that they will not preempt the engine thread. Currently there is no mechanism that allows the agent programmer to change the priorities of the engine thread or the action threads.

Although separately threaded actions will help avoid delays in the engine cycle, they must be used with care. It's the responsibility of the agent programmer to synchronize any interactions between the separately threaded action and the rest of the agent engine. In particular, control panel PACs must synchronize access to any data structures which are used by methods called from the agent engine and also called from event handler code.

For example, assume that some PAC has a `print(String)` method that prints to a text area. Assume also that the PAC has an event handler for a button that prints a message to the same text area whenever the user clicks on the button. If access to the text area is not synchronized (via synchronized methods or synchronized blocks of code) the output from the agent engine and the output from the PAC may be intermixed. In this example the print method could be executing on the engine's execution thread when the user clicks the button. Then the PAC's event handler code would execute on the event handler thread, interrupting the regular print method. Finally, after the event handler code has finished printing its message into the text area, the remainder of the original print argument would be printed. This synchronization failure would only cause confusing output; in other situations, synchronization failures could cause the program to crash or return incorrect results.

Part IV. Appendices

Appendix A. Intrinsics

Built-in actions:
```
void ConnectAction( String actionName,
            Object sourceObject)

Float GetCycleTime()

String GetHostAddress()

String GetHostName()

String GetSystemProperty( String keyString )

void OpenConsole()

void OpenInputDialog( AgentInfo selfAgentInfo,
                        String dialogName,
                        String promptString,
                        Class expectedType )

void OpenOutputDialog(String outputString)

void RegisterAgent( Agent newAgent )

void RemoveRule( String ruleName)
void SendKqmlMessage( KqmlMessage outputMessage,
                  String receiverName,
                  String performative,
                  Object content,
                  String replyWith,
                  String inReplyTo,
                  String language,
                  String ontology,
                  String protocol,
                  String toAgentName,
                  String fromAgentName)

void SetCycleTime(Float cycleTime)

void ShutdownEngine()

void Sleep( int numberOfSeconds )

void SleepUntilMessage()

void SleepWhenIdle()

void SystemOutPrintln( String outputString )
```

Object comparison operators:
EQUALS
NOT_EQUALS
Boolean operators:
AND
OR
NOT

Binding operator:
BIND

Quantifiers:
FOR_ALL

```
EXISTS
```

Numerical comparison operators:

```
=
!=
<
<=
>
>=
```

String functions:

```
String Concat( String firstString,
            String secondString )

String Substring(String originalString,
            int fromIndex, int toIndex)

String SubstringFrom( String originalString,
            int fromIndex )

Integer Length( String targetString )

String Uppercase( String originalString )

String Lowercase( String originalString )

String Trim( String originalString )

Integer IndexOf( String baseString, String substring )

Integer IndexOfFrom(String baseString,
            String substring, int fromIndex)

Integer LastIndexOf(String baseString,
            String substring )

Integer LastIndexOfFrom( String baseString,
            String substring, int fromIndex )
```

Arithmetic functions:

```
+
-
*
/
```

Arithmetic functions take two Number operands and return a Number result. If the actual types of both operands are the same then the return type will be the same as the operand types. Otherwise the return type will be the larger of the operand types, i.e., Long+Integer returns a Long, Float+Integer returns a Float, Float+Double returns a Double.

Mathematical functions:

```
Double Sqrt( Number )
Double Log( Number )
Double Exp( Number )
Double Random()
Long Round( Number )
Long Mod( Long, Long )
Number Power( Number, Number ) *See note (1) below
Number Abs( Number )           *See note (2) below
Number Max( Number, Number )
Number Min( Number, Number )
Long Floor( Number )
Long Ceiling( Number )
Double Sin( Number )
Double Cos( Number )
Double Tan( Number )
Double ArcSin( Number )
Double ArcCos( Number )
Double ArcTan( Number )
```

```
Integer ConvertToInteger( Number )
Float ConvertToFloat( Number )
String ConvertToString( Number )
```

Other Functions

```
Object ConvertTo( Class desiredClass,
            Object existingObject )
```

(1) The return type for Power, Max, and Min is determined by the operand types. If the types of both operands are the same then the return type will be the same as the operand types. Otherwise the return type will be the larger of the operand types, i.e., Max(Long,Integer) returns a Long, Max(Float,Integer) returns a Float, Max(Float,Double) returns a Double.
(2) The return type for Abs is the same as the operand type.

Appendix B. Runtime Agent Definition Language

```
Parentheses, square brackets and angled brackets are to be interpreted as
literals except for angled brackets surrounding a word, e.g. as in <belief-
templates>.  A line such as <template-type> < <instance-name> > is intended
to represent a name such as Location<currentLocation>, or in the case where
the instance is not named, Location<>.
   NULL means that nothing is printed in the agent definition file.
=========================================================================
<agent-definition> ::= <version-number>  <unstructured-comments>
                       <separator-line>  <abbreviated-names>
                       <separator-line>  <initial-objects>
                       <separator-line>  <action-definitions>
                       <separator-line>  <capabilities>
                       <separator-line>  <belief-templates>
                       <separator-line>  <initial-beliefs>
                       <separator-line>  <initial-agency-beliefs>
                       <separator-line   <intial-commitments>
                       <separator-line   <intial-intentions>
                       <separator-line>  <behavioral-rules>
                       <separator-line>

<version-statement> ::= RUNTIME AGENT DEFINITION BNF VERSION: <version>

<version> ::= <numeral> {.<numeral>}* <unstructured-comments> ::= <string>*

<separator-line> ::=  =========================
The separator line may be any length, composed of the SEPARATOR_LINE_CHAR.
The <unstructured-comments> may be anything.

--------------------------ABBREVIATED NAMES--------------------------
<abbreviated-names> ::= ABBREVIATED NAMES <name-table-entry>*

<name-table-entry> ::= ( <abbreviated-name> <full-pkg-class-name> )
```

```
<abbreviated-name> ::= <string>

<full-pkg-class-name> ::= <string>

--------------------------INITIAL OBJECTS---------------------------
<initial-objects> ::= INITIAL OBJECTS  <initial-object>*

# Note: the CTOR_ARGS keyword is now optional.

<initial-object> ::= ( <pkg-class-name>
                 < <instance-name> > {CTOR_ARGS | NULL} <ctor-arg-spec>* )

#Note: You only use quotes on the String when the type of object is String.

<ctor-arg-spec> ::= ( <ctor-arg-type> <ctor-arg-value>* ) |
                 ( java.lang.String "<string>" )

<ctor-arg-type>   ::= <pkg-class-name>

<ctor-arg-value> ::= <string>|<numeral>|<ctor-arg-spec>| DEFAULT | NULL

<string> ::= <alphanumeric>*

<numeral> ::= <digit>*{.<digit>*}

<instance-name> ::= <string> | NULL

<pkg-class-name> ::= <package-name>.<class-name> | <class-name>

<package-name> ::= {<string>.}*<string> <class-name> ::= <string>

---------------------------ACTION DEFINITIONS----------------------
<action-definitions> ::= ACTION DEFINITIONS  <action-definition>*

<action-definition> ::= ( <action-name> PAC_OBJECT <pkg-class-name>
                 < <pac-object-name> > PAC_METHOD <method-name>
                 ( <method-arg-type>* ) )
                 | ( <action-name> SEPARATE_THREAD
                 PAC_OBJECT <pkg-class-name>< <pac-object-name> > )

<action-name> ::= <string>

<method-name>   ::= <string>

<method-arg-type>   ::= <pkg-class-name>

<pac-object-name> ::= <string> | NULL

---------------------------CAPABILITIES-------------------------------------
<capabilities> ::= CAPABILITIES  <capability>*

<capability> ::= ( <action-name> ( <pattern-variable>* )

PRECONDITIONS ( <lhs-pattern>* ) EFFECTS ( <mental-change>* ) )

 ---------------------------BELIEF TEMPLATES--------------------------------
<belief-templates> ::=  BELIEF TEMPLATES  <belief-template>*
```

```
<belief-template> ::= (<template-type>
                 FIELDS [{<field-type> <<field-name> >}* ]
                 CTOR-ARGS [ <arg-type>* ] )

<template-type> ::= <string>

<field-type> ::= <string>

<arg-type> ::= <string>

<field-name>  ::= <string>

---------------------------INITIAL BELIEFS---------------------------------
<initial-beliefs> ::= INITIAL BELIEFS  <initial-belief>*

<initial-belief> ::= ( <template-type> < <instance-name> >
                 CTOR-ARGS <ctor-arg-spec>* )

 ---------------------------INITIAL COMMITMENTS------------------------------
<initial-commitments> ::= INITIAL COMMITMENTS  <commitment>*

<commitment> ::= (<agent-name> [ <time> ] <action-name> ( <bound-element>* ))

<time> ::= <year>:<month>:<day> <hour>:<minute>:<second> |STARTUP | SHUTDOWN

<year> ::= <numeral>

<month> ::= <numeral>

<day> ::= <numeral>

<hour> ::= <numeral>

<minute> ::= <numeral>

<second> ::= <numeral>
---------------------------BEHAVIORAL RULES---------------------------------
<behavioral-rules> ::= BEHAVIORAL RULES <behavioral-rule>*

<behavioral-rule> ::= ( <rule-name> WHEN  <when-clause>
                 IF  <if-clause>
                 THEN  <then-clause>  )

<rule-name> ::= <quoted-string>

<when-clause> ::= <compound-lhs-pattern>*

<if-clause> ::= <compound-lhs-pattern>*

<then-clause> ::= <action-statement>*
                 { <mental-change> | <temp-var-rhs-pattern> }*

=========================== RULE PATTERN SUPPORT
===============================
# This section breaks up the rule "compound-lhs-patterns"  into more specific
# elements which are needed for correct representation.
```

```
 <compound-lhs-pattern> ::= ( <simple-lhs-pattern> ) |
                 ( <compound-lhs-pattern> AND <compound-lhs-pattern> ) |
                 ( <compound-lhs-pattern> OR <compound-lhs-pattern> ) |
                 ( NOT( <compound-lhs-pattern> ) ) | (<quantified-pattern> )

<simple-lhs-pattern> ::= <object-relation> |
                 <numerical-relation> |
                 <bind-pattern> |
                 <message-pattern> |
                 <temp-var-lhs-pattern>

# Example of new object relation using predicate method with args:
# OBJ [INST Time<currentTime>] during( (Time [VAR Time<?t1>]),
# (Time [VAR Time<?t2>]) )
# The comma in the arg list is optional

<object-relation> ::= OBJ <lhs-pattern-element>
                 <operator-keyword>
                 <lhs-pattern-element>|
                 OBJ <lhs-pattern-element>
                 <predicate-method-name> ( <arg-element-spec>* )

<arg-element-spec> ::= ( <arg-class-name> <lhs-pattern-element>* )

<operator-keyword> ::= EQUALS | NOT EQUALS | ELEMENT-OF | NOT-ELEMENT-OF

 <numerical-relation> ::= NUM <lhs-pattern-element>
                 <rel-op-symbol>
                 <lhs-pattern-element>

<rel-op-symbol> ::=   =  |  !=  |  <  |  <=  |  >  |  >=

<bind-pattern> ::= BIND <pattern-variable> | BIND <named-instance-var>

<temp-var-lhs-pattern> ::= SET_TEMPORARY <pattern-variable> TO
                 <lhs-pattern-element>

<quantified-pattern> ::= <quantifier> <quant-var-list>
                 ( <compound-lhs-pattern> )

<quantifier> ::= FOR_ALL | EXISTS

<quant-var-list> ::= <top-level-pattern-variable>+

<pattern-element> ::= <lhs-pattern-element> | <rhs-pattern-element>

<lhs-pattern-element> := <pattern-variable> |
                 <message-pattern-variable> |
                 <bound-element> |
                 <named-instance-var> |
                 <function>

<rhs-pattern-element> := <lhs-pattern-element> |
                 <return-variable> |
                 <new-object-element>

#The second option is a special case for handling KqmlMessage casting.
#This is because the content field is an object and needs to be cast
```

```
#to its specific type.
<pattern-variable> ::= [ VAR <pkg-class-name>
                 < <var-name> >{.<field-name>}* ] |
                 [VAR ( ( <pkg-class-name> ) KqmlMessage
                 < <var-name> >.content ) {.<field-name>}* ]

<message-pattern-variable> ::= [ VAR <pkg-class-name> < <var-name> >
                 {.<field-name>}* ] | [VAR ( ( <pkg-class-name> )
                 KqmlMessage < <var-name> >.content ) {.<field-name>}* ]

# The <top-level-pattern-var> is a restricted form of pattern variable
# which is used as the target variable in <quantified-pattern> and
# <retraction>

<top-level-pattern-var> ::= [ VAR <pkg-class-name> < <var-name> > ]

<var-name> ::= <string>

<field-name> ::= <string>

<bound-element> ::=  [ VAL  <pkg-class-name>  <value> ] |
                 [ VAL  <pkg-class-name>  <ctor-arg-spec>* ]

<value> ::= "<string>" | <numeral> | NULL

 <named-instance-var> ::= [ INST <pkg-class-name>
                 < <instance-name> > {.<field-name>}* ]

# The <top-level-named-instance-var> is a restricted form of named instance
# variable used as the target variable in <retraction>

<top-level-named-instance-var>::= [INST <pkg-class-name> <<instance-name> >]

<return-variable> ::= [ RVAR <pkg-class-name> < <var-name> > ]

<new-object-element> ::= [ NEW <pkg-class-name>  <ctor-arg-element-spec>* ]

<ctor-arg-element-spec> ::= ( <ctor-arg-class-name> <rhs-pattern-element>* )

<function> ::= <math-function> | <arithmetic-function> | <string-function>

<arithmetic-func> ::= [ AFUNC <pattern-element>
                 <arithmetic-symbol>
                 <pattern-element> ]

<arithmetic-symbol> ::=  +  |  -  |  *  |  /

<math-function> ::= [MFUNC <math-func-keyword> ( <math-func-operand-list> ) ]

<math-func-keyword> ::= Sqrt | Log | Exp | Random | Round | Mod | Power |
                 Abs | Max | Min | Floor | Ceiling | Sin | Cos | Tan |
                 ArcSin | ArcCos | ArcTan | ConvertToInteger |
                 ConvertToFloat

<math-func-operand-list> ::= NULL | <pattern-element> |
                 <pattern-element> <pattern-element>

<string-function> ::= [ SFUNC <string-func-keyword>
```

```
                 ( <string-func-operand-list> ) ]

<string-func-keyword> ::= Concat | Substring | Length | Uppercase |
                 Lowercase | Trim | IndexOf | LastIndexOf

<string-func-operand-list> ::= <pattern-element> |
                 <pattern-element> <pattern-element> |
                 <pattern-element> <pattern-element> <pattern-element>

==========================================================================
#This section lays out the agent's beliefs about its own agency and
#all of the other agents it knows about.  It does this in a general
#intial beliefs manner (i.e. it uses the format for an initial belief)
#It will look like an initial belief to the agent engine parser.
#The <comm-info> for the self agent cannot be null.

 <initial-agency-beliefs>    ::= INITIAL AGENCY BELIEFS
                 <self-spec> { <agency-tool-spec> | NULL }
                 { <debugger-spec> | NULL }
                 { <remote-agents-spec> | NULL } | NULL

<self-spec>        ::= ( SELF <agent-name>  [ <agent-address> ]
                 { [<comm-info>] }+ [ <keys> ]  [ <agencies> ] )

<agency-tool-spec> ::= ( AGENCY_TOOL  <remote-agent> )

<debugger-spec> ::=  ( DEBUGGER  <remote-agent> )

<remote-agents-spec> ::= REMOTE_AGENTS { ( <remote-agent> ) }*

<remote-agent>    ::= <agent-name>  [ <agent-address> ]
                 [<comm-info>] [ <keys> ]  [ <agencies> ]

<agent>           ::= <String>  (quoted string)

<agencies>        ::= <agency> { , <agency> } *

<agency>          ::= <String>  (quoted string)

<comm-info>       ::= <rmi> | <socket> | NULL

<rmi>             ::= RMI : rmiRegistry-port-number

<socket>          ::= SOCKET  : socket-port-number : full-pkg-class-name

<agent-address>   ::= <IPaddress> | CURRENT_IP_ADDRESS

<IPAddress>       ::= address  (number or name)

<keys>            ::= <public-key>  <private-key> | NULL

<public-key>      ::= byte[]

<private-key>     ::= byte[]

==========================================================================
#RHS support
<rhs-pattern> ::= ( <action-statement> ) |
                 ( <mental-change> ) |
                 ( <temp-var-rhs-pattern> )

 <action-statement> ::= DO <action-name> ( <rhs-pattern-element>
                 { , <rhs-pattern-element> }* )|
                 DO <return-variable> = <action-name>
                 ( <rhs-pattern-element> { , <rhs-pattern-element> }* ) |
                 DO SendKqmlMessage( <mssg-pattern-element>
                 <sender-pattern-element>
                 <receiver-pattern-element>
                 <performative-pattern-element>
                 <reply-with-pattern-element>
                 <in-reply-to-pattern-element>
                 <to-pattern-element>
                 <from-pattern-element>
                 <language-pattern-element>
                 <ontology-pattern-element>
                 <content-pattern-element> )|
                 DO <target-element> <method-name>
                 ( <ctor-arg-element-spec>
                 { , <ctor-arg-element-spec> }* |
                 NULL ) # For the "run" method |
                 DO <target-element> run ( ) SEPARATE_THREAD |
                 DO <return-variable> = <target-element>
                                  <method-name> ( <ctor-arg-element-spec>
                                  { , <ctor-arg-element-spec> }* | NULL )

<target-element> ::= <pattern-variable> | <named-instance-var>

<method-name>  ::= <String>

<mssg-pattern-element> ::= <rhs-pattern-element>

<sender-pattern-element>       ::= SENDER <rhs-pattern-element> | []

<receiver-pattern-element>     ::= RECEIVER <rhs-pattern-element> | []

<performative-pattern-element> ::= PERFORMATIVE <rhs-pattern-element> | []

<reply-with-pattern-element>   ::= REPLY_WITH <rhs-pattern-element> | []

<in-reply-to-pattern-element>  ::= IN_REPLY_TO <rhs-pattern-element> | []

<to-pattern-element>           ::= TO <rhs-pattern-element> | []

<from-pattern-element>         ::= FROM <rhs-pattern-element> | []

<language-pattern-element>     ::= LANGUAGE <rhs-pattern-element> | []

<ontology-pattern-element>     ::= ONTOLOGY <rhs-pattern-element> | []

<content-pattern-element>      ::= CONTENT <rhs-pattern-element> | []

<mental-change> ::= <assertion> | <retraction>

<assertion> ::= ASSERT ( <instance-name> <object-pattern> ) |
                 ASSERT ( <replacement> )
```

```
<instance-name> ::= <bound-element> | NULL

<object-pattern> ::= <pattern-variable> | <return-variable> |
                 <new-object-element> | <bound-element> |
                 <named-instance-var>

<replacement> ::= SET_VALUE_OF <pattern-variable> TO <rhs-pattern-element>|
                 SET_VALUE_OF <named-instance-var> TO <rhs-pattern-element>

# A retraction target will be either a <top-level-pattern-variable> or a
# <top-level-named-instance-var>.  These are restricted forms of pattern
# vars or named instance vars which cannot have any subobjects specified.

<retraction> ::= RETRACT ( <top-level-pattern-variable> ) |
                 RETRACT ( <top-level-named-instance-var> )

<temp-var-rhs-pattern> ::= SET_TEMPORARY <pattern-variable>
                 TO <rhs-pattern-element>
--------------------------------------------------------------------------
```

Appendix C. Operators and Patterns

NOTE: In the following sections describing operators and pattern types, some simple examples are used to illustrate the behavior of patterns. Many of these examples show the agent's mental model consisting of several `Integer` and/or `String` objects, and the agent's rules contain patterns that match agains `Integers` and/or `Strings`. These data types were chosen to make the examples as simple as possible. Although it's possible to use `Integer` and `String` objects as beliefs in an agent's mental model, most non-trivial agents use PAC instances to store their knowledge of the world.

The discussions in the sections below do not take *refraction* into account. Refraction is the term used in rule-based systems to describe the suppression of a response to repeated stimuli. This term is borrowed from neuroscience, where it refers to a neuron's response to repeated stimuli. In the AgentBuilder run-time system, a rule will be activated only if all of its left-hand-side patterns are satisfied, and at least one of the patterns matches against a *new* belief. Rule activation will be suppressed if all patterns match against old beliefs.

Some examples show rule *definitions*, which contain the generic patterns, and rule *activations* that result from pattern matching the rules against the mental model. The rule definitions show the variables in the patterns, e.g., `?i` represents a variable of type Integer with name `?i`, and `?s` represents a `String` variable. The rule activations show the *values* bound to the variables in each activation, with the given mental model. Only the bindings that lead to an activation of the rule are shown.

BIND

A `BIND` pattern is the simplest type of pattern. A `BIND` pattern will evaluate to true if there exists a binding for the variable in the pattern. For example, assume that `?s` is a variable of type `String`. The pattern `( BIND ?s )` will evaluate to `true`, and will bind a value to the variable `?s`, for every `String` object in the mental model. Consider the example shown in Figure 68:

```
MENTAL MODEL:
String<s1> "foo bar"
String<s2> "yada yada yada"

RULE: Bind and Print
IF
( BIND ?s )
THEN
DO SystemOutPrintln( ?s )
```

Figure 68. Bind Example

This rule, with the given mental model of two new `String` beliefs, will fire twice in the same cycle. The pattern `( BIND ?s )` will evaluate to true twice, and each time it will bind the variable `?s` to one of the `String` values in the mental model. The first time the rule fires it will print the string `foo bar`, the second time it will print `yada yada yada`.

Classes and Subclasses

Variables will bind to objects which are of the class defined for the variable, or any subclass (or subclass of a subclass, etc.) of the defined class. Consider, for example, a hierarchy of classes that represent various shapes in a graphical application. Assume that the `Square` class is a subclass of the `Rectangle` class, and the `Rect-angle` class defines and implements a method `setWidth(int)`. The `Square` class implements its own version of `setWidth(int)`. The `Square setWidth` overrides the `Rectangle setWidth` and sets both the length and the width to the specified value, to maintain squareness.

In the example shown below, assume that the `?rectangle` variable is defined to bind to objects of class `Rectangle`. The `Set Widths` rule will fire twice, once with `?rectangle` bound to the `Square` instance, once with `?rectangle` bound to the `Rectangle` instance. When `?rectangle` is bound to the `Square s1` the `Square` method `setWidth(int)` will be used to perform the specified action. When `?rectangle` is bound to the Rectangle r1 the Rectangle method setWidth(int) will be used.

MENTAL MODEL: `Square<s1> Rectangle<r1>`

RULE: `Set Widths IF (BIND ?rectangle) THEN DO ?rectan-gle.setWidth(42)`

See the `BIND` patterns vs. `EXISTS` patterns section below for an explanation of `BIND` patterns in some unusual contexts.

EQUALS/NOT_EQUALS

The `EQUALS` operator is used to perform an equality comparison between two objects of the same class, using the equals method defined in the class of the objects. For example, consider a pattern that compares a `String` variable and a `String` literal: `( ?string EQUALS "yada yada yada" )`. When this pattern gets evaluated the agent engine will find all possible bindings for the `?string` variable then invoke the `String::equals(String)` method on each binding, using the `String` literal `yada yada yada` as the argument.

The NOT_EQUALS operator is used to perform an inequality comparison between two objects of the same class, using the equals method defined in the class of the objects but with a false expected value.

NUMERICAL RELATIONS

The numerical relation symbols `=, !=, <, <=, >`, and `>=` can only be used between numerical operands. The numerical operands can be any numerical objects or primitive values, and the types of the two operands do not need to be the same. For the following examples assume that `?i` is an `Integer` variable and `?f` is a `Float` variable. The following patterns are all correct patterns: `(?i = 23)`, `(23.4<= ?f)`, and `(?i = ?f)`. Note that these example patterns would not be used together in the same rule, because they are mutually exclusive conditions. The last example shows a comparison between an `Integer` and a `Float`, which will evaluate to `true` if both objects contain the same value.

It's also possible to compare the values in numerical objects (e.g., `Integers` or `Floats`) to primitive values extracted from objects in the mental state. For example, assume a `PriceQuote` PAC has an attribute named `quantity` which is an `int` value, assume `?priceQuote` is a variable of type `PriceQuote`, and assume `currentQuan-`

`tity` is a named instance of `Integer`. Then `( ?priceQuote.quantity <= currentQuantity )` is a valid numerical pattern, even though it compares an `int` to an `Integer`. Numerical patterns may contain any mixture of numerical objects or numerical primitive values; the values are automatically extracted from the objects and used in the comparison. In numerical relations there's no need to invoke methods `(intValue(), floatValue(), etc.)` to extract values from an object before the values are used in a comparison.

It's possible to compare numerical objects with the `EQUALS` or `NOT_EQUALS` operator but the types of both objects must agree. If `?i` is an `Integer` variable and `23` is an `Integer` value, then the pattern `( ?i EQUALS 23 )` is a valid pattern which will have the same truth value as the numerical relation pattern `( ?i = 23 )`. It's **not** possible to mix object types in a pattern with an `EQUALS` operator, so if `23.0` is a `Float` value then a pattern such as `( ?i EQUALS 23.0 )` is **not** a valid pattern. For comparisons of numerical objects the numerical relation patterns (i.e., using `=` or `!=`) are recommended.

NOTE: In a LHS pattern the equal sign, `=`, represents an equality comparison between numerical objects or values. In a LHS pattern it does **not** represent assignment of a value to a variable, as it would in a programming language such as C or Java. The LHS pattern `( ?i = 23 )` compares the bindings for `?i` to the integer value `23`. It does **not** assign a value of `23` to the variable `?i`. The pattern **binds** to the value `23` if and only if there is an `Integer` in the mental model with the value `23`.

ARITHMETIC OPERATORS

The arithmetic operators `+, -, *,` and `/` require numerical operands. The types of the operands may differ, as long as both are numeric types. If the types of both operands are the same then the type of the result will be the same as the operand types. Otherwise the result type will automatically be the larger of the operand types. For example `Long+Integer` returns a `Long`, `Float+Integer` returns a `Float`, `Float+Double` returns a `Double`.

If the operation is division and both operands are integral types (`Integer, Short, Long,` or `Byte`, or their primitive counterparts) then integer division will be used (e.g., `5 / 2` yields `2`). If the operation is division and either or both operands are non-integral types then floating-point division will be used (e.g., `5.0 / 2` yields `2.5`).

AND, OR, NOT

The boolean operators `AND`, `OR`, and `NOT` can be applied to any patterns. Use of the `AND` operator is fairly straightforward: an `AND` pattern is true if and only if both of its subpatterns are true. The `NOT` operator also behaves as expected: a `NOT` pattern is true if and only if its subpattern is false. (See the `BIND` patterns vs. `EXISTS` patterns section below for a description of a pattern where `NOT` may perform differently than expected.)

OR patterns

Rules with `OR` patterns may fire less often or more often than expected and should be used with caution. The following examples illustrate the behavior that can be expected from `OR` patterns.

Example 1

The first example shows a mental model consisting only of several `Integer` objects with instance names (e.g., `i1`) which are not used here. Rule 1 will fire once for each `Integer` object in the belief base which has a value less than `2` and greater than or equal to `4`. This is an example of an `OR` pattern which behaves as expected. See Figure 69.

```
MENTAL MODEL:
Integer<i1> 1
Integer<i2> 2
Integer<i3> 3
Integer<i4> 4

RULE: Rule 1
IF
( ( ?i < 2 ) OR ( ?i >= 4 ) )
THEN
ASSERT String<> Rule 1 fired

RULE ACTIVATIONS:
RULE: Rule 1
IF ( ( 1 < 2 ) OR ( 1 >= 4 ) )
THEN
ASSERT String<> Rule 1 fired

RULE: Rule 1
IF
( ( 4 < 2 ) OR ( 4 >= 4 ) )
THEN
ASSERT String<> Rule 1 fired
```

Figure 69. Example Use of OR

As expected, Rule 1 fires twice because there are two bindings which satisfy the pattern `((?i < 2) OR (?i >= 4))`. The two successful bindings are shown in the list of rule activations; the unsuccessful bindings (i.e., the bindings to 2 and 3) caused the evaluation to fail and so they do not appear in the list of activations.

Example 2

This example shows the unusual behavior that is possible in `OR` patterns when different variables are used in the clauses. Given the mental model shown below, Rule 2 fires twice even though there is only one Integer that satisfies the first clause of the OR pattern. See Figure 70 for an example.

```
MENTAL MODEL:
Integer<i1> 1
Integer<i2> 2
String<s1> "abc"
String<s2> xyz

RULE: Rule 2
IF
( ( ?i = 2 ) OR ( ?s EQUALS yada ) )
THEN
ASSERT String<> Rule 2 fired

 RULE ACTIVATIONS:
RULE: Rule 2
IF
( ( 2 = 2 ) OR ( "abc" EQUALS yada ) )
THEN
ASSERT String<> Rule 2 fired

RULE: Rule 2
IF ( ( 2 = 2 ) OR ( "xyz" EQUALS yada ) )
THEN
ASSERT String<> Rule 2 fired
```

Figure 70. Another Example of OR Usage

The reason for the two firings is that there are two sets of bindings that satisfy the `OR` pattern in Rule 2. The pattern is satisfied with `?i` bound to `2` and `?s` bound to `abc`, and the pattern is satisfied with `?i` bound to `2` and `?s` bound to `xyz`.

Example 3

This example shows the behavior that is possible in `OR` patterns when the clauses are not mutually exclusive. Given the mental model shown in Figure 71, Rule 3 fires THREE times.

```
MENTAL MODEL:
Integer<i1> 1
Integer<i2> 2
String<s1> abc
String<s2> xyz

 RULE: Rule 3
IF
( ( ?i = 2 ) OR ( ?s EQUALS "abc"  )
THEN
ASSERT String<> Rule 3 fired

 RULE ACTIVATIONS:
RULE: Rule 3
IF
( ( 1 = 2 ) OR ( "abc" EQUALS "abc" ) )
THEN
ASSERT String<> Rule 3 fired

RULE: Rule 3
IF
( ( 2 = 2 ) OR ( "abc" EQUALS "abc" ) )
THEN
ASSERT String<> Rule 3 fired

RULE: Rule 3
IF
( ( 2 = 2 ) OR ( "xyz" EQUALS "abc" ) )
THEN
ASSERT String<> Rule 3 fired
```

Figure 71. Another Example Use of OR

The `OR` pattern gets tested with each possible pair of bindings, i.e., with the ordered pairs `(1,abc)`, `(2,abc)`, `(1,xyz)`, and `(2,xyz)`. Three of these pairs cause the `OR` pattern to evaluate to true, so the rule is activated three times. Only the `(1,xyz)` pair causes both clauses of the `OR` pattern to be false, so it doesn't appear in the list of activations.

QUANTIFIED PATTERNS

Quantified patterns are patterns containing one or more quantified variables, which are marked with the `EXISTS` or `FOR_ALL` keywords. The `EXISTS` keyword marks a variable as existentially quantified; the `FOR_ALL` keyword marks a variable as universally quantified. NOTE: The scope of quantified variables is restricted to the quantified pattern. A quantified variable cannot be used in more than one pattern in a rule. The following example in Figure 72 shows an **incorrect** usage:

```
RULE: Incorrect Example
IF
( FOR_ALL ?i ( ?i <= 4 ) )
(?i >= 0 )
THEN ...
```

Figure 72. An Incorrect Example

In this incorrect example the same variable is used in a quantified pattern (the first pattern) and in the second pattern. This is disallowed because the scope of the quantified variable `?i` is restricted to the quantified pattern. Using the same variable elsewhere in the rule implies a connection between the two usages of `?i`, but there is no connection because of the restricted scope of the quantified variable.

A `FOR_ALL` pattern will activate a rule once if all bindings for the quantified variable(s) satisfy its subpattern. The behavior of this type of pattern is fairly straightforward in most usages, but special care must be taken when mixing quantified variables with non-quantified variables.

An `EXISTS` pattern will activate a rule once if there are one or more bindings for the quantified variable(s) which satisfy the subpattern in the `EXISTS` pattern. The behavior of this type of pattern is fairly straightforward in most usages, but special care must be taken when mixing quantified variables with non-quantified variables. The examples below show the behavior of several types of `EXISTS` patterns.

All examples use the following simplified example mental model of Figure 73, which consists of several Integer objects and String objects, some of which are named instances.

```
Mental Model:
Integer<> 1
Integer<> 2
String<s1> "abc"
String<s2> "abc"
```

Figure 73. Example Mental Model

Rule 1 contains a regular object pattern (i.e., not a quantified pattern). The rule will be activated once for every Integer object in the belief base which is greater than or equal to `0`, so Rule 1 will be activated twice. See Figure 74.

In Rule 2, the object pattern from Rule 1 is now a subpattern in an `EXISTS` pattern. Rule 2 will be activated only once, even though there are two Integer objects in the mental model which satisfy the test in the subpattern. The variable `?i` is a *`quantified`* variable so it doesn't matter how many bindings will satisfy the test in the subpat-

```
RULE: Rule 1
IF
( ?i >= 0 )
THEN ...
```

Figure 74. Example Rule 1

tern, as long as there is at least one. The mental model could contain hundreds of Integer objects with non-negative values and Rule 2 would still fire only once. Rule 2 is shown in .

```
RULE: Rule 2
IF
( EXISTS ?i ( ?i >= 0 ) )
THEN ...
```

Figure 75. Example Rule 2

Rules 3 and 4 are examples of badly-written rules. In each case the quantified variable (i.e., the Float variable `?f` in Rule 3, and the `String` variable `?s` in Rule 4) doesn't match the variable in the subpattern. The pattern in Rule 3 asks the question:

Does there exist a `Float` object in the mental model such that there is a `String` object in the mental model equal to `abc`? See Rule 3 in Figure 76.

```
RULE: Rule 3 - BAD PATTERN
IF ( EXISTS ?f ( ?s EQUALS "abc" ) )
THEN ...
```

Figure 76. Example Rule 3

This pattern is not satisfied by any combination of objects in the mental model, because there are no `Float` objects, so Rule 3 will not fire at all. The fact that the subpattern is satisfied (twice) doesn't matter--evaluation is halted due to the lack of a binding for the `Float` variable.

The pattern in Rule 4 asks the question:

Does there exist a `String` object in the mental model such that there is an `Integer` object in the mental model with a non-negative value?

```
RULE: Rule 4 - BAD PATTERN
IF ( EXISTS ?s ( ?i >= 0 ) )
THEN ...

RULE: Rule 4b
IF ( EXISTS ?s (( ?s EQUALS "abc" ) AND ( ?i >= 0 )))
THEN ...

RULE: Rule 4c
IF
( ( ?s EQUALS "abc" ) AND ( ?i > 0 ) )
THEN ...
```

Figure 77. Rule 4 with a Bad Pattern

The pattern in Rule 4 in Figure 77 also has a mismatch between the quantified variable and the variable in the subpattern, but Rule 4 will fire twice. There is at least one binding for a `String` variable (even though the value is not tested at all) and there are two `Integer` bindings that satisfy the test in the subpattern, so the rule is fired twice. Rules 4b and 4c are shown for comparison. The pattern in Rule 4b is equivalent to the pattern in Rule 4, for *this* mental model. (In general they are not equivalent). For this example mental model, Rule 4b will fire twice and Rule 4c will fire four times. These rules are shown here for comparison with Rule 4, to show why Rule 4 fires twice given the example belief base.

There are situations where rules such as Rule 4b or Rule 4c are used in correct programs. It's possible and sometimes useful to mix quantified variables and non-quantified variables in subpatterns, as was done in Rule 4b. The pattern in Rule 5 asks the question:

Does there exist (at least one) `PriceQuote` object for every `Product` object in the mental model?

```
RULE: Rule 5
IF
(EXISTS PriceQuote<?q>
   (Product<?p>.Name EQUALS PriceQuote<?q>.ProductName))
THEN ...
```

Figure 78. Example Rule 5

This rule will fire once for every `Product` in the mental model which has at least one `PriceQuote` associated with it (based on a comparison of the names in each object). In this pattern the `Product` variable `?p` is not quantified, the `PriceQuote` variable `?q` is quantified. This type of pattern would be used in situations where you want to fire the rule if there is at least one `PriceQuote` for a `Product`, but you don't want the rule to fire more than once if there are multiple `PriceQuotes` associated with the same `Product`.

Vacuously True Quantified Patterns

Some quantified patterns may evaluate to true in situations where there are no matching beliefs. For example, consider the following mental model and the two example rules shown in Figure 79:

Rule 1 asks the question: *Is it true that every `Float` in the mental model has a value less than `3`?*

The answer is yes because there are no `Float` objects in the belief base, hence there are no `Float` objects that could fail the test in the

```
Mental Model:
Integer<> 4
Integer<> 5

RULE: Rule 1
IF ( FOR_ALL ?f ( ?f < 3 ) )
THEN ...

RULE: Rule 2
IF
( NOT ( EXISTS ?f ( ?f >= 3 ) ) )
THEN ...
```

Figure 79. Vacuously True Quantified Patterns

subpattern. In logic this kind of pattern is labeled "vacuously true", because the truth value of the pattern depends on the fact that there are no elements to test. In the AgentBuilder run-time system the pattern will evaluate to true (in this particular example), but the activation of the rule will be suppressed because of refraction.

Rule 2 is equivalent to Rule 1; Rule 2 asks the question: *Is it true that there does not exist a `Float` in the mental model with a value greater than or equal to `3`?*

This is of course true because there are no `Floats` in the mental model, hence there are no `Floats` with value `>= 3`. The pattern in Rule 2 also evaluates to true (in this particular example) but the activation of the rule is also suppressed because of refraction.

To ensure that the the rules get activated, modify rules 1 and 2 by adding a pattern that will always bind to a new belief. Rules 1b and 2b show the modified rules 1 and 2, with the added `BIND` pattern. The added pattern binds the built-in `currentTime` instance which is maintained by the run-time system and which is guaranteed to be a new belief each cycle. Therefore the activation of rules 1b and 2b will not be suppressed by refraction. The rules will be activated whenever their first patterns evaluate to true. See Figure 80.

```
RULE: Rule 1b
IF
( FOR_ALL ?f ( ?f < 3 ) )
( BIND currentTime )
THEN ...

RULE: Rule 2b
IF
( NOT ( EXISTS ?f ( ?f >= 3 ) ) )
( BIND currentTime )
THEN ...
```

Figure 80. Ensuring Rule Activation

BIND Patterns and the EXISTS Patterns

A `BIND` pattern will contribute to the activation of a rule once per new matching belief object. An `EXISTS` pattern will contribute to a single activation if there is at least one new matching belief object. The example shown in will make this more clear. In this first example assume all the `Integer` objects have been stored into the mental model during the last cycle, so we don't need to worry about refraction (yet).

Example 1

Rule 1 shows a `BIND` pattern with an `Integer` variable. This the simplest type of pattern. A `BIND` pattern is satisfied if there exists a binding for its variable. Given the mental model containing two Integer objects, Rule 1 will fire three times, once for each Integer binding.

Rule 2 shows an `EXISTS` pattern enclosing a `BIND` pattern. Given the same mental model Rule 2 will only fire once. Rule 2 will fire once

```
BELIEFS
Integer<> 23
Integer<> 34
Integer<> 45

RULE: Rule 1
IF ( BIND ?i )
THEN ...

RULE: Rule 2
IF ( EXISTS ?i ( BIND ?i )
THEN ...
```

Figure 81. Example 1. More on Binding and Existentials

regardless of the number of `Integer` objects in the mental model, as long as there is at least one new `Integer` object.

Testing for the absence of an object involves a problem due to refraction. The activation of a rule will be suppressed if there is not at least one matching new belief used in at least one of the patterns. A simple rule such as Rule 4 will never be activated because the only time the pattern is satisfied is when there are no matching objects, hence there are no new matching objects to satisfy the refraction test. Rule 5 shows the recommended way to test for the absence of an object. The `(BIND currentTime)` pattern will always be satisfied because the `currentTime` maned instance is a built-in object in the mental model. And the `currentTime` instance is always a new belief every cycle, so any rule containing a pattern such as `(BIND currentTime)` will never have its activation suppressed due to refraction. See Figure 82.

```
RRULE: Rule 3 - Will never be activated
IF
(NOT (BIND ?i))
THEN ...

 RULE: Rule 4 - correct
IF
(NOT (BIND ?i))
(BIND currentTime)
THEN ...
```

Figure 82. Rule 5 - the Final Example

Appendix D. Default Ontology Object Model (Printable)

```
Class Name:
com.reticular.agentBuilder.agent.perception.RmiCommInfo
Description: The Supporting class for the Pac Comm System.
This contains all of the necessary information
for implementing RMI style communication.
Package: com.reticular.agentBuilder.agent.perception
Attributes:
   int rmiRegistryPort;
Methods:
   public Object clone()
   public  RmiCommInfo( int rmiPortNumber )
   public  RmiCommInfo()
   public String getCommType()
   public String toBNFString()
   public boolean equals( Object param0 )
   public int getRmiRegistryPort()
   public void setRmiRegistryPort( int rmiPortNumber )
```

```
Class Name:
com.reticular.agentBuilder.agent.mentalState.Time
Description: The Supporting class for time in the sytem.
The string and number integers are for
getting at the time representation.
Package: com.reticular.agentBuilder.agent.mentalState
Attributes:
   Long number;
   String string;
Methods:
   public Long getNumber()
   public Object clone()
   public String getString()
   public String toString()
   public  Time( int year ,  int month ,  int day ,  int
hour ,  int minute ,  int second )
   public  Time()
   public boolean after(
com.reticular.agentBuilder.agent.mentalState.Time param0 )
   public boolean before(
com.reticular.agentBuilder.agent.mentalState.Time param0 )
   public boolean equals( Object param0 )
```

```
Class Name:
com.reticular.agentBuilder.agent.mentalState.Agent
Description: This class contains the information that an
agent
"believes" about itself.  An instance of this
class exists for each agent known.  The instance
about oneself is called the "SELF" instance.
Package: com.reticular.agentBuilder.agent.mentalState
Attributes:
   String userName;
   com.reticular.agentBuilder.agent.perception.AgentInfo
agentInfo;
Methods:
   public Agent(
com.reticular.agentBuilder.agent.perception.AgentInfo
agentInfo )
   public Agent(
com.reticular.agentBuilder.agent.perception.AgentInfo
agentInfo ,  String[] agencies )
   public Object clone()
   public String getCommType()
   public String getIPAddress()
   public String getName()
   public String getUserName()
   public String toBNFString()
   public String toString()
   public boolean equals( Object param0 )
   public
com.reticular.agentBuilder.agent.perception.AgentInfo
getAgentInfo()
   public void print()
   public void setAgentInfo(
com.reticular.agentBuilder.agent.perception.AgentInfo param0
)
   public void setIPAddress( String param0 )
   public void setName( String param0 )
   public void setUserName( String param0 )
```

```
Class Name:
com.reticular.agentBuilder.agent.perception.KqmlMessage
Description: This class represents the KQML container
for the system.  All KQML messages
are represented as KqmlMessage class
instantiations.
Package: com.reticular.agentBuilder.agent.perception
Attributes:
   String from;
   String inReplyTo;
   String language;
   String ontology;
   String performative;
   String protocol;
   String receiver;
   String replyWith;
   String sender;
   String senderURL;
   String sentTime;
   String to;
   Class contentType;
   Object content;
Methods:
   public Class getContentType()
   public  KqmlMessage()
   public Object clone()
   public Object getContent()
   public String getFrom()
   public String getInReplyTo()
   public String getLanguage()
   public String getOntology()
   public String getPerformative()
   public String getProtocol()
   public String getReceiver()
   public String getReceiverName()
   public String getReplyWith()
   public String getSender()
   public String getSenderName()
   public String getSenderURL()
   public String getSentTime()
   public String getTo()
   public String toString()
   public boolean equals( Object param0 )
   public void setContent( Object param0 )
   public void setContentType( Class param0 )
   public void setFrom( String param0 )
   public void setInReplyTo( String param0 )
   public void setLanguage( String param0 )
   public void setOntology( String param0 )
   public void setPerformative( String param0 )
   public void setProtocol( String param0 )
   public void setReceiver( String param0 )
   public void setReceiverName( String param0 )
   public void setReplyWith( String param0 )
   public void setSender( String param0 )
   public void setSenderName( String param0 )
   public void setSenderURL( String param0 )
   public void setSentTime( String param0 )
   public void setTo( String param0 )
```

```
Class Name:
com.reticular.agentBuilder.agent.perception.AgentInfo
Description: The Supporting class for the Pac Comm System.
It contains the information needed for communication.
Package: com.reticular.agentBuilder.agent.perception
Attributes:
   String IPAddress;
   String name;
   com.reticular.agentBuilder.agent.perception.RmiCommInfo
rmiCommInfo;
Methods:
   public AgentInfo( String agentName ,  String IPAddress ,
com.reticular.agentBuilder.agent.perception.RmiCommInfo
rmiCommInfo )
   public  AgentInfo()
   public Object clone()
   public String getIPAddress()
   public String getName()
   public String toBNFString()
   public String toString()
   public boolean equals( Object param0 )
   public
com.reticular.agentBuilder.agent.perception.RmiCommInfo
getRmiCommInfo()
   public void setRmiCommInfo(
com.reticular.agentBuilder.agent.perception.RmiCommInfo
param0 )
```

```
Class Name:
com.reticular.agentBuilder.agent.perception.PacCommSystem
Description: The Support for the Pac Communication.
Package: com.reticular.agentBuilder.agent.perception
Attributes:
Methods:
   public Object clone()
   public  PacCommSystem(
com.reticular.agentBuilder.agent.perception.AgentInfo
agentInfo ,  String pacName )
   public void sendKqmlMessageToAgent(
com.reticular.agentBuilder.agent.perception.KqmlMessage
param0 )
```

Appendix E. Agent Description (Printable)

```
Agent-name: HelloWorld
Creation-time: Wed Aug 26 13:30:41 PDT 1998
Description: Hello World agent with a full GUI and
demonstrating
connections between the agent and the user interface.
Location:
Author: sonny
Vendor: Acronymics, Inc.
Ontologies: [Quick Tour Ontology]
Agencies: [Hello World Agency]

Actions:
Name:Start  based on:HelloWorldFrame::run
Name:Print  based on:HelloWorldFrame::print

Commitments:

JAVA Instances:

PACs:
com.reticular.agents.helloWorld.HelloWorldFrame
com.reticular.agentBuilder.agent.perception.PacCommSystem
com.reticular.agentBuilder.agent.perception.KqmlMessage
com.reticular.agentBuilder.agent.mentalState.Time
com.reticular.agentBuilder.agent.perception.RmiCommInfo
com.reticular.agentBuilder.agent.perception.AgentInfo
com.reticular.agentBuilder.agent.mentalState.Agent

PAC Instances:
Name:currentTime  Type:Time Initial Pac:False
Name:SELF  Type:Agent Initial Pac:False
Name:startupTime  Type:Time Initial Pac:False
Name:myHelloWorldFrame  Type:HelloWorldFrame Initial
Pac:False

Rules:
Name: Print Greeting
Description: Activated by a message from the
myHelloWorldFrame PAC.
It calls the "Print" action which in turn fires an action
to write
out a string to the interface.
( %incomingMessage.sender EQUALS  "HelloWorld:PAC"  )
( %incomingMessage.performative EQUALS  "achieve"  )
( %incomingMessage.contentType EQUALS String  )
( %incomingMessage.content EQUALS  "Say Hello"  )
DO Print ( Concat (  "HelloWorld! the time is: " ,
currentTime.string ) )
DO SleepUntilMessage (  )

Name: Build HelloWorldFrame
Description: Activated by the agent belief instance
"SELF".  The SELF belief is
automatically created by the agent engine at startup.
The RHS causes the myHelloWorldFrame instance to be
created.
( BIND startupTime )
ASSERT(  "myHelloWorldFrame"  HelloWorldFrame (
PacCommSystem ( SELF.agentInfo,  "HelloWorld:PAC"  ) ))

Name: Quit
Description: Activated by a message from the
myHelloWorldFrame PAC.
```

```
It calls the built in action "shutdownEngine".
( %incomingMessage.sender EQUALS  "HelloWorld:PAC"  )
( %incomingMessage.performative EQUALS  "achieve"  )
( %incomingMessage.contentType EQUALS String  )
( %incomingMessage.content EQUALS  "Quit"  )
DO ShutdownEngine (  )

Name: Launch Interface
Description: Activates after the creation of the
myHelloWorldFrame instance.
It connects the appropriate actions to the appropriate
methods,
and launches the interface onto a separate thread.  The
rule then
executes the sleepUntilMessage action which causes the
agent to sleep.
( BIND myHelloWorldFrame )
DO ConnectAction (  "Print" , myHelloWorldFrame )
DO ConnectAction (  "Start" , myHelloWorldFrame )
DO Start (  )
DO RemoveRule (  "Build HelloWorldFrame"  )
DO RemoveRule (  "Launch Interface"  )
DO SleepUntilMessage (  )
```

Index

Index

Index

Index

Index

Index

Index

Index