AGENTBUILDER

An Integrated Toolkit for
Constructing Intelligent
Software Agents

User's Guide

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Introduction

Chapter 1

Introduction

Chapter Overview

You can find the following information in this chapter:

- AgentBuilder Features
- AgentBuilder Development Tools
- RunTime System
- Technical Support
- System Requirements
- Using This Guide
- If You Never Read Manuals...
- Installation Instructions
- Product Family
- Frequently Asked Questions (FAQ)

A. AgentBuilder Features

AgentBuilder:

- Makes it easy to create intelligent software agents
- Requires no special expertise in intelligent agent technology or network communications
- Constructs agents with built-in capabilities for autonomous operation, monitoring their environments, reasoning, and communicating with other agents and users
- Provides a suite of graphical programming tools for specifying agent behavior and operation
- Utilizes a high-level, agent-oriented programming language; programming is accomplished by specifying intuitive concepts such as beliefs, commitments, and actions
- Provides tools for analyzing the problem domain
- Provides tools for defining agencies—collections of intelligent agents
- Provides tools for defining interaction between agents; i.e., agent protocols
- Provides tools for testing and debugging agents and agencies
- Is a Java-based, cross-platform toolkit for creating cross-platform, agent-based applications; creates agents that are Java programs
- Supports easy integration and use of existing software libraries (Java, C and C++)

B. AgentBuilder Development Tools

AgentBuilder provides graphical tools for supporting all phases of the agent construction process. Programming software agents (sometimes called Agent-Oriented Programming) is accomplished by specifying intuitive concepts such as the beliefs, commitments, and actions of the agent.

AgentBuilder provides a comprehensive set of tools for programming software agents. Figure 1 illustrates the use of some of these tools.

AgentBuilder provides tools for:

- Organizing and controlling the development project
- Analyzing the problem domain
- Specifying interaction protocols
- Defining the agency architecture
- Examining running agencies
- Specifying agent behavior
- Creating run-time executable agents
- Viewing and debugging agents

Project Accessory Classes (PACs) contain the problem-specific code that each agent requires for its operation. For example, a database agent might have a PAC for performing SQL queries. PACs can be made up of classes and packages from a variety of sources including legacy and off-the shelf software. PACs can be coded in Java, C or C++.

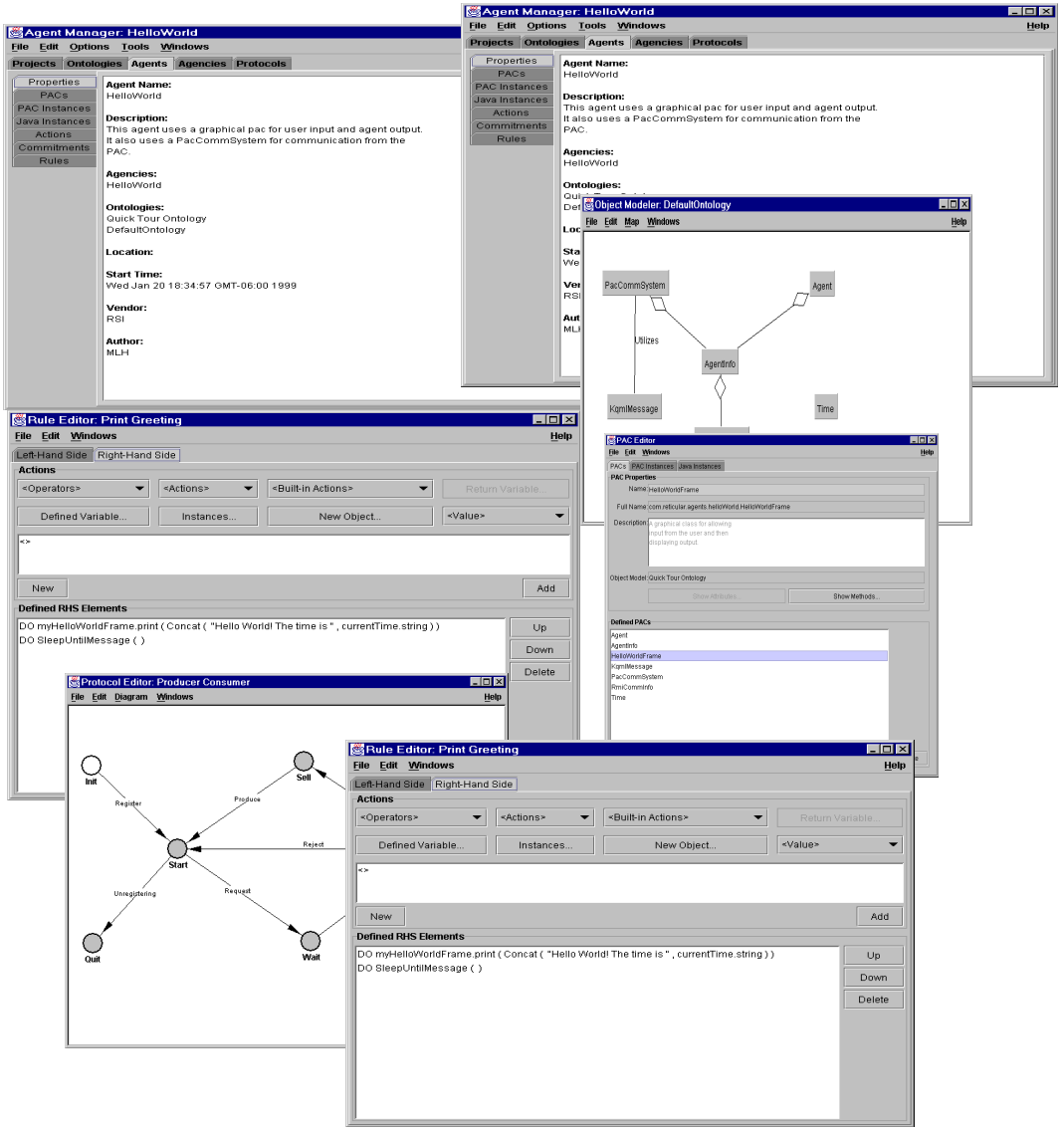

Figure 1. AgentBuilder Tools

C. RunTime System

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 and the private action library (i.e., agent actions and user interface libraries). The agent program is executed by the run-time agent engine; the combination of the agent program and the agent engine create an executable agent.

D. Technical Support

Acronymics, Inc. is committed to providing prompt technical support. If the answer to your problem can't be found in the User's Guide, the built-in help system, or the documentation found at *www.agentbuilder.com/Documentation*, our technical support staff will be glad to assist you.

There are two avenues by which you can reach technical support:

- Telephone
- Email

The type of support to which you are entitled depends upon your version of AgentBuilder. When requesting technical support have the following information available:

- Your license number or invoice number and version of the software.
- Description of your problem: What occurs and when, how severe the problem is, what error messages appear on your screen, and anything else of relevance.
- The error log generated by the program.
- A profile of your computer (manufacturer, model, and hardware).

Telephone

To schedule telephone support, please send an email to

i**nformationAgent@agentbuilder.com** with **Subject**: ***support***.

We will contact you and schedule a support call for you. Telephone support is available on a scheduled basic only to AgentBuilder Pro users. Please be at the computer with AgentBuilder installed when

you talk to a support engineer. Please try to resolve your probem by email before using telephone support.

Email Support

Email support is available to all AgentBuilder users by emailing your question to:

`support@agentbuilder.com`.

Other Support

A list of Frequently Asked Questions is also maintained. Go to the AgentBuilder website at http://www.agentbuilder.com/Documentation/FAQ.html to review the FAQ.

E. System Requirements

This section provides the information required to install and familiarize yourself with AgentBuilder. Please check that your computer system has the hardware and software installed that is required for running AgentBuilder. Install the AgentBuilder software and explore the software using the Quick Tour of AgentBuilder.

AgentBuilder only requires a Java development environment for developing user-defined classes (PACs). The Java development environment must be compliant with at least the Java 1.1.8 API.The current version supports Java 1.3. Java 1.1.8 is required for LINUX and Macintosh support. Only the user-defined classes and the AgentBuilder libraries are required for runtime execution of agents.

AgentBuilder is distributed with the JRE (Java Runtime Environment) for each supported platform. A Java virtual machine is included with the JRE. Both the AgentBuilder Toolkit and the Run-Time System execute on this Java Virtual Machine.

Personal Computers

Windows 98/NT/2000

The operating system requirements are Windows 98, Windows 2000 or Windows NT. The minimum recommended hardware is a Pentium 200 MHz with a minimum of 32 MB of RAM. A 300 MHz machine with 128K of memory is highly recommended.

Macintosh Support

AgentBuilder is not “officially” supported on the Macintosh platform with this release. However, with some limitations, AgentBuilder can be used on the Macintosh.

The minimum requirements for running a JVM on the Macintosh are: (see http:/developer.apple.com/java/download.htm):

- 64 M of memory
- Macintosh with PowerPC Processor
- Mac OSX
- At least 13 M of free disk space.

UNIX Workstations

Currently Solaris and LINUX are supported. Other UNIX platforms can be supported if the appropriate JREs is available. Please contact `support@agentbuilder.com` for information for support on additional platforms.

Solaris

Only Solaris 2.5.1 or higher is supported. The recommended hardware is a SUN SPARC or higher with 64 MB or RAM.

Linux

The i386 LINUX port is the only supported platform. The recommended minimum hardware requirement is a Pentium 200 MHz with 32 MB of RAM. The following libraries will also need to be installed.

- glibc-2.0.7-7
- glibc-devel-2.0.7-7
- libc.so.5.44 or greater
- recent version of ld.so

F. Using This Guide

Style

Information you enter into the computer is indicated with a Courier Oblique type font:

this is stuff you Type

Computer generated responses and the names of icons, dialogs, files and directories are indicated using an Helvetica bold font:

The Computer Said This; Agents File

RADL listings are also printed in the Helvetica font:

A Sample RADL Code Listing

Code snippets are indicated in a Courier font:

```
this is a code snippet(for you);
```

Button Labels, Menu Bars and Menu Items are indicated using a **Helvetica Bold** font:

Menu Item or **Cancel**

Menu items in a menu bar are indicated by:

Menu Bar Item → Menu Item1 → Menu Item Level 2

This means that for the **Menu Bar Item**, the **Menu Item 1** (a hierarchical menu) is selected with sub-item **Menu Item Level 2**.

G. If You Never Read Manuals...

...that's okay with us. If you've never built an intelligent agent and are not familiar with software development, its highly recommended that you first read and work through “Part II. Getting Started with AgentBuilder.” You shoud also review the Reference sections of the documentation. If the compulsion to sit down and start working with the tool is overwhelming then you need to at least make sure everything is properly installed.

Table 1. Checking out AgentBuilder

	Do this	***Comments***
1.	If AgentBuilder isn't already installed on your computer, follow the steps in “Installation Instructions” on page 15 to properly install the system.	
2.	Start the program.	For UNIX users, at the UNIX prompt type the command `AgentBuilder`. For Microsoft Windows users, choose the **AgentBuilder** program icon from the programs menu. In either case if a window appears that is titled **Project Manage**r then the program has successfully been launched.

Table 1. Checking out AgentBuilder

	Do this	***Comments***
3.	To get started, double click the **System Repository** icon	
4.	double click the **Example Project** icon	
5.	double click the **Hello World Agency** icon	Now you should see two icons that represent two agents.
6.	Click on the **HelloWorld** icon and choose the **Agent Manager** from the tools menu	This brings up the tools for programming individual agents. By choosing different tabs, you will see the various components that make up the **HelloWorld** agent. By double clicking on any of the components in the middle panel, the editor for that component is started
7.	Have Fun	

H. Installation Instructions

Before installing the AgentBuilder tools, ensure that you have the most recent version of the software and the README file for that version. **Follow the detailed instructions in the README for installation.** The latest versions are available from the Agent-Builder web site. When downloading the software make sure to choose the version appropriate for the platform being used.

I. Product Family

The AgentBuilder product family[1] includes two different products. *AgentBuilder Lite* is ideally suited for developing single-agent, stand-alone applications. The low cost of AgentBuilder Lite also makes it ideal for software developers who are investigating intelligent agent technology or building their first agent applications. *AgentBuilder Pro* supports development of networks of communicating agents and agents and includes sophisticated tools for building and running multi-agent systems.

Acronymics, Inc. also offers a variety of software maintenance and support services, consulting services, and custom software development.

AgentBuilder Lite

AgentBuilder Lite is the entry level product for intelligent agent software developers. AgentBuilder Lite provides tools for constructing single-agent, stand-alone applications. AgentBuilder Lite includes licenses for:

- Project control tools including Project Manager and Project Dictionary tools.
- Ontology Manager including concept mapping and object modeling tools
- Agent Manager tools for creating RADL-based agent programs
- Agent Console with capability to view and control a single agent
- Run-Time Engine†

1. The product descriptions and specifications provided here are subject to change without notice.

- Support through AgentBuilder Web site access and mailing list membership
- Sample ontological libraries
- Single-user license

Academic versions of AgentBuilder Lite are available to accredited universities.

† Each agent engine requires a separate run-time license.

AgentBuilder Pro

AgentBuilder Pro includes all of the tools of AgentBuilder Lite with the addition of the following:

- Agency Manager including tools for creating and managing multiple intelligent software agents. This includes the agency viewer tools for examining remote agent operation and protocol editors for specifying interagent protocols
- Multi-agent debugger support
- Support for optional learning and planning modules
- Single-user license
- Email and limited FAX and telephone support

Academic versions of AgentBuilder™ Pro are available to accredited universities.

J. Frequently Asked Questions (FAQ)

We'll be adding to this list as we receive questions, comments and feedback. :

Q **How do I get the latest FAQ?**

A *Go to the AgentBuilder website at http://www.agentbuilder.com/Documentation/FAQ.html*

Q **What is AgentBuilder?**

A *AgentBuilder is an integrated tool suite for constructing intelligent software agents. AgentBuilder consists of two major components - the Toolkit and the Run-Time System. The AgentBuilder Toolkit includes tools for managing the agent-based software development process, analyzing the domain of agent operations, designing and developing networks of communicating agents, defining behaviors of individual agents, and debugging and testing agent software. The Run-Time System includes an agent engine that provides an environment for execution of agent software. For more detailed information, see the product description and our white paper at http://www.agentbuilder.com*

Q **What platforms does AgentBuilder support?**

A *AgentBuilder requires a Java Virtual Machine capable of executing Java 1.2.x. AgentBuilder will work on any platform with this capability. AgentBuilder Lite has been tested on Solaris, Windows NT, Windows 2000, Windows XP and Linux platforms. Installation programs are available for Windows, Linux and Solaris. Acronymics does not provide installation programs for other platforms.*

Q **Can I build multiple agents using AgentBuilder Lite?**

A *Yes! AgentBuilder Lite is capable of building multiple agents. In fact, this User's Guide describes building a multi-agent system for electronic commerce - a simple shopping agent with two storefront agents (see Chapter 10). AgentBuilder Pro, however, has more sophisticated tools for multiple agent development.*

Q **How much does AgentBuilder cost?**

A *A current price list is available on the AgentBuilder website. You may also place your order online using our secure server. If you have any questions, please contact AgentBuilder Sales at (480) 615-8543 or via email: informationAgent@agentbuilder.com with Subject: sales.*

Q **Is there an evaluation copy of AgentBuilder available?**

A *Evaluation copies of AgentBuilder Pro are available. There are no evaluation copies for AgentBuilder Lite. This User's Guide and a Reference Manual are available for download. In addition, for the academic community, we have special discounted pricing.*

Q **If I buy AgentBuilder Lite now, are there any special price discounts for purchasing the Pro version?**

A *We do have a special pricing plan that allows the cost of an AgentBuilder Lite license to be applied towards the purchase of an AgentBuilder Pro license (Note: This does not apply to academic versions). For additional information, please contact AgentBuilder Sales at infor-*

mationAgent@agentbuilder.com Subject: pricing or call (480) 615-8543.

Q **Do you have distributors in countries other than the U.S.?**

A *Yes, on a limited basis. Please contact us directly for details. We are making all distributions available electronically. However, we do ship CD-ROMs and printed manuals worldwide.*

Part I. Agents and Agent Development

Chapter 2

Introduction to Agents

Chapter Overview

You can find the following information in this chapter:

- Introduction to Intelligent Agents
- Characteristics of Intelligent Agents
- Intelligent Agent Architectures

A. Introduction to Intelligent Agents

What are Intelligent Agents?

The concept of an intelligent software agent has captured the popular imagination. People like the idea of delegating complex tasks to software agents. These agents can make airline reservations, order books from an on-line store, find out about the latest song from a favorite musician, or monitor stock portfolios. Software agents can roam the Internet to locate information for us. Sophisticated software agents can negotiate the purchase of raw materials for a factory, schedule factory production, negotiate delivery schedules with a customer's software agent, or automate the billing process. However, developing intelligent agents requires specialized knowledge and can be difficult, time-consuming, and error-prone. New tools are needed to make it easier for software developers to build these sophisticated software agents.

Intelligent software agents are a new class of software that act on behalf of the user to find and filter information, negotiate for services, easily automate complex tasks, or collaborate with other software agents to solve complex problems. Software agents are a powerful *abstraction* for visualizing and structuring complex software. Procedures, functions, methods, and objects are familiar software abstractions that software developers use every day. Software agents, however, are a fundamentally new paradigm unfamiliar to many software developers.

The central idea underlying software agents is that of *delegation*. The owner or user of a software agent delegates a task to the agent and the agent *autonomously* performs that task on behalf of the user. The agent must be able to *communicate* with the user to receive its instructions and provide the user with the results of its activities. Finally, an agent must be able to *monitor* the state of its

own execution environment and make the decisions necessary for it to carry out its delegated tasks.

There are two approaches to building agent-based systems: the developer can utilize a single stand-alone agent or implement a multi-agent system. A stand-alone agent communicates only with the user and provides all of the functionality required to implement an agent-based program. Multi-agent systems are computational systems in which several agents cooperate to achieve some task that would otherwise be difficult or impossible for a single agent to achieve. We term these multi-agent systems *agencies*. Agents within an agency communicate, cooperate, and negotiate with each other to find a solution to a particular problem.

Why are They Important?

Every day, software developers are tasked with constructing ever larger and more complex software applications. Developers are now building enterprise-wide and global applications that must operate across corporations and continents. More and more corporations need to integrate their information systems with those of their suppliers and customers. New systems must link multiple organizations and multiple application platforms into a unified information management system that uses the World Wide Web and distributed object technologies.

Next-generation systems must provide global connectivity through a variety of internets and intranets. Users of these systems will include office and factory workers, suppliers, mobile workers, workgroups, customers, and remote workers. Application platforms will vary from desktop personal computers to large multiprocessor mainframes. The kinds of applications that must communicate and operate with each other will vary in complexity from programs as simple

as Intuit's *Quicken* to complex enterprise applications such as SAP's *R/3*.

Developing applications for these existing and emerging application domains requires powerful new methods and techniques for conceptualizing and implementing software systems. Intelligent software agents provide a powerful problem-solving paradigm that is well-suited to developing complex enterprise applications.

Why are They Difficult to Build?

Agents and agent technology have been an active area of research in the artificial intelligence and computer science community for a number of years. Many universities have developed intelligent software agents. A number of companies deliver software agents capable of performing a wide variety of specialized tasks. However, each of these agents had to be handcrafted for a particular application. Building an intelligent software agent is a difficult and time-consuming task that requires an understanding of advanced technologies such as knowledge representation, inferencing, network communications methods and protocols, etc. Sophisticated applications often require expertise in machine learning and machine planning technology.

A developer using intelligent agents in a new application must decide on the overall agent processing architecture, the agent's reasoning (inferencing) mechanism and associated pattern matching technology, internal knowledge and data representations, agent-to-agent communications protocols and message formats. In addition, if the agent is to learn from its environment or its owner then some kind of machine learning technology will be required. Sophisticated agents may require a planning capability which will require that the developer select a planning algorithm and implementation. If the application requires multiple communicating agents then the

developer needs to establish a robust communications protocol between the agents. This will require that the developer have knowledge and expertise in the underlying communications technologies used for interagent communications.

AgentBuilder - a Toolkit for Agent Construction

Software developers need a set of tools that will aid them in developing agent-based applications. Tools are needed that can help the software developer analyze the application domain; formally recognize and describe the concepts, relationships, and objects relevant to that domain; and specify the behavior of the agent(s) operating in the domain. The software developer also needs tools that can specify a collection of agents, analyze and specify the messages and message protocols between agents, and execute and evaluate the actions of the agents. The *AgentBuilder Toolkit* which provides these capabilities.

The AgentBuilder product consists of two major components: the development tools and the run-time execution environment. The development tools are used for analyzing an agent's problem domain and for creating an agent program that specifies agent behavior. The run-time system provides a high-performance agent engine that executes these agent programs.

Agents constructed using AgentBuilder communicate using the Knowledge Query and Manipulation Language (KQML) [Finin, *et al*, 1994a; Finin, *et al,* 1994b; Finin, *et al*, 1994c; Labrou *et al*, 1994; Labrou, 1996] and support the performatives defined for KQML (described later in this paper). In addition, AgentBuilder allows the developer to define new interagent communications commands that suit his particular needs.

The AgentBuilder toolkit and the run-time engine are implemented using the Java programming language. Thus, AgentBuilder will run on any platform that supports Java development. Agents created with

AgentBuilder are themselves Java programs and will execute on any platform with a Java virtual machine.

AgentBuilder allows software developers with no background in intelligent systems or intelligent agent technologies to quickly and easily build intelligent agent-based applications. AgentBuilder reduces development time and development cost and simplifies the development of high-performance, robust, agent-based systems.

B. Characteristics of Intelligent Agents

Intelligent Software Agents

Before defining the characteristics of an intelligent agent we first look at the general characteristics of a *software agent*. A software agent is viewed as an autonomous software construction; i.e., one that is capable of executing without user intervention.

We place two additional constraints on the software before defining such a construct as a software agent: an agent must have the ability to communicate with other software or human agents and the ability to perceive and monitor the environment. The ability to communicate implies that the agent has the ability to cooperate with other agents (after all, cooperation is required in order to receive and acknowledge a communication). Cooperation is of paramount importance and is the primary reason for using multiple agents in a software architecture. Wooldridge and Jennings argue that this cooperation implies a social ability to interact with other agents (or humans) [Wooldridge & Jennings, 1995]. Other researchers use much less rigorous definitions for an agent. Some require only the capability for autonomous operations [Franklin & Graesser, 1996]. Others such as Russell and Norvig insist on the capability to perceive and affect the environment [Russell & Norvig, 1995]. Smith, *et al* [Smith, Cypher, & Spherer, 1994] view an agent as a persistent software entity dedicated to a specific purpose. We define a software agent as a software component that:

- executes autonomously
- communicates with other agents or the user
- monitors the state of its execution environment

Having defined the general characteristics of a software agent, we can begin to address the question of what makes a software agent an

intelligent software agent — this means that we must address the broader question of what is meant by the term *intelligent software*. It is certainly beyond the scope of this document to argue the definition of intelligent software. (The interested reader should consult one of the many texts on artificial intelligence such as [Rich, 1983] or [Russell & Norvig, 1995].) Newell argues that for software to be considered intelligent it should possess the following capabilities or attributes [Newell, 1988]:

- Able to exploit significant amounts of domain knowledge
- Tolerant of errorful, unexpected, or wrong input
- Able to use symbols and abstractions
- Capable of adaptive, goal-oriented behavior
- Able to learn from the environment
- Capable of operation in real-time
- Able to communicate using natural language

A strong argument can be made that an intelligent software agent need not have all of the capabilities and attributes described above. For example, many applications do not require a *real-time* response, merely a *timely response*. Other applications involve only agent-to-agent interaction and thus do not require the ability to communicate using natural language. Numerous researchers have shown that highly capable intelligent agents can be constructed without having a learning capability. Hayes-Roth views intelligent agents as necessarily having the capability to perform three functions:

- Perceive dynamic conditions in the environment
- Take action to affect conditions in the environment
- Reason to interpret perceptions, solve problems, draw inferences, and determine actions [Hayes-Roth, 1995]

IBM researchers define intelligent agents as:

> "*software entities that carry out some set of operations on behalf of a user with some autonomy and employ either knowledge or representation of the user's goals and desires"* [Gilbert et al, 1996].

Table 2. Attributes of an Intelligent Agent

Agent	Executes autonomously
	Communicates with other agents or the user
	Monitors the state of its execution environment
Intelligent Agent	Able to use symbols and abstractions
	Able to exploit significant amounts of domain knowledge
	Capable of adaptive goal-oriented behavior
Truly Intelligent Agent	Able to learn from the environment
	Tolerant of errorful, unexpected, or wrong input
	Capable of operation in real-time
	Able to communicate using natural language

Wooldridge and Jennings [Wooldridge & Jennings, 1995] not only require autonomy, perception and reactivity but further expand the definition to include proactivity (i.e., agents must not simply act in response to the environment, they must be able to exhibit goal-directed behavior by taking initiative). Nwana argues that for an agent to be considered "smart" it must be able to *learn* as it reacts and/or interacts with its external environment. Learning can take the form of improved performance by the agent over time as it performs various tasks [(Nwana, 1996].

Truly intelligent agent software will thus possess the capabilities of agent software (autonomy, communicability, perception) and the capabilities of intelligent software (ability to exploit knowledge and tolerate errors, reason with symbols, learn and reason in real time, and communicate in an appropriate language). Thus it seems clear that intelligent agent software should not be viewed as being either "smart" or "dumb" but, rather, should be viewed as having intellectual capabilities lying along a continuum. Software with more intelligence will have greater capabilities. In certain applications, intelligent agents with limited capabilities will be all that is required. Minsky argues that large networks of very simple communicating and cooperating agents each performing only simple actions may be all that is required for intelligent processing [Minsky, 1985].

It should be pointed out that a software component possessing only some of these capabilities can hardly be considered an intelligent agent. For example, an *expert system* is typically able to exploit knowledge, use symbols and abstractions and is capable of goal-oriented behavior. However, expert systems generally can't execute autonomously, learn, or communicate and cooperate with other agents. Thus, in most cases, we do not consider them intelligent agents.

What Isn't An Agent

Intelligent agent technology has been the victim of hyperbole and exaggeration. Patti Maes notes that current commercially available agents barely justify the name *agent* yet alone the adjective *intelligent* [Maes, 1995)]. MIT researcher Foner argues:

> *"... I find little justification for most of the commercial offerings that call themselves agents. Most of them tend to excessively anthropomorphize the software and then conclude that it must be an agent because of that very anthropomorphization, while simultaneously failing to provide*

> *any sort of discourse or "social contract" between the user and the agent. Most are barely autonomous, unless a regularly-scheduled batch job counts. Many do not degrade gracefully, and therefore do not inspire enough trust to justify more than trivial delegation and its concomitant risks." [Foner, 1993]*

It is important to take care in classifying agent software as to its degree of intelligence. The term *intelligence* has strong historical and emotional connotations. We believe that it is better to assess intelligent agent software in terms of its competence. A highly competent agent will exhibit all of the attributes shown in Table 1 (to some degree). Agents with relatively low competence will exhibit significantly fewer of these attributes.

Agent Classification

There are probably as many ways of classifying intelligent agent software as there are researchers in the field. Nwana provides a typology defining four types of agents based on their abilities to cooperate, learn, and act autonomously; she terms these *smart agents, collaborative agents, collaborative learning agents,* and *interface agents* [Nwana, 1996]. Figure 2 depicts how these four types of agents utilize the capabilities described above.

Collaborative Agents

Collaborative agents emphasize autonomy and cooperation to perform tasks by communicating and possibly negotiating with other agents to reach mutual agreements. These agents are used to solve distributed problems where a large centralized agent is impractical (e.g., air traffic control). Central to this class of agent is a well-defined agent communications language such as KQML, which is described later in this paper.

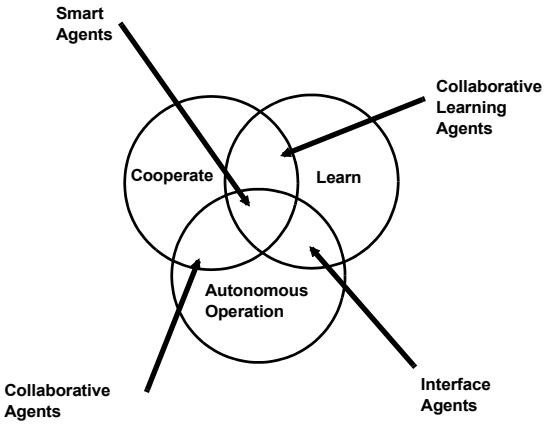

Figure 2. Agent Typology

Interface Agents

Interface agents are autonomous and utilize learning to perform tasks for their users. The inspiration for this class of agents is a personal assistant that collaborates with the user. This class of agent is used to implement assistants, guides, memory aides, and filters; perform matchmaking and referrals; or buy and sell on behalf of the user.

Mobile Agents

Mobile agents are computational processes capable of moving over a network (possibly a wide area network such as the Internet or World Wide Web), interacting with foreign hosts, gathering information on behalf of the user, and returning to the user after performing their assigned duties. Mobile agents are implementations of remote programs (i.e., programs developed on one machine and delivered to a second machine for subsequent execution). Many of

the issues that must be addressed in remote programming must also be addressed in dealing with mobile agents. These include:

- *program naming* – assigning names to agent programs to distinguish one from another
- *program authentication* – authenticating the implementor of an agent program
- *program migration* – moving a program from one machine to another
- *program security* – ensuring that a program does not do harm to the executing machine

The popular press and the trade press have in many cases treated mobile agents as the only embodiment of intelligent agents. As Nwana points out quite emphatically, "mobility is neither a necessary nor sufficient condition for agenthood." [Nwana, 1996]

Information Agents

One of the most popular uses for intelligent agents is for finding, analyzing and retrieving large amounts of information. *Information agents* are tools to help manage the tremendous amount of information available through networks such as the World Wide Web and Internet [Cheong, 1996]. Information agents access the network looking for particular kinds of information, filter it, and return it to their users. Information agents are designed to mitigate the information overload caused by the availability of large amounts of poorly cataloged information. These agents typically use HTTP protocols to access information, although they may also take advantage of KQML or other agent communications languages for interagent communications.

Heterogeneous Agent Systems

Heterogeneous agent systems refer to a collection of two or more agents with different agent architectures. Because of the wide variety of application domains, it is unlikely that any one agent architecture will be used exclusively across all domains; for each domain, the most appropriate agent architecture will be selected. Agents in these heterogeneous systems will communicate, cooperate, and interoperate with each other. A key requirement for this interoperation is the availability of an agent communications language that will allow different kinds of software agents to communicate with each other.

For purposes of this document we consider an intelligent agent to be any software program that can operate autonomously (i.e., without user intervention), learn the habits and preferences of the user, take instructions from and communicate with the user and/or other agents, and perform the duties that the user or other agent assigns it. Intelligent agents are specialized to provide a high level of competence and capability in narrow domains. An intelligent agent must be able to develop high-level plans and goals and attempt to satisfy those goals.

C. Intelligent Agent Architectures

Background: Mentalistic Agents

This section provides a detailed description of an intelligent agent implementation. The motivation for the AgentBuilder agent architecture is the seminal work of Shoham [Shoham, 1990; Shoham,1991; Shoham, 1993; Shoham, 1995] in developing cognitive, i.e., mental model-based, agents. Shoham defined an agent programming language (AGENT-0) for writing agent programs. He theorized that cognitive agents possess a mental state which is composed of various mental elements: beliefs, commitments, capabilities, and commitment rules.

Shoham's Work

Shoham describes two primitive modalities of mental state: *belief* and *commitment*. Commitment is treated as a decision to act rather than a decision to pursue a goal. A strong condition is placed on belief; namely that agents have perfect memory of, and faith in, their beliefs and only relinquish a belief if they learn a contradictory fact. Beliefs persist by default. Furthermore, the absence of a belief also persists by default but in a slightly different sense. If an agent does not believe a fact at a certain time (as opposed to believing the negation of the fact), then the only reason the agent will come to believe it is if the agent learns it.

AGENT-0 defines two different action types: *private actions* and *communicative actions*. Private actions are the primary method agents use to accomplish tasks that affect the agent's environment. For example, a database agent might include private actions that execute SQL queries on a database. Communicative actions are the mechanism for exchanging messages with other agents.

While agent behavior can be arbitrarily complex, it is produced using relatively simple operations. Agents can receive messages, send messages, perform private actions, and update their own mental models. The behavior of an agent is governed by its *program*. An AGENT-0 program consists of initial beliefs, initial commitments, the capabilities of the agent, and the commitment rules. Initial beliefs and initial commitments are present in the mental state at agent start-up. Even though initial commitments are instantiated when the agent starts executing they may not be applicable until some specific time in the future. Capabilities define the actions that the agent can perform and are fixed for the lifetime of the agent. Commitment rules determine the actions performed and the mental changes of the agent in all situations.

Agent Mental Models

Shoham's AGENT-0 research was extended by Thomas, who developed PLAnning Communicating Agents – PLACA [Thomas, 1993; Thomas, 1994], an agent programming language similar to AGENT-0 with extensions for planning. AgentBuilder further extends the work of Shoham and Thomas and provides a new, more practical, user-friendly, agent programming language. This object-oriented language is called the *Runtime Agent Definition Language* (RADL). AgentBuilder provides graphical tools for creating RADL programs that execute in AgentBuilder's run-time system. The following paragraphs describe the runtime language and agent architecture in more detail.

Beliefs

Beliefs are a fundamental part of the agent's mental model. Beliefs represent the current state of the agent's internal and external world and are updated as new information about the world is received. An agent can have beliefs about the world, beliefs about another

agent's beliefs, beliefs about interactions with other agents, and beliefs about its own beliefs.

Mental model integrity is improved because belief instances and pattern variables are always of known type, so the agent's inference engine can ensure that comparisons between mental model elements are always valid. This prevents accidental matches between beliefs and variables of differing types.

Runtime efficiency is improved because belief templates allow the agent's inference engine to focus its search activities when looking for beliefs that match specified rule conditions. In most situations only one or, at most, a small subset of the beliefs need be examined.

Capabilities

A *capability* is a construct used by the agent to associate an action with that action's necessary preconditions. The necessary preconditions (sometimes known as *executability* or *primary* preconditions) are the preconditions that must be satisfied before execution of the action [Thomas, 1993]. An agent's list of capabilities defines the actions that the agent can perform provided that the necessary preconditions are satisfied. A capability is static and holds for the lifetime of an agent. However, the actions an agent can perform may change over time because changes in the agent's beliefs may alter the truth value of precondition patterns in the capability.

For example, consider an agent used to control access to a database. One of the actions of such an agent is to perform queries. A simple capability for the `MakeQuery` action is:

```
Action: MakeQuery (database, ?Query)
Preconditions: database.Status IS Online
```

In this example, the precondition is satisfied when the agent believes the database is on-line. Only then can the execution of the `MakeQuery`

action proceed. Although this capability will not change during the lifetime of the agent, the agent's ability to perform the `MakeQuery` action will depend on the agent's belief about the status of the database.

The AgentBuilder agent architecture classifies actions in two main categories, private actions and communicative actions, as did Shoham. Private actions are actions that affect or interact with the environment of the agent and do not depend on interaction with other agents. Communicative actions are defined as actions that send messages to other agents.

Capabilities are used by the agent to decide whether to adopt commitments (described below). An agent will not adopt a commitment to perform an action if the agent can never be capable of performing that action. Capabilities are also used to determine whether a committed action can be executed.

Commitments

A commitment is an agreement, usually communicated to another agent, to perform a particular action at a particular time. The usual sequence of operations will be as follows: one agent, say agent "Alice", will send a commitment request in a message to another agent, say agent "Betty". Betty will accept or reject the request based on the details of the request, her behavioral rules and current mental model (beliefs, existing commitments, etc.). Finally, Betty will send a message to Alice indicating acceptance or rejection of the request.

If Betty accepts the request she agrees to perform the requested action at the requested time, if possible. It should be noted that a commitment is not a guarantee that an action will be performed; more precisely, a commitment is an agreement to attempt a particu-

lar action at a particular time if the necessary preconditions for that action are satisfied at that time.

When the current time equals the commitment time (assuming Betty adopted the commitment), Betty must test the necessary preconditions of the committed action to ensure that the action can be executed. Betty must have a capability corresponding to the committed action (otherwise she could not have adopted the commitment), and the capability will have zero or more precondition patterns that define the necessary preconditions for the action. To test the preconditions Betty must match the precondition patterns against her current beliefs. If all patterns evaluate to true, Betty can initiate execution of the committed action, otherwise she cannot—an agent should not attempt to execute an action for which the necessary preconditions fail even if the agent is committed to that action.

In general, successful execution of an action may be beyond the agent's control. For example, agent Betty may have committed to make an inquiry into a database on behalf of Alice. Even if the necessary preconditions are met (e.g., Betty believes the database is currently functioning) and Betty is able to initiate execution, the action may still fail (e.g., a disk could crash during the database access). Betty must monitor the execution so she will be able to send a message back to Alice to report the success or failure of the commitment.

Behavioral Rules

In Shoham's model, all actions were performed only as the result of commitments. AgentBuilder has extended the idea of a commitment rule to a include a general *behavioral rule*. Behavioral rules determine the course of action an agent takes at every point throughout the agent's execution. Behavioral rules match the set of possible responses against the current environment as described by the agent's current beliefs. If a rule's conditions are satisfied by the environ-

ment, then the rule is applicable and the actions it specifies are performed.

Behavioral rules can be viewed as WHEN-IF-THEN statements. The WHEN portion of the rule addresses new events occurring in the agent's environment and includes new messages received from other agents. The IF portion compares the current mental model with the conditions that are required for the rule to be applicable. Patterns in the IF portion match against beliefs, commitments, capabilities, and intentions. The THEN portion defines the agent's actions and mental changes performed in response to the current event, mental model, and external environment. These may include:

- mental model update
- communicative actions
- private actions

A behavioral rule is allowed to have any combination of the possible actions and mental changes outlined above.

The following listing describes the format for the behavioral rules.

```
NAME rule name
WHEN
    Message Condition(s)
IF
    Mental Condition(s)
THEN
    Private Action(s)
    Mental Change(s)
    Message Action(s)
```

The following paragraphs provide an example and a description of behavioral rules. The agent used in this example is a seller agent representing a grocery store in an electronic marketplace. One of

the agent's behavioral rules is shown along with a snapshot of the agent's mental model at a particular point in time.

The agent's mental model is composed of several user-defined PAC[1] instances (e.g., `InventoryRecord` and `SellerFrame` instances), several built-in PAC instances (e.g., `Time` and `Agent` instances), and two Java instances. Some of the instances are named instances, such as the `currentTime` and `accountBalance` instances. Some of the instances are not named, such as the `InventoryRecord` instances. The `currentTime` instance is marked with a `*` to indicate that it changed during the last cycle. Figure 3 shows an agent's mental model.

This mental model contains three Agent beliefs, one for the store agent called the `Store 2` agent and two other known agents. There are three `InventoryRecord` beliefs, one for each product sold by this store. The `SellerFrame` is a graphical interface (derived from `java.awt.Frame`) used to display the store agent's activities to the user. The mental model contains a belief about the store's current account balance, stored in a `Float` object, and a belief about the status of the graphical interface.

The rule shown below in Figure 4 is used to accept a message containing a price quote request from a known buyer agent, fill in this store's price, then return the message to the buyer agent. This is just one rule of approximately a dozen rules that define the behavior of this agent in a highly simplified electronic marketplace.

The `WHEN` section of the rule is used to test an incoming message from another agent and the `IF` section of the rule is used to test conditions in this agent's mental model. All of these conditions in the `WHEN` and

1. A PAC is a Project Accessory Class. These are the user-defined classes that represent concepts in the domain and also define the actions of the agent.

```
Mental Model (Beliefs)

* Time<currentTime> Mon Jun 08 14:23:08 PDT 1998

  Time<startupTime> Mon Jun 08 14:22:44 PDT 1998

  Agent<SELF> Agent Name: Store 2  Address: harding.reticu-
lar.com

  Agent<Buyer> Agent Name: Buyer  Address: quincy.reticu-
lar.com

  Agent<Store 1> Agent Name: Store 1  Address: hard-
ing.reticular.com

  InventoryRecord<> ProductName = Milk, Quantity = 41 gal-
lon, UnitPrice = 2.24

  InventoryRecord<> ProductName = Bread, Quantity = 28
loaf, UnitPrice = 2.33

  InventoryRecord<> ProductName = Bananas, Quantity = 32
lb., UnitPrice = 0.55
```

Figure 3. An Agent's Mental Model (Beliefs)

IF sections are implicitly ANDed together, i.e., the set is satisfied if and only if *all* conditions are satisfied. If any condition is not satisfied then the rule is not applicable to the situation. Evaluation stops as soon as a single condition fails.

If, however, all conditions are satisfied then the rule will be activated and then its actions and mental changes are put on the agenda for execution. In this example the rule will send a return message

```
"Receive Bid Request"
WHEN
1. ( %message.sender EQUALS ?buyer.name )
2. ( %message.performative EQUALS "ask-one" )
3. ( %message.contentType EQUALS PriceQuote )
4. ( %message.content.productName EQUALS ?invento-
ryRecord.productName )
5. ( %message.content.quantity <= ?inventoryRecord.quantity
)
IF
1. ( Interface_ready_to_print EQUALS true )
THEN
1. ( DO PrintStatus( Concat( "Sent price quote to ", %mes-
sage.sender )) )
2. ( SET_VALUE_OF %message.content.price TO ?invento-
ryRecord.unitPrice )
3. ( SET_VALUE_OF %message.content.storeName TO SELF.name )
4. ( DO SendKqmlMessage( %message, SELF.name, %mes-
sage.sender, "tell",,
                         %message.replyWith,,,,,,) )
5. ( DO SleepUntilMessage() )
```

Figure 4. An Example Rule

containing a complete price quote to the buyer agent that sent a price quote request.

`%message`e is a variable of type `KqmlMessage`; the leading character `%` is used to indicate that this is a message variable which binds to incom-

ing messages. The dot-separated list following a variable name represents subobjects, or subobjects of subobjects, etc. In the first pattern, for example, `%message.sender` accesses the sender field (a `String`) of the `KqmlMessage` binding. In the fourth pattern, `%message.content.productName` means "bind to an incoming message, then access the content subobject of the message, then access the productName subobject of the content".

`?buyer` is a variable of type `Agent`, which is a built-in class provided with AgentBuilder to represent beliefs about agents (e.g., name, address, etc.). The leading character `?` is used to indicate that this variable binds to `Agent` instances in the mental model. `?buyer.name` evaluates to a `String`, so the first pattern in the `WHEN` section is a comparison between the `String` objects bound to two variables.

The second pattern in the `WHEN` section is similar to the first except that it compares a field in the message to a `String` constant. If evaluation of the first pattern is successful then evaluation proceeds to the second pattern, using the same binding for the `%message` variable as was used in the first pattern.

The third pattern checks the *type* of the content of the message and compares it to the `PriceQuote` class, which is a PAC designed for this problem domain. If the content object is a `PriceQuote` object then rule evaluation proceeds to the fourth pattern. The fourth pattern checks the *value* of one subobject in the content of the message.

`?inventoryRecord` is a variable of type `InventoryRecord`, another PAC designed for this problem domain. This has fields for `productName`, `quantity`, and `unitPrice`. The fifth `WHEN` condition compares the quantity field in the `PriceQuote` from the message with the quantity field in an `InventoryRecord` in the mental model. Note

that the InventoryRecord binding used in pattern 5 is the *same* binding that was used in pattern 4. We test the quantity field in the InventoryRecord which has the same product name as the PriceQuote in the message.

The first and only IF condition in this rule is a test of a named instance. This pattern evaluates to true if the mental model contains a Boolean instance named "Interface_ready_to_print" with a value of true. The Boolean instance needed by this pattern gets asserted by another rule, the Launch Interface rule, which fires before the Receive Bid Request rule.

The left-hand side of the Receive Bid Request rule can be paraphrased as follows, from the point of view of the store agent:

WHEN I receive a KQML message sent from a known buyer agent, and the performative in that message is "ask-one", and the content type in that message is PriceQuote, and the product name in the message's PriceQuote is the same as the product name in one of my InventoryRecords, and (according to that same InventoryRecord) I have at least as much product on hand as the quantity requested in the PriceQuote, and IF the interface is ready to receive a PrintStatus command, THEN ...

The THEN section of this example rule consists of three actions and two mental changes. The first THEN pattern is an action statement with the user-defined action PrintStatus. This invokes a method on the SellerFrame interface PAC, which prints the status of the store agent so the user can follow the progress of the transaction. The argument for the PrintStatus action is constructed using the built-in Concat function, which concatenates two strings into a single string.

The next two THEN patterns modify fields in the KQML message that was tested in the WHEN section. The first pattern fills in the price field of the PriceQuote content object, using a value from the same InventoryRecord object used in patterns 4 and 5 in the WHEN section. Next, the storeName field in the PriceQuote content object is filled in using the agent's belief about its own name, taken from the SELF Agent belief. The SELF belief is automatically added to the agent's mental model by the run-time system.

After the message has been modified, it is sent back to the buyer agent via the built-in action SendKqmlMessage. This action takes several arguments such as message, sender, receiver, performative, etc. (The ordering of the arguments may be difficult to understand in a printed example such as this but they are easy to specify in the Rule Editor.)

In this example the received message is being sent back to the buyer, the sender is now the store agent (i.e., SELF.name), the receiver will be the original sender (i.e., %message.sender), the performative is "tell", and the reply-with field of the original message is used to set the in-reply-to field of the new outgoing message. The empty argument slots indicate that all other fields in the message should be left as they are (the content object has already been changed from its original values by the SET_VALUE_OF statements).

Finally, after the content object has been updated and the message sent back to the buyer agent, the store agent goes to sleep by invoking the built-in SleepUntilMessage action. The store agent will automatically be re-awakened by its controller when another message is received.

Intentions

An *intention* is an agreement, usually communicated to another agent, to achieve a particular state of the world at a particular time. Intentions are similar to commitments in that one agent performs action(s) on behalf of another. However, a commitment is an agreement to perform a single action whereas an intention is an agreement to perform whatever actions are necessary to achieve a desired state of the world.

Requests containing intentions allow agents to communicate in terms of high-level goals and allow the receiving agent (the one who adopts the intention) the freedom to achieve the state of the world using whatever actions are appropriate for that agent. This is more efficient and more robust than requesting a specific sequence of commitments. The agent performing the actions has a better understanding of its own area of expertise within the problem domain and may be able to skip unneeded actions, redo actions when necessary, find alternate actions that will achieve the goal, etc.

In order to achieve the goal specified by an intention, the agent must be able to construct plans to achieve that goal, monitor the success of the actions performed, and construct alternate plans if the original plan fails. Thus, support for processing intentions requires an additional level of sophistication and capability in the agent.

Agent Interpreter

Figure 5 illustrates the AgentBuilder intelligent agent architecture. In this architecture an interpreter continually monitors incoming messages, updates the agent's mental model and takes appropriate actions.

At start-up, an agent is initialized with initial beliefs, initial commitments, initial intentions, capabilities, and behavioral rules. A non-trivial agent requires at least one behavioral rule; the other elements

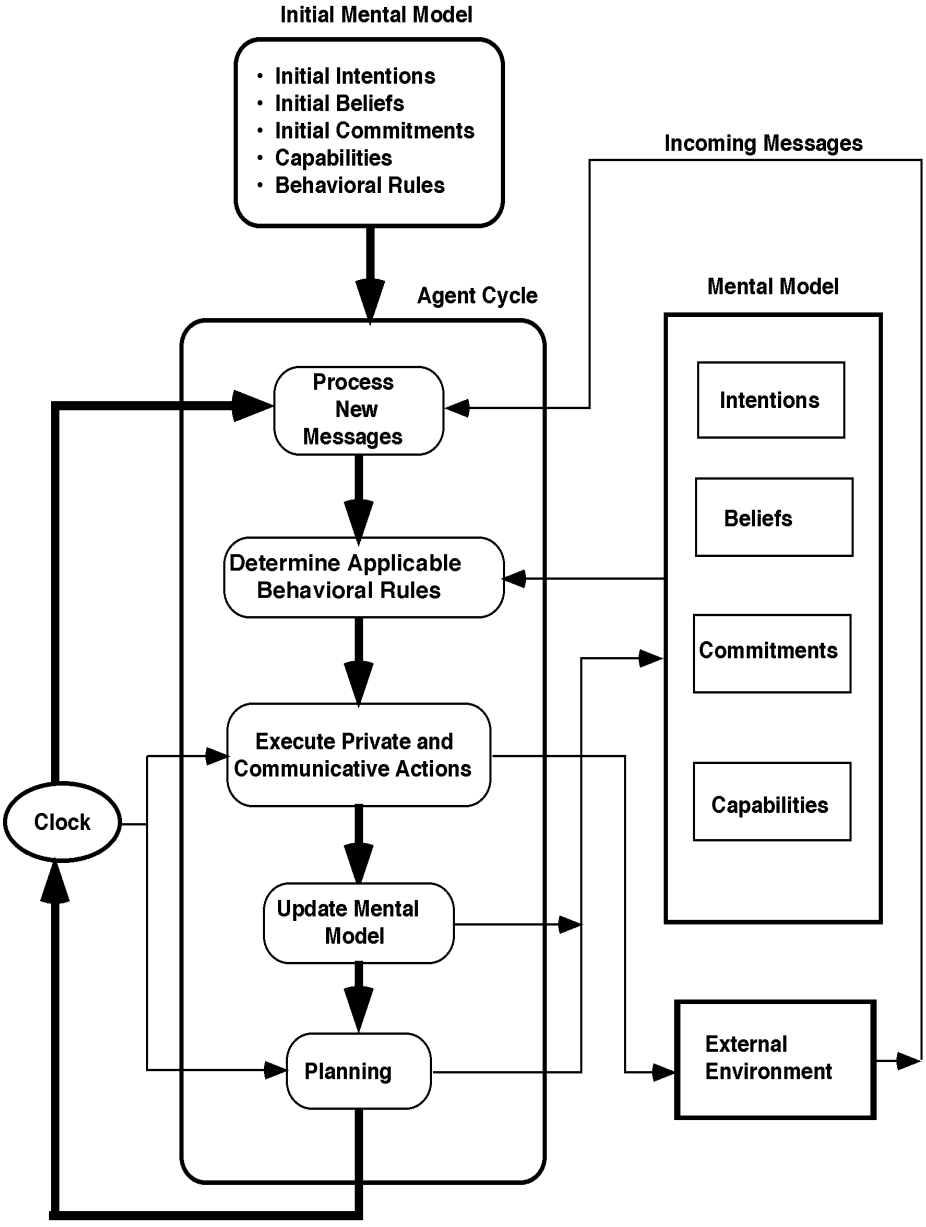

Figure 5. Agent Execution Process

are optional. For example, if an agent has no initial commitments then the agent is not initially committed to doing anything (for itself or anyone else). The same logic applies to initial beliefs and initial intentions. If the capabilities list is empty the agent will not be able to perform any actions.

The mental model contains the current beliefs, commitments, intentions, capabilities, and rules of the agent. Although rules and capabilities are static, the agent's beliefs, commitments, and intentions are dynamic and can change over the agent's lifetime.

The agent execution cycle consists of the following steps:

- processing new messages
- determining which rules are applicable to the current situation
- executing the actions specified by these rules
- updating the mental model in accordance with these rules
- planning

Processing a new message requires identifying the sender and determining the sender's authenticity; then the message is parsed and made part of the mental model. The next step is determining which rules match the current situation. Pattern matching compares the elements of the mental model with conditional patterns in the behavioral rules to determine which rules are satisfied. A rule is marked for execution when all of its conditions are satisfied; the rule is then placed on the agent's *agenda* for execution. Rule execution consists of performing private and communicative actions and making mental changes. During execution, all of the actions (private and communicative) are executed sequentially.

Next, the agent's mental model is updated by adding mental elements (assertions) or removing mental elements (retractions) as specified by the executing behavioral rules. The final step in the cycle requires developing a plan for the agent. Planning is performed by a planning

module attached to the agent. An agent's planning module must develop plans that satisfy goals specified by the agent's intentions.

KQML

The Knowledge Query and Manipulation Language (KQML) is a high-level language intended to support interoperability among intelligent agents in distributed applications. It is both a message format and a message-handling protocol to support run-time knowledge-sharing among agents. KQML is an *interlingua*, a language that allows an application program to interact with an intelligent system. It can also be used for sharing knowledge among multiple intelligent systems engaged in cooperative problem solving. This language, originally developed as part of a DARPA Knowledge Sharing initiative, is becoming a *de facto* standard for interagent communications languages [Finin, *et al*, 1994a; Finin, *et al,* 1994b; Finin, *et al*, 1994c; Labrou *et al*, 1994; Labrou, 1996].

A KQML message consists of a performative, the content of the message, and a set of optional arguments. The performative specifies an assertion or a query used for examining or changing a Virtual Knowledge Base (VKB) in the remote agent. A listing of the defined performatives for KQML [Labrou, 1996] is provided in Table 3.

AgentBuilder agents provide support for all of the performatives specified for the Knowledge Query and Manipulation Language (KQML). Agents constructed with AgentBuilder can communicate and interoperate with any other agent that also "speaks" KQML. Thus, AgentBuilder agents can communicate with existing agents or agents constructed using other methods, tools, and/or architectures other than those supported by AgentBuilder.

Table 3. Summary of Reserved Performatives

Name	Meaning for Sender S and Recipient R with Virtual Knowledge Base (VKB)
achieve	S wants R to make something true of its physical environment
advertise	S wants R to know that S can and will process a message like the one in :content
ask-one	S wants one of R's instantiations of the :content that is true of R
ask-all	S wants all of R's instantiations of the :content that is true of R
ask-if	S wants to know if the :content is in R's VKB
broadcast	S wants R to send a message to all agents that R knows of
broker-all	S wants R to find all responses to a <performative> (some agent other than R is going to provide that response)
broker-one	S wants R to find one response to a <performative> (some agent other than R is going to provide that response)
delete-all	S wants R to remove all matching sentences from its VKB
delete-one	S wants R to remove one matching sentence from its VKB
deny	the negation of the sentence is in S's VKB
discard	S will not want R's remaining responses to a previous multi-response message
error	S considers R's earlier message to be malformed
eos	the end of stream marker to a multiple-response (stream-all)
forward	S wants R to forward the message to the :to agent (R might be that agent)
insert	S asks R to add the :content to its VKB
next	S wants R's next response to a message previously sent by S
ready	S is ready to respond to a message previously received from R
recommend-all	S wants to learn of all agents who can respond to a <performative>
recommend-one	S wants to learn of an agent who can respond to a <performative>
recruit-all	S wants R to get all suitable agents to respond to a <performative>
recruit-one	S wants R to get one suitable agent to respond to a <performative>
register	S announces to R its presence and symbolic name
rest	S wants R's remaining responses to a previously sent by S

Table 3. Summary of Reserved Performatives

Name	Meaning for Sender S and Recipient R with Virtual Knowledge Base (VKB)
`sorry`	S understands R's message but cannot provide a more informative reply
`standby`	S wants R to announce its readiness to provide a response to the message in `:content`
`stream-all`	multiple-response version of `ask-all`
`subscribe`	S wants updates to R's response to a performative
`tell`	the sentence in S's VKB
`transport-address`	S associates its symbolic name with a new transport address
`unachieve`	S wants R to reverse the act of a previous achieve
`undelete`	S wants R to reverse the act of a previous delete
`uninsert`	S wants R to reverse the act of a previous insert
`unregister`	S wants R to reverse the act of a previous `register`
`untell`	the sentence is not in S's VKB

Chapter 3

Agent Communications Languages

Chapter Overview

You can find the following information in this chapter:

- An Introduction to KQML
- KQML Semantics
- KQML Parameters
- KQML Performatives

A. KQML

The *Knowledge Query and Manipulation Language* (KQML) is a language and a protocol that supports network programming specifically for knowledge-based systems or intelligent agents.

KQML is primarily concerned with pragmatics and, secondarily, with semantics. Pragmatics among computer processes includes knowing who to talk with and how to find them, as well as knowing how to initiate and maintain an exchange. KQML is a language and a set of protocols which support computer programs in identifying, connecting to, and exchanging information with other programs [Finin et al., 1994b]. We discuss the specifications and implications of the KQML language in the following sections.

KQML Language Description

KQML is a high-level language intended to support interoperability among intelligent agents in distributed applications. It is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML can be used as a language for an application program to interact with an intelligent system. It can also be used for two or more intelligent systems to share knowledge in support of cooperative problem solving.

Layer of Communication

KQML is most useful for communication among agent-based programs, in the sense that the programs are autonomous and asynchronous. Autonomy entails that agents may have different, and even conflicting, agendas. Thus, the meaning of a KQML message is defined in terms of constraints on the message sender rather than the message receiver. This allows the message receiver to choose a course of action that is compatible with other aspects of its function. Of course, most useful agent architectures strive for maximal

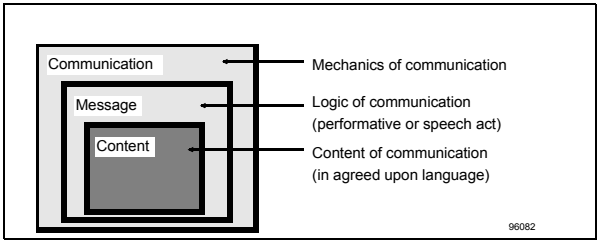

Figure 6. The Three Layers of the KQML Language

cooperation among agents, but complete cooperation is not always possible.

There are several levels at which agent-based systems must agree, at least in their interfaces, in order to successfully interoperate [Finin et al., 1994c]:

- *Transport* – how agents send and receive messages
- *Language* – what the individual messages mean
- *Policy* – how agents structure conversations
- *Architecture* – how to connect systems in accordance with constituent protocols

Conceptually, the KQML language involves only three main layers. The Transport and Architecture layers, which the agents are concerned with separately, can be collapsed into one layer with KQML. Thus, the KQML language can be viewed as being divided into three layers: the *content layer*, the *message layer* and the *communication layer,* as illustrated in Figure 6.

The *content layer* is the actual content of the message in the program's own representation language. KQML can carry any representation language, including languages expressed as ASCII strings and those expressed using a binary notation. All of the KQML

implementations ignore the content portion of the message except to the extent that they need to determine its boundaries.

The *communication layer* encodes a set of features to the message which describe the lower level communication parameters, such as the identity of the sender and recipient, and a unique identifier associated with the communication.

The *message layer* forms the core of the language. It determines the kinds of interactions one can have with a KQML-speaking agent. The primary function of the message layer is to identify the protocol to be used to deliver the message and to supply a speech act, or performative, which the sender attaches to the content. The performative signifies that the content is an assertion, a query, a command, or any of a set of known performatives. Because the content is opaque to KQML, this layer also includes optional features which describe the content: its language, the ontology it assumes, and some type of more general description, such as a descriptor naming a topic within the ontology. These features make it possible for KQML implementations to analyze, route, and properly deliver messages even though their content is inaccessible.

A KQML message consists of a performative, its associated arguments which include the real content of the message, and a set of optional arguments. The main focus of KQML is on its extensible set of performatives, which defines the permissible operations that agents may attempt on each other's knowledge and goal stores at run time. The performatives comprise a substrate on which to develop higher-level models of inter-agent interaction such as contract nets and negotiation.[1]

The optional arguments of the KQML message describe the content in a manner which is independent of the syntax of the content

1. http://www.cs.umbc.edu/kqml/whats-kqml.html

language. For example, a message representing a query about the location of a particular airport might be encoded as:

```
(ask-one     :content (geoloc lax (?long
             ?lat))
             :ontology geo-model3)
```

In this message, the KQML performative is `ask-one`, the content is `(geoloc lax (?long ?lat))` and the assumed ontology is identified by the token `:geo-model3`. The same general query could be conveyed using standard Prolog as the content language in a form that requests the set of all answers as [Finin et al., 1994b]:

```
(ask-all     :content
             "geoloc(lax,[Long,Lat])"
             :language standard_prolog
             :ontology geo-model3)
```

In addition, KQML provides a basic architecture for knowledge sharing through a special class of agents called *communication facilitators* which coordinate the interactions of other agents.

KQML String Syntax

A KQML message is expressed as an ASCII string using the syntax defined in this section. This syntax is a restriction on the ASCII representation of Common LISP Polish-prefix notation.

The ASCII-string LISP list notation was originally chosen because it is readable by humans, simple for programs to parse, particularly for many knowledge-based programs, and transportable by many inter-application messaging platforms. However, no choice of message syntax will be both convenient and efficient for all messaging APIs.

Unlike LISP function invocations, parameters in performatives are indexed by keywords and therefore are order independent. These

keywords, called *parameter names,* must begin with a colon (:) and must precede the corresponding parameter value. Performative parameters are identified by keywords rather than by their position due to a large number of optional parameters to performatives [Finin et al., 1994c].

The BNF for KQML, given in Figure 7, assumes definitions for <ascii>, <alphabetic>, <numeric>, <double-quote>, <backslash>, and <whitespace>. "*" means any number of occurrences, and "–" indicates set difference. Note that <performative> is a specialization of <expression>. Also note that in length-delimited strings, e.g. "#3"abc", the whole number before the double-quote specifies the length of the string after the double-quote.

KQML Semantics

The semantic model underlying KQML is a simple and uniform context for agents to view each others' capabilities. Each agent appears as if it manages a *knowledge base* (KB). That is, communication with the agent is with regard to this KB base. For example, there are questions about what a KB contains, statements about what a KB contains, requests to add or delete statements from the KB, or requests to use knowledge in the KB to route messages to appropriate other agents.

The implementation of an agent is not necessarily structured as a knowledge base. The implementation may use a simpler database system or a program using a special data structure as long as wrapper code translates that representation into a knowledge-based abstraction for the benefit of other agents. Thus, we say that each agent manages a *virtual knowledge base* (VKB).

When defining performatives, it is useful to classify the statements in a VKB into two categories: *beliefs* and *goals*. An agent's *beliefs* encode information it has about itself and its external environment, including the VKBs of other agents. An agent's *goals* encode states

```
<performative> ::= (<word> {<whitespace> :<word> <whitespace> <expression>}*)

<expression> ::= <word> | <quotation> | <string> |
                 (<word> {<whitespace> <expression>}*)

<word> ::= <character><character>*

<character> ::= <alphabetic> | <numeric> | <special>

<special> ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
              @ | $ | % | : | . | ! | ?

<quotation> ::= '<expression> | `<comma-expression>

<comma-expression> ::= <word> | <quotation> | <string> | ,<comma-expression>
                       (<word> {<whitespace> <comma-expression>}*)

<string> ::= "<stringchar>*" | #<digit><digit>*"<ascii>*

<stringchar> ::= \<ascii> | <ascii>-\-<double-quote>
```

NOTE: In length-delimited strings, e.g. "#3"abc", the whole number before the double-quote specifies the length of the string after the double-quote.

Figure 7. KQML String Syntax in BNF

of its external environment that the agent will act to achieve. Performative definitions make reference to either or both of an agent's goals and beliefs, e.g., that the agent wants another agent to send it a certain class of information. The English-prose performatives make reference to these terms, but this view of the VKB is especially important in the formal semantics of KQML [Finin et al., 1994c].

Agents talk about the contents of their VKB and others' VKBs using KQML, but the encoding of statements in VKBs can use a variety of representation languages. That is, the KQML performative "tell" is used to specify that a particular string is contained in

an agent's belief store, but the encoding of that string can be a representation language other than KQML. This is the content layer of the language which was discussed previously.

The only restrictions on the specific representations are that they be sentential. Expressions using the given representation can be viewed as entries in a VKB. In addition, sentences have an encoding as an ASCII string so that sentences can be embedded in KQML messages.

KQML Parameters

Performatives take parameters identified by keywords. The structure of parameters was briefly discussed in the syntax section of this document. The following sections define the meaning of some common performative parameters by coining their keywords and describing the meaning of the accompanying values. Since the following parameters are used heavily, this will facilitate brevity in the performative definitions.

The specification of reserved parameter keywords is useful in two main ways. First, it is useful to mandate some degree of uniformity on the semantics of common parameters to reduce programmer confusion. Also, it is useful to support some level of understanding of performatives with unknown names but with known parameter keywords. The following parameters are reserved in the sense that any performative's use of parameters with those keywords must be consistent with the definitions below. The reserved parameter keywords are summarized in Table 4 on page 64.

```
:sender <word>
:receiver <word>
```

These parameters convey the actual sender and receiver of a performative, as opposed to the virtual sender and receiver in the `:from` and `:to` parameters of networking performatives.

```
:reply-with <expression>
:in-reply-to <expression>
```

If the <expression> is the word "nil" or this parameter is absent from the performative, then the sender does not expect a reply. If the <expression> is the word "t" then the sender expects a reply. Otherwise, the sender expects a reply containing a :in-reply-to parameter with a value identical to <expression>.

```
:content <expression>
:language <word>
:ontology <word>
```

The :content parameter indicates the "direct object" (in the linguistic sense) of the performative. For example, if the performative name is "tell" then the :content will be the sentence being told. The <expression> in the :content parameter must be a valid expression in the representation language specified by the :language parameter, or KQML if the :language parameter does not appear. Furthermore, the constants used in the expression must be a subset of those defined by the ontology named by the :ontology parameter, or the standard ontology for the representation language if the :ontology parameter does not appear.

Both :language and :ontology are restricted to only take <word> as a value, and therefore complex terms (e.g., denoting unions of ontologies), are not allowed. Eventually, it will be possible to support a calculus of ontologies and languages, but its proper place is in performatives that define new KQML names. This way, only those agents that can process extensional performatives are expected to understand such a calculus.

```
:force <word>
```

If the value of this parameter is the word "`permanent`," then the sender guarantees that it will never deny the meaning of the performative. Any other value indicates that the sender may deny the meaning in the future. This parameter works to help agents avoid unnecessary truth-maintenance overhead. The default value is "`tentative`."

Table 4. Summary of Reserved Parameter Keywords and their Meanings

Keyword	Meaning
`:content`	the information about which the performative expresses an attitude
`:force`	whether the sender will ever deny the meaning of the performative
`:in-reply-to`	the expected label in a reply
`:language`	the name of representation language for the `:content` parameter
`:ontology`	the name of the ontology (e.g., set of term definitions) used in the `:content` parameter
`:receiver`	the actual receiver of the performative
`:reply-with`	whether the sender expects a reply, and if so, a label for the reply
`:sender`	the actual sender of the performative

KQML Performatives[1]

A KQML message is called a *performative*. The term is from Speech Act Theory because the message is intended to *perform* some action by virtue of being sent. This document defines a substantial number of performatives in terms of what they connote about the sender's knowledge. However, the performatives defined

1. The list of KQML performatives is documented in [Finin et al., 1994c]: http://www.cs.umbc.edu/kqml/kqmlspec/spec.html

herein are neither necessary nor sufficient for all agent-based applications. Therefore, agents need not support the entire set of defined performatives. A majority of the agents will only support a small subset. In addition, agents may use performatives that do not appear in this specification. New performatives should be defined precisely as specified in a later section.

Table 5. Summary of Reserved Performatives

Name	Meaning for Sender S and Recipient R with Virtual Knowledge Base (VKB)
achieve	S wants R to make something true of its physical environment
advertise	S wants R to know that S can and will process a message like the one in :content
ask-one	S wants one of R's instantiations of the :content that is true of R
ask-all	S wants all of R's instantiations of the :content that is true of R
ask-if	S wants to know if the :content is in R's VKB
broadcast	S wants R to send a message to all agents that R knows of
broker-all	S wants R to find all responses to a <performative> (some agent other than R is going to provide that response)
broker-one	S wants R to find one response to a <performative> (some agent other than R is going to provide that response)
delete-all	S wants R to remove all matching sentences from its VKB
delete-one	S wants R to remove one matching sentence from its VKB
deny	the negation of the sentence is in S's VKB
discard	S will not want R's remaining responses to a previous multi-response message
error	S considers R's earlier message to be malformed
eos	the end of stream marker to a multiple-response (stream-all)
forward	S wants R to forward the message to the :to agent (R might be that agent)
insert	S asks R to add the :content to its VKB
next	S wants R's next response to a message previously sent by S
ready	S is ready to respond to a message previously received from R
recommend-all	S wants to learn of all agents who can respond to a <performative>

Table 5. Summary of Reserved Performatives

Name	Meaning for Sender S and Recipient R with Virtual Knowledge Base (VKB)
recommend-one	S wants to learn of an agent who can respond to a <performative>
recruit-all	S wants R to get all suitable agents to respond to a <performative>
recruit-one	S wants R to get one suitable agent to respond to a <performative>
register	S announces to R its presence and symbolic name
rest	S wants R's remaining responses to a previously sent by S
sorry	S understands R's message but cannot provide a more informative reply
standby	S wants R to announce its readiness to provide a response to the message in :content
stream-all	multiple-response version of ask-all
subscribe	S wants updates to R's response to a performative
tell	the sentence in S's VKB
transport-address	S associates its symbolic name with a new transport address
unachieve	S wants R to reverse the act of a previous achieve
undelete	S wants R to reverse the act of a previous delete
uninsert	S wants R to reverse the act of a previous insert
unregister	S wants R to reverse the act of a previous register
untell	the sentence is not in S's VKB

The performative names are reserved. An application is not KQML-compliant if it uses these performatives in ways that are inconsistent with the definitions given by the KQML developers and outlined in this document. These reserved performatives should be used when possible to increase overall interoperability.

All performatives are described in detail in Appendix 1, "KQML Performatives" on page 281.

New Performatives

The primary dimension of KQML extension is through the definition of new performatives. The definitions of new performatives must explicitly describe all permissible parameters and default values for parameters that do not appear in particular messages. A performative definition may coin new parameter names. [Finin et al., 1994c].

Definitions of new performatives should follow the style of the definitions in this section. A definition should convey the following [Finin et al., 1994c]:

- the performative name;
- all parameters keywords that the performative may contain;
- syntactic categories and semantics for all values of parameters with non-reserved keywords;
- any additional syntactic and semantic constraints for values of parameters with reserved keywords;
- the default values of all absent parameters;
- the semantics, in terms of a statement the sender is making of itself, of the performative name applied to the parameters.

B. KQML Conclusions

KQML is a language and associated protocol by which intelligent software agents can communicate to share information and knowledge. We believe that KQML will be important in building the distributed agent-oriented information systems of the future.

KQML offers an abstraction of an information agent (provider or consumer) at a higher level than is typical in other areas of computer science. In particular, KQML assumes a model of an agent as a knowledge-based system (KBS) [Finin et al., 1994b]. The KBS model easily subsumes a broad range of commonly used information agent models, including database management systems, hypertext systems, server-oriented software (e.g. finger daemons, mail servers, HTML servers, etc.), simulations, and more. Such systems can usually be modeled as having two virtual knowledge bases; one represents the agent's information store (i.e., beliefs), and the other represents its intentions (i.e., goals).

We are hopeful that future standards for interchange and interoperability languages and protocols will be based on this very powerful and rich model. This will avoid the built-in limitations of more constrained models (e.g., that of a simple remote procedure call or relational database query) and also make it easier to integrate truly intelligent agents with simpler and more mundane information clients and servers.

KQML has something it seeks from distributed systems work. This involves the right abstractions and software components to provide basic communication services. Current KQML-based systems have been built on the most common transport layers, mainly TCP/IP. The real contributions that KQML makes are independent of the transport layer. KQML interface implementations will be based on whatever is seen as the best transport mechanism.

The contribution that KQML makes to Distributed AI (DAI) research is to offer a standard language and protocol that intelligent agents can use to communicate among themselves as well as with other information servers and clients. Permitting agents to use whatever content language they prefer will make KQML appropriate for most DAI research. In the

continuation of the design of KQML it would be beneficial to build in the primitives necessary to support all of the interesting agent architectures currently in use. KQML should prove to be a good tool for DAI research, and, if used widely, should enable greater research collaboration among DAI researchers.

Chapter 4

Agent Development Process

Chapter Overview

You can find the following information in this chapter:

- Process Overview
- Project Management
- Domain Analysis
- Agency Definition
- Agent Behavioral Specification
- Agent Programming
- Debugging Agents and Agencies

A. The Process

As noted in the previous discussion, specifying an agent's mental model requires defining its initial beliefs, initial commitments, initial intentions, capabilities and behavioral rules. The key to building intelligent agents is having an efficient mechanism for specifying behavioral rules and other components of the mental model. AgentBuilder provides a graphical interface for easily and quickly defining a collection of agents and specifying their mental models and behaviors.

Developing an intelligent software agent is similar to other software development activities in that the software developer must perform the traditional steps of analysis, design, implementation, testing and debugging, integration, and maintenance. In many ways agent software development is similar to object-oriented software development.

A software developer using traditional object-oriented programming techniques must identify the objects of interest and specify the various interactions among those objects. The developer can define objects as very high-level abstractions (e.g., a bank account) or very low-level abstractions (e.g., a pushbutton in a graphical user interface).

In contrast, developing intelligent software agent programs (sometimes called *agent-oriented programming*) consists of identifying the roles and functions of various agents and then specifying each agent's behavior. Agent-oriented programming is very similar to object-oriented programming except the software developer works with complex entities (agents) at much higher levels of abstraction than is normally done in object-oriented programming. It is much easier for a software developer to develop complex software and systems using these high levels of abstraction.

An *agent-programming language* is a high-level language used to specify the behavior of the agent for any given situation. The AgentBuilder toolkit provides an object-oriented language called RADL (Runtime Agent Definition Language) to create agent programs.

An *agent engine* (i.e., agent execution environment) is required for executing the agent program. This engine must be able to execute on a wide variety of platforms, provide high performance, and support creation of sophisticated agent-based application programs.

AgentBuilder provides facilities for creating agent-based applications programs and includes tools for:

- organization and control of the development project
- problem domain analysis
- agency architecture definition
- specifying agent behavior
- viewing and debugging executing agents

The following paragraphs describe this process in more detail. Figure 8 illustrates the process of intelligent agent construction using the AgentBuilder toolkit.

Organize Project

Developers of agent-based software create intelligent agents for a variety of uses and applications. The AgentBuilder tools allow the developer to organize projects and associate particular agents and collections of agents (i.e., agency) with those projects. Developers can reuse agents developed for one project on a related project.

Likewise, developers will reuse domain knowledge gained in the course of analyzing a particular domain. For example, a developer building an e-mail agent will develop an ontology for e-mail and

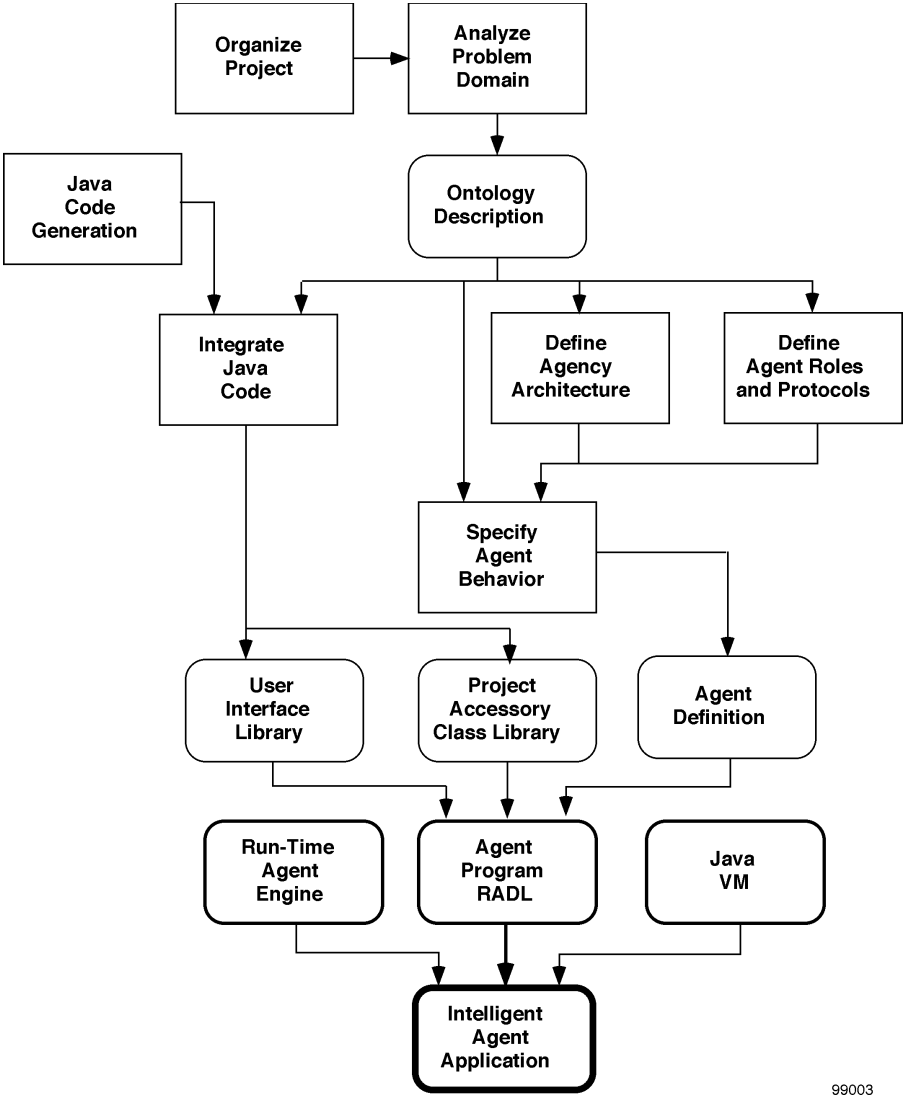

Figure 8. Constructing Intelligent Agents

can then reuse this e-mail ontology on a new project requiring development of a spam-filtering agent.

Analyze Problem Domain

The developer will need to perform an analysis of the problem domain in order to understand the functional and performance requirements of his agents and agent-based solution. AgentBuilder provides tools for analyzing and structuring the domain and codifying information about that domain. Domain analysis is facilitated using conceptual mapping tools and object modeling tools. The object model specifies all of the objects in the domain and the operations they can perform. Another product of the domain analysis is an ontology for that particular domain. This ontology is a formal description of the problem domain.

An ontology gives meaning to the symbols and expressions used to describe a domain. For one agent to properly understand the meaning of a message from another agent, both agents must ascribe the same meaning to the symbols (constants) used in that message. The ontology maps these symbols to a well-understood meaning for the problem domain [FIPA, 1997].

Define Agency Architecture

After completing the domain analysis, the software developer will normally decompose the problem into functions that can be performed by one or more intelligent agents. The developer must identify each agent and its role in solving the overall problem. The developer can then create a skeletal agent and define the basic characteristics of that agent with respect to its interaction with other agents.

After identifying the agents and their roles, the agent developer defines the interagent communication protocols. Protocol editors

are provided that make it easy for the developer to specify the messages and handshaking required between agents.

Specify Agent Behavior

After completing the agency definition, the developer then specifies the behavior of each agent. Agent development is the process of defining agent behavior. AgentBuilder provides tools for specifying behavioral rules, initial beliefs, commitments, intentions, and agent capabilities.

The AgentBuilder toolkit supports creation of a user interface library and an agent actions library and creates an agent definition file; the user interface and action classes comprise the Project Accessory Class (PAC) library. The user interface library can be used to construct the user interface for the agent. Although many agents will not require any interaction with the user, other agents will need information from the user and will provide the user with feedback information about the status of the agent's processing.

The developer specifies the actions of the agent in the agent actions library. For each agent action, the developer performs the following steps:

- defines the action name and parameter list
- associates the action with a method from an object defined in the object model
- imports existing class libraries or develops the Java classes that will implement the action
- stores these classes in the agent actions library

The agent definition file contains a detailed specification of the agent's initial mental model and behavior. This file is used with the agent actions library and the user interface library to fully specify

agent activities and behavior. These three components comprise the *agent program*.

Create Agent Application

The final step in the agent construction process consists of loading the agent program into the Run-Time Agent Engine. The agent engine is a high-performance execution mechanism that interprets the agent program and performs the actions specified in the user interface and agent actions libraries. An agent is composed of the agent program and the run-time engine. The Run-Time Engine is described in the next section of this document.

Agent and Agency Debugging

AgentBuilder provides tools that support all phases of the software agent development process. AgentBuilder provides a debugging environment to support debugging, testing, and integration of agents and agencies. In agent-oriented programming, no low-level source code debugging is necessary because the developer works with a high-level abstraction—the intelligent agent. However, a capability for high-level debugging of an agent's mental model is required.

The AgentBuilder agent debugger allows the developer to examine the agent's mental model. The developer can also step through the agent's operation cycle-by-cycle and examine the mental model as the agent executes. The developer can specify breakpoints and run the agent until a breakpoint is encountered.

Part II. Getting Started with AgentBuilder

The purpose of this section is to demonstrate how to use AgentBuilder to construct several example agents by providing step-by-step instructions. We recommend that you work through these examples before starting your own agent development work. Prior to starting this section, read the previous overview of the AgentBuilder tools. If you need more information about the tools and their operation as you work through this section, refer to the remaining chapters in this User's Guide.

We have also included an **Example Project** and several example agnecies in the System Repository. The agents in this example project are the same as the ones we will develop in this guided, step-by-step introduction to AgentBuilder. You can examine these pre-constructed agents at your leisure. However, we strongly recommend that you work through the examples described in this section.

While all of the agents described in this section are simple, these step-by-step exercises for constructing them will provide you with the hand's-on experience and training you need to get started building your own agents. This section covers most of the processes involved in building agents including construction of object models, construction of basic agent behavioral rules, and execution and debugging your completed agent. This section shows you how to construct a number of related **Hello World** agents. Each successive agent has increased functionality and uses more advanced tools in the AgentBuilder toolkit.

In this step-by-step introduction you will build:

- **ExampleAgent1** — agent with single rule that prints "Hello World" to the built-in console window.
- **ExampleAgent2** — agent that prints 'Hello World' to the console once every 10 seconds, and variations on this agent that demonstrate other aspects of AgentBuilder.

- **ExampleAgent3** — agent that demonstrates the incorporation of a PAC into rules and the run-time creation of objects.
- **ExampleAgent4** — agent that utilizes a GUI-based PAC and demonstrates message passing between the user interface and an agent.
- **SimpleBuyer** — agent that utilizes a GUI-based PAC and demonstrates how a buyer can purchase a product from the least expensive store agent.
- **SimpleSeller** — agent that utilizes a GUI-based PAC and demonstrates how a store can respond to a buyer agent's request for a product price.

Note that this section is not intended to completely describe all the built-in functions, PACs, objects, tools, etc. in AgentBuilder, nor is it intended to provide a tutorial on rule-based programming. You should consult other sections of the User's Guide and other references to increase your understanding in these areas. This section provides a starting point for acquainting you with many of the features of AgentBuilder and shows you how to get started building agents with this toolkit.

Chapter 5

Getting Started

Chapter Overview

You can find the following information in this chapter:

- A Brief Introduction to AgentBuilder
- Quick Tour
- Building Agents Step by Step

A. Introduction

This chapter provides an overview and introduction to the AgentBuilder toolkit. The first section provides a quick tour of the complete toolkit. This tour is designed to provide you with a brief introduction to the major components of AgentBuilder and introduce you to the agent construction process. This section is intended to provide a broad overview of the tool and give you an overall idea of the range of the tools, how they are used, and how they fit together. Subsequent chapters in this document explain each of the tools and the development process in more detail.

The AgentBuilder toolkit is a complex set of tools and requires an understanding of the general theory of intelligent agents as well as programming experience. While this guide cannot provide you with programming experience it does provide all of the necessary theory you will need to get started developing intelligent agents. After reading this section, you will want to try building your own agents. However, before doing so, we strongly urge you to read the rest of this guide.

As you explore AgentBuilder, you may discover unfamiliar tools or it may not be clear to you how to accomplish some task. Nearly all of your questions will be answered by reading the documentation in this user's guide. Throughout the quick tour, you may encounter cross references, terms, or concepts which are unfamiliar to you. We recommend that you keep reading. Later chapters will explain these concepts and terms in more detail.

Before starting, please read the following important section.

Menus, Combo-Boxes and Accumulators

AgentBuilder uses a variety of interface components. You should understand how to use menu bars and pop-up menus as well as the

functions of the various buttons on your mouse. In addition, AgentBuilder uses a "Combo-Box" which is a special graphical component that combines the features of a text edit field and a pull-down menu.

Building Complex Patterns with the Accumulator Paradigm

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 9 shows a portion of the **Mental Condition Rule Editor** used in AgentBuilder. It provides a good example of the use of the accumulator concept.

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

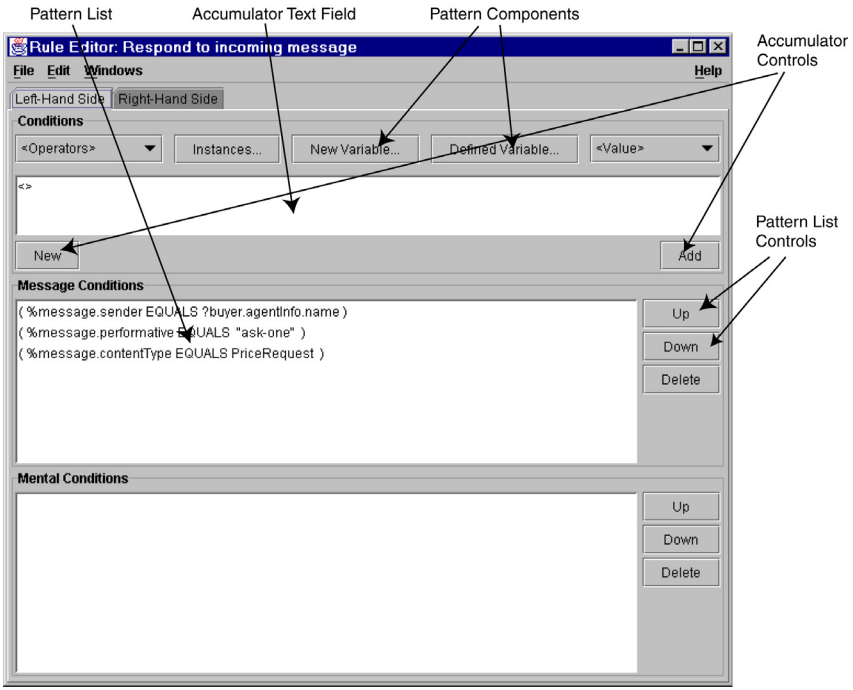

Figure 9. Accumulator Building Complex Patterns

move the pattern from the accumulator to the associated pattern list. Clicking on the **New** button will clear the accumulator text field.

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 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 (currently not implemented). 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.

Variable Naming Conventions

We recommend the following naming convention for variable names:

Use `%name` for KqmlMessage variables intended to bind to incoming messages, e.g., `%incomingMessage` or `%message`.

Use `?name` for any variables intended to bind to objects in the agent's mental model (including any *stored* KqmlMessage objects). For example, a variable intended to bind to any String objects in the mental model might be named `?string` or `?s`, or perhaps something even more descriptive such as `?bookTitle`. A variable intended to bind to KqmlMessage objects stored in the agent's mental model (as opposed to binding to incoming messages) might be named `?storedMessage`.

Use `$name` for temporary variables or return variables. Temporary variables store values, typically the result of a function, so they can easily be used throughout the rule. For example, a temporary variable `$price_plus_tax` might be used to store the value `(?price + (?price * 1.07))`. Return variables store the value returned from an action, e.g., `$myIPAddress = GetHostAddress()`. Temporary variables and return variables differ from regular variables in that regular variables get their values from the agent's mental model, temporary and return variables get their values from statements in a rule.

This name convention is only a suggestion; you're free to choose whatever variable names you wish. As you build complex rules the usefulness of a naming convention will become more apparent.

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.

B. Quick Tour

The way you start the Project Manager will depend on whether you are on a Windows or UNIX machine. On Windows, the Project Manager is accessible from the **Start** Menu. The default installation folder is **AgentBuilder Toolkit**. The folder will be different if you have specified an alternate folder under the **Start** Menu. Under the **AgentBuilder Toolkit** folder, there is an **AgentBuilder** icon. Selecting the **AgentBuilder** icon will start the Project Manager.

On UNIX, the AgentBuilder toolkit is installed in the directory where the system administrator unpacks the AgentBuilder files. Typically, the AgentBuilder toolkit will be installed in *`/usr/local/AgentBuilder`*. As long as the AgentBuilder `bin` directory is in your path, you will be able to start AgentBuilder by typing *`agentBuilder`* at the UNIX prompt. You must also ensure that the `AGENTBUILDER_HOME` environment variable is correctly specified if it is not installed in the default location.

The AgentBuilder Project Manager is the first tool you will encounter in using the toolkit. The Project Manager provides you with an overview of the agent development process and the agents you have under development. Clicking on an item in the tree panel on the left side of the tool will display a description for that item in the window on the right side of the tool. This window is called the *Description Panel.* There are three types of objects displayed in the left panel: projects, agencies, and agents. You can adjust the width of the two panels by selecting the vertical bar that separates the

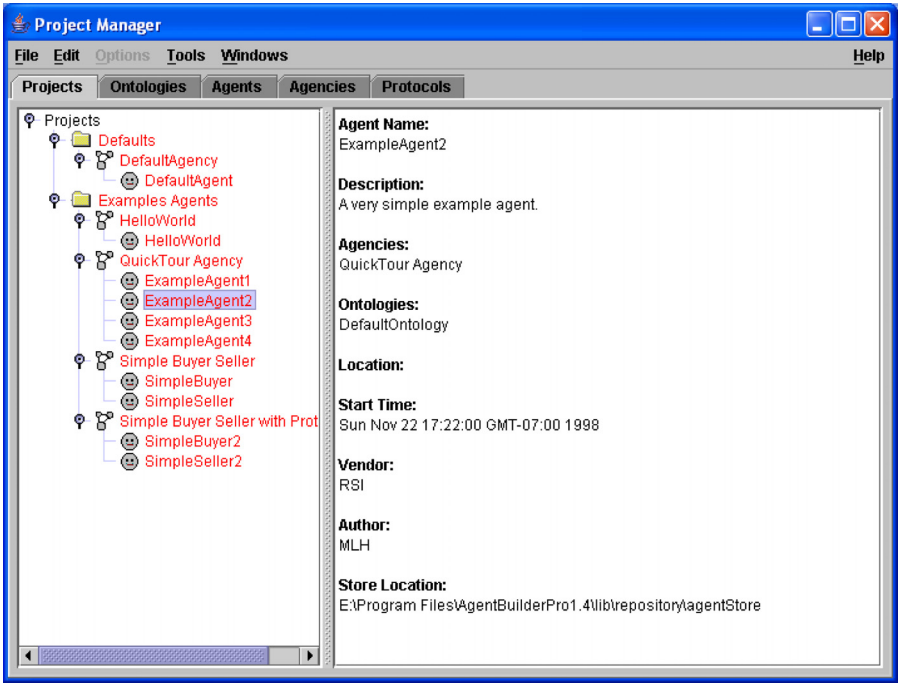

Figure 10. The AgentBuilder Project Manager

panels and dragging it to the left or right. Figure 10 shows the Project Manager.

The main use of the Project Manager is for creating and manipulating the items shown in the left panel. If you click on the **HelloWorld** agent icon in the tree, the agent's properties will be presented in the description panel.

Clicking on an agent and then selecting a tool causes the tool to start and use the selected item. For example, if you click on the **HelloWorld** agent (in the **HelloWorld Agency**) and then select the **Agent Manager** (by clicking on the **Agents** tab) you will open the **Agent Manager** and load the **HelloWorld** agent.

Some of the tools are not available in the AgentBuilder Lite version of AgentBuilder. For example, if you are using Lite you will not be able to use the Protocol and Agency Managers.

The tree is a hierarchical structure made up of projects, agencies defined for these projects, and the agents contained within each agency. Projects can only contain agencies, and agencies can only contain agents. All agents must be contained in an agency, including projects with only a single agent. This containment policy is enforced by the tool. To create new items in the tree, select the container object (i.e., the project or agency) and then select **New** from the **File** menu. You can do the same thing with a right mouse click on the item and then select **New Agent** from the pop-up menu. Both techniques have the same effect. The objects in red are defined as read-only; this means they are examples provided with the tool and cannot be modified. However, you will see how these agents can be copied and then pasted into you personal projects where you can modify them as desired.

The **Examples Agents** project folder you see contains agencies and agents constructed for you. These are provided for learning and demonstration. There are three example agencies: **Hello World Agency, Simple BuyerSeller Agency** and the **Quick Tour Agency**. Each of these contains example agents you can run and examine. It is useful to note that an agent can belong to one or more agencies. This means that the same agent may appear in multiple agency folders. As an agent is modified, the modifications are global in scope, i.e., they are applied across all agencies of which the agent is a member. This is sometimes confusing to new developers using the folder paradigm.We'll be examining the **HelloWorld** agents again in this Quick Tour.

The **Defaults**[1] project folder contains the default agency and default agent. The values of these are applied as default settings used by the system when creating new agents and agencies. Examining either of these will show you the defaults new agents are given at creation time. These defaults are configurable.

Selecting the **Edit → Properties** menu item will bring up a dialog which will allow you to modify the various properties of the tool. These properties include: the user's name, the user repository and directories for tool-generated output, error logging information, "look and feel," font size, and colors for the foreground and background. The user repository is a persistent store where all of the data structures created and used by the tool are stored.

There is also a right-click popup menu on the tree items that allows you to copy, past, delete, run or edit an item depending on the characteristics of the item.

1. The term *defaults* is used here in the sense of a default property value inherited from a parent class.

Ontology Manager

The Ontology Manager is designed to help you build ontologies for your agents. An ontology is a specification of a conceptualization and defines the domain knowledge needed by the agent to function in its environment. To start the Ontology Manager, click on the **Ontologies** tab while in the **Project Manager**. Figure 11 shows the Ontology Manager window.

The Ontology Manager features a tree view showing all of the ontologies you have created or imported. The Ontology Manager is similar to the Project Manager in that clicking on one of the ontology icons results in a display of information about the selected ontology in the description panel. This information includes: the developer's description of the ontology, the ontologies that use this

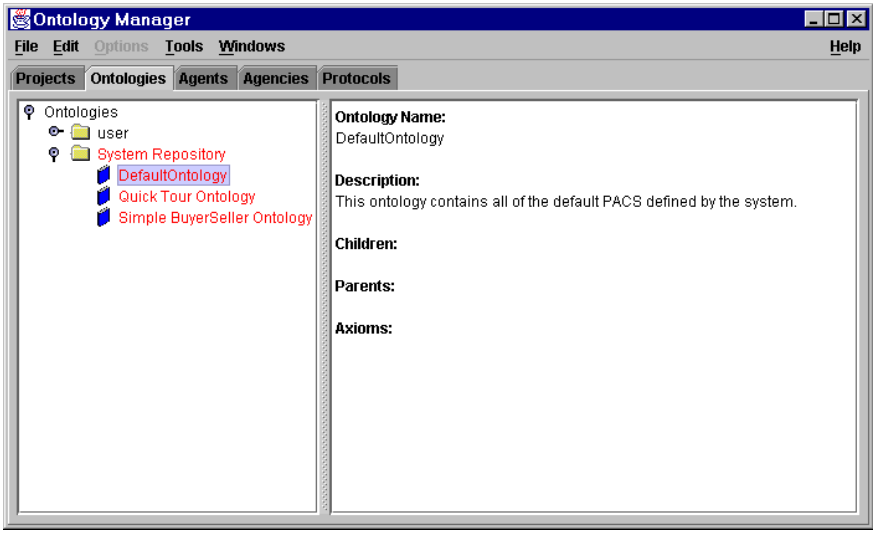

Figure 11. The Ontology Manager

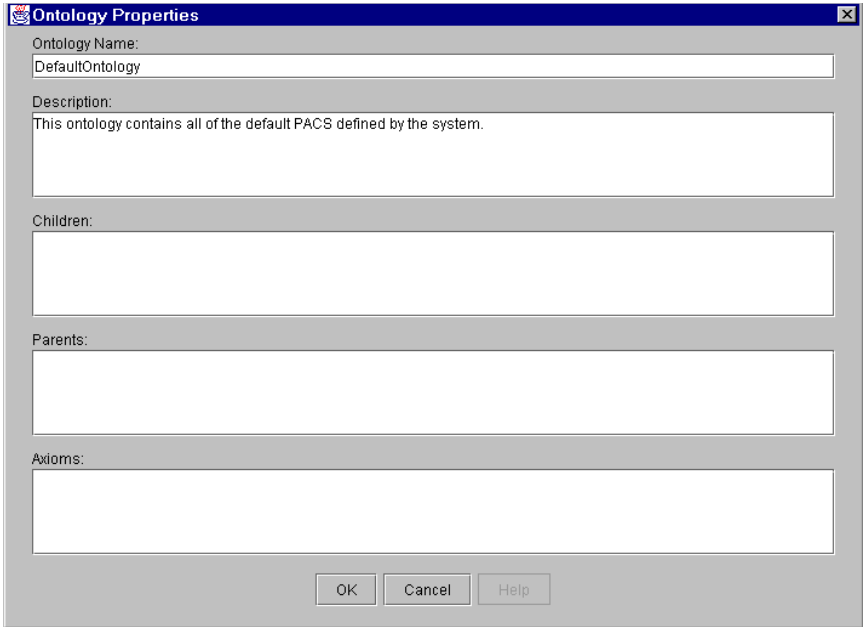

Figure 12. The Ontology Properties Panel

ontology, the ontologies used by this ontology, and all axioms defined for this domain.

Clicking on the **Hello World Ontology** causes the ontology properties to be displayed in the properties panel. To alter any of these properties, right-click on an ontology and click on **Properties…**. This will bring up the **Ontology Properties** dialog shown in Figure 12. All information shown in this dialog can be edited.

The **Ontology Properties** dialog allows the user to edit descriptive information about the ontology. When creating new ontologies you must supply (as a minimum) the ontology name. The children, parents, and axioms fields are described in the AgentBuilder Reference Manual.

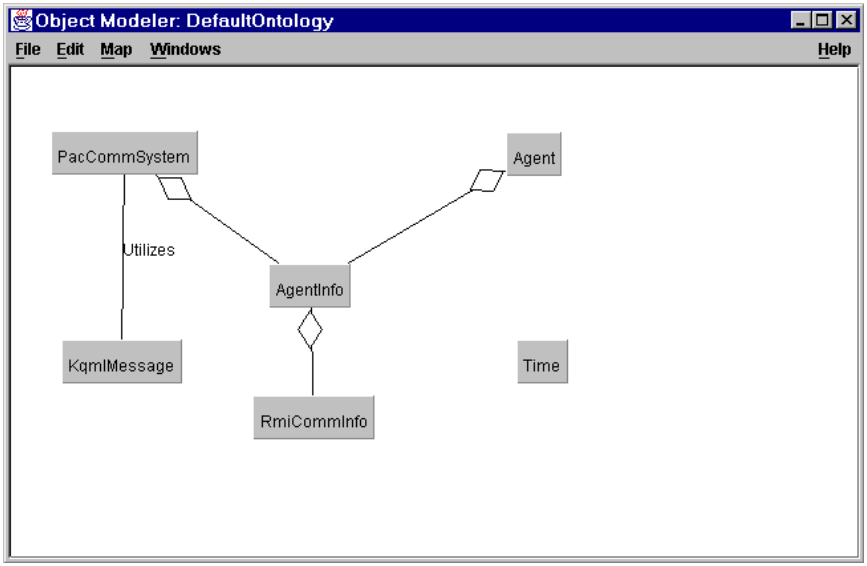

Figure 13. Object Modeler

Object Modeler

The Object Modeler is used for defining an object model for the agent application domain. This tool is used to aid in developing the classes for a particular domain. The classes represent entities or concepts in the agent's domain that the agent can instantiate and manipulate. The agent can invoke any of the methods specified for a class in the object model.

The Object Modeler can be started by clicking on the **Default Ontology** and then selecting the **Object Modeler** menu item (from the **Tools** menu). The Object Modeler tool will be displayed and will be preloaded with an object model for the selected ontology. The **Object Modeler** loaded with the **Default Ontology** object model is shown in Figure 13.

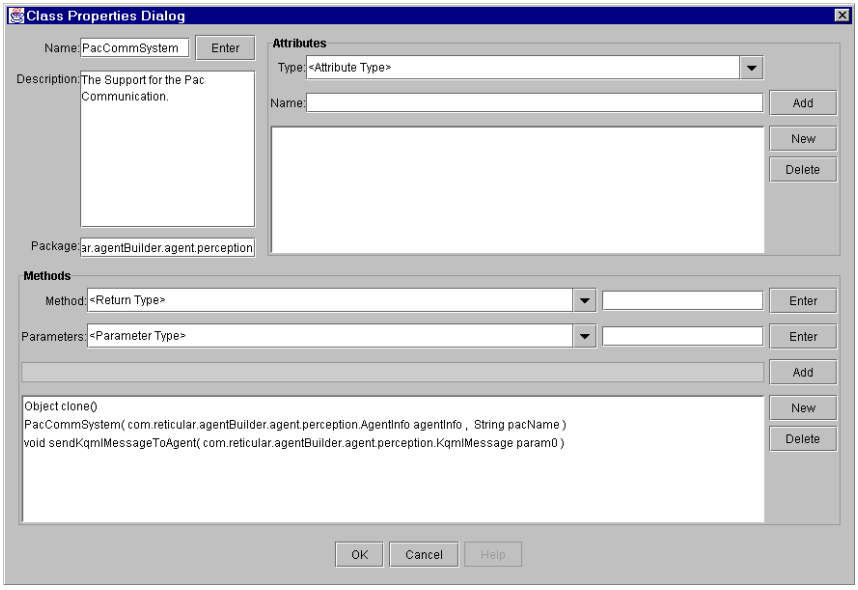

Figure 14. Class Properties for the Agent Class

Double-click on the **PacCommSystem** object and the **Object Properties** dialog will appear. As shown in Figure 14, all of the methods and attributes of the object are displayed. After you enter new attributes and methods in this panel the agent can use them. The object models built in the Object Modeler are imported into the Agent Manager and used to define the objects in the Agent's domain. Only the attributes and methods specified for a class in the object model are visible in the agent tools.

New objects are created by moving the cursor into the object model graphical window and right-clicking the mouse or by using the **New Object** menu item in the **Map** menu pull-down of the Object Modeler.

New objects can be created from existing Java class files. The Object Modeler assists the user in importing existing class files. To create an object based on an existing class file, click **File → Import Class Files…** and use the dialog to import classes by entering class and package names in their appropriate text areas and clicking **Add**. This allows you to skip the tedious work of entering object attributes as well as ensuring that the method and attribute naming is correct.

It should be noted that all attributes, parameters and return values need to use their fully qualified class names (with the exception of Java types).

The Ontology Manager also provides a Concept Mapper tool. This tool is used for defining concepts and conceptual relations and is discussed in detail in the AgentBuilder Reference Manual.

To close any of the tools, select **Close** from the **File** menu.

Agent Manager

The Agent Manager tool is used for constructing individual agents. Start the Agent Manager from the **Project Manager** window by selecting an agent (e.g., the **HelloWorld** agent) and then selecting the **Agents** tab. The Agent Manager tool is shown in Figure 15.

You can only run a single Agent Manager at any given time. However, you can run the Agent Manager on any agent that you define in the Project Manager. If you start the Agent Manager without selecting an agent first, the tool will open with no agent. At this point you can either create a new agent or load an existing one using the **File → New** or **File → Open…** menu items. You can safely modify any part of an agent without permanently changing it. The agent's definition is only saved if you use the **File → Save** command.

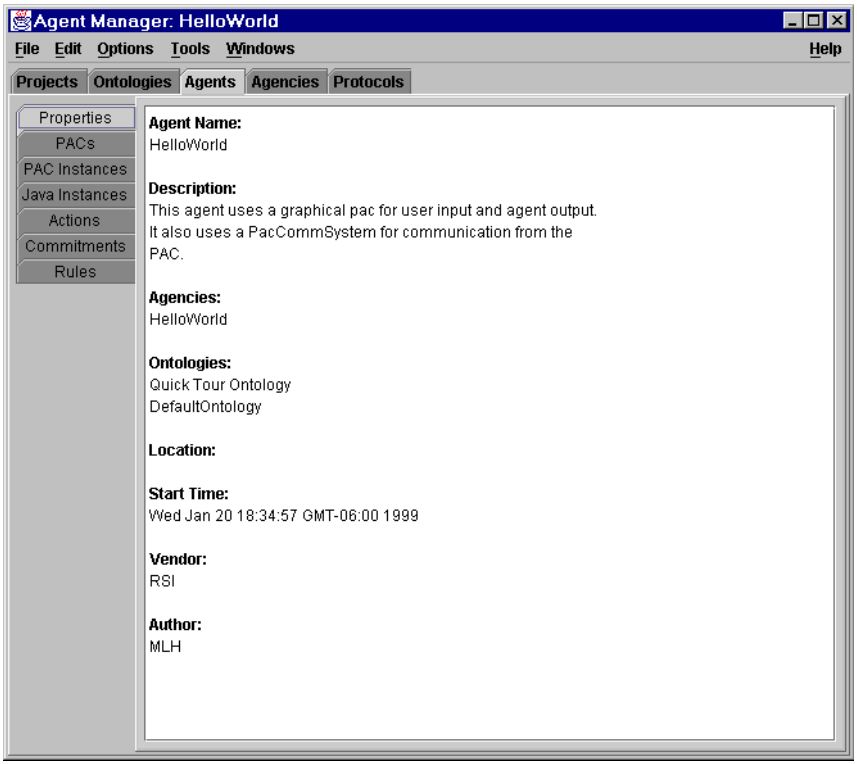

Figure 15. The Agent Manager

The Agent Manager is divided into several panels. The tabbed panel on the left displays all the components of an agent. You can examine the defined structures of the agent by clicking on the tab for each one. Tabs allow you to display but not directly edit Properties, PACs, PAC Instances, Actions and Rules.

The **Properties** panel shows high-level information about the agent. This includes the agent's name, description, agencies and ontologies used, network location, the date created, author, and vendor.

Some of this information is editable by clicking on the properties item in the **Edit** menu.

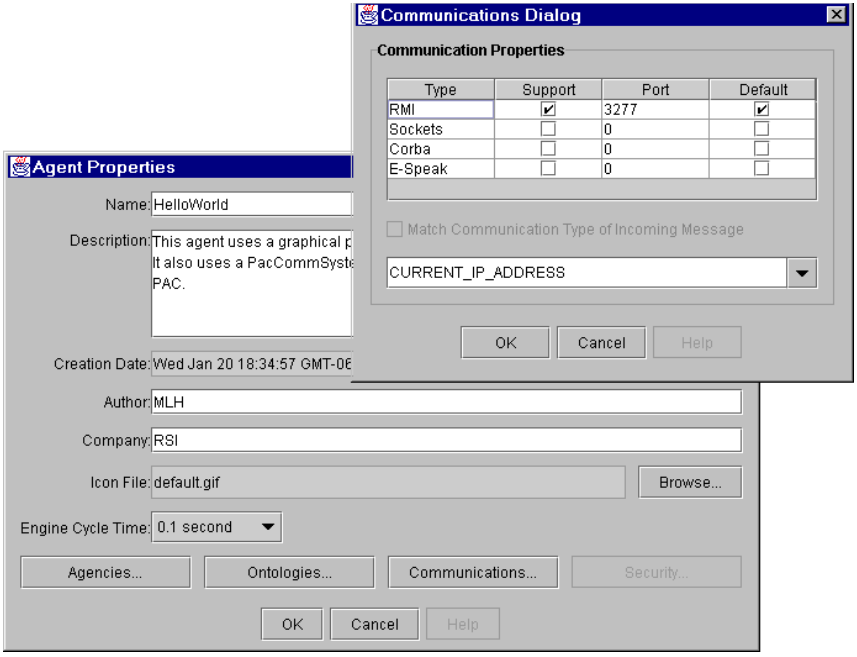

Figure 16. Agent Properties

Use the agent properties dialog for creating and defining a new agent. To create an agent, select **File → New** in the menu bar. The resulting dialog allows you to edit properties and select an agency, ontology and communication information for the agent. For example, clicking on the **Communications…** button will display the agent communication dialog. Figure 16 shows the **HelloWorld** communications dialog.

The agent **Communications Dialog** allows you to specify the type of communication to use and the specific attributes needed for that type of communication. For example, the **HelloWorld** agent is using

RMI (remote method invocation) at port 3932 and is using the IP address of the local host. By specifying the `CURRENT_IP_ADDRESS` the user is telling the agent to use the network address of the host computer. Each agent needs its own unique port number for communications.

To examine any of an agent's data items, click on the tab for it in the tab panel. Figure 17 shows what a rule looks like in the **Agent Manager** rule display. The **Print Greeting** rule is selected from the list of rules. This causes the **Print Greeting** rule to be displayed in the description panel. Double-clicking on an item brings up the appropriate editor for that item.

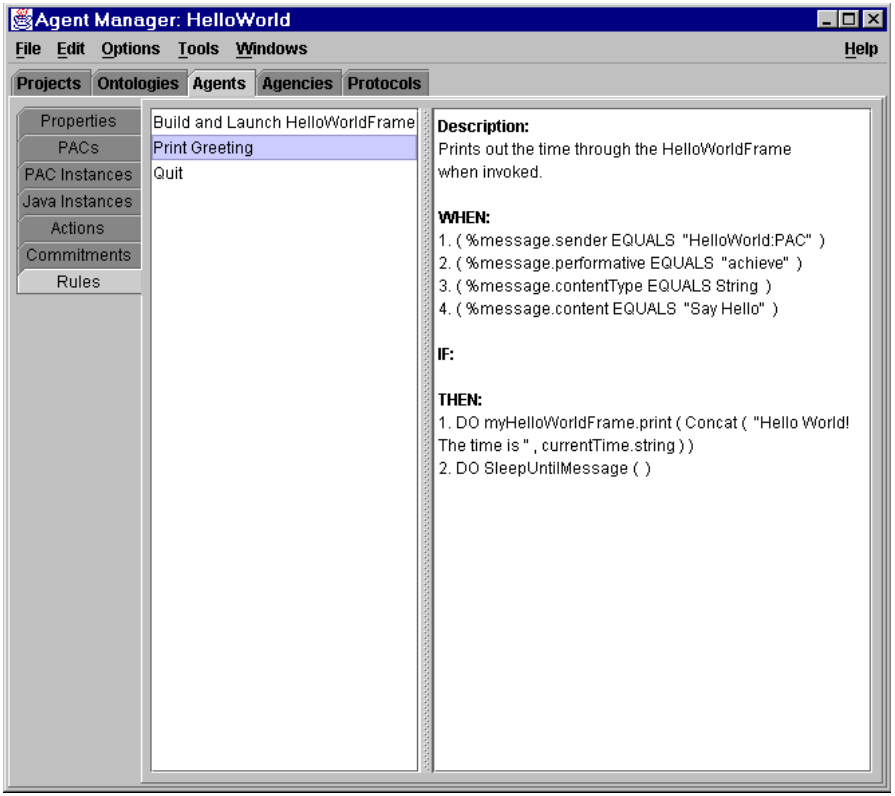

Figure 17. Agent Manager Rule Panel

PAC Editor

Project Accessory Classes (PACs) are classes used by the agents. PACs are analogous to the hands and eyes of an agent. They provide a mechanism that allows the agent to interact with its environment. The PAC Editor allows you to specify needed PACS and can be started from the Agent Manager by clicking on the **Tools → PAC Editor** menu or by selecting the **PAC** tab in the **Agent Manager** pane and double-clicking on a PAC name in the **Agent Manager** window. Figure 18 shows the basic **PAC Editor** panel that will be displayed.

The PAC Editor is used for defining and importing classes for the agent. The PACs represent Java classes in the agent's domain. The **HelloWorld** agent, for example, has several PACs. One of them is the `HelloWorldFrame` PAC. This PAC is a class that the agent can instantiate and manipulate. The agent can invoke any of the methods specified in the PAC definition.

All PACs are created from object models defined in an ontology. You can import objects from the ontologies by selecting **Import…** from the **File** menu in the PAC Editor. This pops up an **Import** dialog that is used to select the Object Model for importing. You can either import all of the classes from an object model or a selected subset of the classes. If the ontology changes, you can use the PAC Editor for updating PACs. This dialog is invoked by selecting **Update…** in the **File** menu.

You can create initial PAC instances using another view of the PAC Editor. Click on the **PAC Instances** button at the top of the window to get the PAC Instance Editor. Figure 19 shows the PAC Instance Editor with the **myHelloWorldFrame** instance shown.

The PAC Instance Editor allows you to specify the PAC instances which should be created at startup. These are used to define the agent's initial mental model. You can use the pull-down menu

located below the **Description** text field in the **PAC Properties** panel. This menu is normally labeled **<PAC>** and can be used to select an instance of interest. Clicking on the **Initial PAC Instance** check box will bring up a **Specify Attribute Values** dialog and allow you to

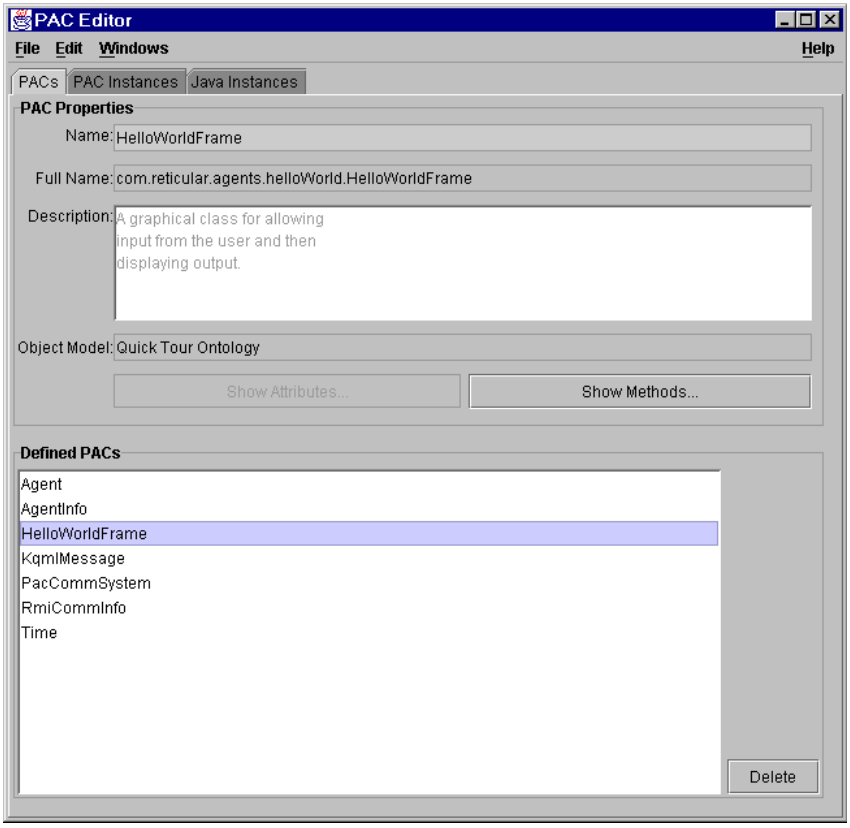

Figure 18. The PAC Editor

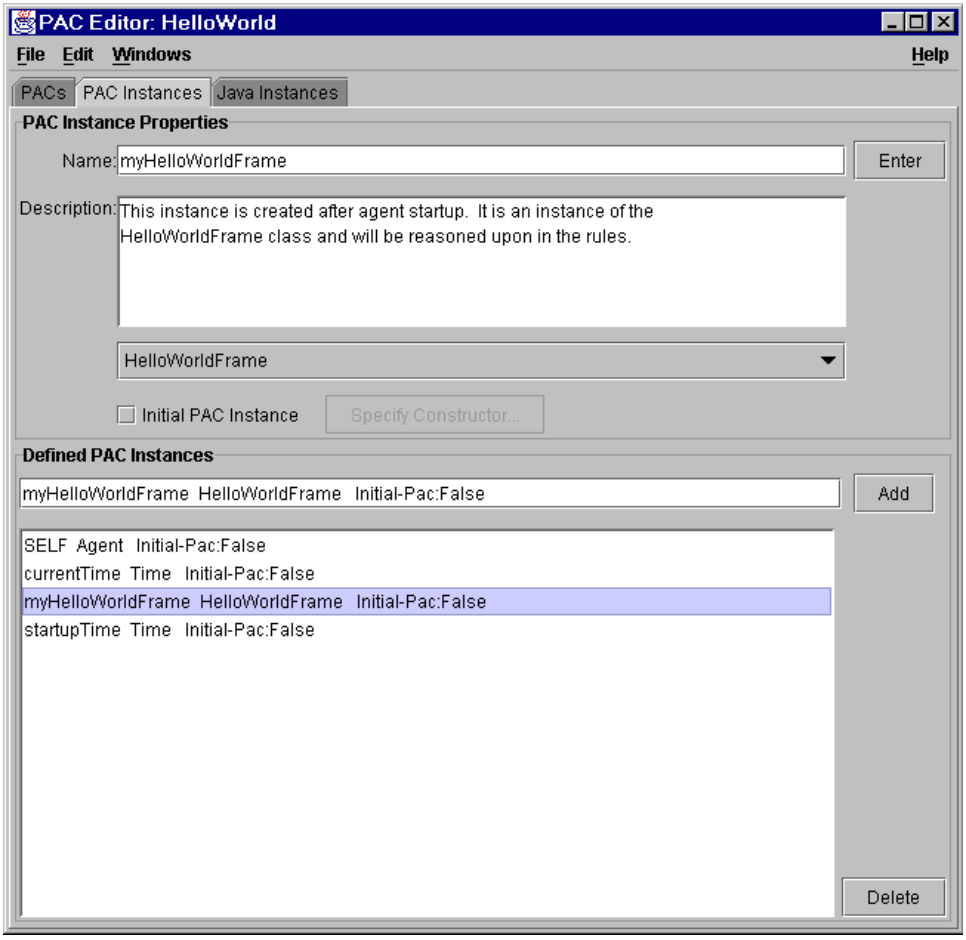

Figure 19. The PAC Instance Editor

specify the values for the constructor of the initial PAC Instance. For more information see the PAC Editor section of the Agent-Builder Reference Manual.

Action Editor

The Action Editor is started by selecting the **Tools →Action Editor** menu item in the Agent Manager or by selecting the **Actions** tab item and then double-clicking on an action in the **Agent Manager** window. The Action Editor is used to define the private actions the agent can perform. Creating private actions is equivalent to tagging a PAC method with a symbolic label. This is accomplished by defining a name, picking a PAC, and choosing the method. If the method is `run()`, then you can run the action on a separate thread. The run-time system automatically handles this threading. The Action Editor is shown in Figure 20.

Figure 20 also shows the definition of the `Print` action for the **HelloWorld** agent. This action is mapped to the `print` method of the HelloWorldFrame PAC.

Note that defining actions isn't strictly necessary. You can directly invoke methods on objects when specifying actions in rules.

Rule Editor

The Rule Editor is one of the key editors in AgentBuilder. It uses the information specified in the other editors to define the behavioral rules for an agent. The behavioral rules specify how the agent interacts with its environment. All rules are specified as WHEN-IF-THEN constructs. The WHEN and IF sections are conditions that must be met before the THEN part of the rule is executed. The WHEN applies to temporal events such as recently received messages. The IF applies to beliefs the agent has about its internal and external environment. The THEN section specifies internal and external actions the agent should take when the conditions are met.

The Rule Editor is started from the Agent Manager by selecting **Tools → Rule Editor** menu item or by selecting the **Rules** tab panel and then double-clicking on a rule in the **Agent Manager** list. The Rule Editor provides two different views. One view shows the

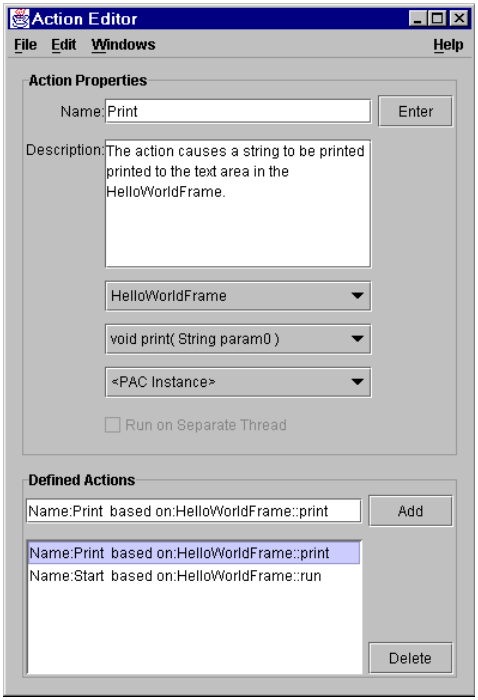

Figure 20. Action Editor

left-hand side of a rule, and the second view shows the right-hand side of a rule. Checkboxes labeled **Left-Hand Side** and **Right-Hand Side** are provided in the **Panel Options** area of the editor. Figure 21 shows the **Left-Hand Side** panel of the rule editor. You can toggle between the two views by clicking either of the buttons in the **Panel Options** at the top of the window. The left-hand side (LHS) allows you to specify the conditions for the rule. The right-hand side(RHS) allows you to specify the actions performed when the rule is executed.

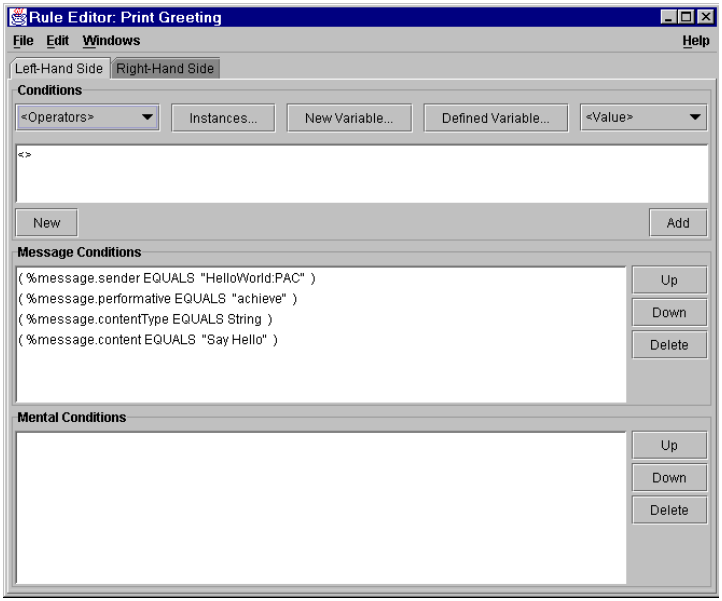

Figure 21. Rule Editor Showing LHS of the Print Greeting Rule

In this example, the left-hand side of the rule consists of several patterns that test fields in an incoming message. If an incoming message passes all tests on the left-hand side then the actions on the rule's right-hand side will be executed. The **Right-Hand Side** panel of the Rule Editor is shown in Figure 22. To view the other rules in the **HelloWorld** agent, select the **File → Open** menu item in the Rule Editor.

The other rules in the **HelloWorld** agent have elements that condition on new messages arriving in the agent. To see another rule, select the **File →Open** menu item from the Rule Editor and then double-click on the desired rule in the dialog list.

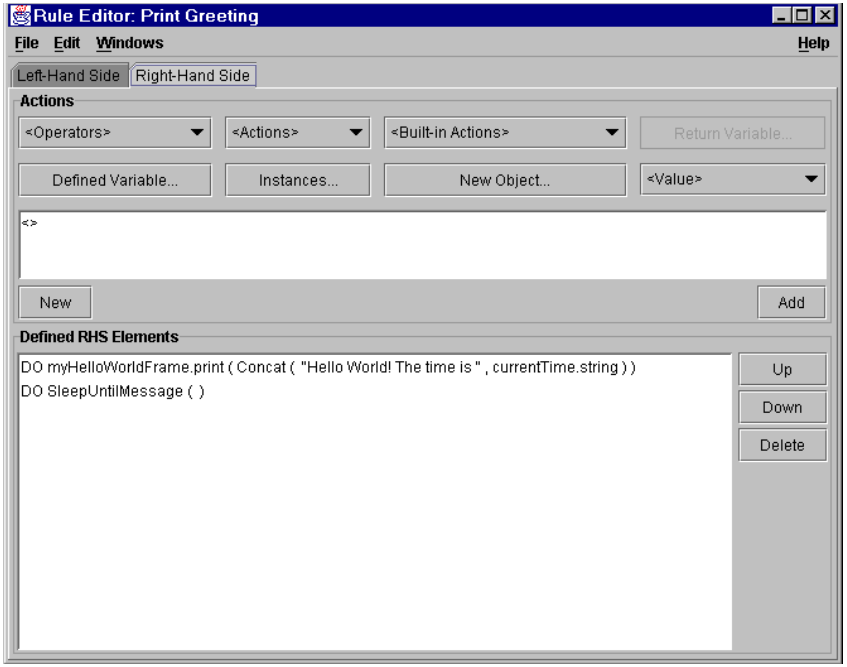

Figure 22. Rule Editor Showing RHS of Print Greeting Rule

Running the HelloWorld Agent

RADL files are produced by the AgentBuilder construction process. All of the information needed by an agent for execution is encoded in its RADL file with the exception of the agent's PACs. PACs are Java class files and must be in the system `CLASSPATH`. You can generate the RADL file for an agent by selecting **Options → Generate Agent Definition** menu item in the Agent Manager. A file dialog will be displayed, and you can enter the name and location of the RADL file. Once this information is entered, subsequent RADL generation will default to the same name and location. Figure 23 is a partial RADL file listing for the HelloWorld agent.

```
AgentBuilder version
Using Reticular Agent Defintion Language Formating
Copyright Reticular Systems Inc.
Agent Name: HelloWorld
Agent Description: Hello World agent with a full GUI and demonstrating
connections between the agent and the user interface.
Agencies belonged to:
[  "Hello World Agency" ]
==========================================================================
ABBREVIATED NAMES
( HelloWorldFrame com.reticular.agents.helloWorld.HelloWorldFrame )
==========================================================================
INITIAL OBJECTS
==========================================================================
ACTION DEFINITIONS
( Start SEPARATE_THREAD PAC_OBJECT  HelloWorldFrame <>)
( Print PAC_OBJECT  HelloWorldFrame <> PAC_METHOD print ( String))
==========================================================================
CAPABILITIES
==========================================================================
BELIEF TEMPLATES
==========================================================================
INITIAL BELIEFS
==========================================================================
INITIAL AGENCY BELIEFS
( SELF "HelloWorld"  [CURRENT_IP_ADDRESS] [RMI:2010] [] [  "Hello World Agency" ]  )
( AGENCY_TOOL "null"  [harding.reticular.com] [RMI:2000] [] [  "Hello World Agency" ]  )
REMOTE_AGENTS
==========================================================================
INITIAL COMMITMENTS
==========================================================================
INITIAL INTENTIONS
==========================================================================
BEHAVIORAL RULES
("Print Greeting"
WHEN
( OBJ [ MVAR KqmlMessage<%incomingMessage>.sender]  EQUALS [ VAL String "HelloWorld:PAC"  ]   )
( OBJ [ MVAR KqmlMessage<%incomingMessage>.performative]  EQUALS [ VAL String "achieve"  ]   )
( OBJ [ MVAR KqmlMessage<%incomingMessage>.contentType]  EQUALS [ VAL Class String  ]   )
( OBJ [ MVAR ( String )(KqmlMessage<%incomingMessage>.content)]  EQUALS [ VAL String "Say Hello"  ]   )
IF
THEN
(  DO Print( [ SFUNC Concat(  [ VAL String "HelloWorld! the time is: "  ]  [ INST Time<current-
Time>.string] ) ]) )
(  DO SleepUntilMessage( ) ))
("Build HelloWorldFrame"
WHEN
IF
(  BIND [ INST Time<startupTime>]   )
THEN
( ASSERT ( [ VAL String "myHelloWorldFrame"  ]  [ NEW HelloWorldFrame (PacCommSystem [ NEW PacCommSystem
(AgentInfo [ INST Agent<SELF>.agentInfo] ) (String [ VAL String "HelloWorld:PAC"  ] ) ]) ] ) ))
("Quit"
WHEN
( OBJ [ MVAR KqmlMessage<%incomingMessage>.sender]  EQUALS [ VAL String "HelloWorld:PAC"  ]   )
( OBJ [ MVAR KqmlMessage<%incomingMessage>.performative]  EQUALS [ VAL String "achieve"  ]   )
( OBJ [ MVAR KqmlMessage<%incomingMessage>.contentType]  EQUALS [ VAL Class String  ]   )
( OBJ [ MVAR ( String )(KqmlMessage<%incomingMessage>.content)]  EQUALS [ VAL String "Quit"  ]   )
IF
THEN
(  DO ShutdownEngine( ) ))
("Launch Interface"
WHEN
IF
(  BIND [ INST HelloWorldFrame<myHelloWorldFrame>]   )
THEN
(  DO ConnectAction( [ VAL String "Print"  ] , [ INST HelloWorldFrame<myHelloWorldFrame>] ) )
(  DO ConnectAction( [ VAL String "Start"  ] , [ INST HelloWorldFrame<myHelloWorldFrame>] ) )
(  DO Start( ) )
(  DO RemoveRule( [ VAL String "Build HelloWorldFrame"  ] ) )
(  DO RemoveRule( [ VAL String "Launch Interface"  ] ) )
(  DO SleepUntilMessage( ) ))


==========================================================================
```

Figure 23. AgentBuilder RADL File Listing

After you've created an agent you'll want to run it. There are three ways that agents can be started. You can invoke the Run-Time system by selecting the **Options → Run Agent** menu item in the Agent Manager, you can run the agent from the **Project Manager**, or you can invoke the Run-Time system directly and specify an existing RADL file. For instructions on how to directly invoke the Run-Time system, please see the Run-Time System section of the AgentBuilder Reference Manual.

Selecting **Run Agent** starts the agent execution. A file dialog will query you for the file name and location of the RADL file. The tool then generates the RADL file and passes it to the Run-Time system. The next dialog you'll see is the run-time system **Agent Engine Options** dialog. This dialog will allow you to specify run-time options before starting the agent. Figure 24 shows the dialog. After setting run-time options, click on **OK**. This will start the Run-Time system, parse the RADL file, start execution of the agent engine, and display an engine console. For the **HelloWorld** example, the **Hello World** dialog is displayed.

The engine console allows you to view the output from the Run-Time system and terminate or reset the agent. Figure 25 shows both the engine console and the **HelloWorldFrame** interface.

The top panel of the console contains a text area that allows you to view output that is directed to standard out. You can freeze, save or clear the panel as desired. The lower text panel is a window for error display. The **Exec** menu allows you to stop the engine as well as restart it.

The other window shown in the example is the **HelloWorldFrame** interface. This is displayed as a result of the `Start` private action. By clicking on the **Say Hello** button or the **Quit** button you can interact with the agent. Clicking on this panel's **Say Hello** button

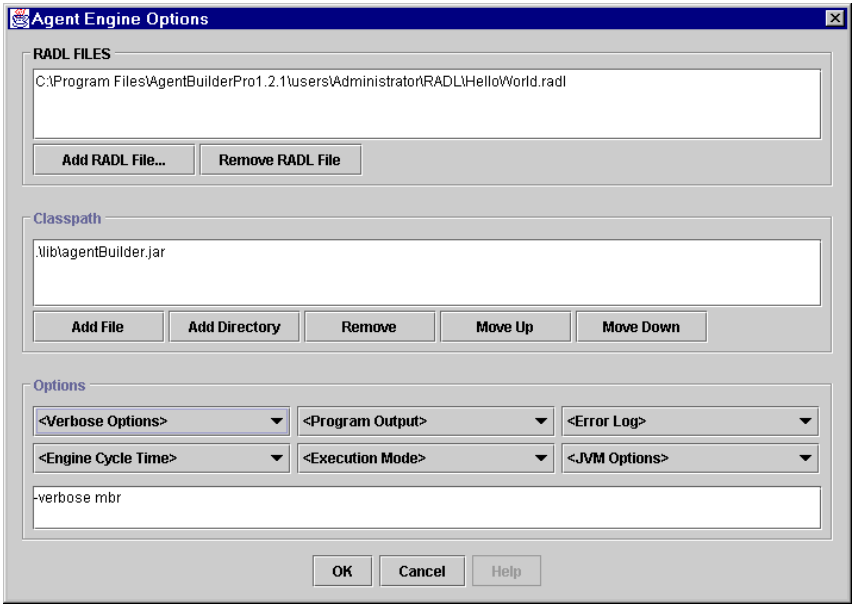

Figure 24. The Agent Engine Launcher Dialog

causes the `Print` rule to be executed. Clicking on **Quit** button causes the `Quit` rule to be executed.

Select the **File → Reset** menu item in the console to restart the agent and redisplay the HelloWorldFrame. Select the **File → Quit** menu item from the console's **File** menu to close the console and terminate the agent and its run-time system.

On-Line Help

AgentBuilder provides an on-line help system. This help system provides on-line documentation that you can access while using the various AgentBuilder tools. Figure 26 below shows the **Help** viewer. The help system is organized as a series of HTML pages. You can navigate the help system using your default web browser.

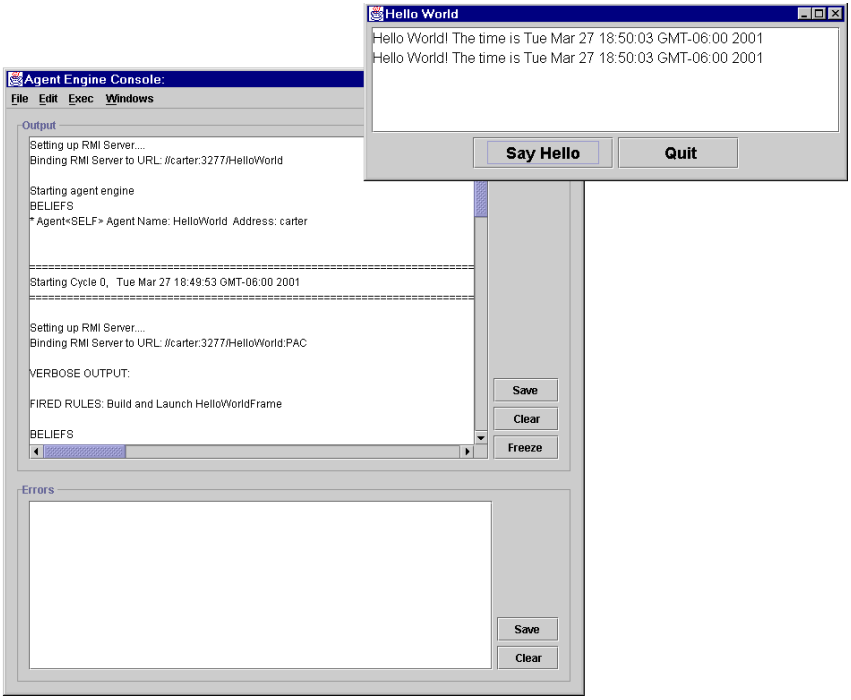

Figure 25. AgentBuilder Engine Console and HelloWorld Frame

Overview of Typical Agent Development

Agent-based systems development is a complex and iterative process. Like most software engineering processes, the developer must first develop a problem description. The next step is the problem and domain analysis. AgentBuilder provides the ontology tools to assist the user in developing concept maps and object models of the agent's domain. Several different ontologies may be required to capture this knowledge.

Once these first steps are completed the design phase begins. This involves decomposing the system into agencies and agents and assigning responsibility and functionality for each agent. Some-

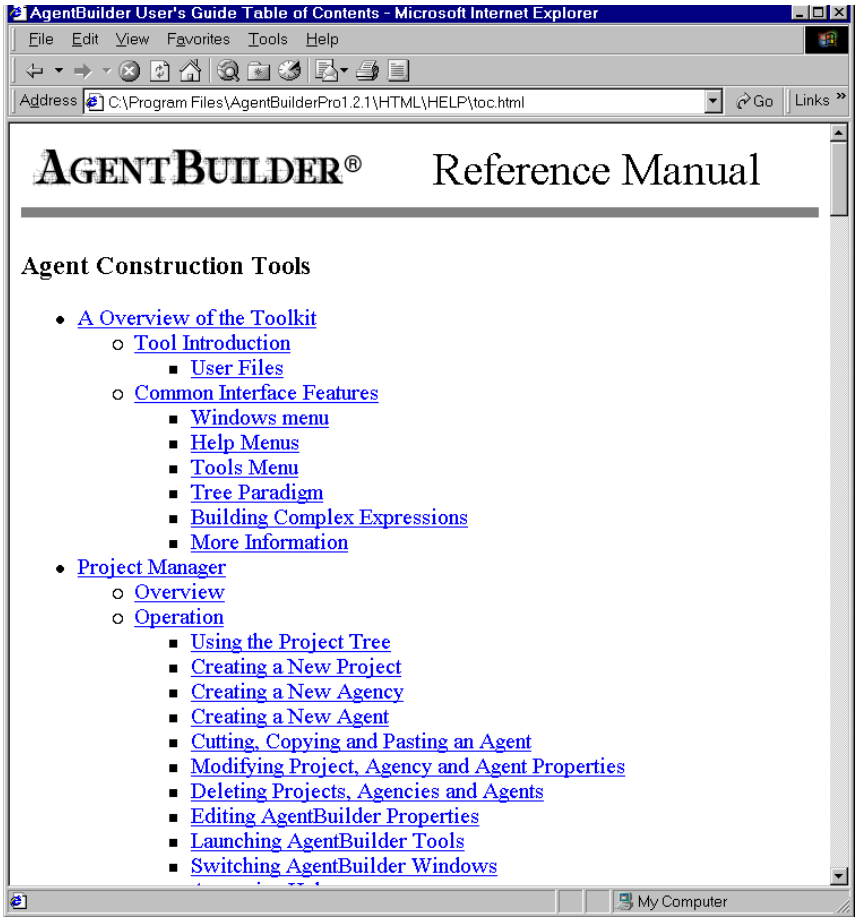

Figure 26. AgentBuilder Help Viewer

times several different agencies may be required. After the agencies and agents are defined, the agent interaction protocols must be specified. This involves deciding how information and commands are processed and communicated.

The next phase is the detailed design of the agents. At this stage, the PACs required by each agent must be fully defined. This step usu-

ally requires refining the ontologies developed in the analysis phase. If PACs have been developed previously, then the developer must import them into object models.

At this point, the developer is ready to write the agent's behavioral rules. This is typically the most challenging part of the agent construction processes. It may take several iterations of development and debugging to get the rules to function properly. The rules must process incoming messages, generate outgoing messages, perform appropriate actions, and update the internal mental state of the agent.

The development process described here is not a precise recipe but consists of general guidelines that we have found to be useful. Sometimes it's appropriate to iterate or skip various steps during the development process. For more detailed information on the agent development process see the section entitled "Agent Development Process" on page 71.

Chapter 6

Building Simple Agents - Example Agent 1

In this chapter you will learn how to:

- Create a Project
- Define an Agency
- Create an agent for that agency
- Create a behavioral rule for the agent
- Run your simple agent and examine the output

This example shows you how to create a simple agent that says "Hello World." This simple agent requires only a single behavioral rule. The steps for constructing this simple agent are detailed in Table 6.

Table 6. Building the Hello World Agent

Step	***Description***
1.	Create the Hello World Project
2.	Create the Hello World Agency
3.	Create the Skeletal Agent
4.	Create the Agent's Behavioral Rule a. Define the Rule in the Rule Editor b. Create the LHS Pattern c. Create the RHS Action
5.	Create the RADL
6.	Run the Agent

Step 1. Create Hello World Project and Agency.

Before creating an agent you must first create a project for managing the agent development process. The first window you see after starting AgentBuilder is the **Project Manager** window (Figure 27). The projects shown in red are system resources provided by Acronymics, Inc. and should not be modified. However, AgentBuilder makes it easy for you to copy agents and reuse the portions of the agent that are most useful to you.Create a project by selecting the **File → New** menu item. This will display the **Project Properties Dialog** box. Enter the project name and description as shown in Fig-

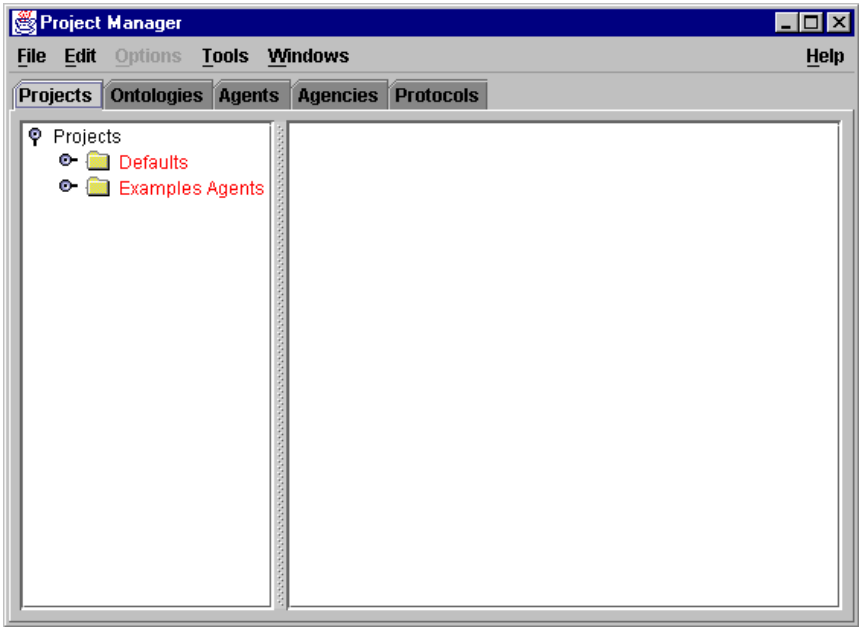

Figure 27. Project Manager Window

ure 28. After entering the name *Quick Tour Project* and a description of the project, click on the **OK** button. This will create a new project folder in the left panel of the **Project Manager** window with the name you specified.

Step 2. Creating the Hello World Agency

Now select the **Quick Tour Project** folder icon. Note that when the **Quick Tour Project** is selected, the previously entered information is displayed in the right panel. With the **Quick Tour Project** selected, click on the right mouse button (i.e., right-click). This will display a pop-up menu from which you can select the **New Agency** pop-up

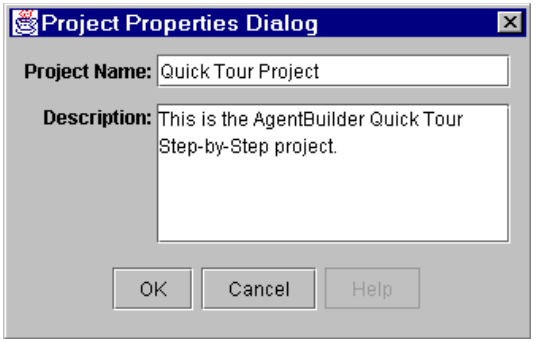

Figure 28. Project Properties Dialog

menu item. This will display an empty **Agency Properties Dialog** where you can enter the name of the agency (e.g. *`Quick Tour Agency`*) as well as a short description of the agency (i.e. its purpose, the scope of the agents, the type of agents, etc.). Fill out this dialog as shown in Figure 29. Click **OK** in the dialog to save the information.

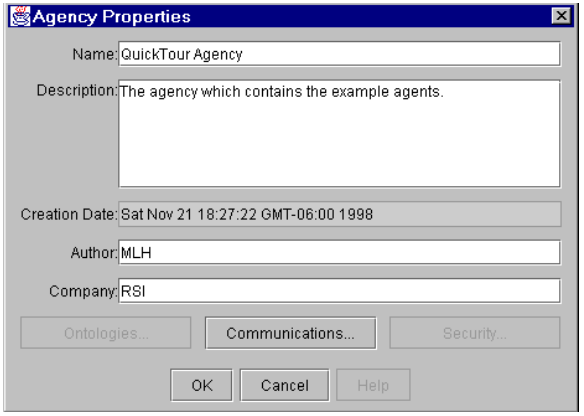

Figure 29. Agency Properties Dialog

Step 3. Create Your First Hello World Agent

Now that you have created an agency, you can populate it with the necessary agents. To create your first **Hello World** agent, select the **Quick Tour Agency** in the left panel of the Project Manager and then select the **File → New** menu item. This will cause the **Agent Properties** dialog window (Figure 30) to appear. An alternate technique is to click on the right mouse button and choose **New Agent** from the popup menu. You can also edit agents by selecting the **Edit** option from the popup menu.

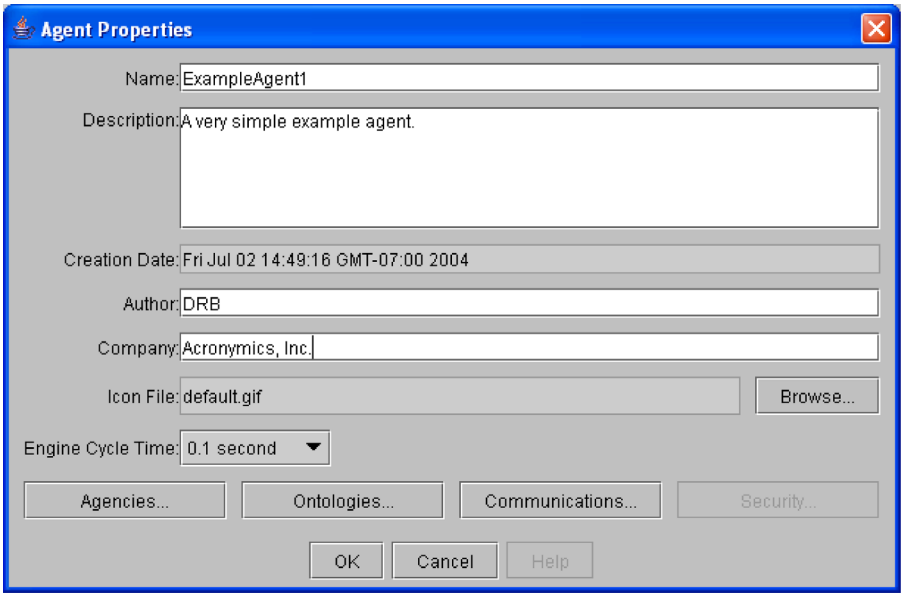

Figure 30. Agent Properties Dialog

Enter the agent's name (in this case, `ExampleAgent1`) as well as a description of the agent, the author's name, and your company name. Other agent properties can be set using the three buttons near the bottom of the dialog (**Agencies…, Ontologies…**, **Com-**

munications…); we will discuss the use of these buttons later. For more information on these buttons see the description in the AgentBuilder Reference Manual. Click on the **OK** button to save the basic properties for this agent (this may take a few seconds depending on the speed of your computer). The Project Manager should now look like the example shown in Figure 31.

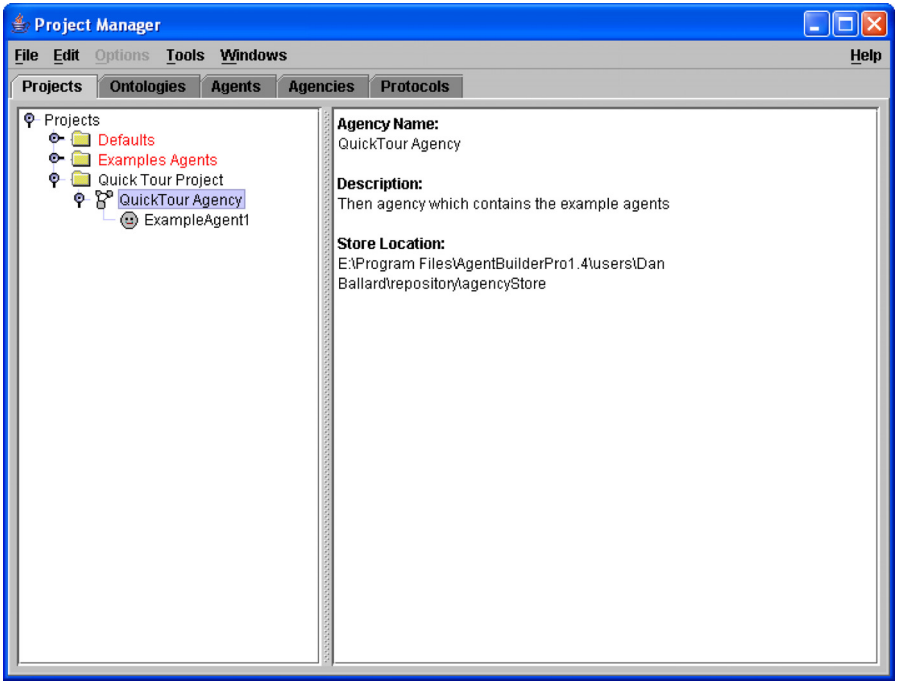

Figure 31. Project Manager Window

Step 4. Create the Agent's Behavioral Rules

Now we can create a rule for the agent so it can print out "Hello World."

Step 4a. Start the Rule Editor

To define the behavior of any agent, you must first open the Agent Manager. The Agent Manager is the primary tool for configuring and defining an agent's behavioral rules and capabilities. Open the Agent Manager by selecting **ExampleAgent1** in the **Project Manager** window and then selecting the **Agents** tab. The system will display the **Agent Manager** dialog for **ExampleAgent1** as shown in Figure 32.

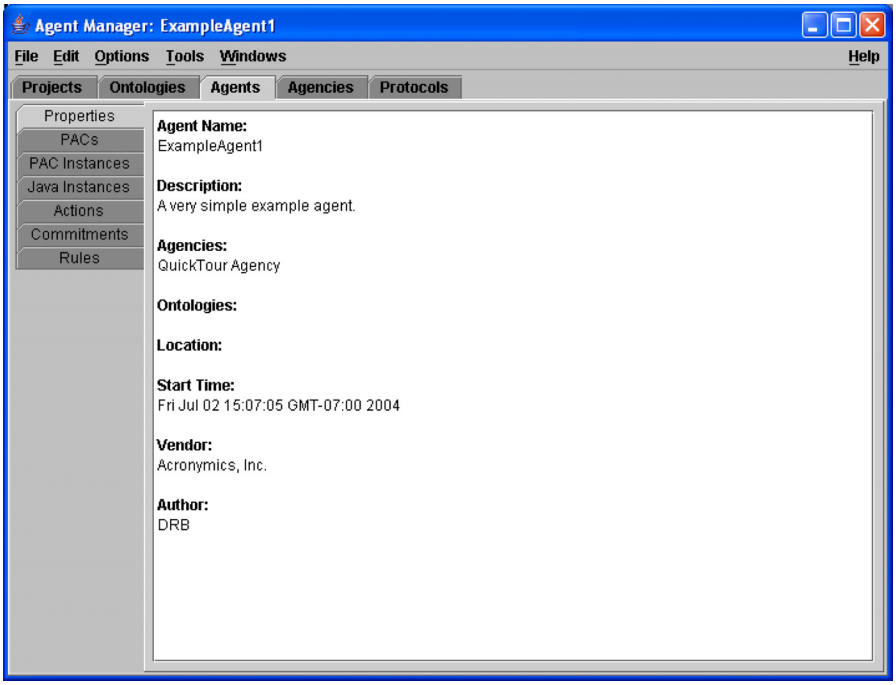

Figure 32. Agent Manager Window for the ExampleAgent1

Using this window, you can launch the various editors (**Action**, **Commitment**, **PAC**, and **Rule**) used for configuring the agent. For this simple agent, we need to create only a single rule. Select **Tools**

→ **Rule Editor** from the **Agent Manager** menu bar. This will display a window as shown in Figure 33.

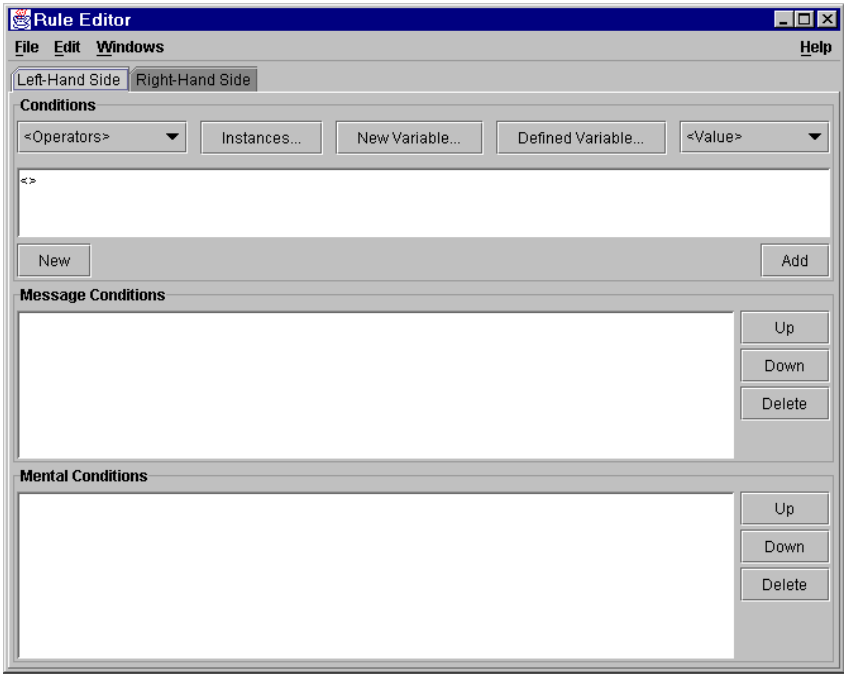

Figure 33. Rule Editor for ExampleAgent1

Now select **File → New → New Rule** and enter the name of the rule and a rule description in the **Rule Properties Dialog**. This dialog is shown in Figure 34. For this example, call the rule *Hello* and enter any description you like. For this agent, there are no message conditions and therefore the LHS pattern will consist of only a single mental condition.

Step 4b. Create LHS Pattern

For this agent, we need to create a rule that will print "Hello World" to the console when the agent starts executing. To do this, create a

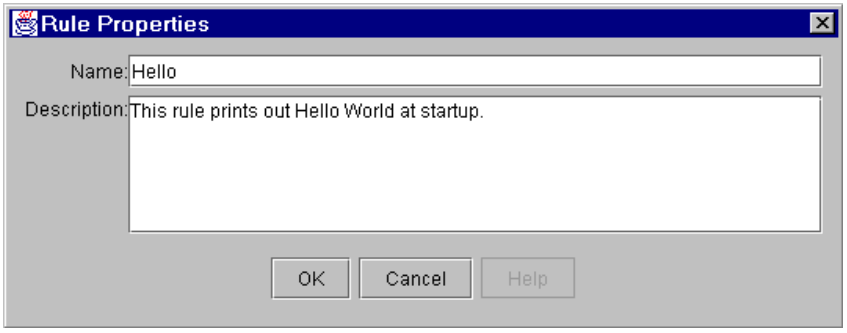

Figure 34. The Rule Properties Dialog

Left-Hand Side (LHS) pattern that matches with the starting time of the agent engine.

A rule cannot fire (i.e., execute it's right-hand side) unless it has at least one LHS pattern that matches a new belief in the agent's mental model. We want a simple rule that will fire during the first engine cycle, therefore we want to write a LHS pattern that we know will be satisfied by a belief in the mental model at engine start-up. The Run-Time System automatically creates a `Time` instance named `startupTime`, and stores it into the mental model. A `BIND startupTime` pattern will look for a `Time` instance named `startup` in the mental model. The pattern will be satisfied if such an instance is found, and so the rule will be fired.

Using the Rule Editor, select **BIND** from the **Operators** pull-down menu in the **Conditions** panel of the Rule Editor. After performing this operation, the text field below the pull-down menu contains the string **(BIND <>)**. This text field is used to *accumulate* the pattern as you construct it. See "Building Complex Patterns with the Accumulator Paradigm" on page 85 for a more detailed explanation of using the accumulator.

Now we want the pattern to *bind* to a built-in instance called `startupTime`.You can access this instance using the **Instances...** button.

After clicking on this button you will see a dialog as shown in Figure 35. This dialog shows the current Instances available. Select

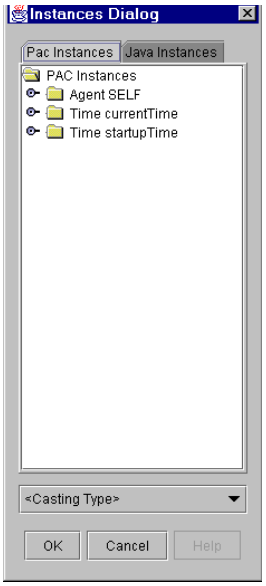

Figure 35. Instance Dialog

the **startupTime** item in the **PAC Instances** dialog (i.e., select the folder labeled **Time startupTime**) and then click on the **OK** button. The **Condition** text field will now contain **(BIND startupTime).** This completes the pattern so you can now click on the **Add** button located below the **Conditions** text area. The Rule Editor will add this pattern to the list in the **Mental Conditions** panel.

Step 4c. Create RHS Action

Select the **Right-Hand Side** tab of the **Rule Editor** to define the action for our `Hello` rule. For this rule, we wish to utilize a built-in action, `SystemOutPrintln`, to print a string to the console device. To invoke this action, select **SystemOutPrintln** from the <**Built-in Actions>** pull-down menu. This will fill in the **Actions** text field

with **DO SystemOutPrintln(<Output>)**. Now, all you have to do is specify the string to print out. To do this, select **String** from the **<Value>** pull-down menu. The system will display a **Value Dialog** (Figure 36). Now enter the string *Hello World* into the **Literal Value** text field and click on **OK**.

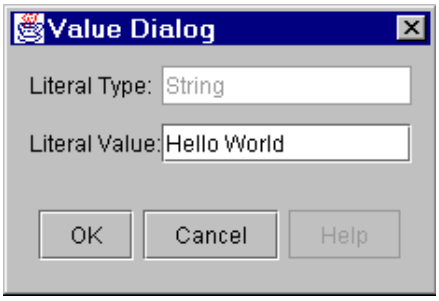

Figure 36. Values Dialog

Now add the action to the list of **Definded RHS Elements** displayed at the bottom of the Rule Editor by clicking on the **Add** button located below the **Actions** text area. The window will now look like that shown in Figure 37. Now select **File → Save** from the **Rule Editor** menu bar to save both the LHS and RHS elements of the rule. The Rule Editor will now display the **Hello** rule we have just defined.

Step 5. Create the RADL file.

Once all the rules are created (in this case there is only one rule), you must generate a RADL file for use by the run-time agent engine. You use the Agent Manager to control generation of the RADL file. You can quickly navigate back to the Agent Manager by selecting **Windows → Agent Manager: ExampleAgent1** in the Rule Editor menu bar.

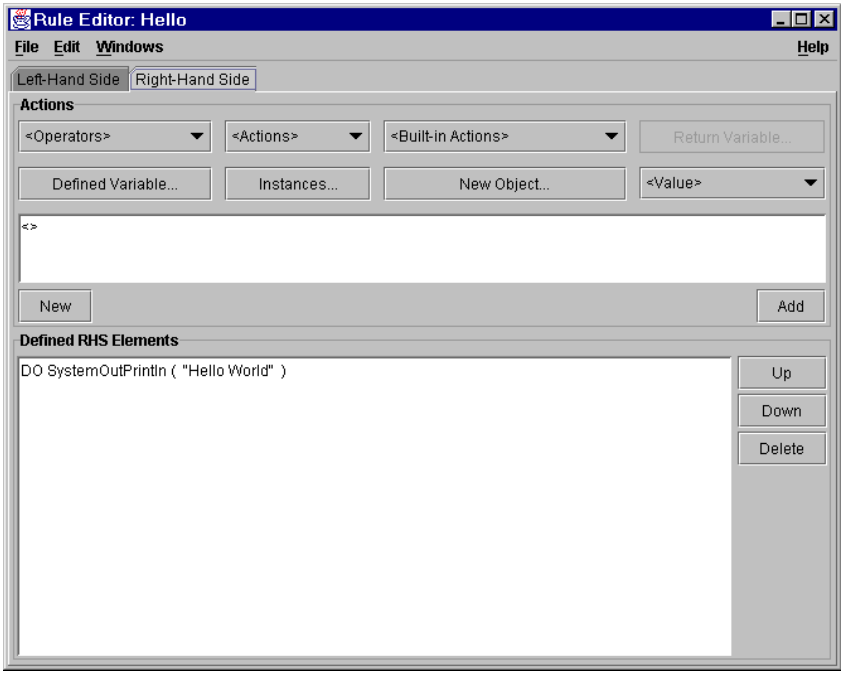

Figure 37. Rule Editor Window After Entering RHS Information

To generate the RADL file, select **Options →Generate Agent Definition...** from the **Agent Manager** window. The Agent Manager will then display a file dialog as shown in Figure 38. You can now save the RADL file in a directory of your choice. Clicking on the **Save** button will generate and save the required agent's RADL file to your hard disk. A dialog box will be displayed informing you when you have successfully generated the RADL file. Typically, this step can be skipped since performing Step 6 automatically creates the RADL file.

Step 6. Run the Agent.

You are now ready to run the agent. Select **Options → Run Agent** from the **Agent Manager** menu. AgentBuilder will display a file

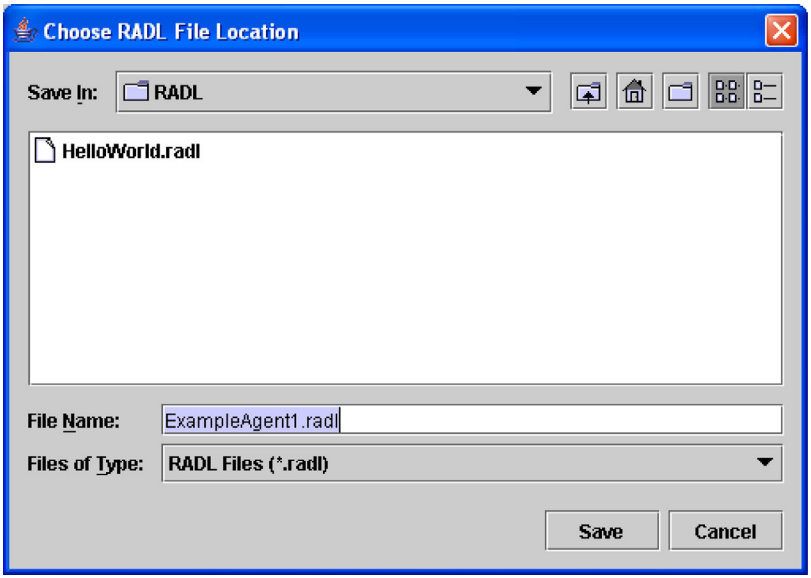

Figure 38. File Dialog for Saving RADL File

manager dialog (Figure 38) that allows you to select a RADL file to be executed in the agent engine. Note that the **Run Agent** command automatically creates a new RADL file.

After you select the proper RADL file, click on the **Save** button. You can then choose the **Option →Run Agent...** menu item and you will be presented with the file chooser dialog**.** After selecting the file, the **Agent Engine Options** dialog shown in Figure 39 will be displayed. This dialog presents several options which you can use to specify the format of the agent's output data display. To see all available agent output, select the **Verbose Options → Everything** item and click on the **OK** button.

After a few seconds, the system will display the agent engine console window as shown in Figure 40. Since we selected the *verbose* option in the **Agent Engine Options** dialog, each cycle of the engine

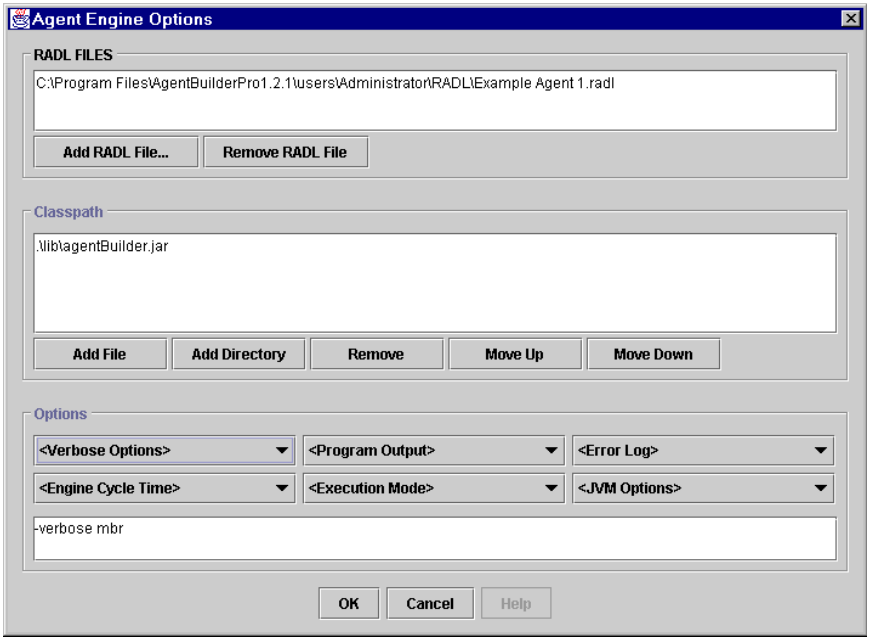

Figure 39. Agent Engine Options Window

is displayed along with the rule fired, the action performed, and current beliefs.

You can cancel verbose mode selection by selecting **Exec → Set Engine Options** and then selecting **Verbose Options → Clear Verbose Options** and clicking on the **OK** button. This will restart the agent engine and display the agent engine console shown in Figure 41. The **Program Output** option in the **Agent Engine Options** panel allows you to suppress printing or route the `Hello World` output string to a file or to the standard output stream on your system (typically the screen).

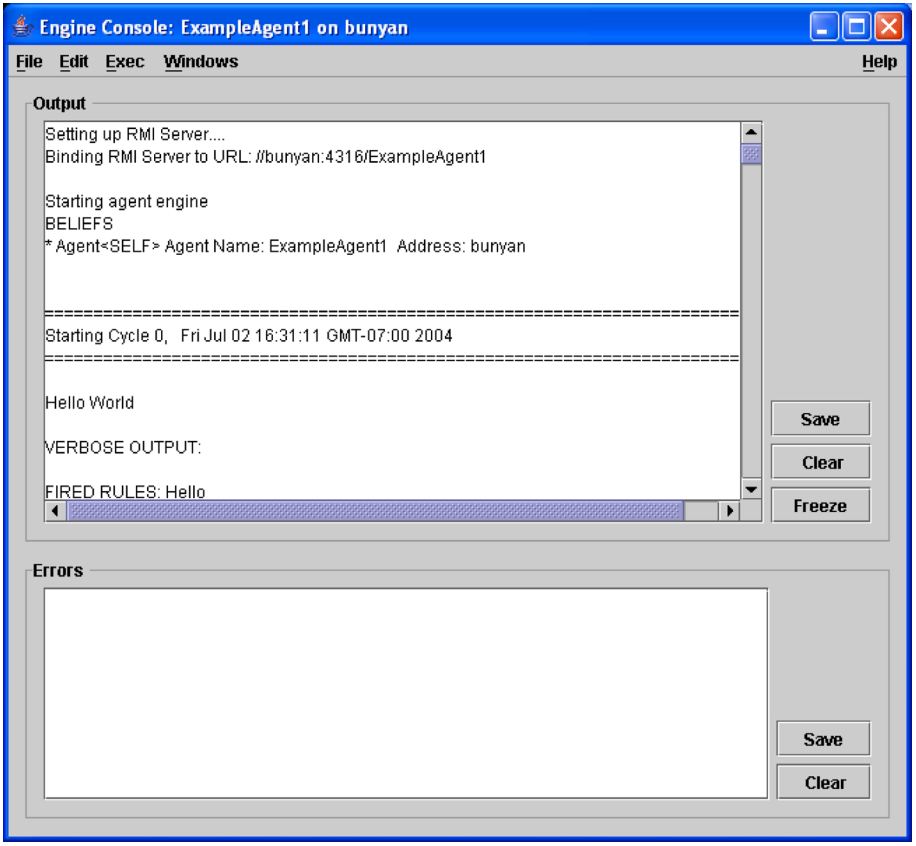

Figure 40. Agent Engine Console Window in Verbose Mode

Note that closing the panel does not shut down the run-time system. Select **File → Exit** to shutdown the runtime system.

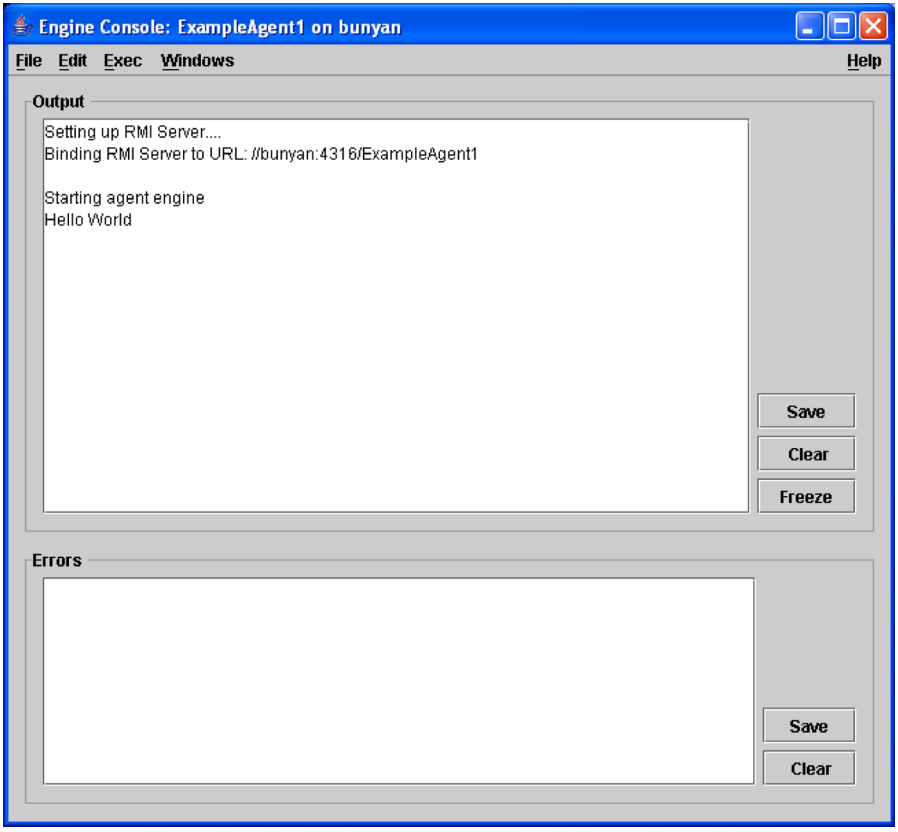

Figure 41. Agent Engine Console Window in Non-Verbose Mode

Chapter 7

A More Complex Agent (Example Agent 2)

In this chapter you will learn how to:

- Create a new agent by copying an old one
- Modify rules to change behavior
- Set and test the mental conditions of the agent
- Create Java objects and reason using them

This section describes the creation of a series of agents that are based on our previous example but have additional features that demonstrate other aspects of agents you can build using AgentBuilder. The next agent we build (**ExampleAgent2**) uses a single rule that will fire repeatedly and print the string `Hello World` followed by the current time at 10 second intervals.

This example shows the process that is typically used in developing an agent. We will reuse the previous agent (**ExampleAgent1**) by modifying its rule and running and testing it again. AgentBuilder makes it easy to reuse agents. The steps used in this process are detailed in Table 7.

Table 7. Creating Agents by Modifying Behavior

Step	***Description***
1.	Copy Previous Agent
2.	Alter Existing Rules a. Open the Rule Editor with Old Rule b. Alter the LHS c. Alter the RHS
3.	Run the Agent

Step 1. Copy Previous Agent

The easiest way to create our new agent is to modify the agent we previously constructed (ExampleAgent1). Open the **Project Manager** by clicking on the **Projects** tab and selecting a previously created agent (in this example, **ExampleAgent1**). Now, right-click on the selected agent and select **Copy** from the pop-up menu. This will copy our initial agent into a temporary holding area. Now select the **Quick Tour Agency,** right-click on it, and select **Paste Agent**. This will create a new agent that is an exact duplicate of the initial agent. The new agent's name is pre-appended with **CopyOf**. To rename this agent, select the copied agent in the **Project Manager,** right-click on it, and select **Properties…** from the pop-up menu. You can now change the name of this new agent to *ExampleAgent2* and alter the description as required. After updating, click on the **OK** button. Note that the name and description of the agent are displayed in the right panel when the **Example Agent 2** is selected in the right panel (Figure 42).

Step 2. Alter Rule to Run Continuously

We need to alter the rule to make it fire repeatedly. The initial `startupTime` object is used for matching for the first cycle of the engine. However, after the first cycle that object is no longer a new object and will not trigger the rule again. This is referred to as *refraction* and is a property of rule-based systems that prevents a rule from firing unnecessarily. We want to match against an object that changes every cycle so that refraction will not prevent our rule from firing. We utilize the `currentTime` object to accomplish this.

Step 2a. Open the Rule Editor with Hello Rule Loaded

To change the rule, we must use the Rule Editor. To open the Rule Editor for **ExampleAgent2** select the agent in the **Project Manager** pane, click on the **Agents** tab in the **Project Manager** pane and then

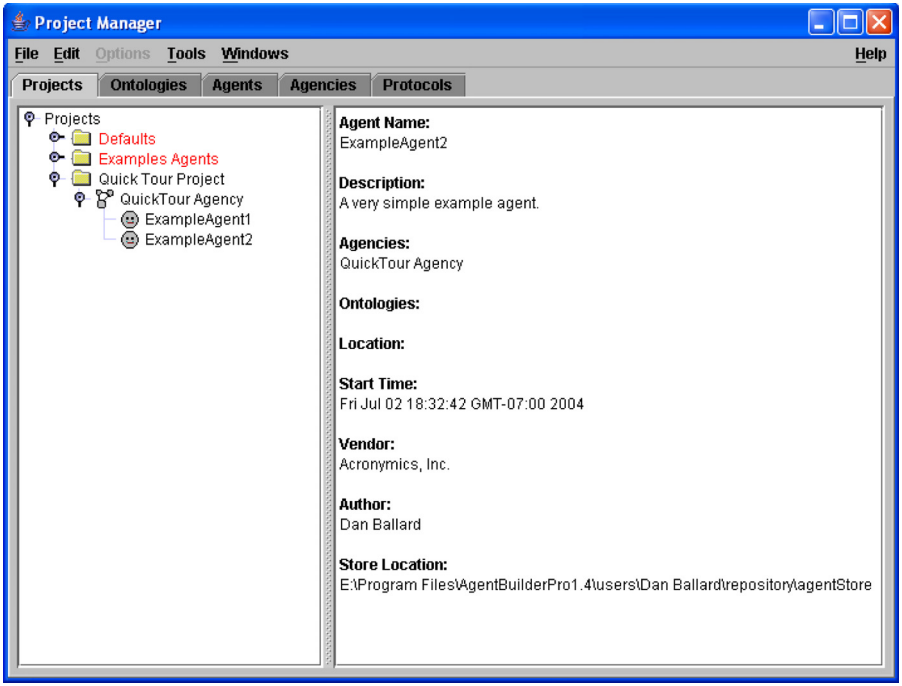

Figure 42. Copying Agents in the Project Manager

select the **Rules** tab on the left panel of the **Agent Manager** pane. When the **Rules** tab is selected, the rule `Hello` will be displayed in the middle panel. When you double-click on the `Hello` rule, the Rule Editor is started with the `Hello` rule loaded. Note that when the `Hello` rule is selected in the Agent Manager, the right-hand panel displays the description and summary for the selected rule. Note that the name of the rule you are working on is displayed in the title bar of the **Rule Editor**.

Step 2b. Alter the LHS Pattern

Since we want this rule to fire repeatedly, we need to remove the (`BIND startupTime`) pattern and replace it with (`BIND currentTime`). Since the `currentTime` (built-in) instance is updated every cycle,

this rule will fire every cycle rather than one time based on the single match with the `startupTime` instance (which only matches on the first cycle). First, select the pattern **(BIND startupTime)** in the **Mental Conditions** list at the bottom of the **Rule Editor** window. Then click on the **Delete** button on the right side of this text area. The selected pattern will be removed. Note that the rule changes are not final until you perform a **Save** operation. Click on the **New** button below the **Conditions** text area to clear the accumulator line. To create the new pattern, select **<Operators>** → **BIND** and then click on the **Instances…** button. From the **Instances Dialog** select the folder **currentTime** and click on the **OK** button. Your new pattern is now complete and can be added to the list of patterns by clicking on the **Add** button below the text field. Alternatively, you can just replace the **startupTime** instance with the **currentTime**. You do this by clicking on **startupTime** in the accumulator and then inserting the **currentTime** instance in the accumulator.

You can modify the description of the rule by selecting **Edit** → **Properties** from the menu bar and entering the appropriate information in the **Rule Properties** dialog. When you have completed these activities the Rule Editor will look as shown in Figure 43.

Step 2c. Modify the RHS Elements

To modify the RHS of the `Hello` rule, simply click on the **Right-Hand Side** tab of the **Rule Editor**. Like the LHS, you need to delete the single RHS element (that was copied from the first agent) listed at the bottom of the **Rule Editor** by selecting the RHS element and clicking on the **Delete** button on the right. Clear the **Actions** panel's accumulator text field by clicking on the **New** button. To enter the new RHS actions, select **<Built-in Actions>** → **SystemOut-Println**.

Now we need to inform AgentBuilder about the string we wish to print. For this modified agent, we want to print "Hello World" fol-

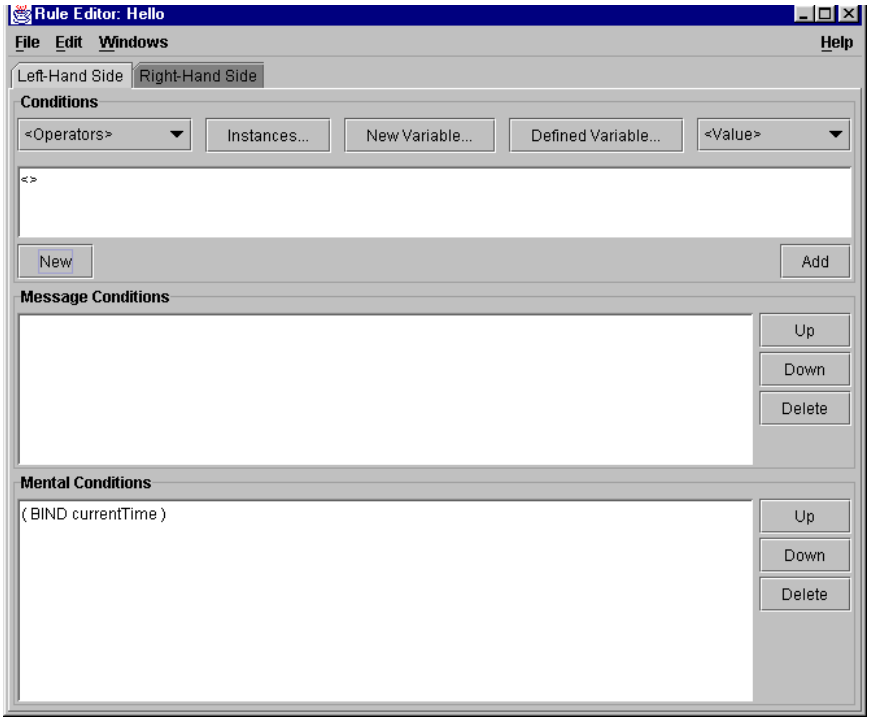

Figure 43. Rule Editor Window After Modifying LHS

lowed by the current time. This will allow us to see how much time has elapsed between the print actions. To accomplish this, we must concatenate the string "Hello World" with a string representing the current time. This can be accomplished by selecting **<Operators> → Concat** (in the **Actions** panel) to add the `Concat` function to the RHS action. You need to supply the two strings that are to be concatenated in the `Concat` parentheses. To enter the first string, select **<Value>->String** and enter *`Hello World:`* in the **Value Dialog** window. The second string is supplied by the built-in `currentTime` object. This can be accessed by clicking on the **Instances…** button and then clicking on the small button located next to **currentTime**. This will display the variables available within the `currentTime`

object. In this case, select **String string** (which is the text representation of the current time) and then click on the **OK** button. This completes the print message; we can now add the action to the **Defined RHS Elements** list by clicking on the **Add** button.

To prevent the agent from printing continuously, we will place a 10 second sleep action into the RHS. To do this select **<Built-in Actions> → Sleep** and then enter the integer value 10 by selecting the **<Value> → Integer** item and entering 10 into the **Value Dialog** box. Now using the **Add** button, add this to the RHS actions and save the rule changes by selecting **File → Save**. The **Rule Editor** should now look as shown in Figure 44.

Step 3. Run the Agent

Before running the agent go to the **Agent Manager** window and select **File →Save** to save the agent. Now select **Options → Run Agent** from the **Agent Manager**. You will get a file dialog prompting you for a location to save the RADL file. Since you copied **ExampleAgent1**, the default filename for saving the RADL information is also **ExampleAgent1**. Change the filename to **ExampleAgent2**. After you dismiss this dialog by clicking on **Save**, an **Agent Engine Options** panel will appear. You can clear the verbose options using the **Verbose Options** pull-down menu in this dialog.

Click on **OK** to start the engine. You should see output like that shown in Figure 45. There will be approximately 10 seconds between each printout. This can vary by ±2 seconds depending on the speed of your processor. Note that you can freeze the display output of the engine by clicking on the **Freeze** button. However, this does **not** stop the agent engine. The agent will continue running, and its output text will be stored in the console and displayed when you click on the **Resume** button. The agent will continue to

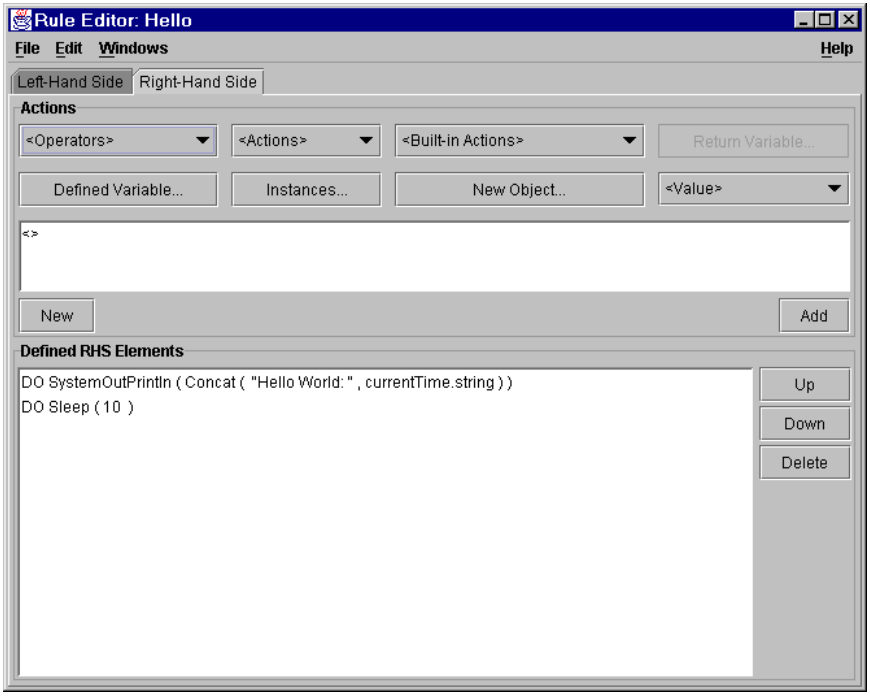

Figure 44. Rule Editor Window After Modifying RHS

output text which will be displayed when you click on the **Resume** button. You can halt the agent engine by selecting **Exec →Terminate Engine** from the menu bar.

Step 4. Adding Rules to Change Agent Behavior

Now let's alter the `Hello` rule and add an additional rule to stop execution after a single printout of "Hello World" while still triggering on the `currentTime` object. To do this, open the Rule Editor from the **Agent Manager** window by selecting the `Hello` rule and then selecting **Tools → Rule Editor**. For this version of our agent we

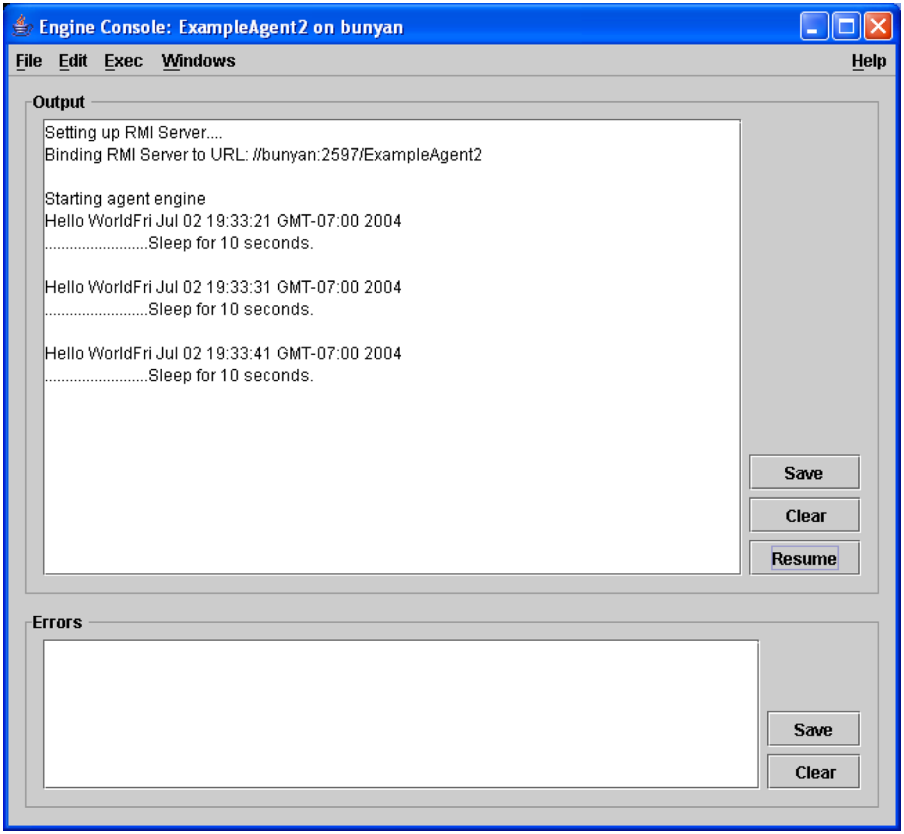

Figure 45. Agent Engine Console Window for Modified Agent

need to insert an ASSERT command in the RHS. Click on the **Right-Hand Side** tab.

Step 4a. Alter Hello rule's RHS

We want to assert a string (or more generally, a belief) into the mental state of the agent and then use that belief to trigger another rule to shutdown the agent engine. To assert a new belief, select **<Operators> → ASSERT** (in the **Actions** panel). An **Assert Dialog**

then prompts you to enter a name for the object being asserted. Enter `PrintFlag` as the name and click on **OK**. We now need to tell the agent what kind of object and the contents of that object that are to be asserted. In this case, we want to assert a String object with a value of "Quit Printing Now." Use the **<Value>** pull-down menu and select **String** and then enter the String "Quit Printing Now" in the **Value** Dialog. Now click on the **Add** button and then save the changes to the rule. The Rule Editor will look like Figure 46.

Step 4b. Create new Quit rule's LHS

Now let's create a rule that will be triggered by the `PrintFlag` belief. First go back to the LHS of the current rule by selecting the **Left-Hand Side** tab on the **Rule Editor**. Select **File → New → New Rule...** from the menu bar. This will clear everything in the **Rule**

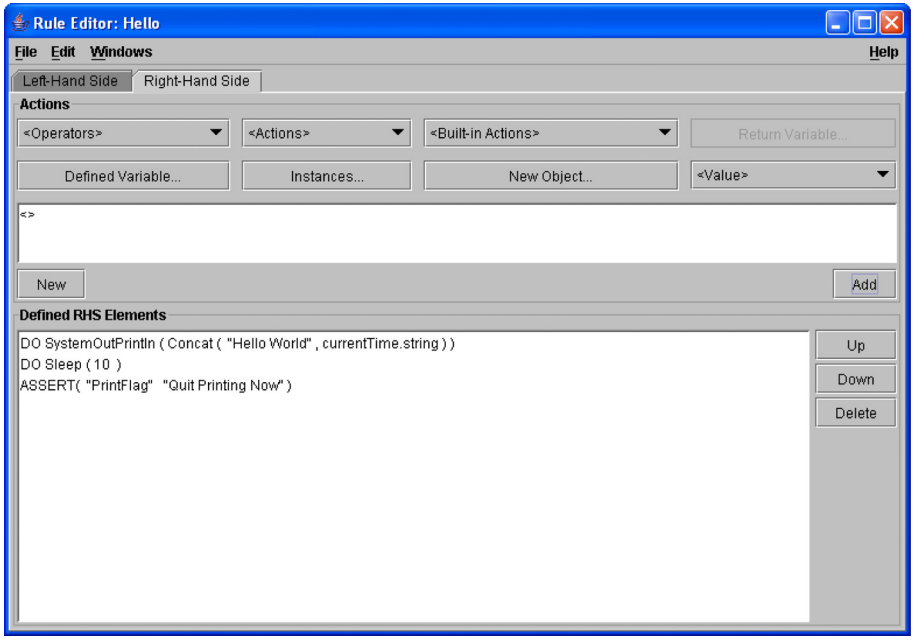

Figure 46. Rule Editor Window with Added Assertion

Editor and allow you to enter a completely new rule into the agent without deleting the current rule. Name the rule *Quit* and put a short description into the description part of the **Rule Properties** panel.

For the LHS of the Quit rule, we want to create a mental condition that will test for the string that was asserted into the mental state by the Hello rule. We must create a variable that can find any instance of a string in the mental state. To do this, click on the **New Variable…** button. This will display the **New Variable** window as shown in Figure 47. Since we want to detect a string we must first click on the **Java Types** tab in the top panel of the window. Now select the **String** class from the Java types listed in the **Java Types** panel. Next, enter a variable name (e.g. *?str*) in the text field below the list[1]. You can add this variable to the list of variables at the bottom of the dialog window by clicking on the **Add** button on the right of the **Variable Name** text field. Click on the **OK** button to confirm entry of this new variable. All we have done is define the new variable. There have been no changes made to the **Rule Editor** at this point.

We are now ready to perform an EQUALS comparison between all Strings in the agent's mental state and the string Quit Printing Now. Do this by selecting **<Operators> → EQUALS** from the **Conditions** panel. We can now use the variable that we created in the previous paragraph by clicking on the **Defined Variable…** button. This will display the **Defined Variable** dialog shown in Figure 48. You can select the proper variable by clicking on the **Java Types** tab at the top of the window and then select the ?str variable (which should be of type String). Clicking on the **OK** button will

1. It is common practice in rule-based languages to represent variable names by preceding the variable name with a "?"; e.g., ?variable. See "Variable Naming Conventions" on page 87.

Figure 47. New Variable Dialog

enter this variable into the `EQUALS` pattern. The second slot can be filled in by selecting **<Value> → String** menu item and filling in the string we wish to match (i.e. *`Quit Printing Now`*). Note that the string you use in this pattern must exactly match the string asserted by the Hello rule. Any difference in spelling, spacing, or capitalization will prevent the pattern from matching. Click on **OK** to dismiss this dialog. Now click on the **Add** button and then save the results by selecting **File → Save** from the menu bar.

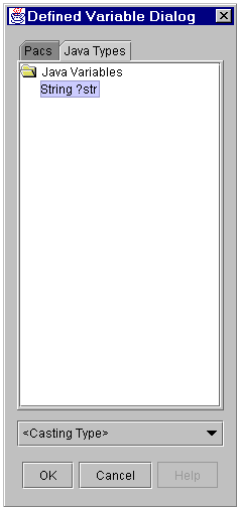

Figure 48. Defined Variable Dialog

Step 4c. Create the New Quit rule's RHS

Select the **Right-Hand Side** tab in the **Rule Editor**. When the `Quit` rule fires, we want to halt the agent engine. Fortunately there is a built-in action to do this (**<Built-in Actions> → Shutd-ownEngine**). Add this action to the RHS elements by clicking on the **Add** button and then save the `Quit` rule. Viewed using the Agent Manager, the rule will appear as shown in Figure 49.

Step 5. Rerun Agent

Now run the agent with the new rules in place by selecting **Options → Run Agent** from the **Agent Manager** menu bar. You will be presented with a dialog for saving the RADL file, and then the Agent Engine Console will appear. Set your desired agent engine options and click **OK** to run the agent. You should see output like that shown in Figure 50. Note that the agent actually printed the

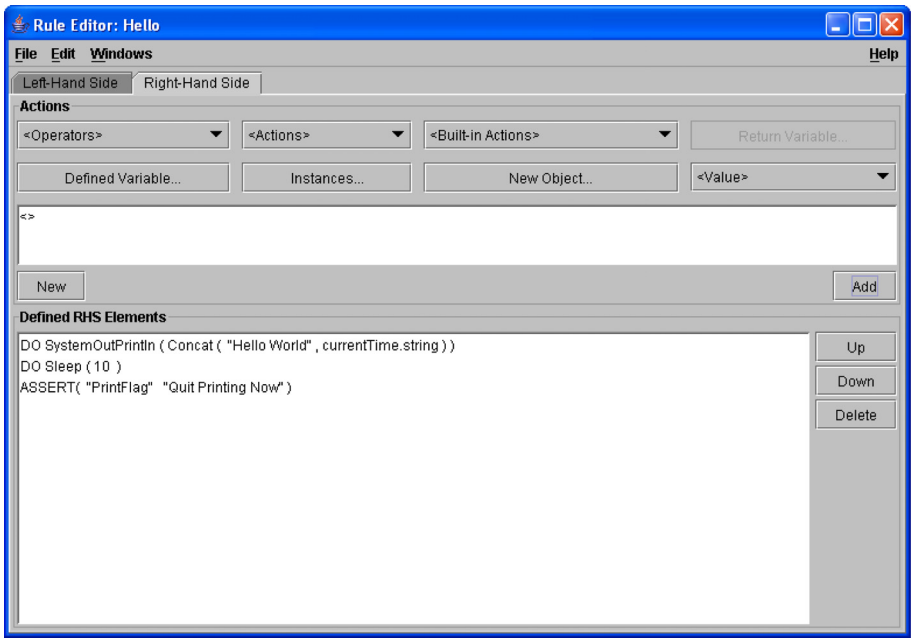

Figure 49. Viewing Rules Using Agent Manager

Hello World message twice! There is a warning about replacing an existing named instance! Why?

In the first cycle, only the `Hello` rule fires. However, in the second cycle (after the 10 second sleep) both the `Hello` rule and the `Quit` rule fire before the engine shuts down. Thus, the `PrintFlag` instance is asserted twice, hence the warning. Also, since the `ShutdownEngine` action happens at the end of the cycle, the `Hello` rule fires a second time.

To prevent this we need to modify the LHS of the `Hello` rule. Open the `Hello` rule by selecting **File → Open** and then select **Hello** from the **Open Rule Dialog** and click on the **OK** button. Figure 51 shows one solution using the `EXISTS` operator. See if you can duplicate it in the **Rule Editor** by following the sequence of steps in Table 8.

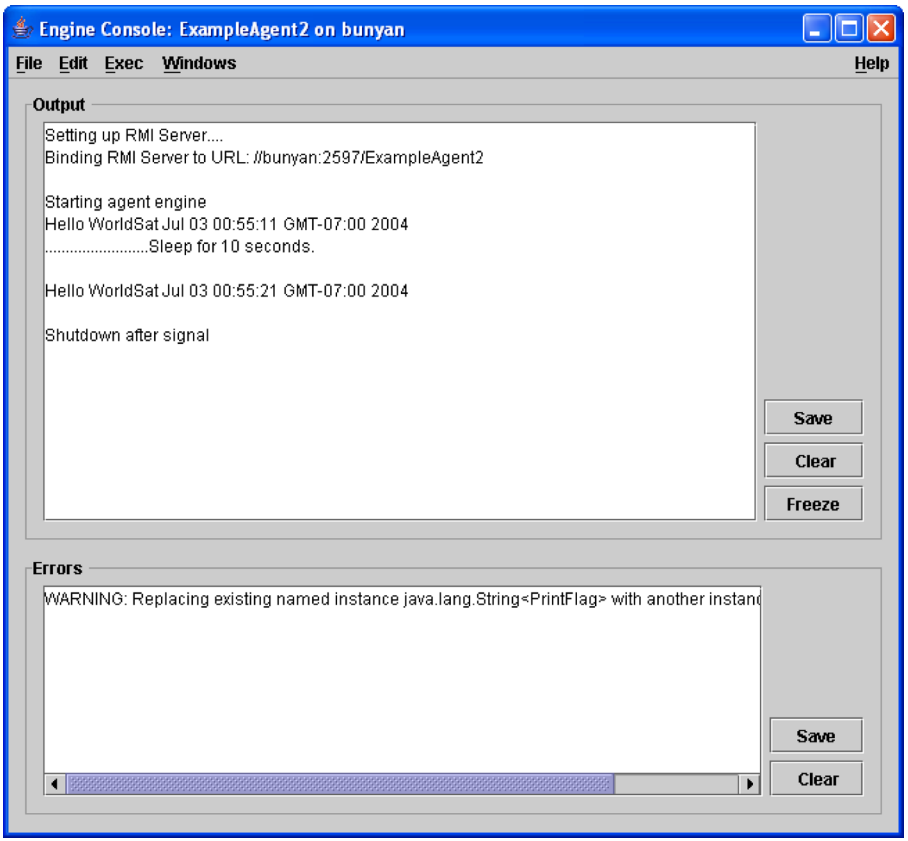

Figure 50. Running the Agent

Rerun the agent to ensure that only a single Hello World output is printed. The sequence of steps to create this pattern is shown in Table 8.

Table 8. Rule Creation (in Mental Conditions)

Step	***Instructions***
1.	Select <Operators> → NOT
2.	Select <Operators> → EXISTS, 1 quantified variable
3.	New Variable, Java Type String, name ?str
4.	Defined Variable, Java Type, ?str
5.	Select <Operator> EQUALS
6.	Defined Variable, Java Type, ?str
7.	<Value> String, “Quit Printing Now”

You can run the agent now to ensure it operates as you wish.

Step 6. Add Initial Objects.

Now we want to add some initial object so that the agent prints out three messages and then quits. To do this we first need to create two initial objects, `currentCount` and `targetCount`. (Note that variables and instances may have spaces embedded in their names. However in most cases, for the sake of readability, we recommend using `currentCount` or `current_count` in place of `current count`.) We will then modify the `Hello` rule’s RHS to update the `currentCount` object and then modify the `Quit` rule to see if the current count has reached the target count.

Step 6a. Create Initial Objects

To create an initial object, select **Tools → PAC Editor** from the **Agent Manager** menu bar. Select the **Java Instances** tab at the top of the **PAC Editor** window. The first object we create, `currentCount`, will be of type `Integer` and initialized with a value of 1.

Enter the name *currentCount* in the **Name** text field inside the **Java Instance Properties** panel. Also enter a short description of the instance and click on the **Enter** button. Select **<Java> → Integer** in the pull-down menu and click on the **Initial Java Instance** check box, then enter a *1* in the **initial value** text field. Finally add this instance to the **Defined Java Instances** list by clicking on the **Add** button next to the text field. Now, click on **Add** in the **Defined Java Instance** panel and the value will be entered in the **Defined Java Instances** text field. Repeat this process for the `targetCount` Java instance, with a 3 as the initial value. Now save the initial objects using the **File → Save** menu item. At the completion of this process you should have a window that looks like Figure 52.

Step 6b. Alter the Hello rule's RHS.

We now need to modify the `Hello` rule's RHS so that it will update the current count every cycle. To do this open the Rule Editor with the `Hello` rule loaded and select the RHS editor. Delete the **ASSERT** action in the **Define RHS Elements** list. Click on the **New** button in the **Actions** panel to clear the **Actions** text field. Click on the **SET_VALUE_OF** operator from the **<Operators>** pull-down menu. Now, to increment the value of `currentCount` by one, click on the **Instances...** pull-down menu and add the **currentCount** instance (in the **Java Instances** tab panel). This indicates that the `currentCount` is the instance to be incremented. Now we wish to increment the `currentCount` by 1, so select the **+** operator from the **<Operators>** pull-down menu. Now using the **Instances…** pull-down menu enter the `currentCount` instance. Next, use the **<Value>** pull-down and enter an `Integer` value of 1. The final state of the Rule Editor's RHS should look like Figure 53. If you want, you may also delete the `(NOT ...)` pattern from the `Hello` rule's LHS. Now save the `Hello` rule.

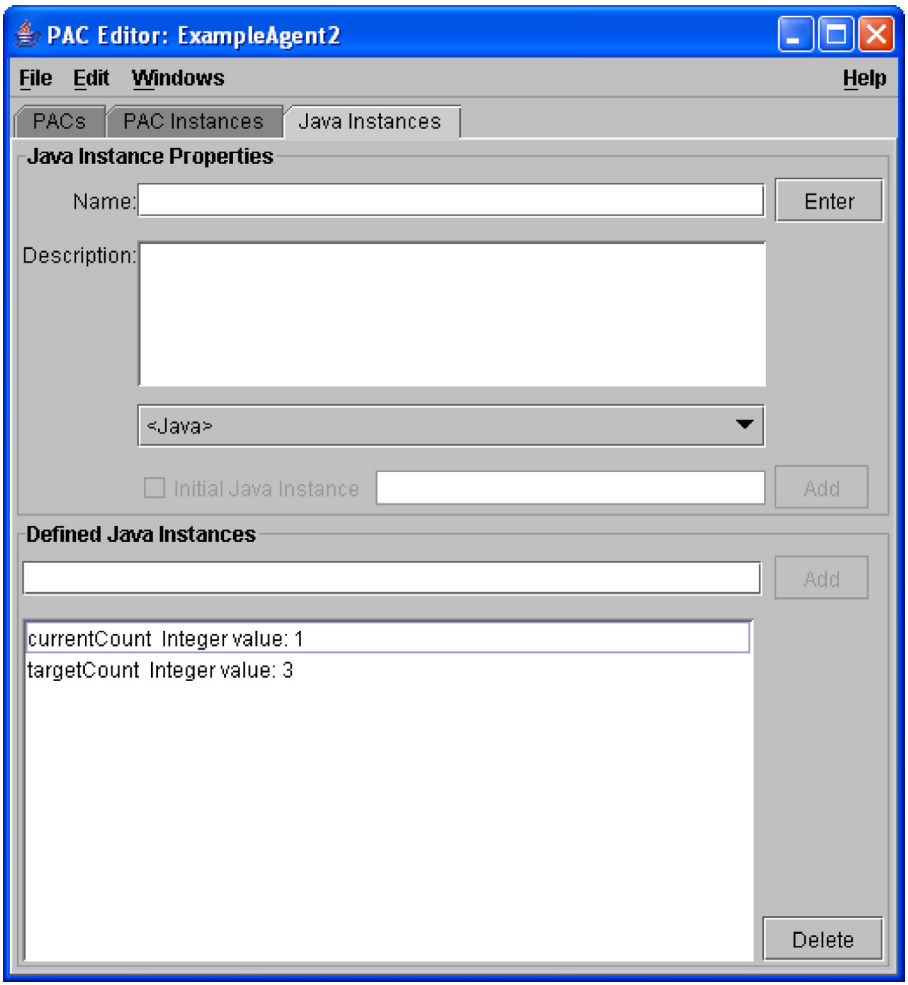

Figure 52. PAC Editor with Two Java Instances

Step 6c. Alter the Quit rule's LHS.

Now we want to alter the `Quit` rule so that it will recognize when the current count reaches the `targetCount`. We will no longer need

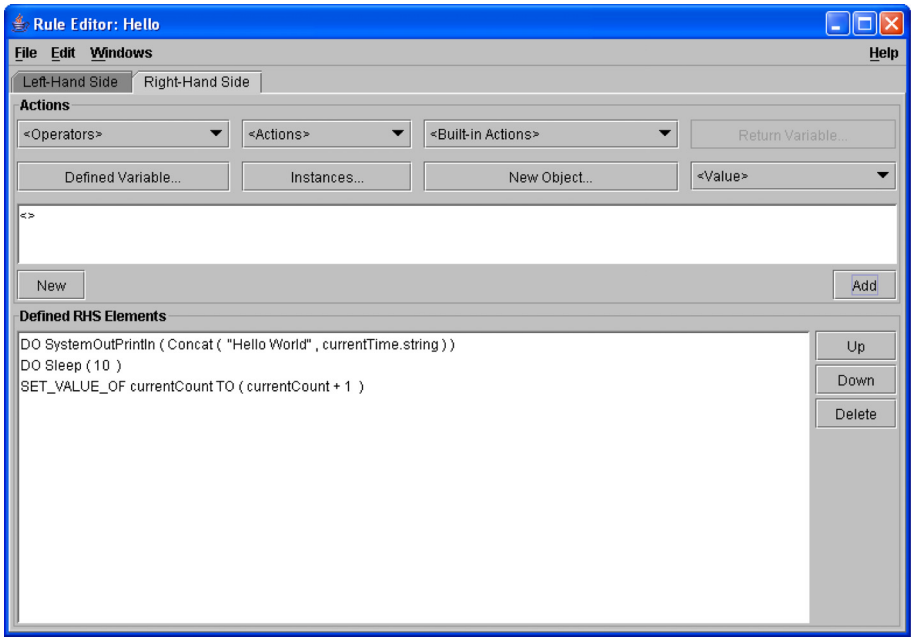

Figure 53. Rule Editor Showing Hello Rule RHS

the pattern to test for the `Quit Printing Now` string, so delete the pattern **(?str EQUALS "Quit Printing Now")** using the **Delete** button.

Click on the **New** button to clear the **accumulator's** text field. Now add a pattern that tests to see if the `currentCount` is equal to the `targetCount`. Click on the **<Operators>** pull-down menu and select the numeric equal sign ('**=**'). You may need to scroll down the list of menu item entries to find the equal sign. Now use the **Instances...** pull-down menu and enter the two Java instances (`currentCount` and `targetCount`). The Rule Editor should now look like Figure 54 for the `Quit` rule's LHS. Now, save the rule and make sure that it runs correctly and prints out "Hello World" (followed by the time) three times in succession.

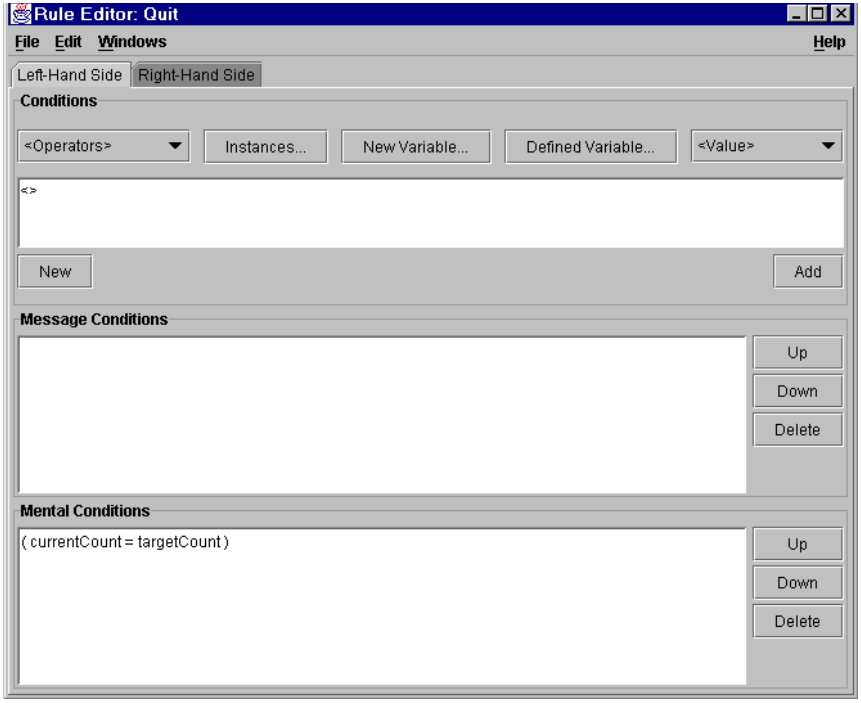

Figure 54. Rule Editor

Chapter 8

Simple Agent with a PAC

In this chapter you will learn about PACs. These accessory classes can be viewed as plug-in modules that extend the behavior of your agents. You will learn how to:

- Create an Ontology
- Use the Object Modeler
- Create a PAC
- Create rules that use the PAC
- Create an agent that uses the PAC

We have been creating agents that are self-contained. Everything that these agents are required to do can be provided by built-in functions, objects, etc. However, most agents require much more varied capabilities than can be provided by an agent development tool. Therefore, AgentBuilder supports the creation of Project Accessory Classes (PACs). PACs are custom classes coded in Java and designed to perform some specific tasks that augment the agent's behavior.

The PACs allow you to develop complex agents capable of performing a wide variety of activities. The developer must create or import the Java classes that will be used as PACs. AgentBuilder also provides support for creating templates based on the object model created inside the Ontology Manager's Object Modeler. Moreover, Acronymics, Inc. also provides libraries of PACs with specific capabilities (email, NNTP news reader, database access, etc.).

In this example, we will create an agent with a simple PAC. As you work through this example you will learn how to integrate custom classes into your agents. The Object Modeler (accessed from the Ontology Manager) allows you to define each class, its methods and attributes (data members). The resulting object model is then imported into the PAC Editor. This allows the PAC to be utilized in any rules that the agent requires. The agent can access methods or attributes in the PAC from either the LHS or RHS of its behavioral rules.

The agent created in this example will print out "Hello World" followed by the time and operates in much the same way as the previous example agents. However, we will utilize user-implemented Java code to perform the printing rather than using the built-in `SystemOutPrintln` action. You will learn how to define an action and connect an action to a user-defined Java method at run-time. You will also learn how values can be passed from the agent's mental

state to an action and then to a method within the PAC. The steps in creating this agent are summarized in Table 9.

Table 9. Simple Agent Using PAC

Step	***Description***
1.	Create New Ontology
2.	Create Hello Object in Object Modeler
3.	Generate Java Template File for the Hello Class
4.	Create New Agent
5.	Import Hello class into a Hello PAC
6.	Create PAC Instance
7.	Create Java Instance
8.	Create Rules to Use the PAC a. Create the Init Rule b. Create the Print Rule c. Create the Quit Rule
9.	Run Agent

Step 1. Create New Ontology

The first step in developing the Hello PAC is to create a new ontology (which we will call the *Quick Tour Ontology*). This can be accomplished by clicking on **Ontologies** in the **Project Manager** window. The Ontology Manager (Figure 55) displays the various ontologies that are available either from the user's personal repository or from the system repository. Note that AgentBuilder pro-

vides some system ontologies that are used in some of the example agents.

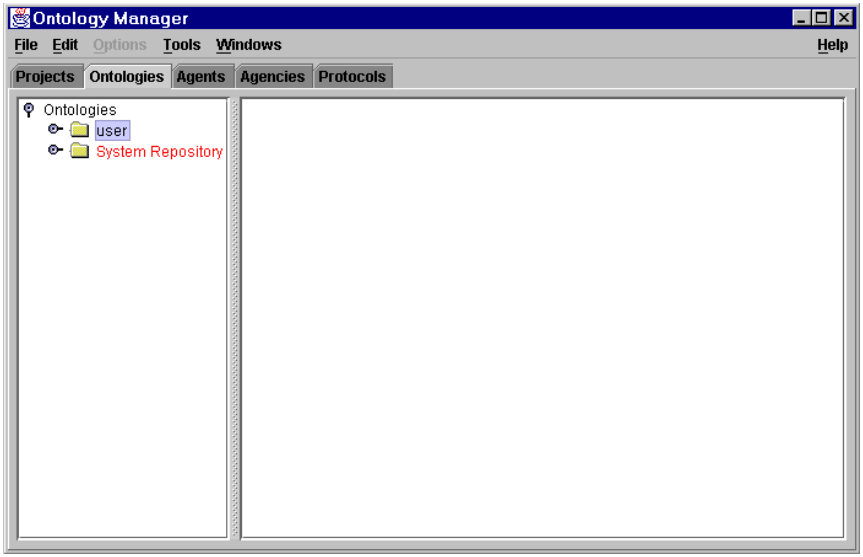

Figure 55. Creating a New Ontology

To create a new ontology, right-click on the folder labeled **user** and select **New Ontology** from the pop-up menu. AgentBuilder will then display the **Ontology Properties** window (Figure 56). Fill in the name of the ontology (e.g., *Quick Tour Ontology*) along with a short description of the ontology and click on the **OK** button to enter the properties into the new ontology.

Step 2. Create a Hello Object in the Object Modeler

Now we need to populate the `QuickTour` ontology with a `Hello` class. To do this, select the **Quick Tour Ontology** from the left panel in the Ontology Manager and then select **Tools → Object**

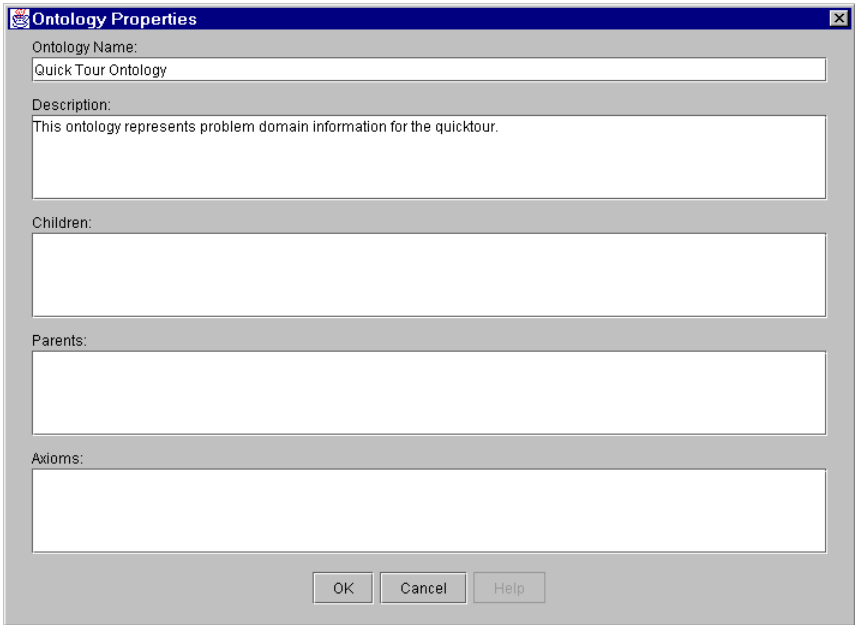

Figure 56. Ontology Properties for Quick Tour Ontology

Model.... This will display a blank canvas where different classes can be organized, defined, and then imported into PACs. To create a node for the Hello class, right-click anywhere on the Object Modeler canvas and select **New Object...** from the pop-up menu. Fill out the name and description of the `Hello` class as shown in Figure 57. Click on the **Enter** button located next to the **Name** text field when you have finished.

Our `Hello` class will only have a single attribute named `greeting` which will be of type `String`. Select **String** from the **<Attribute Type>** combo box and enter the name (e.g., *`greeting`*) in the Name text field of the **Attributes** panel and click on the **Add** button. Note that when you add any attribute the corresponding `set` and `get` methods are automatically added to the **Methods** list at the bottom

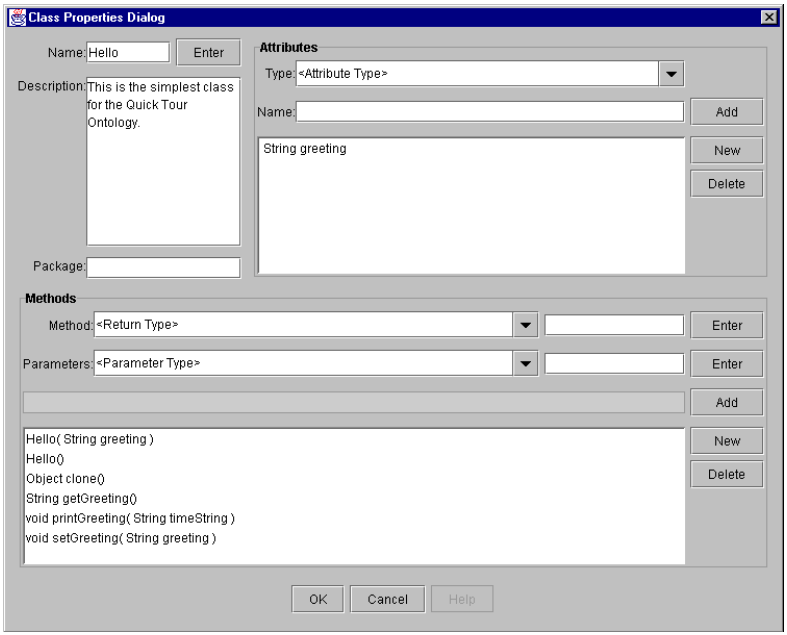

Figure 57. Class Properties for the Hello Class

of the panel. The **<Attributes Type>** combo box is editable, so if an attribute is needed that is not on the list it can be added in the combo box.

Note that the type must be fully qualified with its package name. All new types must be defined in the current object model or in some other ontology. For every class, the tool automatically adds an empty constructor and a clone method. For this class, you will need to define two new methods: a constructor with a `String` input, i.e., `Hello(String val)`; and a `printGreeting(String timeString)` method that will be invoked as one of the agent's actions. The `printGreeting` method will print a greeting stored in the PAC then print the input `timeString`.

To add a method, use the **Methods** panel of the **Class Properties** dialog to enter the method's name (e.g., `printGreeting`) and then specify the return type and parameter types and names. Enter *`printGreeting`* in the top text field in the **Methods** panel, then select `void` for the return type, then click on **Enter** to move the method name into the accumulator. Next, select **String** for the parameter type and type a parameter name (e.g., *`timeString`*) into the lower text field in the **Methods** panel, then click on the lower **Enter** button. Then click on the **Add** button to add `void printGreeting(String timeString)` to the method list.

Follow the same procedure to define the single-input constructor, using method name `Hello` (remember that the constructor method name must be the same as the class name), no return type, and a `String` parameter with a name of your choice (e.g., `val`). You don't need to specify the return type because the tool will recognize that a method with the same name as the class is a constructor for that class, so it will not have a return type.

For methods that have more than one parameter you'll need to sequentially enter the parameter types and names on the second line in the **Methods** panel, clicking on the lower **Enter** button to store each parameter into the accumulator. Then when all parameters have been specified, click on **Add** to store the method signature into the method list. For the current example PAC, after you finish adding the single attribute and the two methods, the **Class Properties Dialog** should look like that shown in Figure 58. If you make a mistake, click on the **New** button to clear the **Methods** panel field and then enter your information again. Click on **OK** to dismiss the **Class Properties Dialog**.

You can examine the **Object Modeler** output. If you do, you will see a display similar to Figure 59. You need to save the object model by selecting **File → Save.**

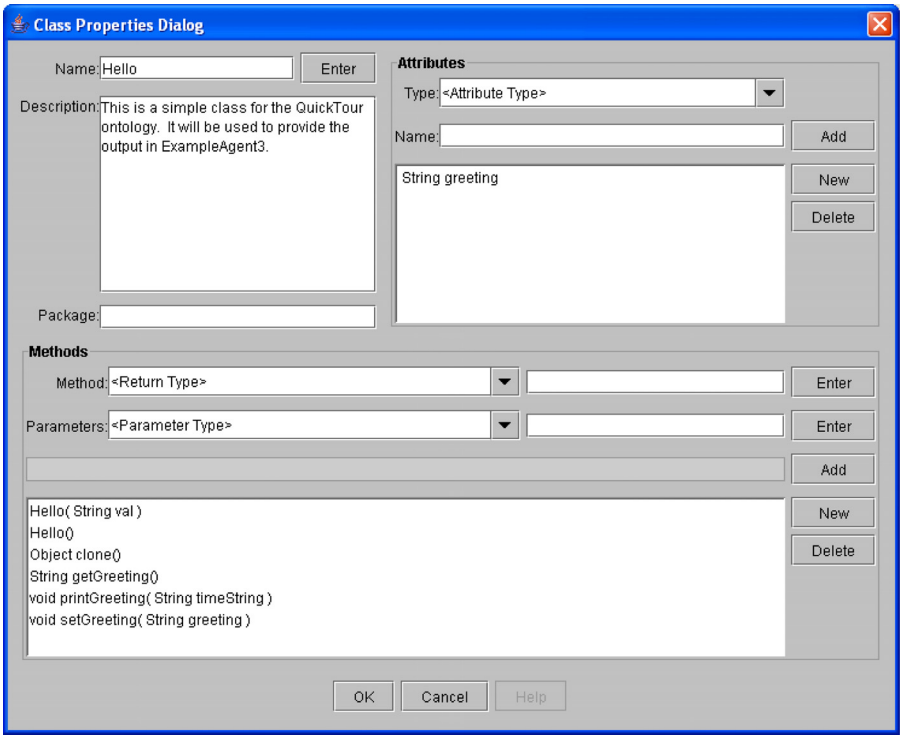

Figure 58. Object Properties Dialog for Hello Class

Step 3. Generate Java Template File for the Hello Class.

To assist you in creating the underlying classes represented by any object in your object model, AgentBuilder provides facilities to automatically generate a file that can act as a template for the class. To generate the file `Hello.java` in the Object Modeler, select **File → Generate Java Files...**. This will bring up an **Export Dialog** window as shown in Figure 60. With this dialog, you can select which objects should have Java files automatically created or you can click on the **Select All** button and select all objects in the cur-

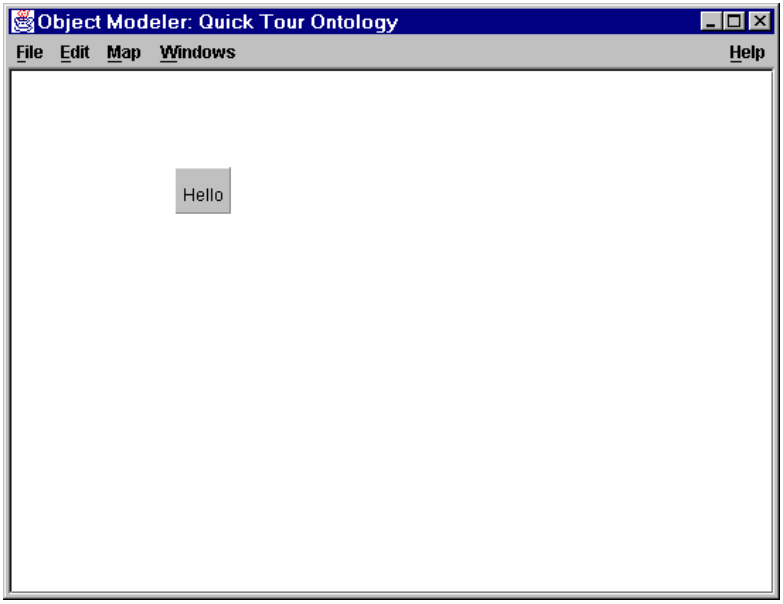

Figure 59. Object Modeler After Defining Hello Class

rent model. Use the **Directory** panel to specify the directory where you wish to save the generated Java files.

Alternatively, you can use the **Browse** button to graphically navigate your computer's file structure. Pressing the **Browse** button will bring up the **Directory Dialog** shown in Figure 61. You can use this dialog to navigate your file space and determine where to save your files. The three buttons near the top of the **Directory Dialog** allow you to go up one level in the directory hierarchy, create a new directory, and specify your home directory. When you have selected the appropriate objects and determined the directory where you wish to save the resulting Java files, click on the **OK** button. Examine the Java `Hello.java` located in the directory that you specified and ensure that your `CLASSPATH` environment contains this directory. Note that `get` and `set` methods are completely defined for you in the generated code.

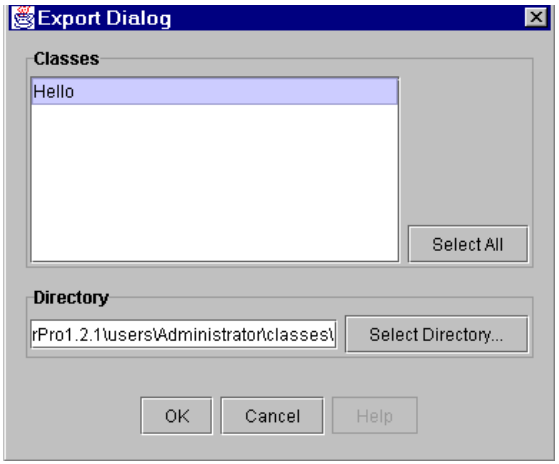

Figure 60. Export Dialog

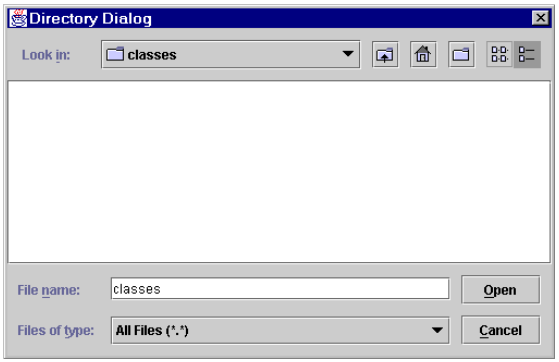

Figure 61. Directory Dialog for Automatically Generated Java Files

Use any text editor (e.g., emacs) to open the `Hello.java` file, then type the line:

```
greeting = val;
```

inside the `Hello(String val)` constructor. Next, in the `Hello()` empty constructor, put the line:

```
greeting = "Hello World";
```

Inside the `printGreeting` method put the line:

```
System.out.println(greeting + ": " + timeString);
```

All PACs must implement the `Cloneable` interface, i.e., they must provide a `clone()` method which returns a `java.lang.Object`. This is needed because PAC instances are sometimes cloned when they are manipulated by the agent engine, such as when PAC instances are used as action arguments. The default clone method generated by the Agent Manager will attempt to invoke `super.clone()`, the `clone()` method of the superclass of the PAC. Usually the superclass is the `java.lang.Object` class, so `super.clone()` will simply create a bitwise copy (i.e., a shallow copy) of the original object. For most PACs this is adequate, but there may be some situations where you'll want to provide your own `clone` method. In a graphical interface PAC, for example, you may want the clone method to simply return a reference to the instance without actually cloning the instance. This could be done by changing the body of the `clone` method to return `this`. In other situations you may need to provide the code for a PAC's `clone` method in order to get a deep copy. In a deep copy, the sub-objects referenced in the original object are themselves cloned; in a shallow copy only the *references* to the sub-objects are copied.

The complete file, as modified, is shown in Figure 62.

You now need to save and compile the `.java` file to produce a `.class` file. If you do not have a JDK or other Java development environment then the built-in `Hello.class` file can be used. This file is provided in the jar file included with the AgentBuilder distribution.

```
/*
 *
 * File: Hello.java
 *
 * Automatically generated by AgentBuilder (tm).
 */

import java.io.Serializable;

public class Hello implements Cloneable, Serializable
{

   //Attribute list

   String greeting;


   //Method list

   /**************************************************************************/
   public  Hello()
   {
                        greeting = "Hello World";
   }

   /**************************************************************************/
   public Object clone()
   {
      Object clonedObject = null;
      try
      {
         clonedObject = super.clone();
      }
      catch( CloneNotSupportedException exception )
      {
         exception.printStackTrace();
      }
      catch( OutOfMemoryError exception )
      {
         exception.printStackTrace();
      }

      return clonedObject;
   }

   /**************************************************************************/
   public void setGreeting( String greeting )
   {
      this.greeting = greeting;
   }

   /**************************************************************************/
   public String getGreeting()
   {
      return(greeting);
   }

   /**************************************************************************/
   public void printGreeting( String timeString )
   {
                        System.out.println(greeting + ":  " + timeString);
   }

   /**************************************************************************/
   public  Hello( String val )
   {
                        greeting = val;
   }

}/* End of: Hello.java*/
```

Figure 62. Hello.java Code Listing as Modified

Step 4. Create new agent

Now we need to create a new agent. Use the Project Manager to create a new agent by selecting the **Quick Tour Agency** and then selecting **File → New...**. Enter the agent's name (*ExampleAgent3*) and a description of the agent as shown in Figure 63. Now click on the **OK** button in the **Agent Properties** window to associate this information with the agent.

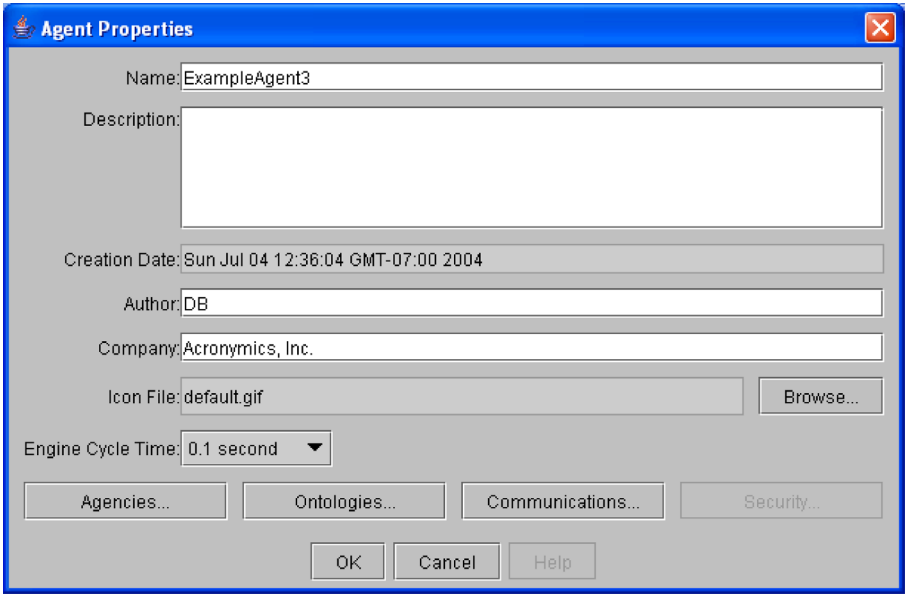

Figure 63. Agent Properties

Step 5. Import Hello class into a HelloPAC

Now we need to import the `Hello` class into a PAC. If you don't have the Agent Manager open for **ExampleAgent3** then go to the AgentBuilder Program Manager and select **ExampleAgent3**. Click on the **Agents** tab. In **Agent Manager**, select **Tools → PAC Editor** from the **Agent Manager** menu bar. This will open the PAC Editor

for **ExampleAgent3**. Now from this editor, select the **File** → **Import…** menu item. This will cause AgentBuilder to display the **Import Dialog** (Figure 64).

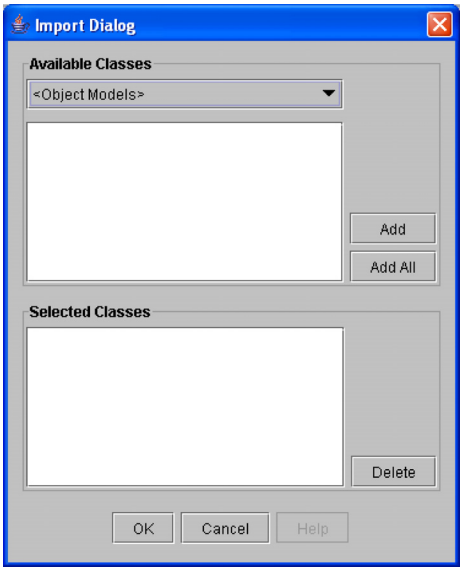

Figure 64. Import Dialog

To import our `Hello` class into a PAC, use the **<Object Models>** combo-box and select **Quick Tour Ontology** and then select **Hello** from the list below the combo-box and click on **Add**. **Hello** will appear in the **Selected Object** list on the dialog. Click on **OK** to confirm the import of the object listed in the bottom **Selected Objects** list. Note that the list at the bottom of the **PAC Editor** window (in the **Defined PACs** panel) now has the `Hello` PAC included. Select the `Hello` PAC in this list and you can see all the attributes. Figure 65 shows what the PAC Editor looks like after importing the `Hello` PAC and selecting the **Hello** package.

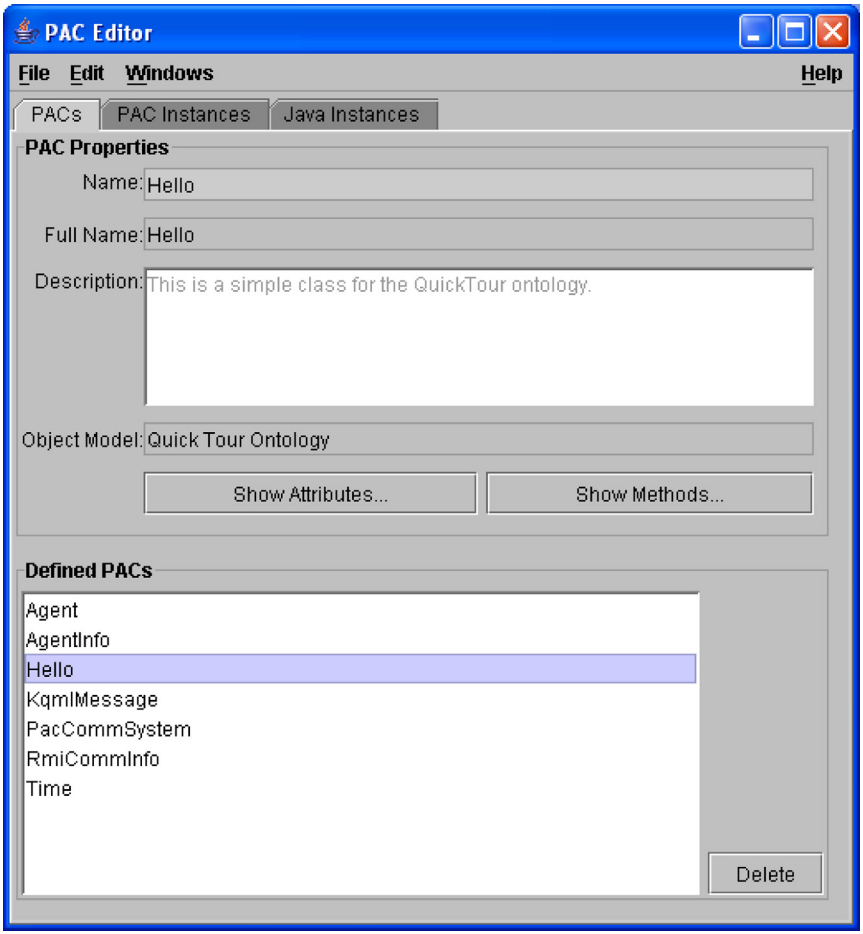

Figure 65. PAC Editor

Step 6. Create PAC Instance

Before creating the agent's rules, we need to first create a PAC instance that we can reference from inside the rules. However, this PAC instance will not be instantiated until the proper rule is fired.

Using the **PAC Editor** on **ExampleAgent3**, click on the **PAC Instances** tab at the top of the panel. Now, enter a name *hello-PACInstance* and description into the appropriate text areas. Use the **<PAC> combo-box** pull-down menu and select **Hello** to tell AgentBuilder what type of PAC to use. Also, be sure that you **do not** click on the **Initial PAC Instance** button. After entering the name, description and PAC type, click on the **Enter** button. This will place the instance in the accumulator line at the top of the **Defined PAC Instances** panel. Now click on the **Add** button to add this to the list below. The PAC Editor should look like Figure 66 when you are finished. Now save the PAC instance using the **File → Save** menu item and close the **PAC Editor** window using the **File → Close** menu items.

Step 7. Create Java Instance

To create this `readyToPrint` PAC Java instance, open the PAC Editor (using the **Tools** menu in the Agent Manager), click on the **Java Instances** button and name the instance *readyToPrint*. Provide a brief description and click on the **Enter** button. Select **Boolean** under the **<Java>** pull-down menu and then click **Add**. Do not click on the **Initial Java Instance** checkbox. Now save this instance in the PAC editor and close the PAC editor by selecting the **File → Close** menu item.

Step 8. Create rules to utilize PAC

There will be three rules used in the agent. The first rule `Init` will create a new instance of our `Hello` PAC and demonstrate how to call a constructor on the PAC to initialize the PAC instance. The final two rules `Print` and `Quit`, are similar to the rules we created in `ExampleAgent2` and will be used to control printing. However, this agent will be using PAC-specific methods to print the greeting

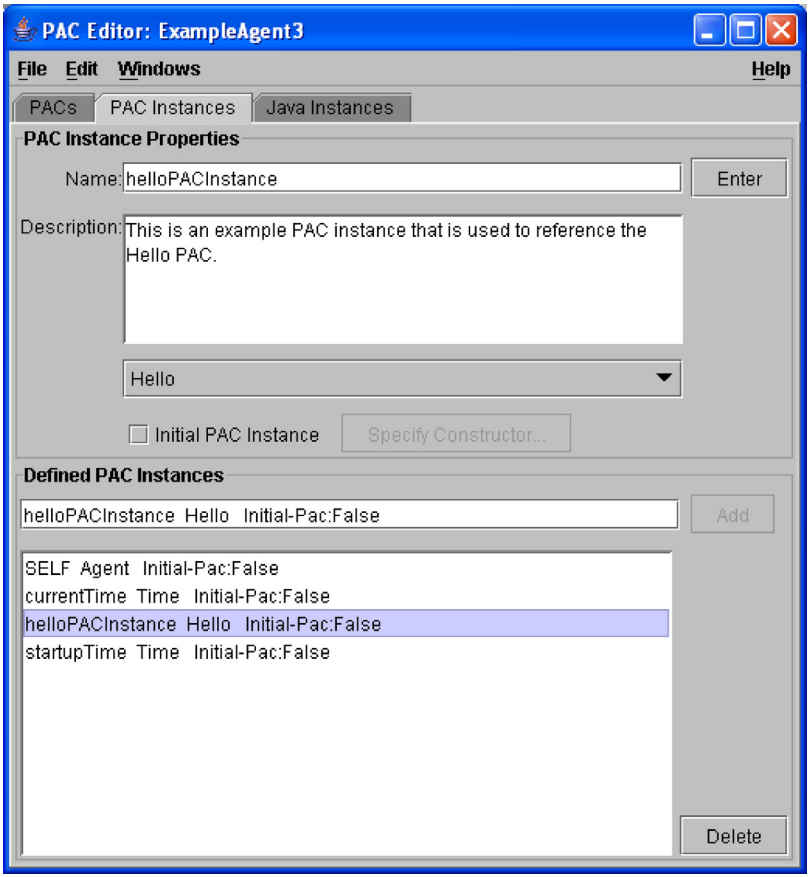

Figure 66. PAC Editor

rather than use the built-in actions we used in our first and second agents. This is simply for demonstration; usually you will want to use built-in actions whenever possible.

Step 8a. Create Init rule.

The `Init` rule is triggered by the `startupTime` instance and will be used to create an instance of the `Hello` PAC at run-time. From the

Agent Manager, select **Tools → Rule Editor** to open the Rule Editor. Use the **Edit → Properties** menu item and select the **Rule Properties** dialog to create a rule with the name *Init*; provide a short description of the rule. On the LHS of the rule, create a mental condition that will bind to the `startupTime`. Click on **Add** to add it to the **Mental Conditions** list.

On the RHS of the rule, we want to create and assert a PAC instance. To do this, we need to call a constructor for the PAC and utilize the `ASSERT` operator. Using the **<Operators>** pull-down menu, select **ASSERT** from the **Actions** panel. The input parameter of the **Assert Dialog** is the name of the instance being asserted, in this case `helloPACInstance`. This will be the name associated with the instance within the agent's mental model. For the second parameter of the `ASSERT`, specify a new instance of `Hello` PAC by clicking on the **New Object...** button. The **New Object Dialog** will then be displayed. Click on the **PACs** tab on the top panel and then select the `Hello` PAC from the list of PACs available. Now select **<Constructor> → Hello(String greeting)** so that you can specify the greeting. The **New Object** dialog should look like Figure 67 after you have finished.

Click on the **OK** button to confirm the creation of this new object. Now create a string (using **<Value> → String**) such as *`Hello World: The time is`*. Click on the **Add** button to create the first RHS element. Now we also want to assert a new string to trigger the actual printing rules. To do this, assert a new belief named `readyToPrint` and use the **<Value>** button to make its type `Boolean` with a value of `true`. Again, click on **Add** to add this expression to the **Defined RHS Elements** list. Now add the element to the RHS and save the rule. Figure 68 shows the `Init` rule viewed using the Agent Manager.

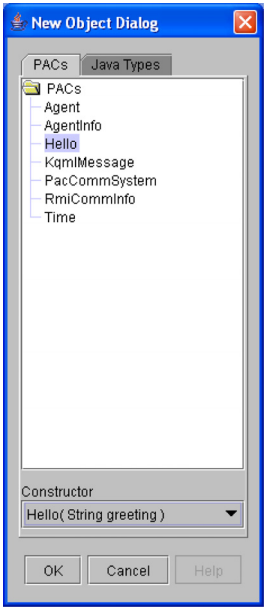

Figure 67. Creation of Hello Object in New Object Dialog

Step 8b. Create Print Rule.

Triggering the print rule should depend on the `readyToPrint` belief asserted by the `Init` rule. Clear the Rule Editor by selecting **File → New → New Rule...**. In the **Rule Properties** dialog, enter the name of the rule (i.e. *`Print`*) and then a short description of the rule. The first pattern should bind to the `Boolean` instance named `readyToPrint`, with a value of `true`. To create the first pattern, select **<Operators> → EQUALS** then click on **Instances…** to bring up the **Instances** dialog. Click on the **Java Instances** tab at the top of the dialog, then select **Boolean readyToPrint**, then click on **OK**. Next, select **<Value> → Boolean** then choose `true` in the field; click on **OK** to complete the pattern in the accumulator. Click on the **Add** button to move the pattern into the pattern list. Add the pattern (BIND currentTime).

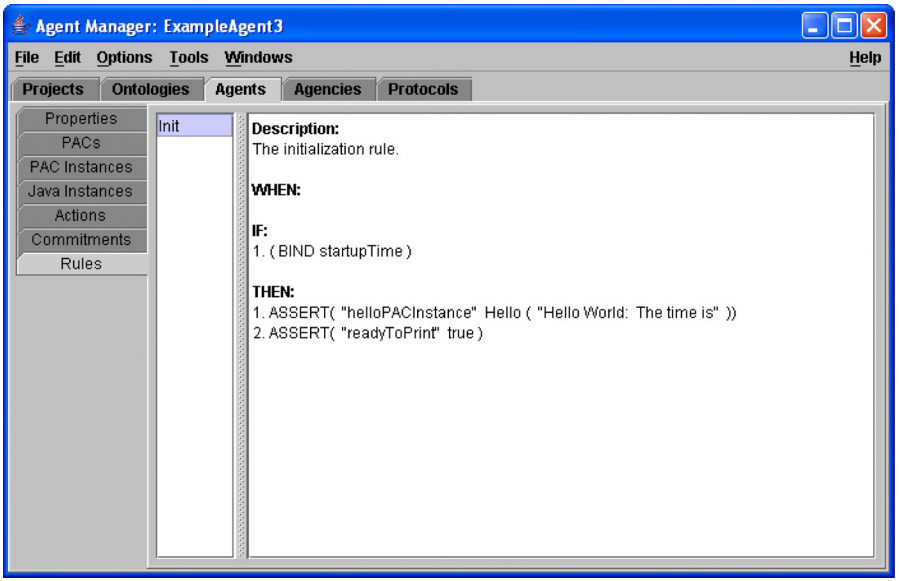

Figure 68. Init Rule in Agent Manager

Now select the RHS of the Rule Editor. The Rule Editor allows you to directly invoke methods on instances and variables. Click on the **Instances…** . By double-clicking on the **helloPACInstanc**e you can select the `printGreeting` method. Select it and then click **OK**. Now we want to use the current time string as the input parameter to this action. This is accessed by clicking on the **Instances…** button and selecting the `string` attribute of the `currentTime` object. You will need to click on the small circle next to the current time object to see the attribute.

Put in a sleep action (for 10 seconds) and increment the value for the current count Java instance. You may need to go to the PAC Editor to create the `currentCount` and `targetCount` Java instances.

Create a new action to increment the `currentCount` variable. Use the `SET_VALUE_OF` action from the **<Operators>** button. These steps are similar to the steps performed in developing **ExampleAgent2**.

See, for example, Step 6 in Chapter 7. Figure 69 shows the RHS and LHS of the `Print` rule viewed in the **Agent Manager.**

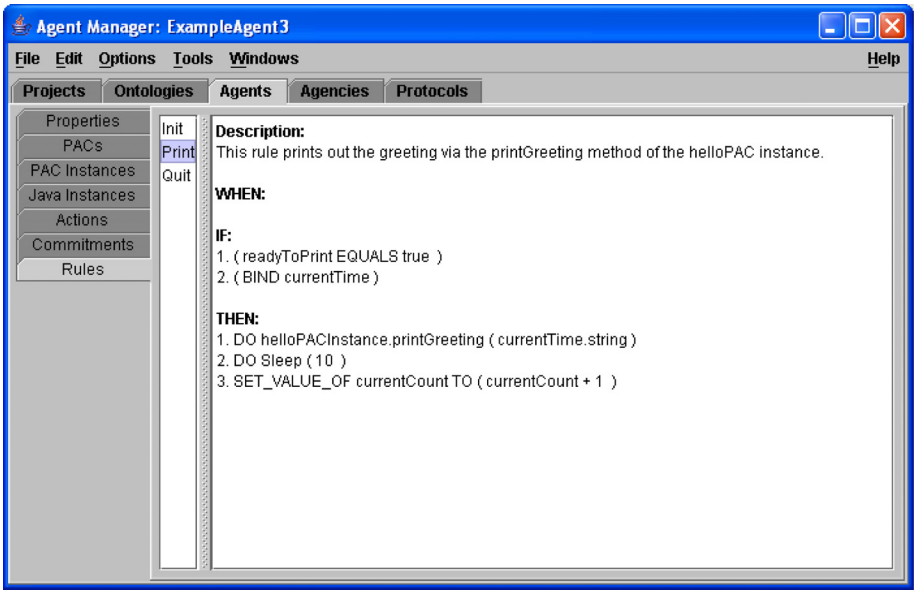

Figure 69. Print Rule in Agent Manager

Step 8d. Create Quit rule.

Save the `Print` rule and then clear the Rule Editor using the **New** and **Delete** buttons. Now enter *`Quit`* in the **Name** text field and a short description of the `Quit` rule. Now enter a LHS condition in the mental state that will cause the rule to fire when the current count reaches the target count value. (Again, if you need detailed instructions for completing this step, refer to the discussion for constructing ExampleAgent2, Step 6c). On the RHS, simply shutdown the agent engine with **<Built-In Actions> → ShutdownEngine**.

Step 9. Run agent

Now save the agent in the Agent Manager by selecting **File** → **Save**. You can now run the agent from the **Options** menu in the Agent Manager and see that it prints three times and then shuts down.

Chapter 9

An Agent with a Graphical PAC

In this chapter you will learn how to add a simple graphical user interface to an agent. You will:

- Create a new ontology
- Create the GUI interface PAC
- Send and receive messages betweent the GUI and the agent

This example agent will utilize a more complicated PAC that provides a GUI interface. The interface PAC will run on its own thread and use a built-in PAC called `PacCommSystem` to communicate with the agent via messages. The `PacCommSystem` class is used to simplify communication between an agent and an interface PAC. In addition, we will utilize an ontology from the system repository that contains the Java code for constructing and displaying the user interface. This interface class, `HelloWorldFrame`, defines a simple interface with two buttons and a text area for printing the agent's greeting. In this example, we have the GUI send KQML messages to the agent in order to demonstrate KQML message processing. Figure 70 shows the `HelloWorldFrame` interface.

Figure 70. A Graphical User Interface for an Agent

The steps involved in constructing this agent and its associated graphical user interface are shown in Table 10.

Step 1. Create appropriate ontology

The first thing that we need to do is develop the proper ontology. Rather than creating a new one, however, it will be more convenient to utilize the Quick Tour Ontology created in **ExampleAgent3**.

Table 10. Creating an Agent with Graphical Interface PAC

Step	***Description***
1.	Create Appropriate Ontology
2.	Create New Agent
3.	Create Initial PAC Instances
4.	Create Rules a. Create the Build HelloWorldFrame rule b. Create the PrintGreeting rule
5.	Run the Agent

From the **Project Manager**, click on the **Ontologies** tab**.** Select the **user** folder in the left panel. Now, open the folder and select the **Quick Tour Ontology**. Select **Object Model** from the **Tools** menu. Now, select the **Import Class Files…** menu item from the **File** menu. This dialog allows you to import class definitions for classes that are already written, in this case the `HelloWorldFrame` class. Type in *`com.reticular.agents.helloWorld.HelloWorldFrame`* into the **class** field and click the **Add** button. The dialog should appear as shown in Figure 71.

Next click **OK**, a warning dialog will appear that notifies you that the class is not implemented as serializable. This is OK, since this is a graphic PAC and will not be sent in a communication. You now have an usable class definition for the `HelloWorldFrame` class in the Quick Tour Ontology **Object Modeler**. Figure 72 shows methods and attributes available in `HelloWorldFrame`. Remember, to examine the properties of an object you must right-click on it and select **Properties…** in the popup menu. Note there are no data members for this class; this means that there were no private or public vari-

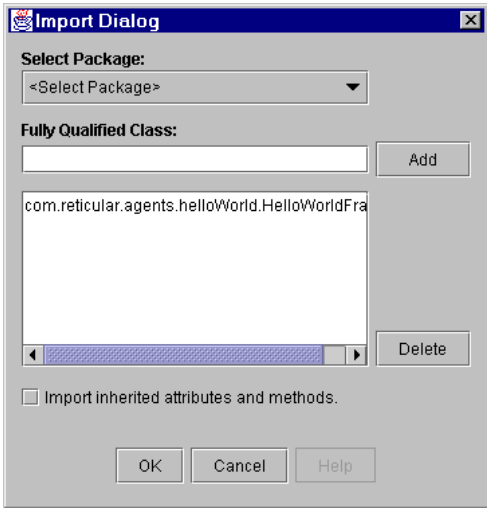

Figure 71. Class Import Dialog

ables with the correct `get` or `set` methods. This is appropriate for a class that is going to run on its own thread using the `run` method. Be sure and save the contents of the **Object Modeler** panel..

Step 2. Create New Agent

Select the **Project Manager** and then select the **Quick Tour Project**. Click on the **Quick Tour Agency**. We can now create a new agent inside the **Quick Tour Agency** (call the agent `ExampleAgent4`). In addition to providing the name, description, author and vendor field information, click on the **Communications** button. This will bring up a window as shown in Figure 73.

This **Communication Dialog** assigns default values for the port number when you create a new agent. There is no reason to change the default for this agent. The default is randomly choosen and varies between 1000 and 6000. In the combo-box, you can either type in

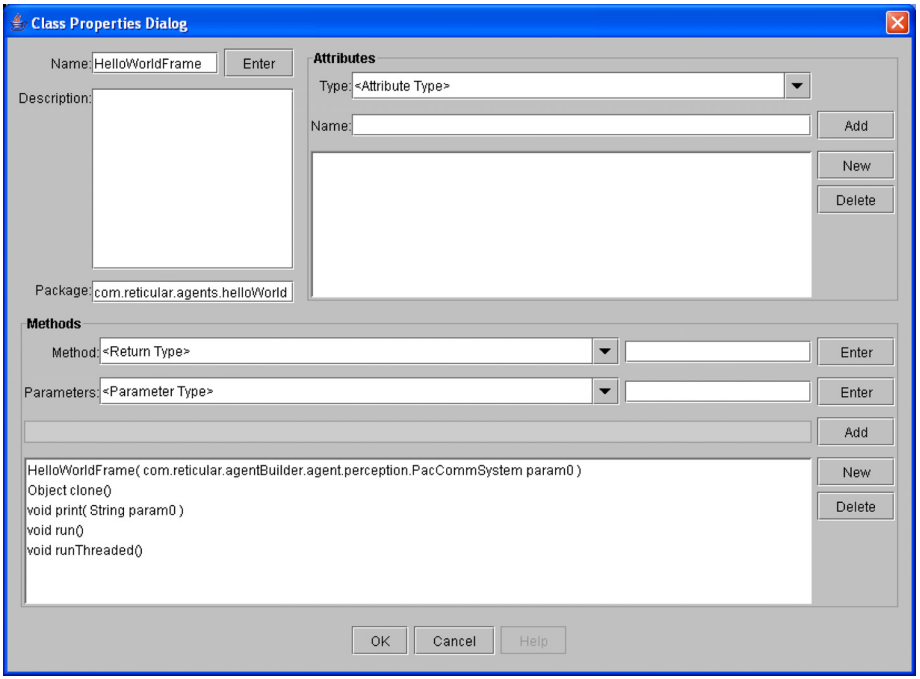

Figure 72. Object Properties Dialog

an IP number or a name and domain. Selecting **CURRENT_IP_ADDRESS** tells the agent to use the IP address of your computer. If you are defining agents that are going to run on multiple machines, then it is necessary to assign specific address to each agent. Click on the **OK** button to continue. If you haven't done so, click the **OK** button on the **Agent Properties** dialog to finish creating the agent

Step 3. Define the PAC Instance

In this example, we need to define an instance of a `HelloWorldFrame` object. However, before we can define a PAC instance, we must first import the PAC as we did in **ExampleAgent3**. Select

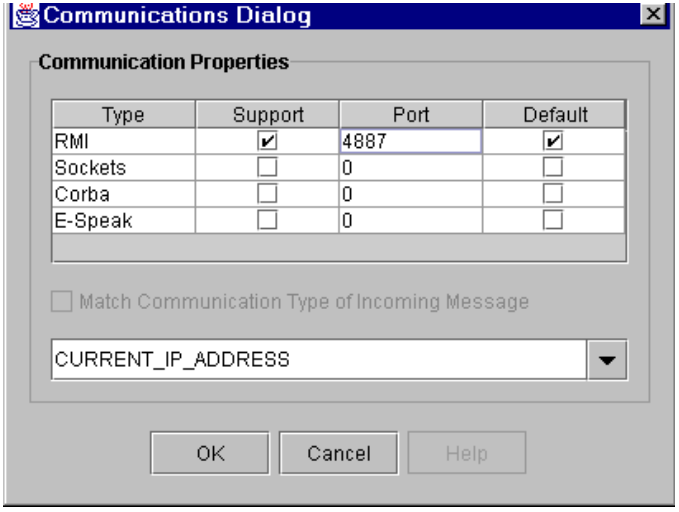

Figure 73. Communications Dialog

ExampleAgent4 in the Project Manager and then click on the **Agents** tab. From the **Agent Manager** menu bar select **Tools → PAC Editor**. In the PAC Editor, select **File → Import** and use the **Import Dialog** to import the `HelloWorldFrame` class into a PAC. In the **<Object Model>** pop-up menu, select **Quick Tour Ontology**. The **HelloWorldFrame** will appear in the **Available Objects** list[1]. Select **HelloWorldFrame** and click on the **Add** button. **HelloWorldFrame** will now appear in the **Selected Objects** list. Figure 74 shows the **Import Dialog**. Click **OK** to dismiss the **Import Dialog**. **HelloWorldFrame** will now appear in the **Defined PACs** list in the PAC Editor.

You can now define a PAC instance named `myHelloWorldFrame` that's of type `HelloWorldFrame`. Clicking on PAC Instances prompts you to save the results. Click on the **PAC Instances** tab at

1. Actually you will see the complete path such as `com.reticular.....\HelloWorldFrame`

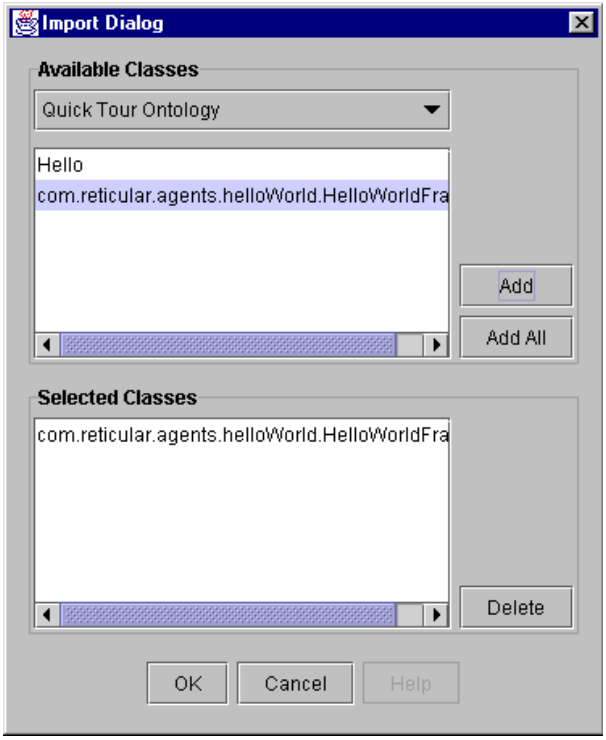

Figure 74. Import Dialog

the top of the PAC Editor and enter the name `myHelloWorldFrame` and a description, then click on the **Enter** button. Next select the **HelloWorldFrame** from the **<PAC>** pull-down menu, then click on the **Add** button. You should **not** click on the **Initial PAC Instance** check box; the construction of this instance should be done at run-time using values from the `SELF` agent belief. Finally, select **File** → **Save** in the **PAC Editor** window. Figure 75 shows the PAC Instance Editor.

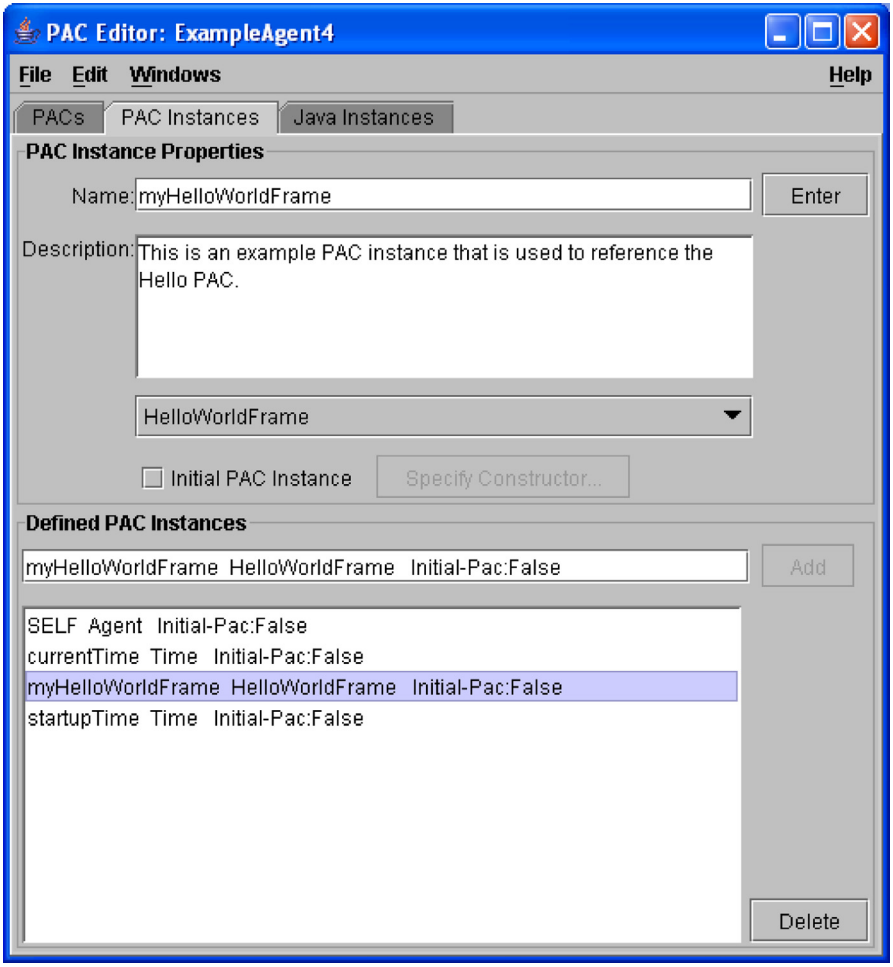

Figure 75. PAC Editor for HelloWorld

Step 4. Create Rules.

Next we wish to create the three rules that are needed in this agent. The three rules are a `BuildAndLaunchHelloWorld` rule, a `Print Greeting` rule, and a `Quit` rule. In addition to the features that we

have seen in the previous agents, this agent will utilize message conditions that test the messages received by the agent from the user interface. The messages come from the `HelloWorldFrame` instance in response to the user's button clicks on the interface.

Step 4a. Create the BuildAndLaunchHelloWorldFrame rule.

This rule will be responsible for building the `HelloWorldFrame` instance from information extracted from the `SELF` agent belief which will be present in the agent's mental model at start-up. The first step is to launch the Rule Editor from the Agent Manager using the **Tools → Rule Editor** menu item. Specify the name (*`Build and Launch HelloWorldFrame`*) and description of the rule using the **Rule Properties** dialog. The LHS is simple and requires a single mental condition: `BIND` to the Time instance named `startupTime` (**<Operators> → BIND** followed by **Instances**, `startupTime`).

Next, create a new variable of type `HelloWorldFrame`, label the new variable *`$helloWorldFrameVar`*. See **ExampleAgent3** if you are having trouble with creating variables.

The RHS will contain three patterns; one for creating a temporary variable of type `HelloWorldFrame`, a second for launching the `run` method on a separate thread and a third for assertion of the `HelloWorldFrame` into the mental state. To create the first pattern, switch to the RHS panel of the rule editor. Then select **<Operators> → SET_TEMPORARY** from the **Actions** panel. This creates a temporary variable. Then click on the **Defined Variable..** button. Choose the **$helloWorldFrameVar** from the PAC panel. The second slot in the SET_TEMPORARY pattern is to be filled with the new `HelloWorldFrame` object, which you specify by clicking on the **New Object…** button, which displays the **New Object** dialog. Note that you can resize this dialog to see the full path and name of the constructor. Figure 76 shows the **New Object Dialog**.

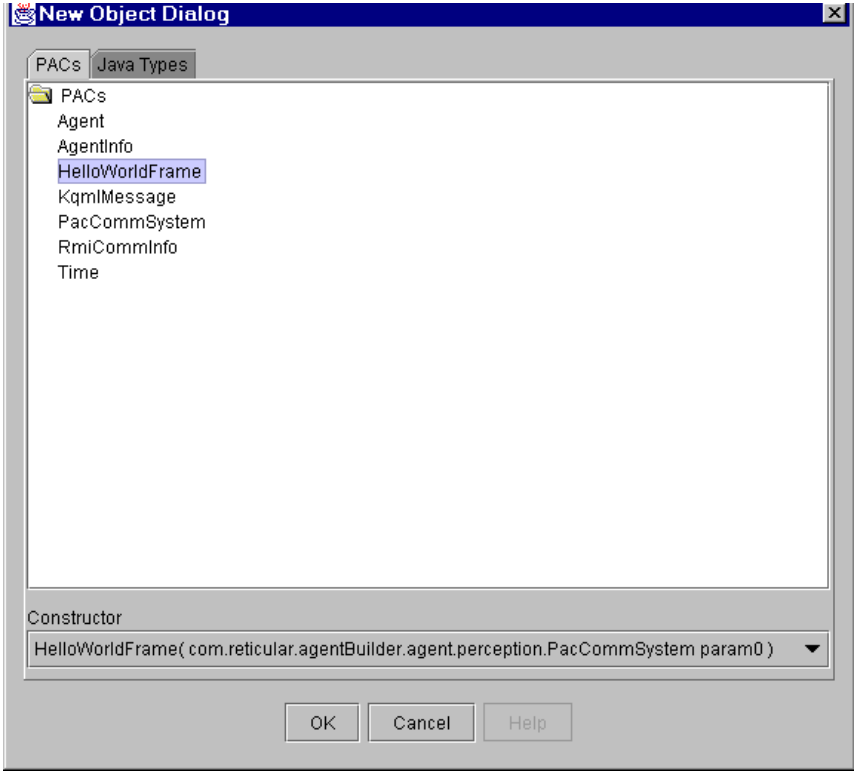

Figure 76. New Object Dialog

Select the **HelloWorldFrame** PAC, then select **<Constructor> → HelloWorldFrame(PacCommSystem)**, then click on **OK**. This stores **HelloWorldFrame(<param0>)** into the accumulator, which indicates that the next step is to specify the argument to the constructor. Click again on the **New Object…** button, select the **PacCommSystem** PAC, then select **<Constructor> → void PacCommSystem(AgentInfo,String)** and click on **OK**. This will write **PacCommSystem(<agentInfo>, <pacName>)** into the `HelloWorldFrame` argument slot in the accumulator. For the `PacCommSystem's` first argument, click on the **Instances…** button then

double-click on **Agent SELF**; this will display the `AgentInfo` attribute. Click on the `AgentInfo agentInfo` attribute then click on **OK**; **SELF.agentInfo** will be written into the `PacCommSystem's` first argument slot in the accumulator. Figure 77 shows the **Instances Dialog**.

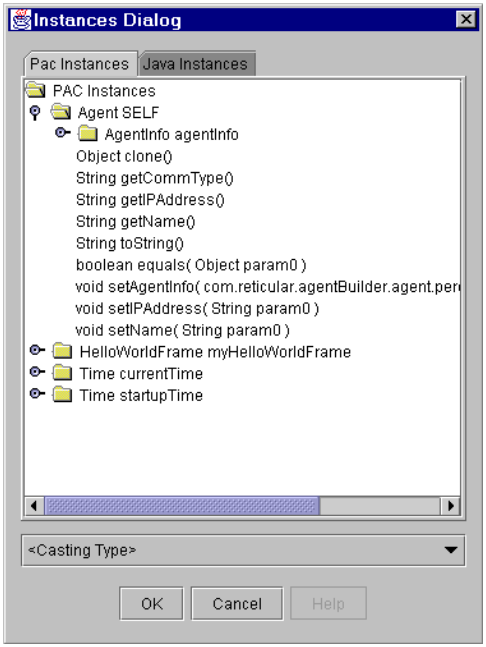

Figure 77. Instances Dialog

For the `PacCommSystem's` second argument, select **<Value>** → **String**, then enter *`HelloWorld:PAC`* into the value field and click on **OK**. This will store the name **HelloWorld:PAC** into the `PacComm-System`. Finally, click on **Add** to add this assertion to the list of RHS patterns, then save the current rule.

The next pattern is much simpler. We wish to start the `HelloWorld-Frame` on its own thread (this is when it displays). Bring up the

Defined Variables dialog, then double click **<?myHelloWorld-Frame>** and choose the **run** method, then select **OK**. This inserts an action pattern into the RHS of the rule. When this pattern is executed, the visible frame of the `HelloWorldFrame` will be shown. Click on **Add** to add this action to the list of RHS patterns. Add the **SleepUntilMessage** built-in action to a new pattern and click **Add**; now save the current rule.

The next pattern asserts the **HelloWorldFrame** into the mental model of the agent. Select **<Operators>** → **ASSERT** in the **Action** panel. Remember that the assert statement needs an instance name which is entered in the assert dialog. The instance name you enter should be *`myHelloWorldFrame`* or the label you gave it in the PAC instances panel. The next slot should be the `HelloWorldFrame` variable, `myHelloWorldFrame`. Choose it using the **Defined Variables...** button in the **Defined Variables** dialog. Finally, click on **Add** to add this assertion to the list of RHS patterns, then save the current rule. Add the action DO SleepUntilMessage() and save the current rule.

If you view the rule using the Agent Manager, click on the **Rules** tab and then select the `BuildAndLaunchHelloWorldFrame` rule. You will see a display similar to Figure 78.

Step 4b. Create the PrintGreeting Rule.

Now we want to create a rule that will detect messages from the user interface and print a greeting in the interface. This can be done by checking to make sure that the sender of the KQML message is named `HelloWorld:PAC`, the performative is `achieve`, and the content of the message is a `String` with the value `Say Hello`. These are the values in the message sent by the interface PAC when the user clicks on the interface's **Say Hello** button.

To implement this behavior, create the `PrintGreeting` rule and add a description. Next we need to create a new variable which will bind to any incoming KQML messages. In the **Conditions** panel of

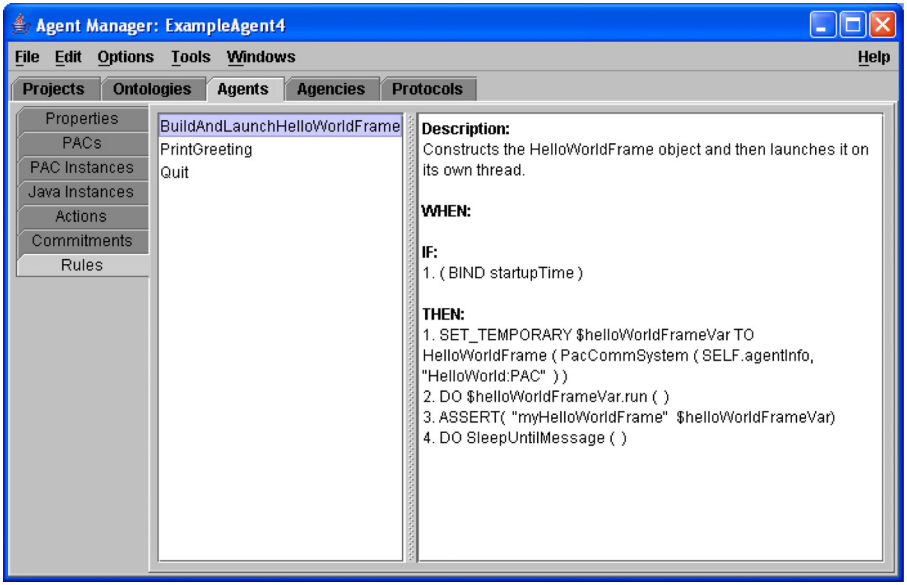

Figure 78. Agent Manager Showing the Build HelloWorldFrame Rule

the LHS Panel of the Rule Editor click on the **New Variable** button then select **KqmlMessage** in the **New Variable Dialog** window. Figure 79 shows the **New Variable Dialog**.

Next, enter the variable name `%message` in the **Variable Name** text field, then click on **Add**. This will display the **Binding Dialog** shown in Figure 80. For this case, we wish to detect new incoming messages, so ensure that the **KQML Message Binding** pull-down has **Incoming** selected (the default case) and click **OK** here and in the **New Variable** dialog. This creates a variable that will be available for use in patterns on the LHS and RHS of the rule.

For the first LHS pattern, select **<Operators>** → **EQUALS** and then click on the **Defined Variables…** button to display the **Defined Variables Dialog** (Figure 81). You can select the **PACs** tab and double click on the **%message**[1] variable to display all of the KQML

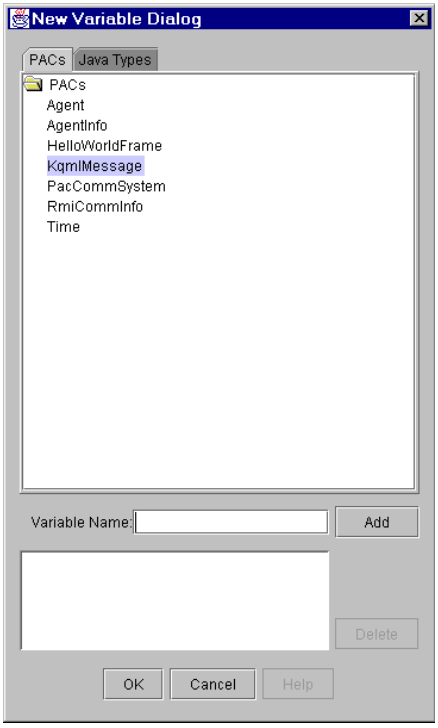

Figure 79. New Message Variable Dialog

message attributes that can be used in message conditions. Select **String sender** and click on **OK**. This will write `%message.sender` into the first slot in the message conditions accumulator of the Rule Editor.

---

1. The choice of %message for the variable name is based on the following naming convention: *%name* for KqmlMessage variables which will bind to incoming messages, *?name* for any variables which will bind to objects in the agent's mental model (including any *stored* KqmlMessage objects), and *$name* for any temporary variables or return variables. This naming convention is only a suggestion; you're free to choose whatever variable names you wish.

Figure 80. The Binding Dialog

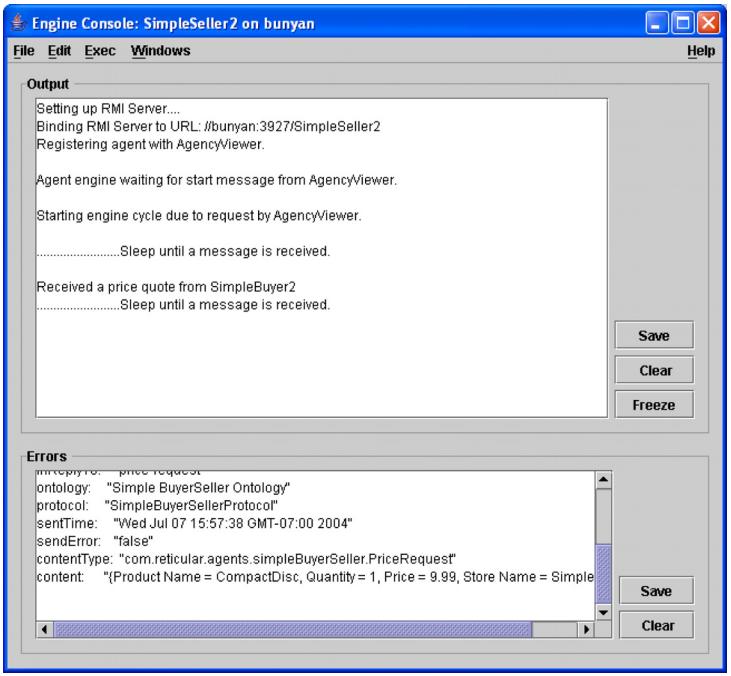

Figure 81. Defined Message Variables

Next, select **<Value>→KQML Message** to display the **Values** dialog. Figure 82 shows the **Values Dialog**. Select **sender**, then click and erase the value in the **<Value>** combo-box. Enter *HelloWorld:PAC* in this combo-box and click **OK**. Now that the pattern in the accumulator is complete click on **Add** in the **Rule Editor**.

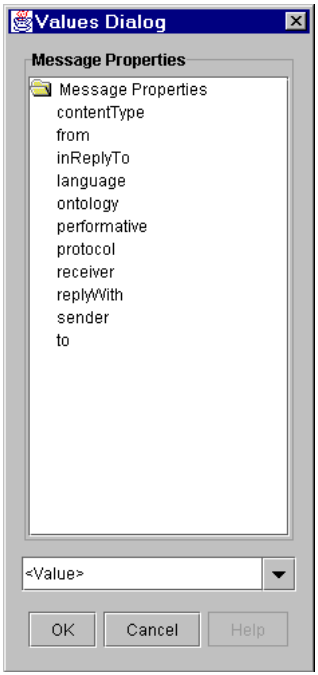

Figure 82. Message Properties

Follow the same process for the `performative` and `contentType` message conditions. The conditions of the rule should make sure that the performative equals the string constant `achieve` and the `Class contentType` is of Class `String`. The values needed for each pattern can be found in the **Values** dialog.

If you make a mistake creating a message condition, you can clear it from the accumulator text field using the **New** button. You can clear an entry from the **Message Condition** list using the **Delete** button.

The fourth LHS pattern in this rule tests the binding in a message variable. In the **Conditions** panel select **<Operators>** → **EQUALS**,

then click on the **Defined Variable..** button to display the **Defined Variables** dialog. Select **Pacs** tab at the top of the dialog, then double-click on the **%message** variable and select **Object content.** Choose **String** from the **<Casting Type>** combo box to type cast the content type to `String` (this allows you to condition with the content as if it were a string), then click on **OK**. This writes `%message.content` into the accumulator. Next, click on **<Value>** → **String**, then enter *`Say Hello`* into the text field and click on **OK**. Note: the string tested for in this mental condition must be exactly the same as the string sent in the message from the interface, otherwise the rule will not fire. Finally, click on **Add** to add the message condition to the list. Figure 83 shows the LHS of the `PrintGreeting` rule in the Rule Editor.

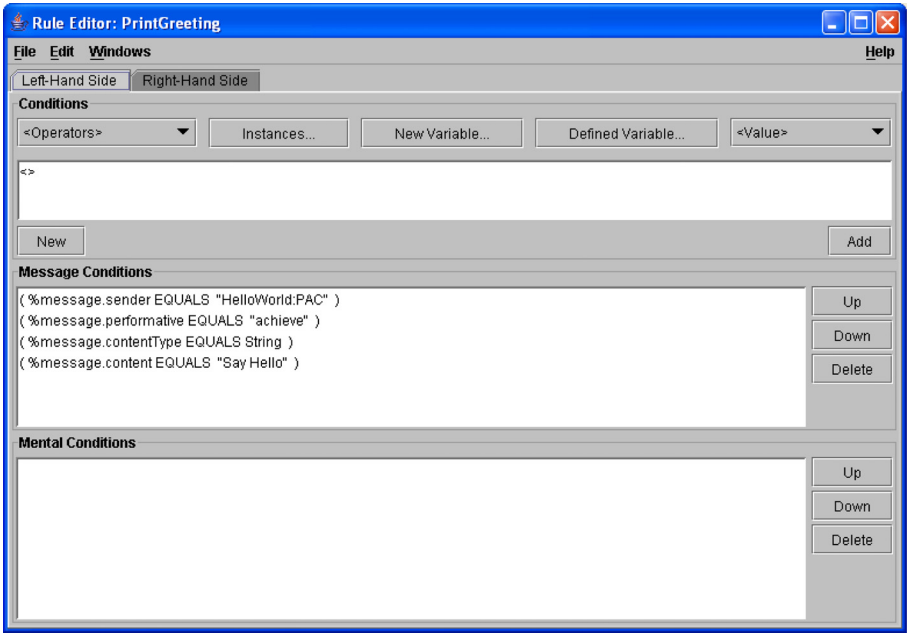

Figure 83. Rule Editor Showing Print Greeting Rule LHS

Now select the **Right-Hand Side** tab in the **Rule Editor**. The RHS of the `PrintGreeting` rule will use the print method on the `myHelloWorldFrame` instance to print a greeting in the PAC interface and will put the agent to sleep until another message arrives. In the **Actions** panel, select the **Instances...** button, then choose the print method shown in the subtree beneath the `myHelloWorldFrame` instance. Selecting **OK** in the dialog writes **DO myHelloWorldFrame.print(<param0>)** into the accumulator.

Use a string concatenation function as the argument to the `Print` method, to combine the standard greeting with the agent's current time belief (as we did with Example Agent 2). Select **<Operators> → Concat,** then select **<Value> → String** and enter *`Hello World! The time is`* , then click **OK**. Note that this string is not used in any comparisons (as was the `Say Hello` string in the message pattern) so you're free to type in any greeting you choose. For the second argument in the `Concat` function, click on the **Instances…** button then double-click on the `Time currentTime` instance, then click on the `String string` attribute under **currentTime**. This will write `currentTime.string` into the slot for the second `Concat` argument, completing the `Print` action pattern. Click on **Add** to add this to the list of RHS patterns.

Finally, add a `SleepUntilMessage` action. This requires the same sequence of steps as was done for the `SleepUntilMessage` action in the previous rule. After both RHS patterns have been added, save the `Print Greeting` rule.

In the `Quit` rule we want to detect when the user clicks on the **Quit** button of the `HelloWorldFrame` and shutdown the agent. The LHS message and mental conditions for this rule are exactly the same as the `Print Greeting` rule except that mental condition should test for the `String Quit` (instead of `Say Hello`). The RHS consists of a single action, the built-in action `ShutdownEngine`. The `Quit` rule should look like that shown in Figure 84.

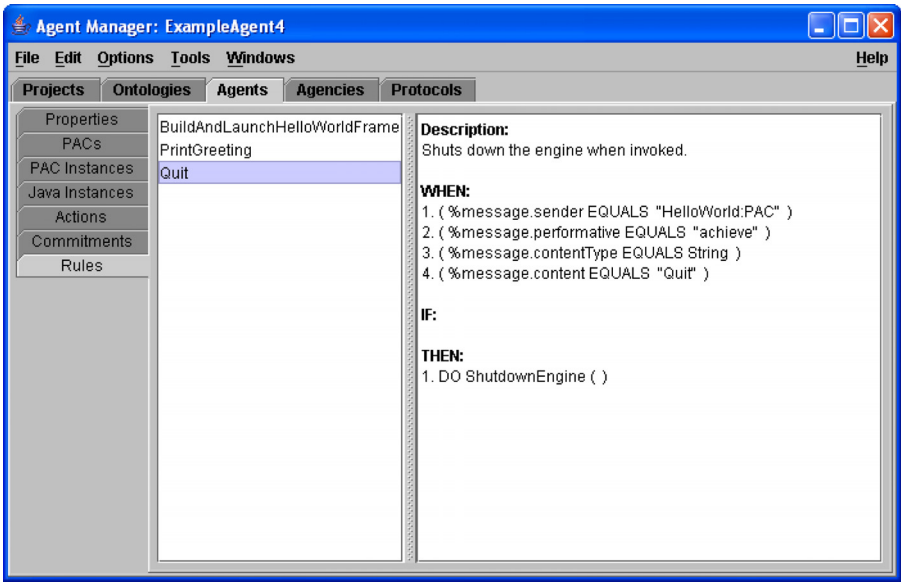

Figure 84. Agent Manager Showing the Quit Rule

Step 5. Run the Agent

You can now run the agent from the Agent Manager. Every time you click on the **Say Hello** button you should get a printout in the **HelloWorldFrame** that says **Hello World! The time is .** When you click on the **Quit** button the **HelloWorldFrame** should disappear and the agent engine should shutdown. The console should remain visible until you select **File → Exit**.

Congratulations! You have completed the introduction to AgentBuilder and now know how to use the basic capabilities of the tools. The next chapter will tell you how to build two agents that communicate with each other using KQML. The final chapter in this User's Guide describes the use the AgentBuilder Pro Agency Viewer and agency construction tools. Be sure and review the Ref-

erence Manual for more detailed information about the operation of the various AgentBuilder tools.

Chapter 10

Creating Agents that Communicate

This chapter provides detailed instruction for creating two agents that can communicate with each other. These two agents take on the roles of a buyer and a seller. This chapter describes

- Creating the Buyer/Seller Ontology
- Creating a Buyer Agent
- Creating a Seller Agent
- Writing the Rules for the Agents
- Running the Agents

This example will demonstrate how two agents communicate with each other. The two agents will run on their own Java virtual machine, so they can be started from AgentBuilder. The agents will communicate with each other via messages using the agent communications module. In addition, we will use the **PriceRequest PAC** located in the system repository's ontology. The **PriceRequest PAC** will be used by the agents to request and send a price quote for a certain product. In this example, an agent, SimpleBuyer, will send a message to another agent, SimpleSeller, requesting a price quote on a certain product. The SimpleSeller agent will respond by returning a message that contains the price of the product

Table 11. Creating Two Agents that Communicate with Each Other

Step	***Description***
1.	Create SimpleBuyerSeller Ontology
2.	Create SimpleSeller agent.
3.	Import PAC Object and create initial Java instance.
4.	Create rules. a. Create the `WaitForIncomingMessage` rule. b. Create the `RespondToIncomingMessage` rule.
5.	Create SimpleBuyer agent.
6.	Import PAC Object.

Table 11. Creating Two Agents that Communicate with Each Other

Step	***Description***
	Create Rules a. Create `PriceQuote` rule. b. Create `Send PriceRequestToStoreAgents` rule. c. Create `ReceivePriceQuotesFromStoreAgents` rule.
7.	Run the Agent

Step 1. Create SimpleBuyerSeller Ontology

We will use the System Repository's **SimpleBuyerSeller Ontology** rather than create one. From the **Project Manager,** select the **Ontologies** tab an then (in the left-hand panel) open the **System Repository**. Now, select the **SimpleBuyerSeller Ontology** and right-click on it. Select **Copy** from the pop-up menu. Now, select the **user** ontology folder and right-click on the ontology. Select **Paste** from the pop-up menu. You now have an editable copy of the **SimpleBuyerSeller Ontology** in your **user** ontology folder. Figure 85 shows the **Simple BuyerSeller Ontology** in the user's ontology folder.

We also need to create a new agency as part of the **Quick Tour Project**. From the **Project Manager**, select the **Quick Tour Project** folder you created earlier. Right-click on this folder to pop up a menu that will allow you to select **New Agency…**. Name this new agency the *`SimpleBuyerSellerAgency`*.

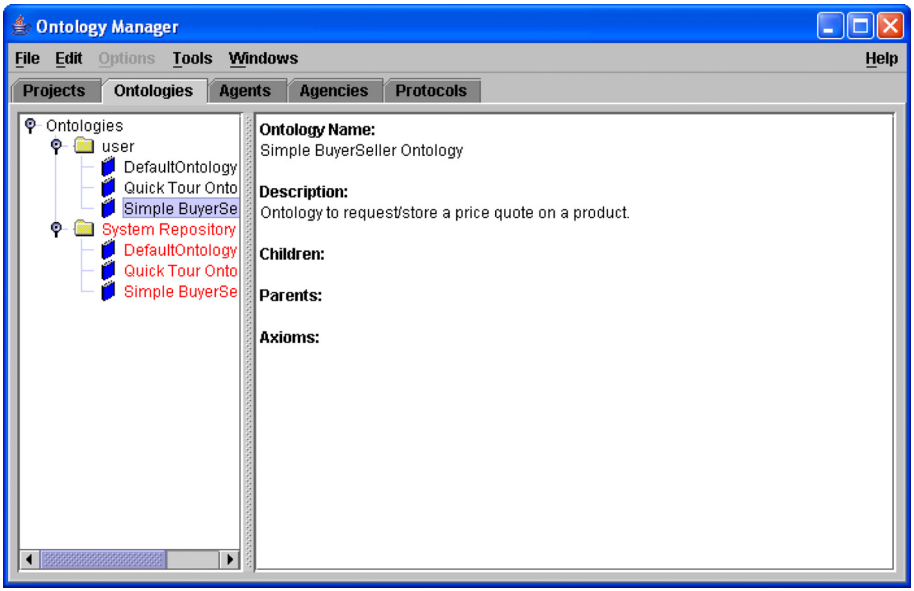

Figure 85. Simple Buyer/Seller Ontology

Step 2. Create SimpleSeller Agent

Go to the **Project Manager** by selecting the **Project** tab. Select the **SimpleBuyerSeller Agency**. You can now create a new agent that belongs to the **SimpleBuyerSeller Agency** by right-clicking on the selected agency and selecting the **New Agent** menu item. Fill in the name, description, author, and vendor field information for the SimpleSeller agent. The agent is given default values for the communication settings. If you want to change them, click on the **Communications…** button and enter an IP address or a new port number. Now, click on the **OK** button to close the **Communications Dialog**. Click the OK button on the **Agent Properties** dialog to close it. The default port number is randomly assigned, so yours may be different from that shown in Figure 86.

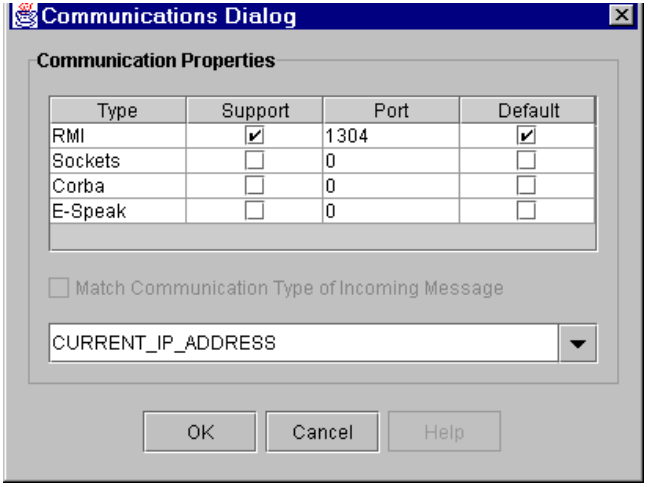

Figure 86. Simple Seller Communications Dialog

Step 3. Import PAC Object and create initial Java instance

In order to import a PAC, we must open the PAC editor. Select the **SimpleSeller** agent in the **Project Manager** window, and choose **Agents** in the tab menu. In the Agent Manager window, select **Tools → PAC Editor** from the menu bar. Now choose **File → Import…**, to import the **PriceRequest** class from the **Simple BuyerSeller Ontology**. This will bring up the window shown in Figure 87. Using the **<Object Model>** pop-up menu, select **Simple BuyerSeller Ontology;** this will list the available classes. Select **PriceRequest** and click on the **Add** button, then on the **OK** button to close the **Import Dialog**. **PriceRequest** should appear in the list of **Defined PACs.** Go ahead and save the current PACs by selecting **File → Save.**.

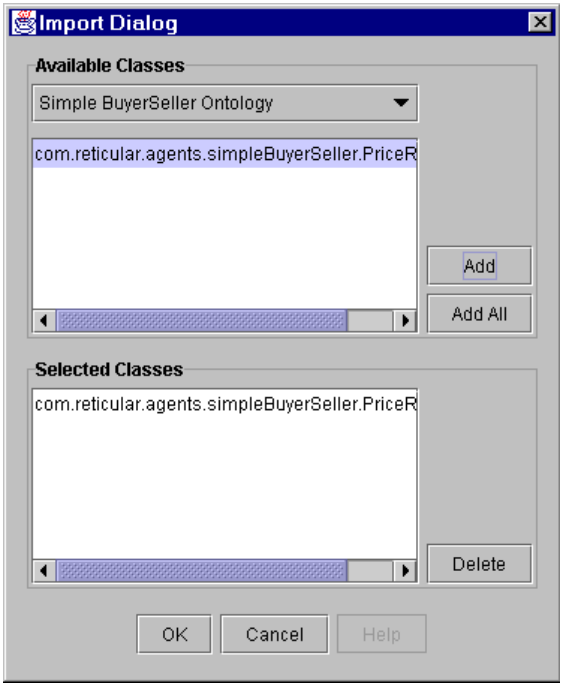

Figure 87. Simple Seller Import Dialog

Since the SimpleSeller agent only needs to know the price for a product, we will use a Java instance. In the **PAC Editor**, select **Java Instance** tab. In **Java Instance Properties**, enter the name *compact-DiscPrice* and a description. Now click on the **Enter** button. From the **<Java>** popup menu select **Float**. Check the **Initial Java Instance** check box, enter *9.99* as the value and click on the **Add** button . From the **Defined Java Instances** panel, click on the **Add** button (see Figure 88). Finally, save the Java Instance by selecting **File** → **Save** in the **PAC Editor** window.

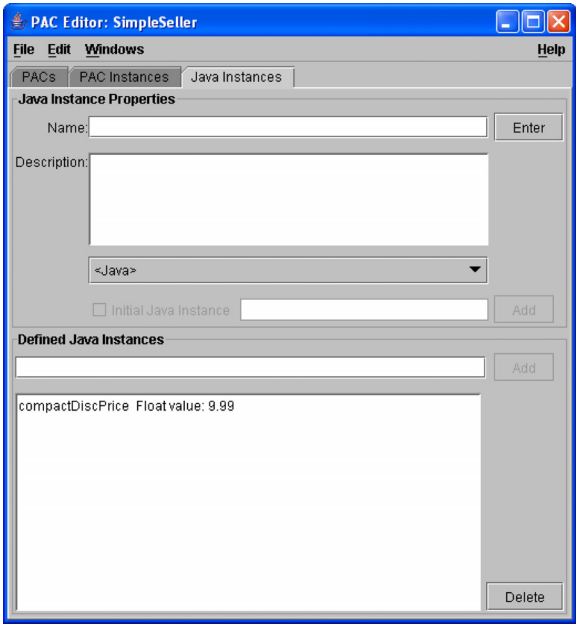

Figure 88. Simple Seller PAC Editor

Step 4. Create rules.

Now we need to create the rules that will handle the incoming message and respond to the agent that sent that message. There are only two rules, `WaitForIncomingMessage` and `RespondToIncomingMessage`. The agent will only respond to messages that meet the message conditions.

Step 4a. Create the WaitForIncomingMessage rule.

This rule will put the agent to sleep until the agent receives a message. First, open the **Rule Editor** by selecting **Tools → Rule Editor** in the Agent Manager's menu bar. Then, select the **Edit → Proper-**

ties menu item. Enter the name and description for the rule in the **Rule Properties** dialog.

On the Left-Hand Side of the rule you need to create a Mental Condition that binds to the Agent instance named `SELF`. You do this by selecting **<Operators> → BIND,** clicking on **Instances…** button, selecting **Agent SELF** and clicking on the **OK** button. Click on the **Add** button in the Rule Editor to add the new pattern to the **Mental Conditions** list.

The right-hand side of the rule will contain a single built-in action. Select **<Built-in Actions> → SleepUntilMessage**, and click on the **Add** button. Now save the rule by selecting **File → Save** in the **Rule Editor** window.

Step 4b. Create the RespondToIncomingMessage rule.

This rule will test all incoming messages to see if the sending agent is requesting a price quote. If this is the case, it will fill in the price and store name in the `PriceRequest` object and send it to the agent that requested it.

To enter a new rule, select the **File → New → New Rule** menu item. Enter the name *`RespondToIncomingMessage`* and a brief description of the rule. On the Left-Hand Side of the rule click **New Variable…** button to create two variables of type `KqmlMessage` and `Agent`. Select **KqmlMessage**, enter *`%message`* as the variable name and click on the **Add** button. This will display the **Binding Dialog**; select **Incoming** and click on **OK** to close the **Binding Dialog**. Now, select **Agent** and enter *`?buyer`* as the variable name, and click on the **Add** button. The **New Variable Dialog** is shown in Figure 89. Click on **OK** to close the **New Variable** dialog.

To create the first Message Condition, select **<Operators> → EQUALS** using the Rule Editor. This will insert **(<> EQUALS <>)** in the accumulator. Next, click on **Defined Variable…**, double click on

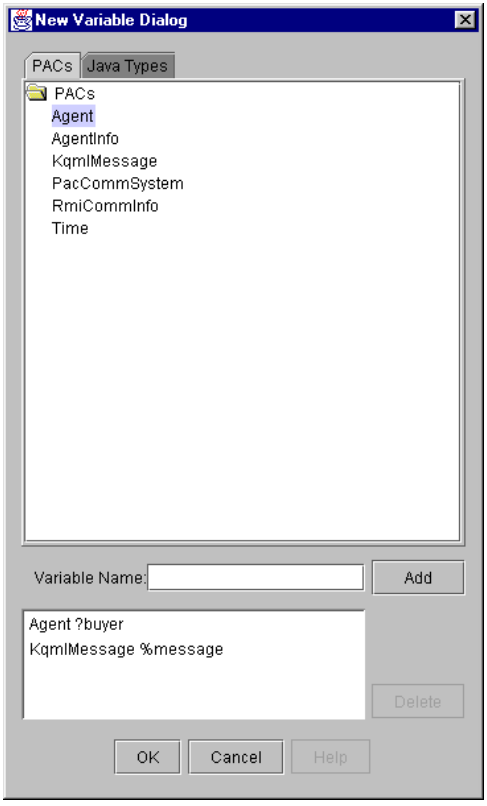

Figure 89. Simple Seller New Variable Dialog

%message, select **sender**, and click on the **OK** button to insert **%message.sender** in the accumulator. To finish the EQUALS condition, click on **Defined Variable…** button, open the **?buyer** folder and then the **agentInfo** folder. Finally select the **name** and click on **OK.** This will enter the pattern **?buyer.agentInfo.name** into the Rule Editor. Click on the **Add** button to add the pattern to the **Message Condition** list.

Follow the same process for the performative and contentType message conditions. The conditions of the rule should ensure that the performative equals the string constant `ask-one` and the **content-Type** is `PriceRequest`. The value for the contentType can be found by selecting **<Value> → Class →PriceRequest**. Figure 90 shows the left-hand side of the **RespondToIncomingMessage** rule.

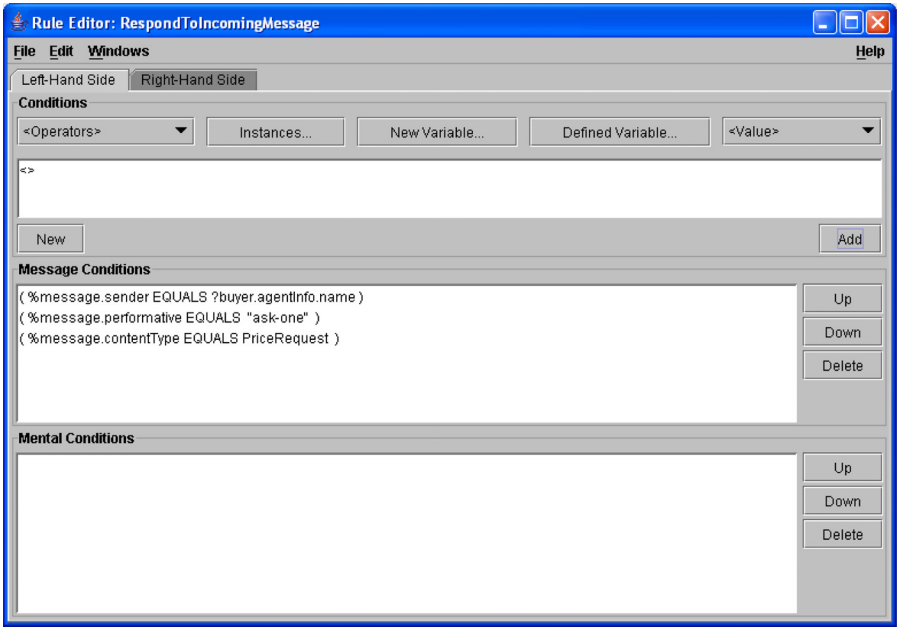

Figure 90. Simple Seller Rule Editor

On the right-hand side of the rule, we will print a message that we received with the price request, reply to the message, and print another message confirming that we have sent the price quote to the requesting agent.

To print a message, select **<Built-in Actions> → SystemOut-Println**. This will add **Do SystemOutPrintln(<output>)** to the accumulator. Next, select **<Operators> → Concat**, this will add

Concat(<string>, <string>) to the accumulator. Now, select **<Value> → String**, this will pop-up a window; type *Received a price quote from* in the **Literal Value** field and click on **OK**. To print the name of the agent requesting a price quote, click on **Defined Variable…**, select **?buyer.agentInfo.name** and click on **OK**. Now, add the action to the **Defined RHS Elements** by clicking on the **Add** button.

Next, we need to fill in the `price` and `storeName` on the `PriceRequest` object we received from the buyer agent. Select **<Operators> → SET_VALUE_OF,** this will add **SET_VALUE_OF <> TO <>** to the accumulator. Click on **Defined Variable…** to select **%message.content**, select **<Casting Type> → PriceRequest**, and select **%message.content.price**. To finish the **Mental Change**, click on **Instances**, select the **Java Instances** radio button, select **compactDiscPrice**, and click on **OK**.

Follow the same process to set the value of **%message.content.storeName** to **SELF.agentInfo.name**. The instance `SELF` can be found in the **Instances** dialog.

Now, we need to send the buyer agent the message with the `PriceRequest` information. In the **Actions** panel, select **<Built-in Actions> → SendKqmlMessage**, this will add **Do SendKqmlMessage(<message>,<receiver>, <performative>,<content>,<replyWith>,<inReplyTo>,<language>,<ontology>, <protocol>, <to>,<from>)** to the accumulator. Since we are going to reply with the same message, we only need to fill in five parameters of the `KqmlMessage`. The first four parameters are the following: `%message, %message.sender, "tell", %message.content.` **<replyWith>**, `%message.replyWith`. Note: you do not need to enter anything in the **<replyWith>** field. The **<replyTo>** field should be set to `%message.replyWith`. **Warning! You will need to explictly cast** `message.content` to `Price Request`. Otherwise, you will receive an

error message at runtime complaining that the object is not serializable.

The next action is another print statement. Follow the process described above to construct a string with the values *Sent price quote to* and `%message.sender`.

Finally, we need to add another action that will put the agent to sleep until it receives another message from an agent. Select **<Built-in Actions> → SleepUntilMessage** to add **Do SleepUntilMessage()** to the accumulator. All we need to do now is add the rule to the list of Defined RHS Elements and save the rule. Your rule should be similar to the one shown in Figure 91.

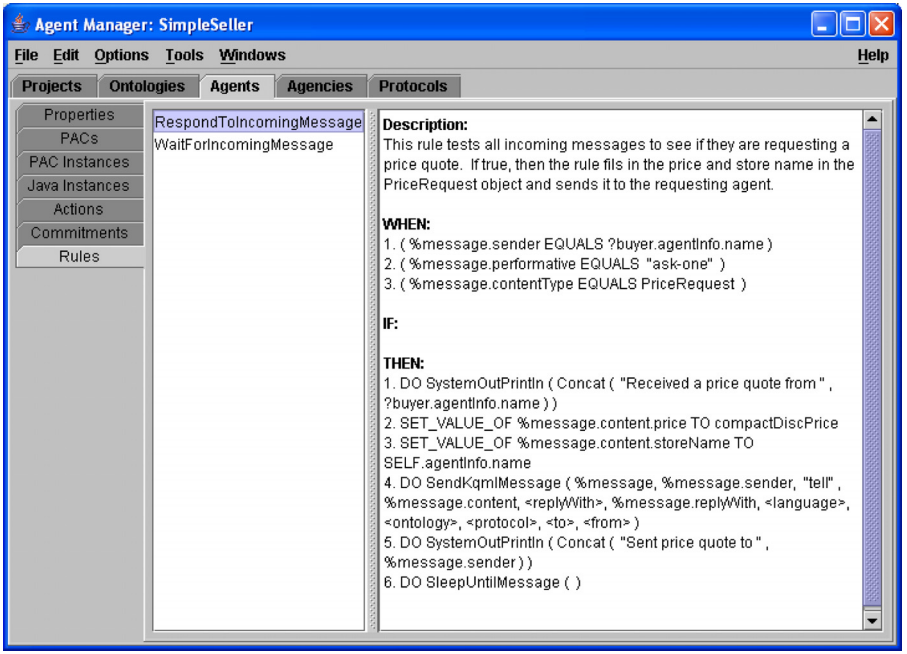

Figure 91. Simple Seller Rules

Step 5. Create SimpleBuyer agent.

To create the SimpleBuyer agent, follow the same procedures described in step 2. The agent properties will be the same as the SimpleSeller agent, except for the name (which is “SimpleBuyer”) and the port number.

Step 6. Import PAC Object.

To import the PAC Object, please refer to Step 3. Note: the Simple-Buyer agent doesn't need the Java Instance that the SimpleSeller agent needed.

Step 7. Create rules.

The SimpleBuyer agent will need rules to create a `PriceRequest` object, send the `PriceRequest` to each agent that it knows about, receive the `PriceRequest` object and print them on the console.

Step 7a. Create the CreatePriceQuote Rule.

In the **Rule Editor Properties** dialog enter the name *`CreatePrice-Quote`* rule and a brief description. On the Rule Editor's Left-Hand side enter a Mental Condition that binds to `SELF` (See second paragraph of Step 4a). On the Right-Hand Side of the rule assert a new `PriceRequest` object. **Select <Operator> → ASSERT**. This will bring up the **Assertion Dialog**, leave the field blank and click on **OK**. This will insert **ASSERT(<>)** in the accumulator. Next, click on **New Object…** select **PriceRequest**, select **<Constructor> → PriceRequest(String param0, int param1, String param2),** and click on **OK**. (Remember, you can drag the corners of the dialog box to increase its size so you can see all the parameters). This will add **PriceRequest(<param0>, <param1>, <param2>)** to the accumulator. Fill in the parameter values with the String *`CompactDisc`*,

Integer 1, and String *CD*. Add the Mental Change to the Defined RHS Elements and save the rule.

Step 7b. Create the SendPriceRequestToStoreAgents rule.

This rule is executed by the `priceQuote` instance created in the previous rule. Clear the Rule Editor using the **File → New → New Rule** menu item. Specify the name (**SendPriceRequestToStoreAgents**) and a description of the rule.

Select the **Left-Hand Side** tab of the Rule Editor. Create a new variable of type `PriceRequest` with the name `?priceQuote` (**New Variable…,** select **PriceRequest**, enter *?priceQuote* in the **Variable Name** field); then add a mental condition to BIND to the `?priceQuote` variable (**<Operators> → BIND** followed by **Defined Variable…**, `?priceQuote`).

You will also need to create a new variable of type `Agent` with name *?agent*.

The next Mental Condition will prevent the agent from sending the message to itself. Select **<Operators> → NOT_EQUALS** in the Conditions panel. Enter *?agent.agentInfo.agentName* and *SELF.agentInfo.agentName* as the parameters and add it to the list of Mental Conditions.

On the Right-Hand Side of the rule, we will construct a `KqmlMessage` and send it to the SimpleSeller agents. Only the first five parameters are needed in order to send the KqmlMessage. The parameter values are `KqmlMessage(), ?agent.agentInfo.name, "ask-one", ?priceQuote, "Price Quote"`. You will find the KqmlMessage() constructor using the New Object... button. When you complete constructing the KqmlMessage, add it to the list of defined RHS elements.

Now, we need to print a message indicating that we are forwarding the message to the Seller agents. Select **<Built-in Actions> →**

SystemOutPrintln, <Operators> → Concat, select **SELF.agentInfo.name** in the **Instances Dialog**, then **<Operators> → Concat**, **<Value> → String**, enter *`Forwarding price quote request to `* in the **Literal Value** field, and select **?agent.agentInfo.name** from the **Defined Variable Dialog**. Add the action to the list of defined RHS Elements.

To finish the rule, add the built-in action, `SleepUntilMessage`, and save the rule. Figure 92 shows the completed SendPriceRequest-ToStoreAgents rule.

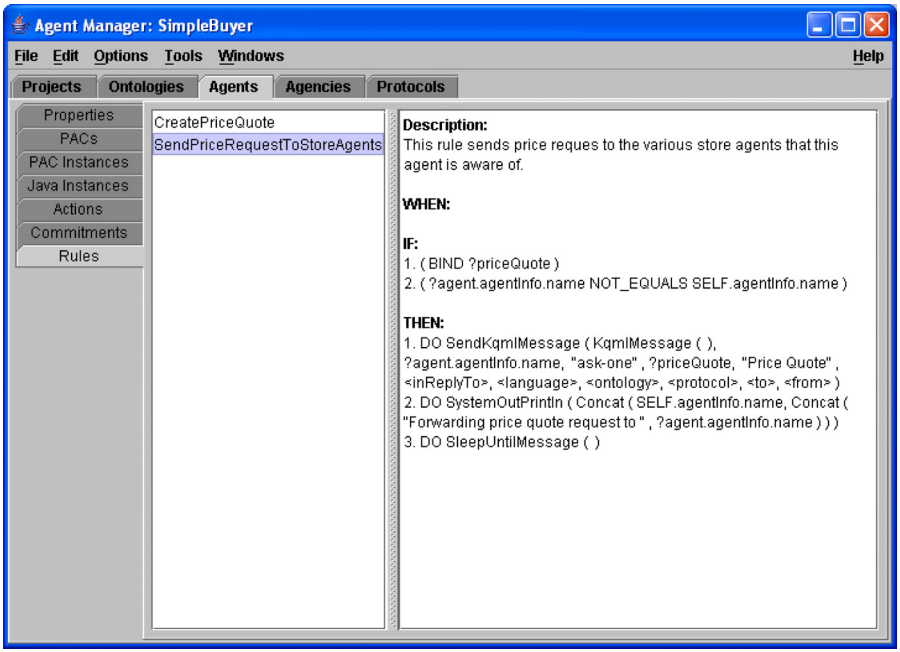

Figure 92. The SendPriceRequestToStoreAgents Rule

Step 7c. Create the ReceivePriceQuotesFromStoreAgents rule.

This rule will receive the price quote from the SimpleSeller agent and print the price of the product. We need to create a `KqmlMessage`

variable (click **New Variable…**, select **KqmlMessage**, enter *%message* in **Variable Name** field, and click on **OK**) and an Agent Variable with the name of **?agent**. Follow the same process as step 4b (third paragraph) to test if `%message.sender` equals `?agent.agentInfo.name`, `%message.performative` equals `tell`, `%message.contentType` equals `PriceRequest`, and `%message.content.productName` equals "`CompactDisc`" (remember that the message content must be cast to `PriceRequest`).

For the Right-Hand Side of the rule all we need to do is print (to the console that sent the message) the price quote of the product that was requested. Select **<Built-in Actions> → SystemOutPrintln**, **<Operators> → Concat**, **<Value> → String**, enter *Received price quote from* , click on **Defined Variable…**, select **?agent.agentInfo.name**, and click on **OK**.

All that's left to do is add the built-in action `SleepUntilMessage` (**<Built-in Actions> → SleepUntilMessage**). Add the action to the list of **Defined RHS Elements** and save the rule. Figure 93 shows the ReceivePriceQuotesFromStoreAgents rule.

Step 8. Run agents.

When running the agents, order is important. The SimpleSeller agent must be started before the SimpleBuyer agent. This will prevent the SimpleBuyer agent from sending a message to an agent that doesn't exist.

The agents can be started from the AgentBuilder application or from the engine application. If you decide to run the agents from the engine, create the RADL files first by selecting **Options → Generate Agent Definition** from the Agent Manager.

To start the agents from the Agent Manager window, open SimpleSeller agent and select **Options → Run Agent**. Now, open the

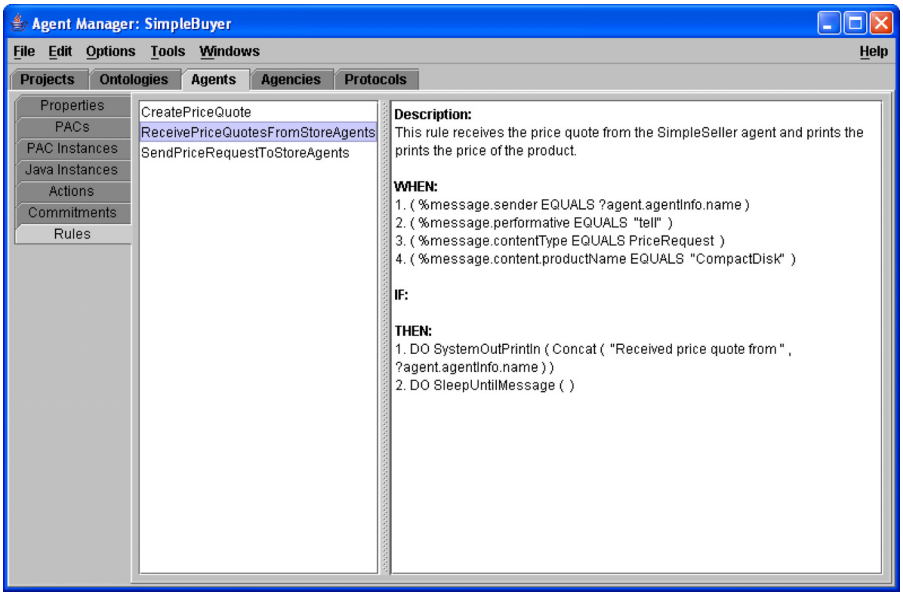

Figure 93. The ReceivePriceQuotesFromStoreAgents rule

SimpleBuyer agent from the same Agent Manager window and select **Options → Run Agent**.You will see the print statements on each of the agent's console window. Figure 94 and Figure 95 show the console windows for both agents.

Now, try adding a RHS pattern to the **ReceivePriceQuotesFrom-StoreAgents** rule that will print out a message something like "The price for the <productName> is <price>. You will need the rule to perform the following actions: **<Built-In Actions> →DoSyste-mOutPrintln**, **<Operators>→ concat**, **<Value>** →"`The price for the` ", **<Operator>** →**concat**, **Defined Variable...** →`%mes-sage.content.productName`, **<Operator>** →**concat**, **<Value>** →String " `is` ", **<Operator>** →**ConvertToString**, **Defined Vari-able...**→ `%message.content.price`. Add the rule to the right-hand side pattersn. You may need to use the **Up** button to ensure that this

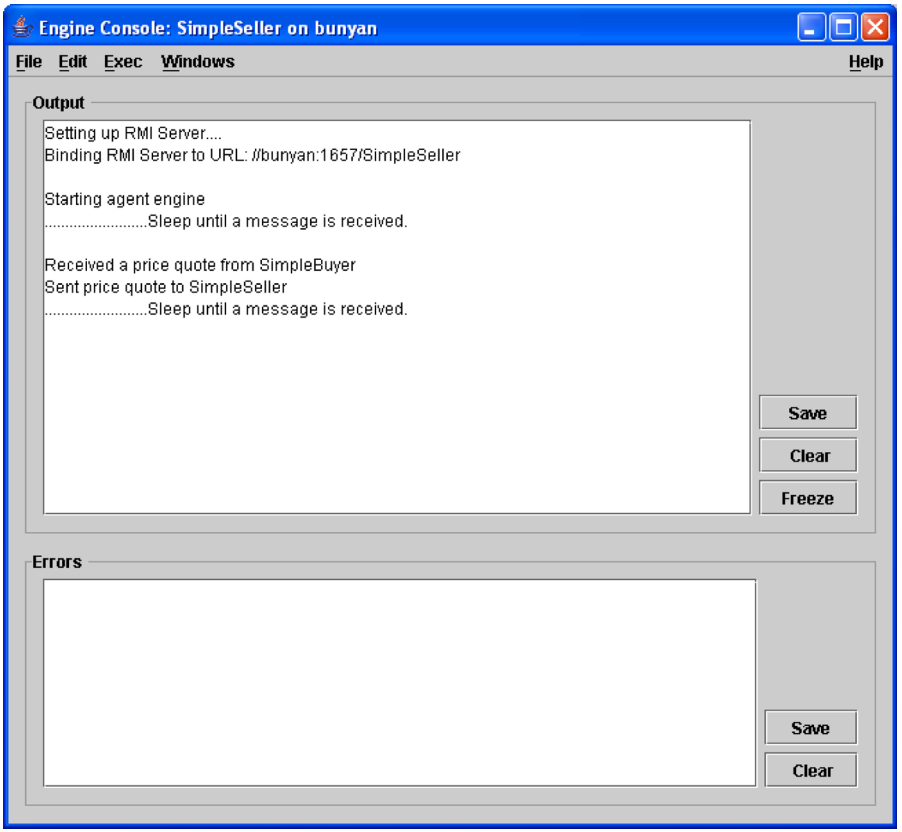

Figure 94. SimpleSeller Agent Console

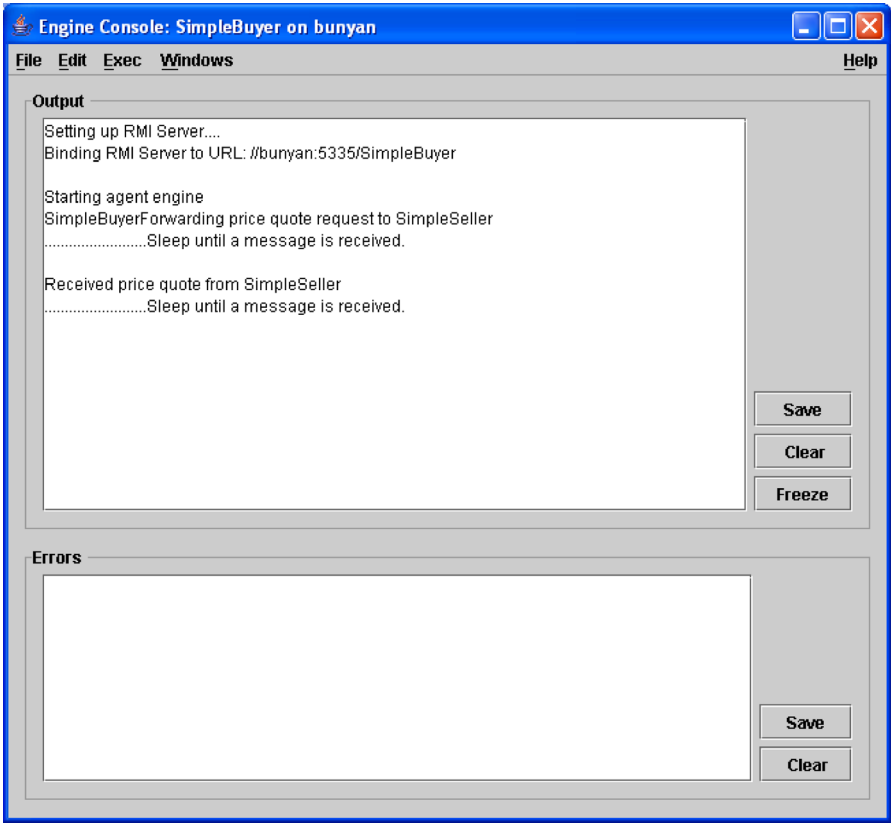

Figure 95. SimpleBuyer Agent Console

actions is performed before the sleep action. Figure 96 shows the completed rule.

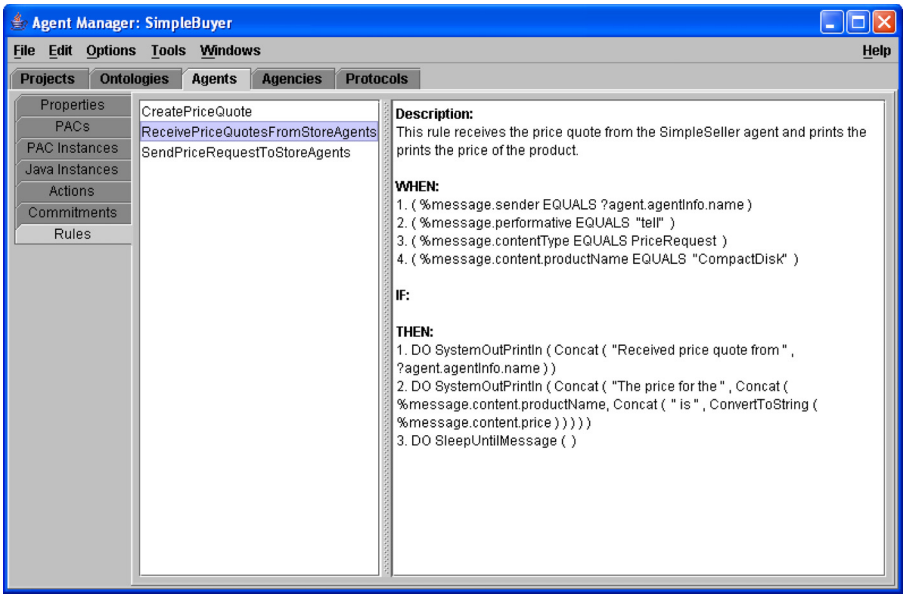

Figure 96. The Modified SimpleBuyer Rule

Chapter 11

Agents that Communicate with CORBA

This chapter provides detailed instruction for creating two agents that communicate with each other using CORBA instead of RMI. These two agents are the same as in the previous chapter. Only their communications mechanism is different. This chapter also describes:

- Modifying Agency and Agent Communication
- Running a CORBA Name Server
- Running the Agents

This example will demonstrate how two agents communicate with each other. The two agents will run on their own Java virtual machine, so they can be

started from AgentBuilder. The agents will communicate with each other via messages using the agent communications module. In addition, we will use the **CorbaPriceRequest PAC** located in the system repository's ontology. The **CorbaPriceRequest PAC** will be used by the agents to request and send a price quote for a certain product. In this example, an agent, SimpleBuyer, will send a message to another agent, SimpleSeller, requesting a price quote on a certain product. The SimpleSeller agent will respond by returning a message that contains the price of the product.

Table 12. Creating Two Agents that Communicate with Each Other

Step	***Description***
1.	Create CORBA Compatible PAC(s)
2.	Create Simple BuyerSeller Ontology
3.	Create SimpleSeller agent.
4.	Import PAC Object and create initial Java instance.
5.	Create rules. a. Create the `WaitForIncomingMessage` rule. b. Create the `RespondToIncomingMessage` rule.
6.	Create SimpleBuyer agent.
7.	Import PAC Object.

Table 12. Creating Two Agents that Communicate with Each Other

Step	***Description***
8.	Create Rules a. Create `PriceQuote` rule. b. Create `SendPriceRequestToStoreAgents` rule. c. Create `ReceivePriceQuotesFromStore agents` rule.
9.	Modify Agency Communications
10.	Modify Agent Communications
11.	Run the Nameserver
12.	Run the Agent

Step 1. Create CORBA Compatible PAC(s)

Any object that will go through the CORBA communication system will need to be specified using the OMG Interface Definition Language (ID). Once you have created the IDL file, you will need to use Sun Microsystems' **idlj** tool to convert the IDL file to a Java source file. We have provided a sample IDL file and the Java file that is created by running the **idlj** tool. You can find these files in the AgentBuilder **src** directory.

Step 2. Create SimpleBuyerSeller Ontology

We will use the System Repository's **SimpleBuyerSeller Ontology** rather than create one. From the **Project Manager,** select the **Ontologies** tab an then (in the left-hand panel) open the **System Repository**. Now, select the **Simple BuyerSeller Ontology** and right-click

on it. Select **Copy** from the pop-up menu. Now, select the **user** ontology folder and right-click on the ontology. Select **Paste** from the pop-up menu. You now have an editable copy of the **Simple BuyerSeller Ontology** in your **user** ontology folder. Figure 97 shows the **Simple BuyerSeller Ontology** in the user's ontology folder.

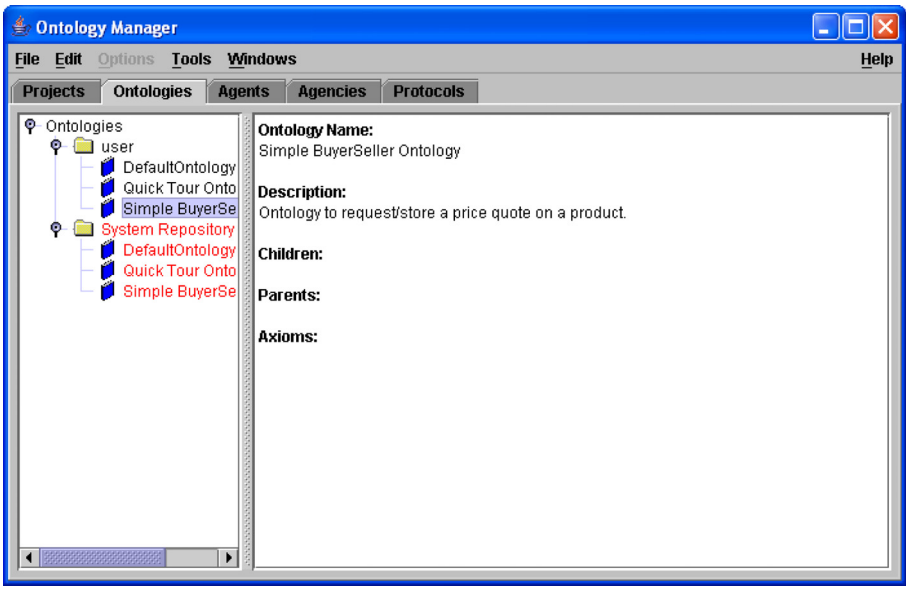

Figure 97. Simple Buyer/Seller Ontology

We also need to create a new agency as part of the **Quick Tour Project**. From the **Project Manager**, select the **Quick Tour Project** folder you created earlier. Right-click on this folder to pup up a menu that will allow you to select **New Agency…**. Name this new agency the *SimpleBuyerSellerAgency*.

Step 3. Create SimpleSeller Agent

Go to the **Project Manager** by selecting the **Project** tab. Select the **SimpleBuyerSellerAgency**. You can now create a new agent that

belongs to the **SimpleBuyerSellerAgency** by right-clicking on the selected agency and selecting the **New Agent** menu item. Fill in the name, description, author, and vendor field information for the SimpleSeller agent. The agent is given default values for the communication settings. If you want to change them, click on the **Communications…** button and enter an IP address or a new port number. (NOTE: The current version of Java has an interesting feature that forces you to double click on the port column, enter the value, and then press the **Enter** key on your keyboard; any other sequence will not properly enter the number and will cause an error message to be displayed.) Now, click on the **OK** button to close the **Communications Dialog**. Now, click the OK button on the **Agent Properties** dialog to close it. See Figure 98.

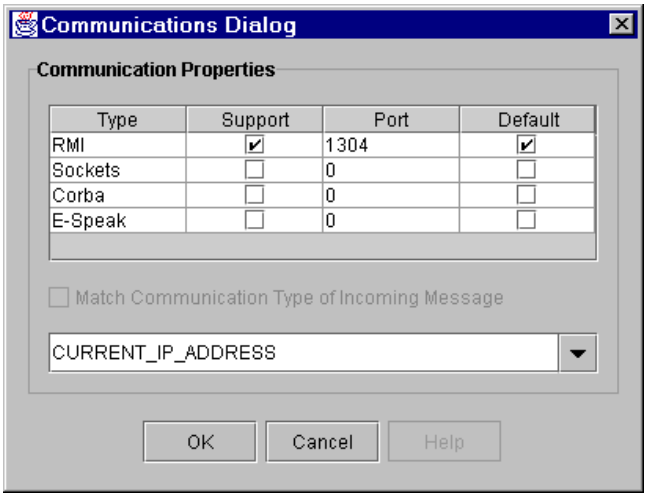

Figure 98. Simple Seller Communications Dialog

Step 4. Import PAC Object and create initial Java instance

In order to import a PAC, we must open the PAC editor. Select the **SimpleSeller** agent in the **Project Manager** window, and choose **Agents** in the tab menu. In the Agent Manager window, select **Tools → PAC Editor** from the menu bar. Now choose **File → Import…**, to import the **PriceRequest** class from the **Simple BuyerSeller Ontology**. This will bring up the window shown in Figure 99. Using the **<Object Model>** pop-up menu, select **Simple BuyerSeller Ontology;** this will list the available classes. Select **PriceRequest** and click on the **Add** button, then on the **OK** button to close the **Import Dialog**. **PriceRequest** should appear in the list of **Defined PACs.** Go ahead and save the current PACs by selecting **File → Save.**.

Since the SimpleSeller agent only needs to know the price for a product, we will use a Java instance. In the **PAC Editor**, select **Java Instance** tab. In **Java Instance Properties**, enter the name *`compactDiscPrice`* and a description. Now click on the **Enter** button. From the **<Java>** popup menu select **Float**. Check the **Initial Java Instance** check box, enter *`9.99`* as the value and click on the **Add** button (see Figure 100). From the **Defined Java Instances** panel, click on the **Add** button. Finally, save the Java Instance by selecting **File → Save** in the **PAC Editor** window.

Step 5. Create rules.

Now we need to create the rules that will handle the incoming message and respond to the agent that sent that message. There are only two rules, `WaitForIncomingMessage` and `RespondToIncomingMessage`. The agent will only respond to messages that meet the message conditions.

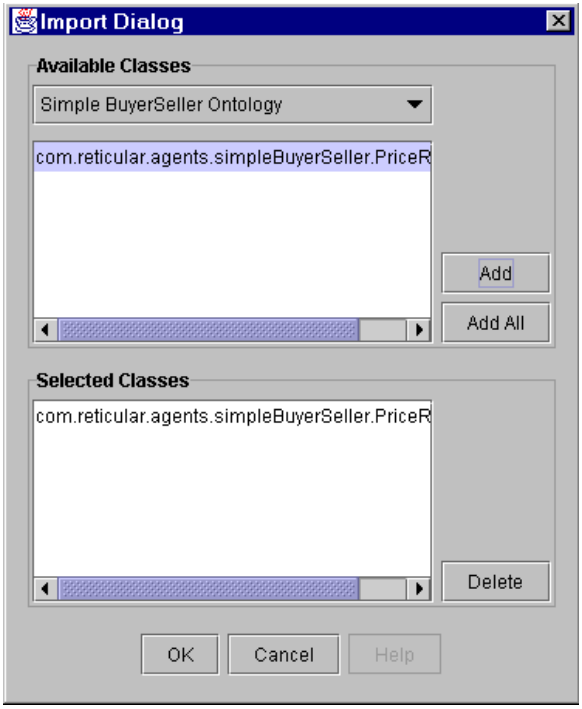

Figure 99. Simple Seller Import Dialog

Step 5a. Create the WaitForIncomingMessage rule.

This rule will put the agent to sleep until the agent receives a message. First, open the **Rule Editor** by selecting **Tools → Rule Editor** in the Agent Manager's menu bar. Then, select the **Edit → Properties** menu item. Enter the name and description for the rule in the **Rule Properties** dialog.

On the Left-Hand Side of the rule you need to create a Mental Condition that binds to the Agent instance named SELF. You do this by selecting **<Operators> →BIND,** clicking on **Instances…** button, selecting **Agent SELF** and clicking on the **OK** button. Click on the

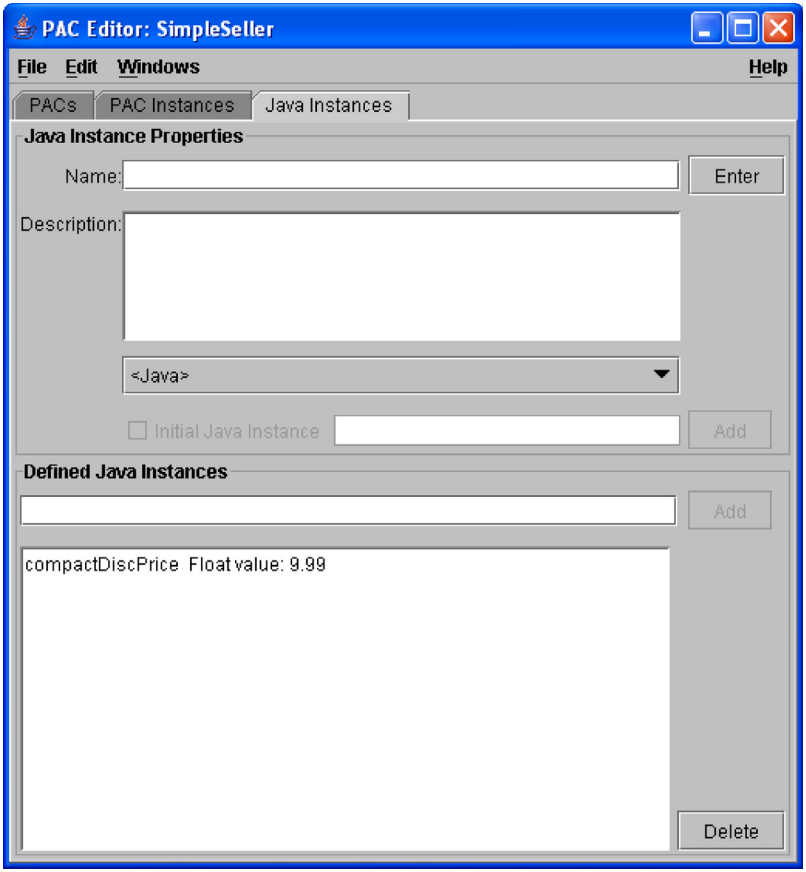

Figure 100. Simple Seller PAC Editor

Add button in the Rule Editor to add the new pattern to the **Mental Conditions** list.

The right-hand side of the rule will contain a single built-in action. Select **<Built-in Actions>** →**SleepUntilMessage**, and click on the **Add** button. Now save the rule by selecting **File** → **Save** in the **Rule Editor** window.

Step 5b. Create the RespondToIncomingMessage rule.

This rule will test all incoming messages if they are requesting a price quote. If this is the case, it will fill in the price and store name in the `PriceRequest` object and send it to the agent that requested it.

To enter a new rule, select the **File → New → New Rule** menu item. Enter the name *`RespondToIncomingMessage`* and a brief description of the rule. On the Left-Hand Side of the rule click **New Variable…** button to create two variables of type `KqmlMessage` and `Agent`. The **New Variable Dialog** is shown in Figure 101. Select **KqmlMessage**, enter *`%message`* as the variable name and click on the **Add** button. This will display the **Binding Dialog**; select **Incoming** and click on **OK** to close the **Binding Dialog**. Now, select **Agent** and enter *`?buyer`* as the variable name, and click on the **Add** button. Click on **OK** to close the **New Variable** dialog.

To create the first Message Condition, select **<Operators> → EQUALS** using the Rule Editor. This will insert **(<> EQUALS <>)** in the accumulator. Next, click on **Defined Variable…**., double click on **%message**, select **sender**, and click on the **OK** button to insert **%message.sender** in the accumulator. To finish the EQUALS condition, click on **Defined Variable…** button, open the **?buyer** folder and then the **agentInfo** folder. Finally select the **name** and click on **OK.** This will enter the pattern **?buyer.agentInfo.name** into the Rule Editor. Click on the **Add** button to add the pattern to the **Message Condition** list.

Follow the same process for the performative and contentType message conditions. The conditions of the rule should ensure that the performative equals the string constant `ask-one` and the **contentType** is `PriceRequest`. The values needed for each pattern can be found by selecting **<Value> → KqmlMessage**. Figure 102 shows the left-hand side of the **RespondToIncomingMessage** rule.

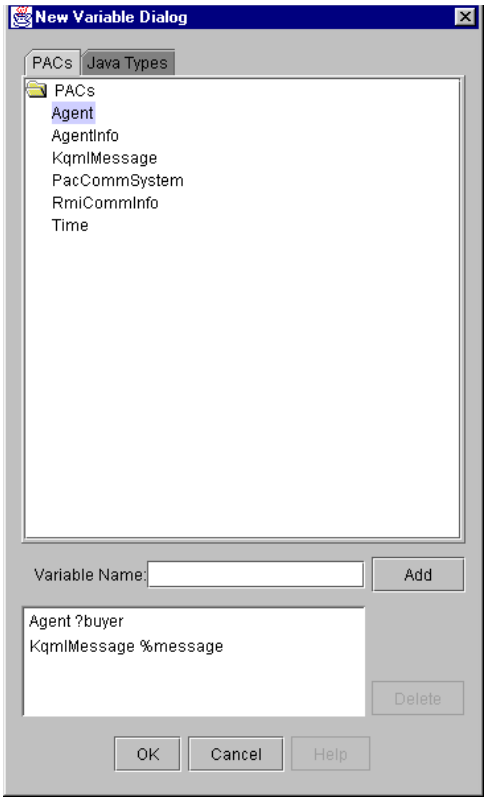

Figure 101. Simple Seller New Variable Dialog

On the right-hand side of the rule, we will print a message that we received with the price request, reply to the message, and print another message confirming that we have sent the price quote to the requesting agent.

To print a message select **<Built-in Actions> → SystemOut-PrintIn**, this will add **Do SystemOutPrintIn(<output>)** to the accumulator. Next, select **<Operators> → Concat**, this will add **Concat(<string>, <string>)** to the accumulator. Now, select **<Value> →String**, this will pop-up a window; type *`Received a price`*

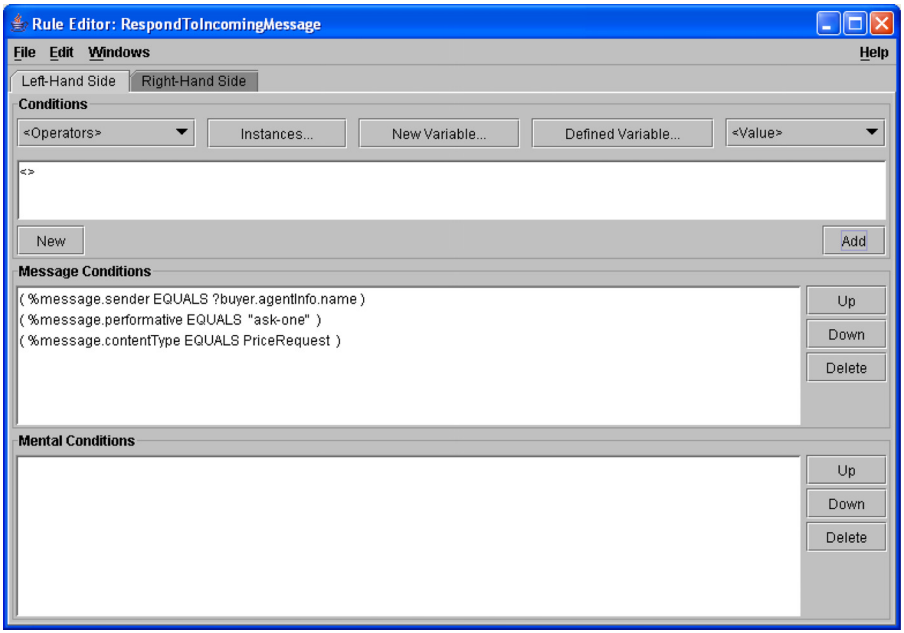

Figure 102. Simple Seller Rule Editor

quote from in the **Literal Value** field and click on **OK**. To print the name of the agent requesting a price quote, click on **Defined Variable…**, select **?buyer.agentInfo.name** and click on **OK**. Now, add the action to the **Defined RHS Elements** by clicking on the **Add** button.

Next, we need to fill in the `price` and `storeName` on the `PriceRequest` object we received from the buyer agent. In the **Mental Changes** panel, select **<Operators> →SET_VALUE_OF,** this will add **SET_VALUE_OF <> TO <>** to the accumulator. Click on **Defined Variable…** to select **%message.content**, select **<Casting Type> → CorbaPriceRequest**, and select **%message.content.price**. To finish the **Mental Change**, click on **Instances**, select the **Java Instances** radio button, select **compactDiscPrice**, and click on **OK**.

Follow the same process to set the value of **%message.content.storeName** to **SELF.agentInfo.name**. The instance `SELF` can be found in the **Instances** dialog.

Now, we need to send the buyer agent the message with the `CorbaPriceRequest` information. In the **Actions** panel, select **<Built-in Actions> → SendKqmlMessage**, this will add **Do SendKqmlMessage(<message>,<receiver>, <performative>,<content>,<replyWith>,<inReplyTo>,<language>,<ontology>, <protocol>, <to>,<from>)** to the accumulator. Since we are going to reply with the same message, we only need to fill in five parameters of the `KqmlMessage`. The first four parameters are the following: `%message, %message.sender, "tell", %message.content.` **<replyWith>,** `%message.replyWith`. Note: you do not need to enter anything in the **<replyWith>** field. The **<replyTo>** field should be set to `%message.replyWith`. **Warning! You will need to explictly cast** `message.content` to Corba `Price Request`. Otherwise, you will receive an error message at runtime complaining that the object is not serializable.

The next action is another print statement. Follow the process described above to construct a string with the values *`Sent price quote to`* and `%message.sender`.

Finally, we need to add another action that will put the agent to sleep until it receives another message from an agent. Select **<Built-in Actions> →SleepUntilMessage** to add **Do SleepUntilMessage()** to the accumulator. All we need to do now is add the rule to the list of Defined RHS Elements and save the rule. Your rule should be similar to the one shown in Figure 103.

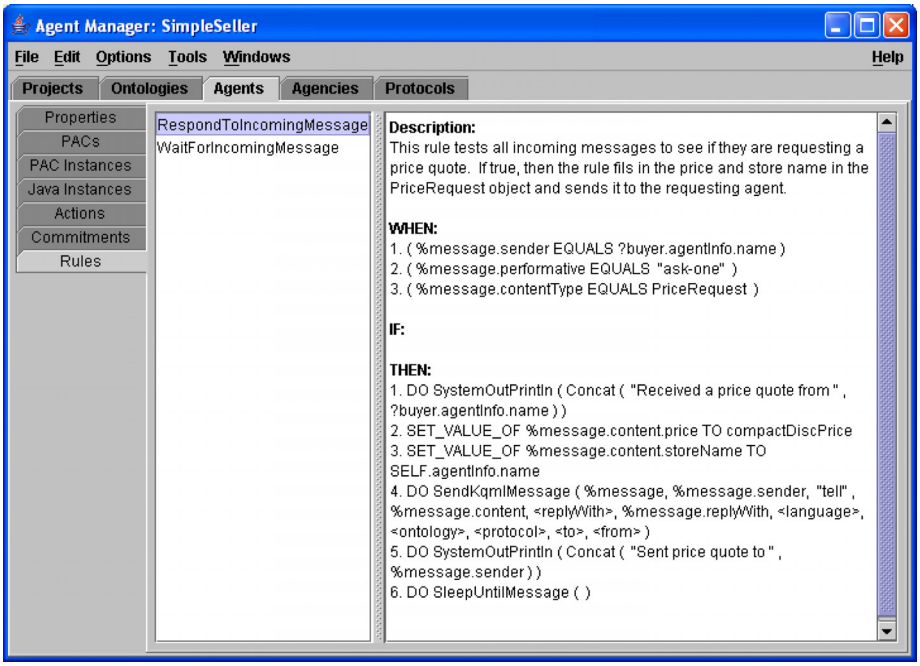

Figure 103. Simple Seller Rules

Step 6. Create SimpleBuyer agent.

To create the **SimpleBuyer** agent, follow the same procedures described in step 3. The agent properties will be the same as the **SimpleSeller** agent except for the name and the port number.

Step 7. Import PAC Object.

To import the PAC Object, please refer to Step 4. Note: the Simple-Buyer agent doesn't need the Java Instance that the SimpleSeller agent needed.

Step 8. Create rules.

The **SimpleBuyer** agent will need rules to create a `PriceRequest` object, send the `PriceRequest` to each agent that it knows about, receive the `PriceRequest` object and print them on the console.

Step 8a. Create Price Quote rule.

In the **Rule Editor Properties** dialog enter the name *`CreatePrice-Quote`* rule and a brief description. On the Rule Editor's Left-Hand side enter a Mental Condition that binds to `SELF` (See second paragraph of Step 5a). On the Right-Hand Side of the rule ASSERT a new `PriceRequest` object. **Select <Operator> → ASSERT**. This will bring up the **Assertion Dialog**, leave the field blank and click on **OK**. This will insert **ASSERT(<>)** in the accumulator. Next, click on **New Object…** select **CorbaPriceRequest**, select **<Constructor> → CorbaPriceRequest(String param0, int param1, String param2),** and click on **OK**. This will add **CorbaPriceRequest(<param0>, <param1>, <param2>)** to the accumulator. Fill in the parameters with the String *`CompactDisc`*, integer 1, and String *`CD`*. Add the Mental Change to the Defined RHS Elements and save the rule.

Step 8b. Create Send price request to store agents rule.

This rule is executed by the `priceQuote` instance created in the previous rule. Clear the Rule Editor using the **File → New → New Rule** menu item. Specify the name (**Send price request to store agents**) and description of the rule. Create a new variable of type `PriceRequest` with the name `?priceQuote` (**New Variable…,** select **CorbaPriceRequest**, enter *`?priceQuote`* in the **Variable Name** field); then add a mental condition to BIND to the `?price-Quote` variable (**<Operators> → BIND** followed by **Defined Variable….**, `?priceQuote`).

You will also need to create a new variable of type `Agent` with name *`?agent`*.

The next Mental Condition will prevent the agent from sending the message to itself. Select **<Operators> → NOT_EQUALS** in the Conditions panel. Enter *?agent.agentInfo.agentName* and *SELF.agentInfo.agentName* as the parameters and add it to the list of Mental Conditions.

On the Right-Hand Side of the rule, we will construct a `KqmlMessage` and send it to the SimpleSeller agents. Only the first five parameters are needed in order to send the KqmlMessage. The parameter values are `KqmlMessage(), ?agent.agentInfo.agentName, "ask-one", ?priceQuote, "Price Quote"`. After constructing the KqmlMessage, add it to the list of defined RHS elements.

Now, we need to print a message indicating that we are forwarding the message to the Seller agents. Select **<Built-in Actions> → SystemOutPrintln, <Operators> → Concat**, select **SELF.agentInfo.name** in the **Instances Dialog**, then **<Operators> → Concat**, **<Value> → String**, enter *forwarding price quote request to* in the **Literal Value** field**,** and select **?agent.agentInfo.name** from the **Defined Variable Dialog**. Add the action to the list of Defined RHS Elements.

To finish the rule, add the built-in action, `SleepUntilMessage`, and save the rule. Figure 104 shows the completed **SendPriceRequestToStoreAgents** rule.

Step 8c. Create Receive price quotes from store agents rule.

This rule will receive the price quote from the SimpleSeller agent and print the price of the product. We need to create a `KqmlMessage` variable (click **New Variable…**, select **KqmlMessage**, enter *%message* in **Variable Name** field, and click on **OK**) and an Agent Variable with the name of **?agent**. Follow the same process as step 4b (third paragraph) to test if `%message.sender` equals `?agent.agentInfo.name`, `%message.performative` equals `tell`, `%message.contentType` equals `CorbaPriceRequest`, and `%mes-`

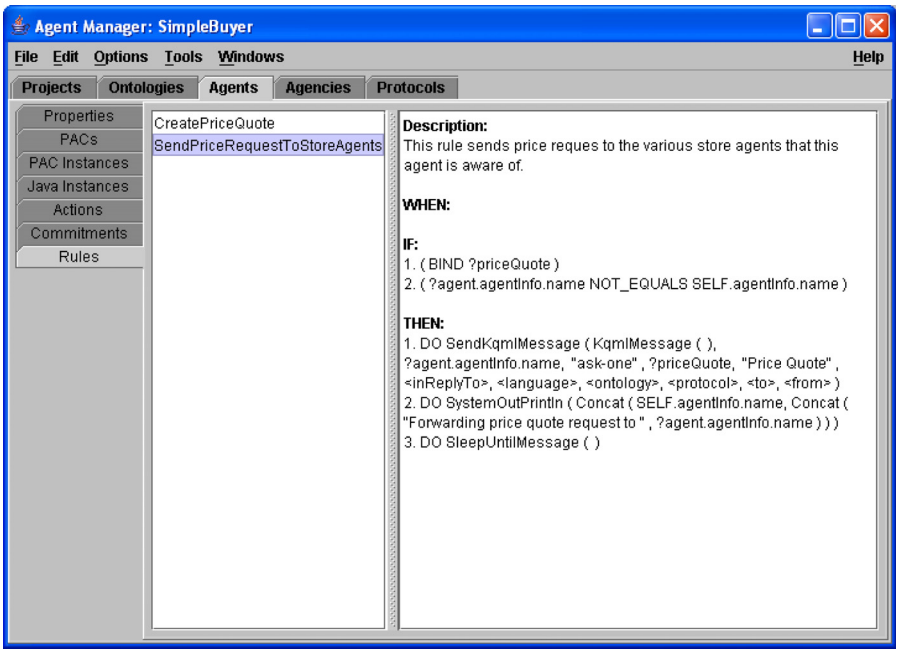

Figure 104. The Send price request to store agents Rule

`sage.content.productName` equals "`CompactDisc`" (remember that the message content must be cast to `PriceRequest`).

For the Right-Hand Side of the rule all we need to do is print (to the console that sent the message) the price quote of the product that was requested. Select **<Built-in Actions> →SystemOutPrintln**, **<Operators> → Concat**, **<Value> → String**, enter *`Received price quote from `* , click on **Defined Variable…**, select **?agent.agentInfo.name**, and click on **OK**.

All that's left to do is add the built-in action `SleepUntilMessage` (**<Built-in Actions> → SleepUntilMessage**). Add the action to the list of **Defined RHS Elements** and save the rule. Figure 105 shows the **ReceivePriceQuotesFromStoreAgents** rule.

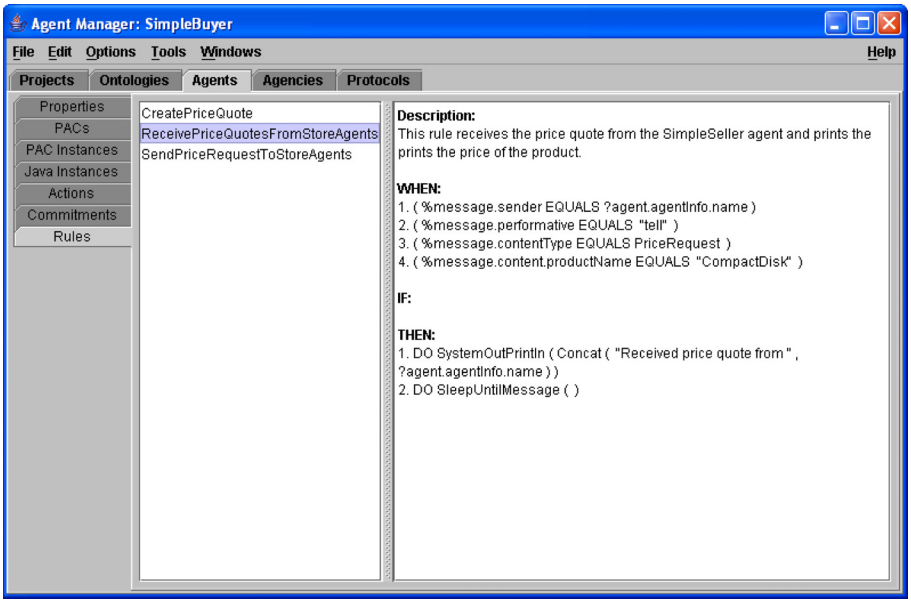

Figure 105. The Receive price quotes from store agents rule

Step 9. Modify Agency Communications

We first need to change the communications being used by the **BuyerSeller** agency. From the Project Manager, select the **BuyerSeller** Agency. Next choose **Edit → Properties...**, to modify the agency's properties. Now click on the **Communications...** button.

Select **CORBA** as the default communication to be used in the agency. Make sure to deselect other communications methods. Specify the port number to be used for communications. Note that this port number will need to be the port number that is used by the CORBA name server. Now click on the **OK** button to close the **Communications** Dialog. Now, click the **OK** button on the **Agency Properties** dialog to close it. See Figure 106. .

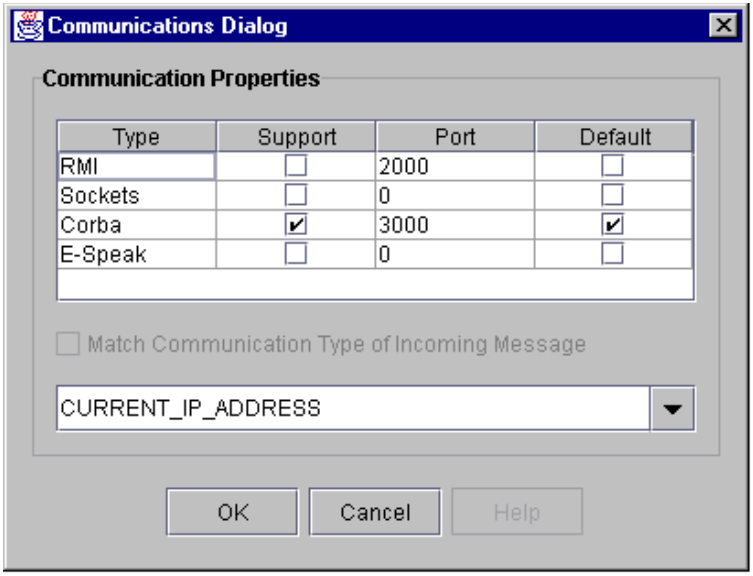

Figure 106. Communications Dialog for CORBA Buyer Seller

Step 10. Modify Agent Communication

Now that the communication for the agency has been established, we now need to set the communication for each agent in the agency. For this agency, we will need to set the communication for the **SimpleBuyer** and for the **SimpleSeller** agents.

From the **Project Manager**, select the Simple Buyer agent. Next choose **Edit → Properties...**, to modify the agent's properties. Now click on the **Communications...** button.

Select CORBA as the default communication to be used by the agent. Make sure to deselect other communications. Specify the port number to be used for communications. Note that this port number will need to be the same port number that was specified for

the agency. Now click on the **OK** button to close the **Communications Dialog**. Now, click the **OK** button on the **Agency Properties** dialog to close it. See Figure 107. .

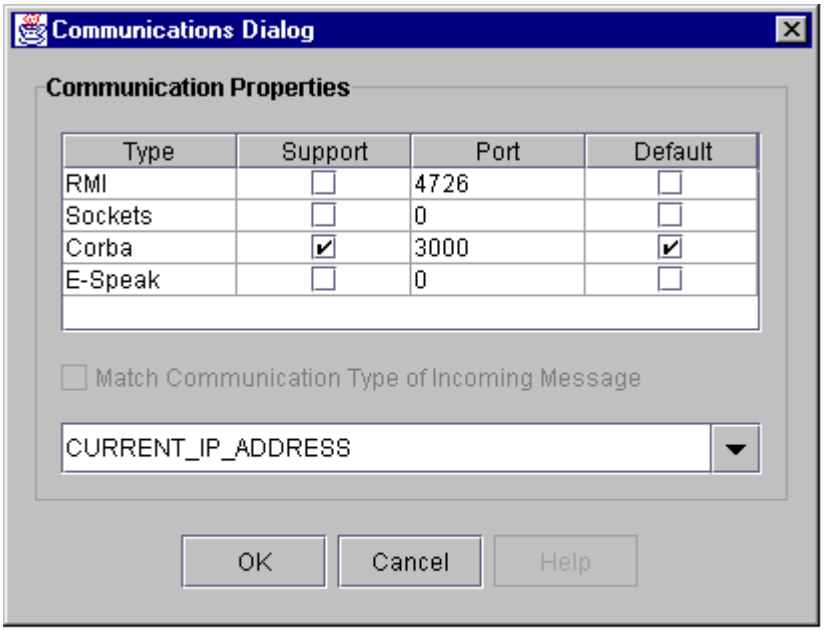

Figure 107. Simple Buyer Communications Dialog

Repeat the above procedure for modifying the communication settings for the **SimpleSeller** agent.

Step 11. Run the Nameserver

Now that the communication has been set for the agency and for all agents in the agency, we now need to run the CORBA name server. Currently, AgentBuilder only supports the CORBA name server that is distributed with Sun's Java Runtime Environment. Please

contact Acronymics, Inc. if you need support for another name server.

Now, open a command-line shell. For Windows, this will be your DOS or Command window and for UNIX, this will be your command-line shell. Run the following command:

<path_to_agentbuilder_installation>/Jre/bin/tnameserv -ORBInitialPort port

This will run Sun's Transient Name Server on the specified port. Remember to use the same port number that was specified for the agency and agents above. When the CORBA name server begins running, the agents can communicate with each other. See Figure 108.

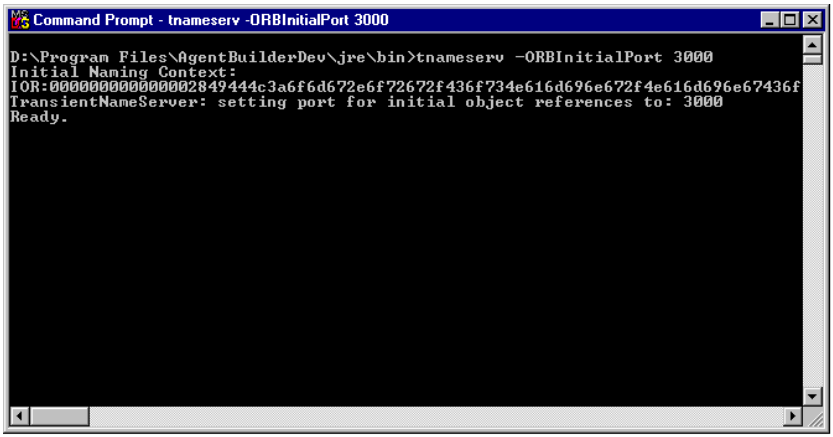

Figure 108. Running the Transient Name Server

Step 12. Run agents.

With the communication set and the name server running, we are now ready to run the agents. When running the agents, the order in

which they are started is important. The **SimpleSeller** agent must be started before the **SimpleBuyer** agent. This will prevent the **SimpleBuyer** agent from sending a message to an agent that doesn't exist.

The agents can be started from the AgentBuilder application or from the engine application. If you decide to run the agents from the engine, create the RADL files first by selecting **Options → Generate Agent Definition** from the Agent Manager.

To start the agents from the from the Agent Manager window, open **SimpleSeller** agent and select **Options → Run Agent**. Now, open the **SimpleBuyer** agent from the same Agent Manager window and select **Options → Run Agent**.You will see the print statements on each of the agent's console window. Figure 109 and Figure 110 show the console windows for both agents.

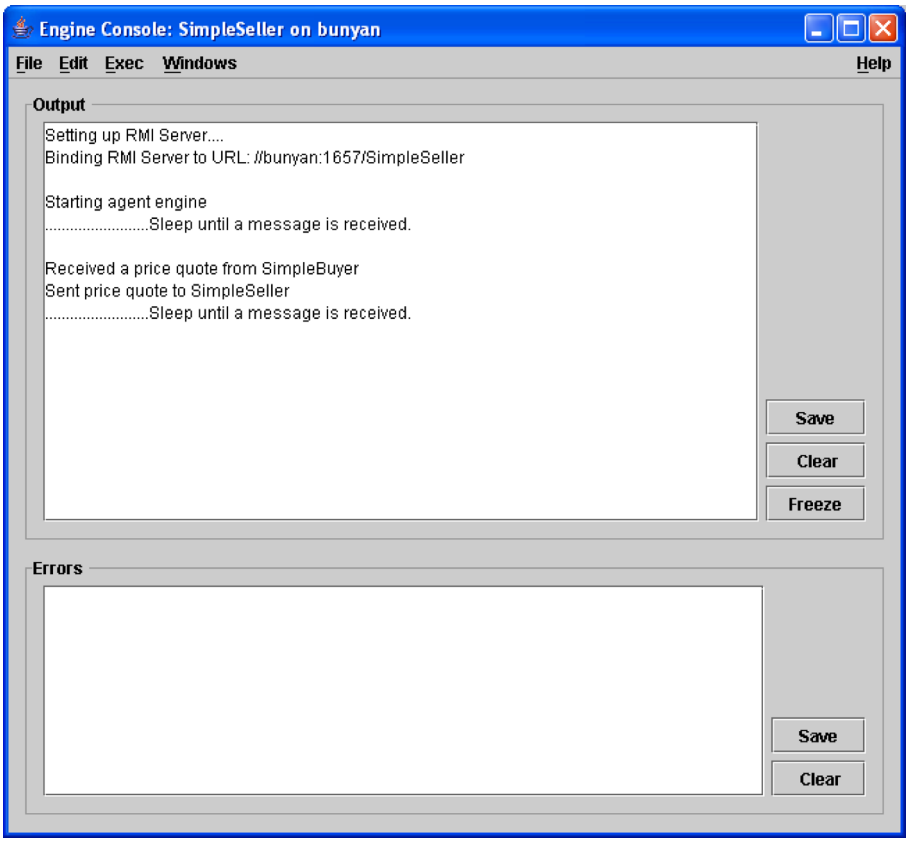

Figure 109. Simple Seller Agent Console

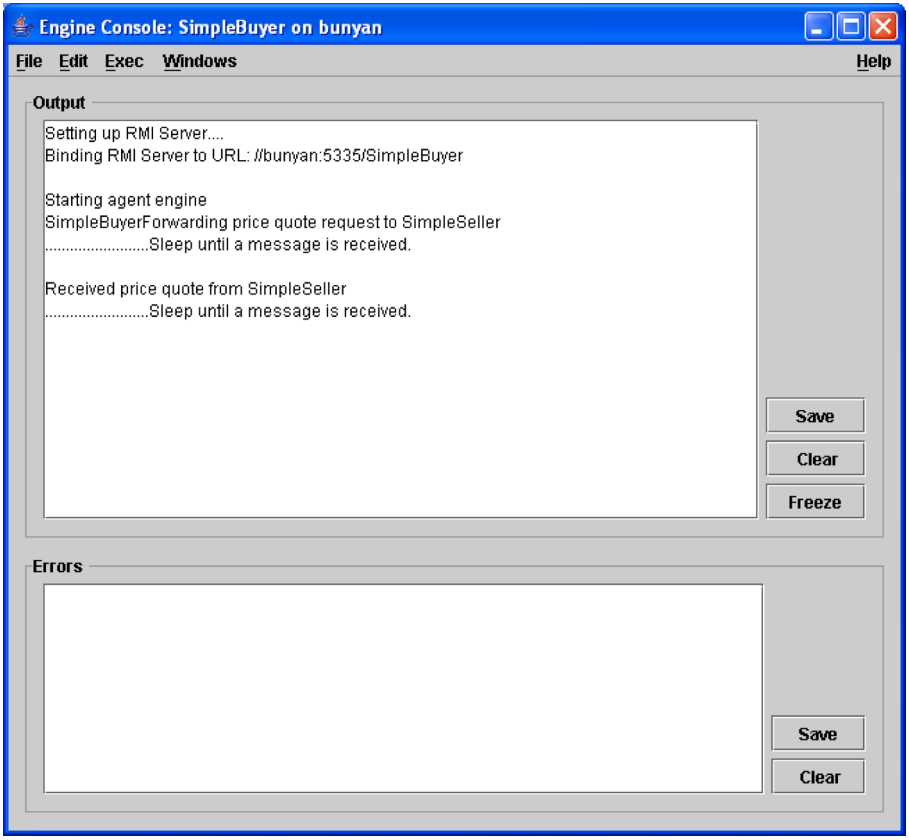

Figure 110. Simple Buyer Agent Console

Chapter 12

Creating and Running Agents using Protocols

This chapter provides detailed information on constructing agents that use a communications protocol. You will learn how to use:

- Roles and protocols
- Protocol Editor
- Agency Viewer
- Debugging Agencies

NOTE: The tools described in this chapter are available only with the AgentBuilder Pro version of the toolkit.

This chapter provides details for recreating the Simple Buyer and Seller agents using agent interaction protocols. These two agents are very similar to the original Simple Buyer and Simple Seller. The primary difference being how their rules are constructed. This chapter describes:

- Creating the **SimpleBuyerSellerWithProtocol** agency.
- Automatically creating the **SimpleBuyer2** and the **SimpleSeller2** agents.
- Constructing the **SimpleBuyerSellerProtocol** - an interaction protocol.
- Applying the protocol to the **SimpleBuyerSellerWithProtocol** agency.
- Finishing the **SimpleBuyer2** and the **SimpleSeller2** agents.
- Running the agents using the **Agency Viewer** tool.

This example demonstrates how to build two agents that communicate with each other using the built-in agent communication subsystem. The agents will communicate with each other using a protocol built using the Protocol Editor. The protocol specifies how the agents interact when they are involved in negotiations. In this particular case, for simplicity, the agents will be performing exactly the same interaction as the **SimpleBuyer** and **SimpleSeller** agents in Chapter 10. The content of the messages is the **PriceRequest** PAC from the **SimpleBuyerAndSeller** ontology. The basics of the interaction is that the **SimpleBuyer** sends a partially completed **PriceRequest** PAC and the **SimpleSeller** completes it and returns it. See the previous chapter for more information about **SimpleBuyer** and **SimpleSeller** interactions.

Table 13. Creating Two Agents with Protocols

Step	***Description***
1.	Create the **BuyerSellerWithProtocol** agency.
2.	Create the **SimpleBuyer2** and **SimpleSeller2** agents.
3.	Copy or create the “Simple Buyer Seller Ontology”
4.	Create the **SimpleBuyerSellerProtocol** with the Protocol Editor. a. Create the two roles. b. Create the three states. c. Create the Request_Quote transition. d. Create the Price_Quote_Reply transition
5.	Import the protocol into the **SimpleBuyerSeller-WithProtocol** agency. a. Assign agents to roles.
6.	Finish the agents
7.	Run the agents in the **Agency Viewer**.
	Create Rules a. Create **PriceQuote** rule. b. Create **SendPriceRequestToStoreAgents** rule. c. Create **ReceivePriceQuotesFrom-StoreAgents** rule.
8.	Run the Agents

Step 1. Create the BuyerSellerWithProtocol agency.

The first step is to create the **BuyerSellerWithProtocol** agency. Go to the Project Manager and select the project containing your other example agents (Quick Tour Projects). From the Project Manager's **File** menu, select **New**. This will display an **Agency Properties Dialog** to appear. Type in the name of the agency (*SimpleBuyerSellerWithProtocol*) and click **OK**.

Step 2. Create the SimpleBuyer2 and SimpleSeller2 agents.

Go to the **Project Manager** and select the new agency you created. Now select **New** from the **File** menu. This will again display an **Agent Properties Dialog**. Fill in the information and click **OK**. The suggested name is *SimpleBuyer2*. Now repeat the process to create a *SimpleSeller2*. You Project Manager should now look like Figure 111.

Step 3. Copy or create the Simple Buyer Seller Ontology from the system repository.

If you have not already done so, copy the **Simple Buyer Seller Ontology** from the **System Repository** to the **user** repository. See Chapter 10, step 1 if you encounter problems. This is a simple ontology; the object model contains a single item, the **PriceRequest** class.

Step 4. Create the SimpleBuyerSellerProtocol with the Protocol Editor.

The next step is to create a protocol for use by the agents. This step involves several steps which must be performed in a specific order. The protocol being created is called the *SimpleBuyerSellerPro-*

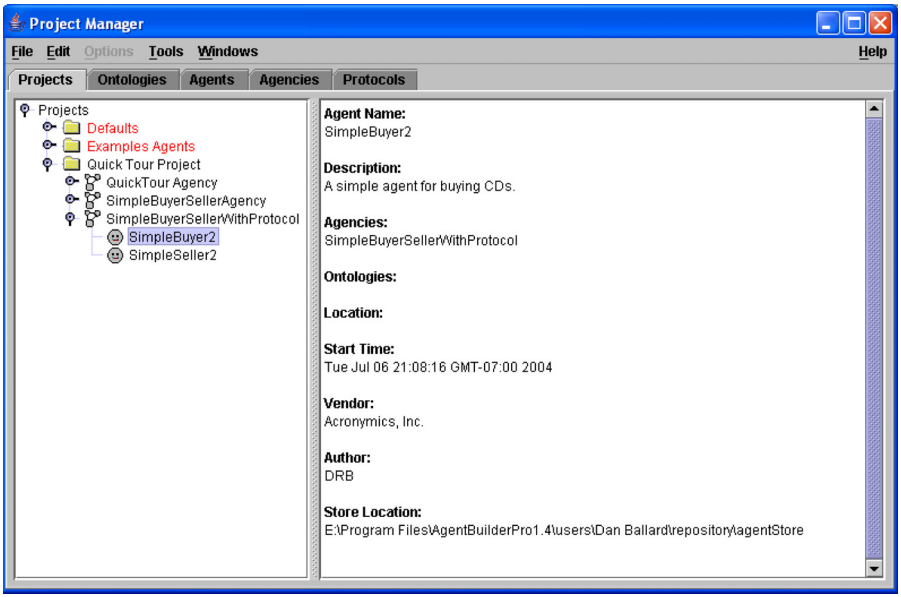

Figure 111. Project Manager View

tocol and will be imported by the new agency. This requires defining two roles, the Buyer and Seller roles and creating a protocol state diagram. A role is the defined pattern of communications that an agent can assume when implementing a protocol. An agent can assume 0,1 or more roles, depending on the protocol. Sometimes it will be appropriate for a buyer to also be a seller and vise-versa. Thus an agent can assume more than one role when appropriate. The state diagram describes the state of the protocol. Do not confuse this with the overall state of the agency. Transitions in the state diagram represent changes in the state of the protocol, i.e. communications between agents.

Go to the protocol manager by selecting the **Protocols** tab in the **Project Manager**. Click on the **User** icon and then select **File →New...** from the **File** menu. Name the new protocol *SimpleBuy-*

erSellerProtocol. Click on **OK** to close the window and continue. The window will appear as shown in Figure 112.

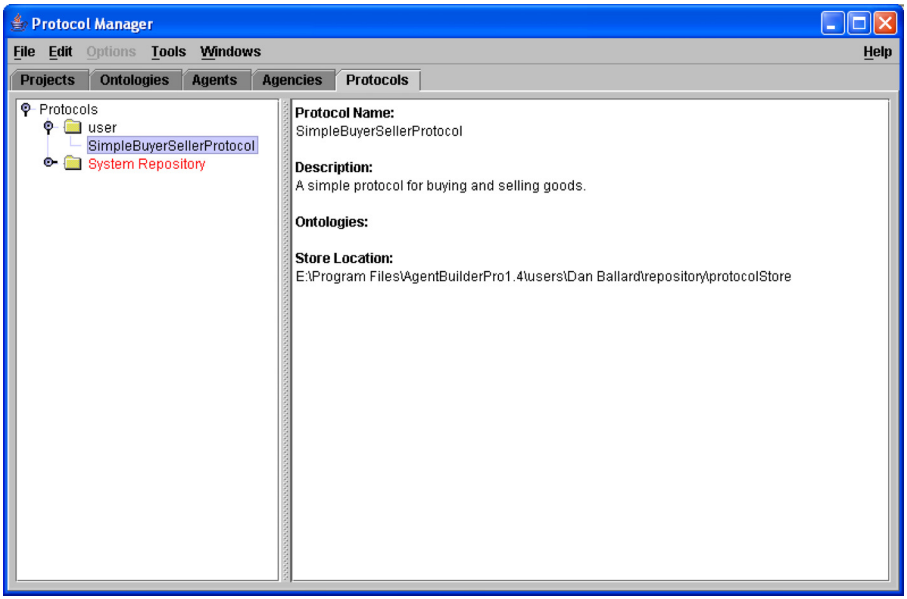

Figure 112. The Protocol Manager

Step 4a. Create the two roles.

The first step in defining a protocol is to specify the roles that exist in the protocol. In this particular protocol there are two roles, the Buyer Role and the Seller Role. Click on the **SimpleBuyerSeller-Protocol** item in the **Protocols** tree. Now select the **Protocol Editor** menu item from the **Tools** menu. This will display the drawing canvas used to graphically construct the protocol. Now select **Roles…** from the **Diagram** menu. This will display a dialog for entering, editing and deleting roles from this protocol. Select the **New Role...** menu item from the **File** menu. Now type in *Buyer-Role* in the name field, a description and then enter *1* in the

Instances field. (The combo box is editable, so delete the label and type *1*.).

The instances allows you to specify a specific number of instances or a range. There are three choices when specifying the value of the instances field: zero or more, one or more, or a specific number. The first two mean that a dynamic number of agents can implement this particular role. For example, it may not be known how many buyers are going to be part of an agency, so you would select "zero or more". If you knew that at least one buyer was needed for the protocol to work, you would choose the "one or more" option. If there needs to be an exact number of agents implementing a role to make the protocol work, specifying the exact number would be appropriate. For example, you might want one and only one facilitator in an agency.

Click **OK**, this will enter the role into the **Roles Dialog**. Now repeat the procedure to create a `SellerRole`. The **Roles Dialog** will now look like Figure 113. When you are finished select **File →Close**.

Step 4b. Create the three states.

The next step in defining a protocol is to define the communication state diagram. To do this return to the **Protocol Editor** (if you haven't already done so). Then put your mouse in the drawing area and right-click. This will display a popup dialog that allows you to create states and transitions.We'll use this to create the states and transitions of the protocol. First create a new state; this is done by selecting the **New State** menu item popup menu. This will display a **State Properties Dialog**. Fill in the **Name** field with *Start*, then add a description, finally, select the type as **Initial** for an **initial state**. This means that the protocol starts from this state. The dialog will appear as shown in Figure 114. Click on **OK**.

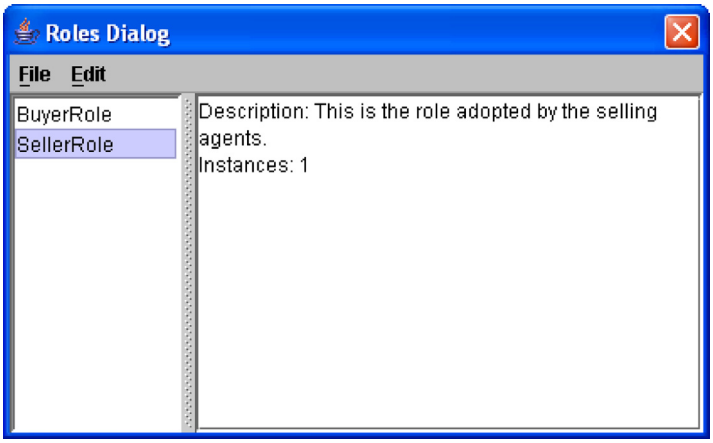

Figure 113. Buyer Seller Roles Dialog

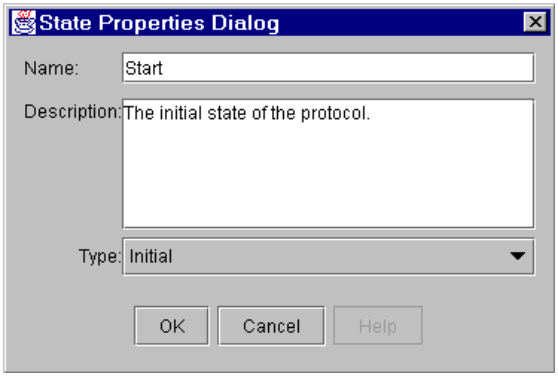

Figure 114. State Properties Dialog

Now create two more states. One non-terminal state will be of type **Standard** and will be called `Request` and a final state will be called `Done`. When you are finished, the diagram should look like Figure 115.

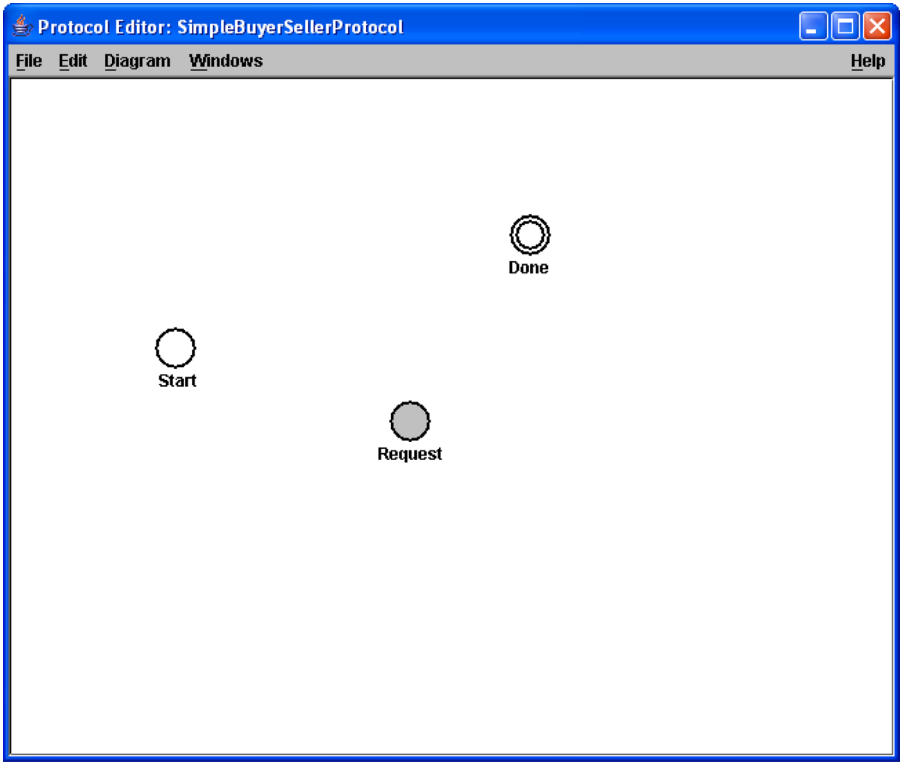

Figure 115. The Protocol Editor

Step 4c. Create the Request_Quote Transition

In this step you will specify the communication that causes the event which produces a state change in the protocol. You will be creating a template for a KQML message that will be sent and received by the agents implementing the roles. You specify various fields in a KQML message which will then be used by the tools to develop message sending and handling rules in the appropriate agents. New rules are created for each agent sending messages for the receiving agents for handling incoming messages. The rule gen-

eration is accomplished when importing the protocol into an agency.

The first step is to go to the drawing area (not over one of the cirches which represents a conversational state) and right-click. Now choose the **New Transition →Normal...** menu item which will cause the cursor to change to cross-hairs. Now select the state you wish the transition to start from (the Start state), hold down the mouse button and move it to the new transition state (the Request state). This will display the **Transition Properties Dialog**. The dialog allows you to specify fields that go into a KQML message. Fill in the name with *Price_Request*. The **Transition Properties Dialog** does not allow white space in the name. Add a description like:

This transition represents a request that the Seller Role agent fill out a PriceRequest PAC and return it.

The transition must be properly filled out for it to be added to the protocol. The **Sender**, **Receiver**, **Performative** and **Content Type** fields must all be specified. Add the following information:

Name: *Price_Request*

Sender: *BuyerRole*

Receiver: *SellerRole*

Performative: *ask-one*

Ontology: *Simple BuyerSeller Ontology*

ContentType: *PriceRequest*

Reply-With: *price-request*

It should be noted that you will not be able to find the **PriceRequest** in the **Content Type** list until the correct ontology is selected. This is because the ontology selection dictates which classes are available for sending. All classes defined in the object model of that ontology are available for selection as the content type of the message.

The **Transition Properties Dialog** will now appear as shown in Figure 116. Select **OK** after entering all of the relevant information. If you do not complete the required fields, you will not be able to close the dialog.

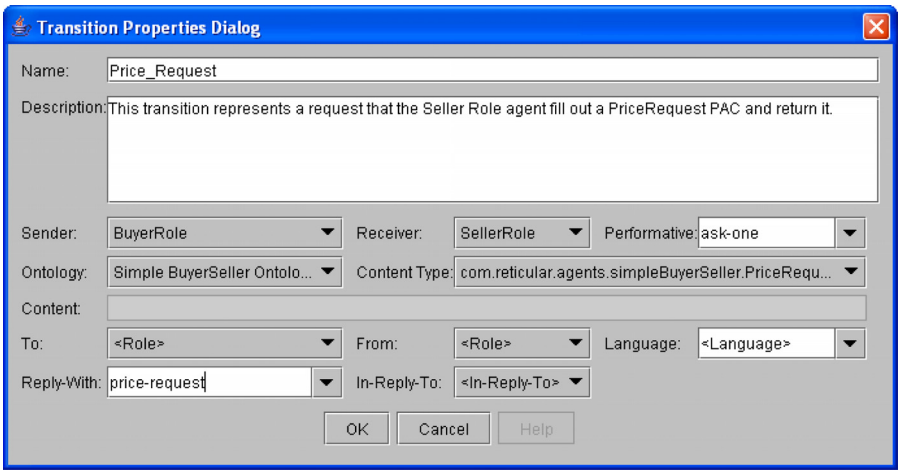

Figure 116. Price Request Transition Dialog

Step 4d. Create the Price_Quote_Reply transition

Create a new transition between the **Request** and **Done** states. The fields should be filled out as follows:

Name: *Price_Quote_Reply*

Sender: *SellerRole*

Receiver: *BuyerRole*

Performative: *tell*

Ontology: *Simple BuyerSeller Ontology*

ContentType: *PriceRequest*

In-Reply-To: *price-request*

When you are finished, the dialog should appear as shown in Figure 117. Now save and close the **Protocol Editor**.

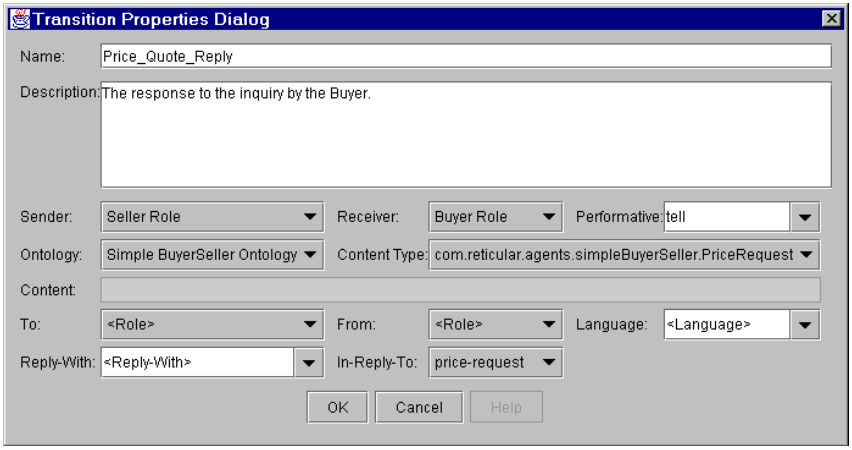

Figure 117. The Transition Properties Dialog

Step 5. Import the protocol into the SimpleBuyerSellerWithProtocol agency.

The next step is to import the protocol into the agency. Protocols are completely decoupled from actual agents and agencies. After the protocols are completed they are imported into an agency (or more than one agency) and the mapping is made between the roles of the protocols and the agents in the agency.

First, open the **SimpleBuyerSellerWithProtocol** agency in the **Agency Manager**. This requires either clicking on the **Agencies** tab and opening the agency using the **File** menu, or clicking on the desired agency in the **Projects** tree and then clicking the **Agencies** tab. Either technique displays the **SimpleBuyerSellerWithProtocol** agency in the **Agency Manager.** In the **Agency Manager,** select the **File → Import Protocols…** menu item. The **Import Protocol Dia-**

log will be displayed with the **SimpleBuyerSellerProtocol** in the list. Select this protocol and then click **OK**.

The next step is to make the role assignments. While in the **Agency Manager**, click on the **Protocols** tab located on the left side of the panel and then select the **SimpleBuyerSellerProtocol**. Now select the **Tools → Role Editor** menu item. This will display the **Role Editor** tool. You can use this tools to assign agents to specific protocol roles. Select the **Buyer Role** in the left side panel and then click **Options → Assign Agents…** menu item. This will display the **Assign Agents Dialog**. Now select **SimpleBuyer2** and click on the **Add** button. You should now have a dialog like that shown in Figure 118. Click on **OK** to close the dialog.

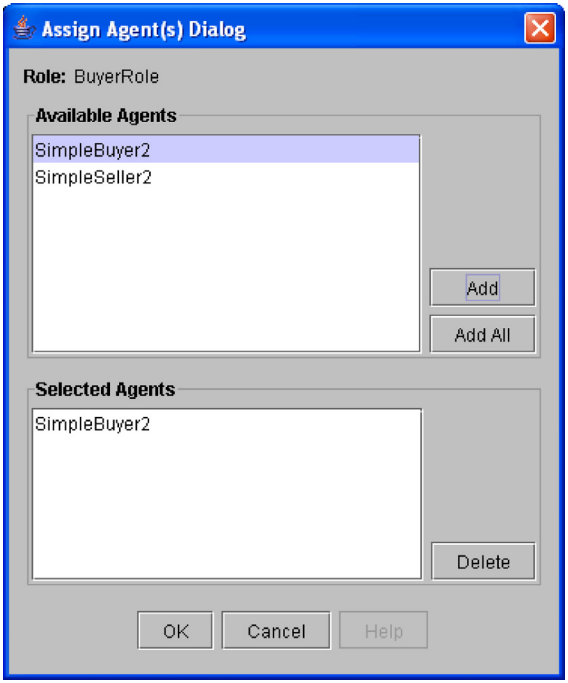

Figure 118. The Assign Agents Dialog

Now select the **SellerRole** in the left panel and follow the same steps except choose **SimpleSeller2** agent for implementing the role. You should now have a Role Editor that looks like Figure 119.

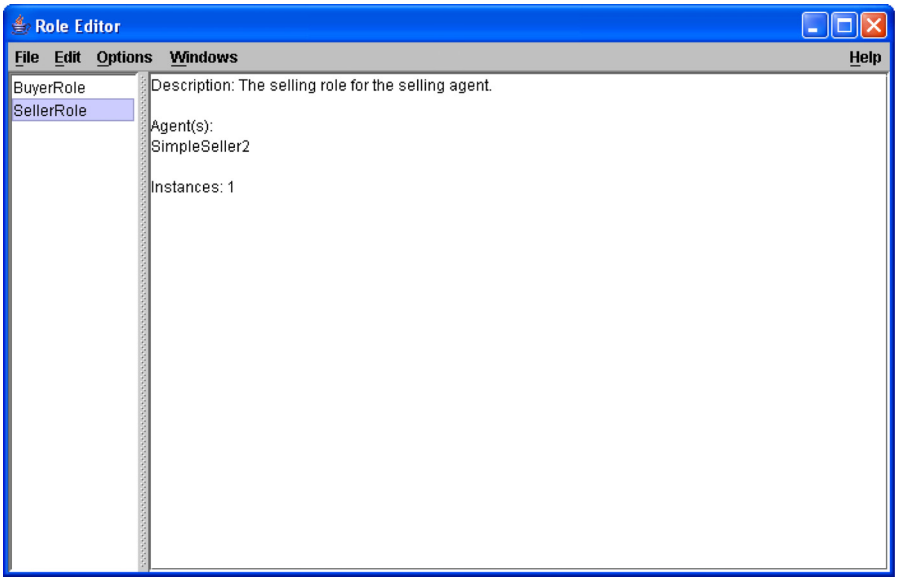

Figure 119. Role Editor

The next step is to update the agents that are affected by the protocol. This step is easily accomplished by choosing the **Options** → **Update All Agents** menu item. This has the effect of generating information (primarily rules) for each agent that assumes a role in the protocol. **Save** and **Close** the **Role Editor** using the **File** menu.

Step 6. Finish the agents

The next step is to complete construction of the agents. The protocol that was imported into the **SimpleBuyerSellerWithProtocol** agency has added several key skeletal rules to the agents but they

must now be completed. The protocol only provides high-level information and you must provide additional detail.

Open the **SimpleBuyer2** agent in the **Agent Manager**, click on the **Rules** tab and select the **SimpleBuyerSellerProtocol Price_Request Message Sender** rule. The **Agent Manager** view will look like Figure 120. The tool has generated two skeletal rules for each agent. It has generated a message handler and a message sender rule for each.

These rules aren't complete because some information must be generated by the developer at agent construction time. For example, the first, second, and fourth parameters need to be added to the **sendKqmlMessage** built-in action. The message handler rules have only conditions and no actions. You must add the appropriate actions.The use of protocols provides two main benefits; every rule that needs to be created for communication is created. Further; many of the conditions and actions are automatically added. However, not all of the information can be specified in the **Protocol Editor**.

It should be noted that the automatically generated rule names are rather large, but the names follow a specified format. The format is protocol-name, transition-name, and rule-type. This provides the automatically generated rules with a unique name. Figure 120 illustrates these rules.

The next steps involve modifying the existing rules of the **SimpleBuyer2** agent to implement the functionality of the original **SimpleBuyer** agent. If you're having trouble remembering the details of the **SimpleBuyer** then review Chapter 10. The complete **SimpleBuyerSellerProtocol Price_Request Message Handler** rule is shown in Figure 121.

The remaining rules should look like those shown in Figure 122 and Figure 123. You need to create an entirely new rule, the `Cre-`

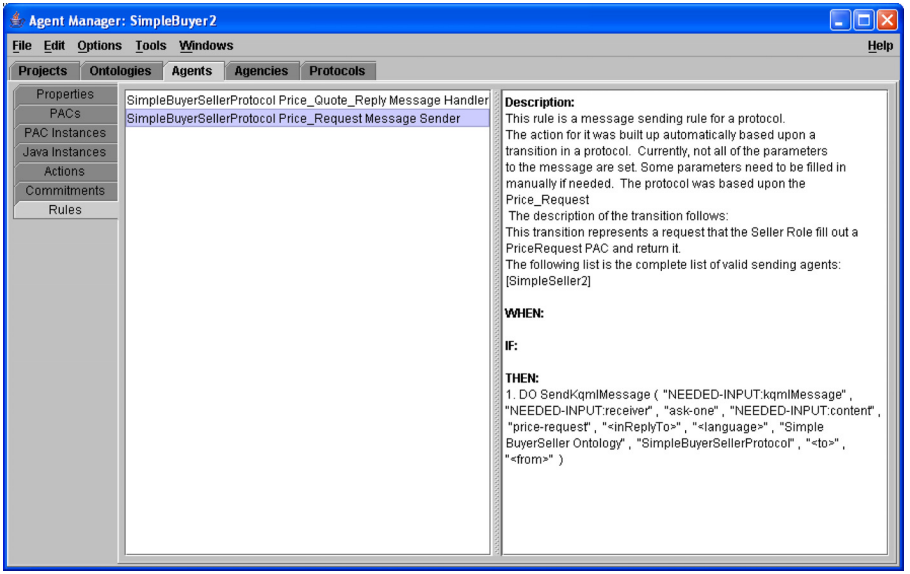

Figure 120. SimpleBuyer2 Skeletal Rules

atePriceRequest rule. In the process of updating the rules, you'll need to edit several of the existing patterns and don't forget that you need to use the **Pac Editor** to import the **PriceRequest** PAC. Also, remember that there are a number of new variables that you must create such as `?priceRequest` and `?agent`.

The next step is to perform a similar task for the **SimpleSeller2** agent. When you're done the rules should look like those shown in Figure 124 and Figure 125. The original **SimpleSeller** agent used only two rules. Notice that this new version uses three. The automatic rule generation produced one rule for handling incoming messages and one for sending a reply. Instead of merging these two rules, they are tied together by asserting `KqmlMessage` and protocol state beliefs into the mental state and then using them as a test condition in the sender rule. Note that we needed to make no changes

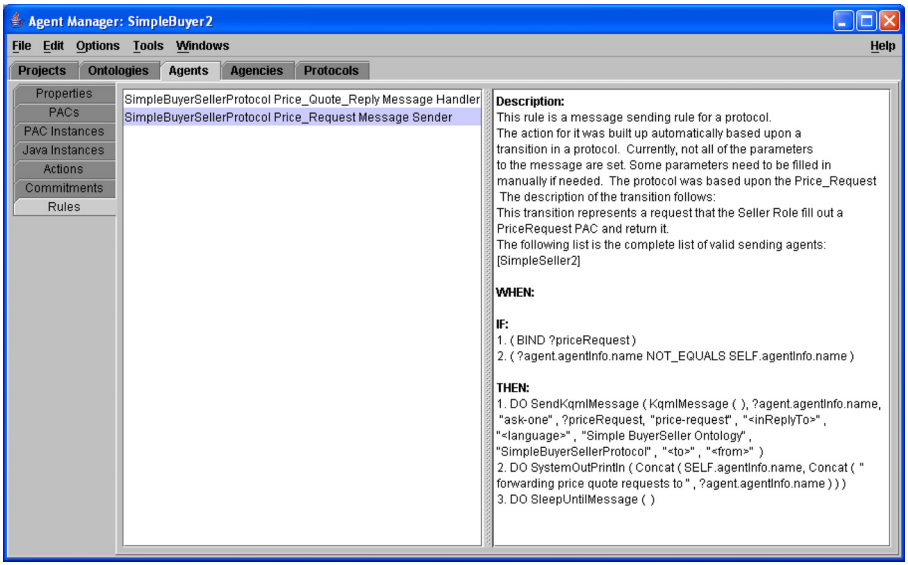

Figure 121. Completed SimpleBuyer2 Rule

to the **SimpleBuyerSellerProtocol Price_Request Message Handler** rule..

You can run each agent from the **Project Manager** by selecting the **Projects** tab, clicking on the agent in the agency hierarchy and selecting the **Tools → Agent Engine** pull-down menu. You should start the **Seller** agent executing before starting the **Buyer** agent. If you have any trouble getting an agent to run you may want to examine the RADL code provided with the example agents in the AgentBuilder distribution.

Compare the RADL code for the agents in the **Example Agents** folder with the RADL code generated for your agents. Pay careful attention to the typing of variables. Also, be sure and perform the explicit cast of KQML message contents as discussed earlier. On a default Windows XP installation the example RADL code can be

```
Rules:
Name: CreatePriceRequest
Description: Creates a new instance of the PriceRequest PAC and places
it in the mental state.
( BIND startupTime )
ASSERT( PriceRequest (  "CompactDisc" , 1 ,  "CD"  ))

Name: Simple Buyer Seller Protocol Price_Quote_Reply Message Handler
Description: This rule is a message handling rule for a protcol.
The conditions for it were built up automatically based upon a
transition in a protocol.  Currently, there are no conditions
being added to make sure the sender is one of the agents in
the protocol.  These can be added manually if needed.
The protocol was based upon the Price_Quote_Reply
 The description of the transition follows:
The response to the inquiry by the Buyer.
The following list is the complete list of valid sending agents:
[Simple Seller 2]
( %incomingProtocolMessage.protocol EQUALS  "SimpleBuyerSellerProtocol"  )
( %incomingProtocolMessage.contentType EQUALS
               com.reticular.agents.simpleBuyerSeller.PriceRequest  )
( %incomingProtocolMessage.performative EQUALS  "tell"  )
( %incomingProtocolMessage.ontology EQUALS  "Simple BuyerSeller Ontology"  )
( %incomingProtocolMessage.inReplyTo EQUALS  "price-request"  )
DO SystemOutPrintln ( Concat (  "Received price quote from " ,
               %incomingProtocolMessage.sender ) )
DO SystemOutPrintln ( Concat (  "The price is " ,
               ConvertToString ( %incomingProtocolMessage.content.price ) ) )
DO SleepUntilMessage (  )
```

Figure 122. Handler Rules for Simple Buyer

found in `C:\Program Files\AgentBuilderPro1.4\Users\yourUser-Name\RADL`.

Step 7. Run the agents in the Agency Viewer.

The next step is to run the agents. The **Agency Manager** provides a tool, the **Agency Viewer**, to help in running more than one agent at a time. This tool allows you to start, stop, pause and reset one or all

```
Name: Simple Buyer Seller Protocol Price_Request Message Sender
Description: This rule is a message sending rule for a protcol.
The action for it was built up automatically based upon a
transition in a protocol.  Currently, not all of the parameters
to the message are set. Some parameters need to be filled in
manually if needed.  The protocol was based upon the Price_Request
 The description of the transition follows:
This transition represents a request that the Seller Role
fill out a PriceRequest PAC and return it.
The following list is the complete list of valid sending agents:
[Simple Seller 2]
IF
( BIND ?priceRequest )
( ?seller.agentInfo.name NOT_EQUALS SELF.agentInfo.name )
THEN
DO SendKqmlMessage ( KqmlMessage (  ),
                ?agent.agentInfo.name ,
                "ask-one" ,
                ?priceRequest,
                "price-request" ,
                "<inReplyTo>" ,
                "<language>" ,
                "Simple BuyerSeller Ontology" ,
                "SimpleBuyerSellerProtocol" ,
                "<to>" ,
                "<from>"  )
DO SystemOutPrintln ( Concat (SELF.agentInfo.name,Conat "forwarding price quote request
to" ,  ?agent.agentInfo.name ) )
DO SleepUntilMessage (  )
```

Figure 123. PriceRequest Message Rule for Simple Buyer

of the agents. It also allows you to inspect messages sent from one agent to another and save the results of a run.

To start the **Agency Viewer**, open the **Agency Manager** with the correct agency. As mentioned previously, there are two different techniques for loading an agency. You can either click on the **Projects** tab in the **Project Manager** and select the agency in the left-hand panel or click on the **Agencies** tab and then use **File → Open** to

```
 Rules:
Name: Init
Description: This rule puts the agent to sleep until a message arrives.
IF
( BIND startupTime )
THEN
DO SleepUntilMessage (  )

Name: Simple BuyerSeller Protocol Price_Quote_Reply Message Sender
Description: This rule is a message sending rule for a protcol.
The action for it was built up automatically based upon a transition in a protocol.
Currently, not all of the parameters to the message are set. Some parameters need
to be filled in manually if needed.  The protocol was based on the Price_Quote_Reply.
 The description of the transition follows:
The response to the inquiry by the Buyer.
The following list is the complete list of valid sending agents:
[Simple Buyer 2]
IF
( ?protocolState EQUALS  "SimpleBuyerSellerProtocol STATE: Request"  )
( ?assertedMessage.protocol EQUALS  "SimpleBuyerSellerProtocol"  )
THEN
DO SystemOutPrintln ( Concat (  "Received a price request from " , ?assertedMessage.sender ))
SET_VALUE_OF ?assertedMessage.content.storeName TO  SELF.agentInfo.name
SET_VALUE_OF ?assertedMessage.content.price TO compactDiscPrice
DO SendKqmlMessage ( ?assertedMessage, ?assertedMessage.sender,  "tell" ,
?assertedMessage.content,  "<replyWith>" ,  "price-request" ,  "<language>" ,
"Simple BuyerSeller Ontology" ,  "SimpleBuyerSellerProtocol" ,  "<to>" ,  "<from>"  )
DO SystemOutPrintln ( Concat (  "Sent a price quote to " , ?assertedMessage.sender ) )
RETRACT( ?protocolState)
RETRACT( ?assertedMessage)
DO SleepUntilMessage (  )
```

Figure 124. SimpleSeller Rules

```
 Name: Simple Buyer Seller Protocol Price_Request Message Handler
Description: This rule is a message handling rule for a protcol.
The conditions for it were built up automatically based upon a
transition in a protocol.  Currently, there are no conditions
being added to make sure the sender is one of the agents in
the protocol.  These can be added manually if needed.
The protocol was based upon the Price_Request
 The description of the transition follows:
This transition represents a request that the Seller Role
fill out a PriceRequest PAC and return it.
The following list is the complete list of valid sending agents:
[Simple Buyer 2]
WHEN
( %incomingProtocolMessage.protocol EQUALS  "Simple Buyer Seller Protocol"  )
( %incomingProtocolMessage.contentType EQUALS
                com.reticular.agents.simpleBuyerSeller.PriceRequest  )
( %incomingProtocolMessage.performative EQUALS  "ask-one"  )
( %incomingProtocolMessage.ontology EQUALS  "Simple BuyerSeller Ontology"  )
( %incomingProtocolMessage.replyWith EQUALS  "price-request"  )
DO
ASSERT(  "SimpleBuyerSellerProtocol STATE: Request" )
ASSERT( %incomingProtocolMessage)
```

Figure 125. The Price_Request Message Handler Rule

load the correct agency. Once the agency is loaded, select the **Tools → Agency Viewer** menu item. This will load the tool with all of the agents from the agency (in this case the **SimpleBuyer2** and **SimpleSeller2** agents will be loaded).

The Agency Viewer uses an icon to represent each agent. Each agent has a default icon. These default icons can be easily replaced. All agent icons can be found in the `<agentBuilder-install-directory>/lib/icons/agentIcons/`. You can create your own icons using any kind of drawing or painting program. To make an icon available for use, place it in this directory. To select an icon, edit the agent icon field in the **Agent Properties Dialog.** There is a browser provided to help you change the icon.

Notice that icon labels use different colors to represent the state of the agent. As the agent's state changes from stopped, registered, or waiting to running you will see the state changes visually. SeeTable 14.

Table 14. Agency Viewer Colors

State	Color Code
Stopped	Red
Registered	Pink
Ready to Begin	Yellow
Running	Green
Pause	Cyan

The first step in running the agency is to click on the **Exec → Register Mode** menu item. This places the tool into registration mode. At this point, agents can join the agency. You can run the agents either locally or remotely (using the agency runtime option *agency-mode*) and they can register with the tool.

The Agency Viewer appears as an agent to the running agents; i.e., they communicate with the tool using KQML messages like all other agents. The major difference is that the Agency Viewer is a special controller type agent. After the register mode is entered, the agents must be started. If all the agents are running on the same computer (which is true in this case) then you can use the **Exec → Run All** command to start all the agents. This, in turn, causes an **Engine Options Dialog** to be display for every agent that is run. In this case, a dialog will be displayed for the **SimpleBuyer2** and **SimpleSeller2** agents. Click on the **OK** button, for each of the **Engine Option Dialog**s. After the agents are started and their con-

soles appear, you will notice that the agents are not running. This is expected because they are waiting for the "begin" command from the **Agency Viewer**. Click on **Exec → Begin** to start the agents.

At this point the agents icon's will turn green, indicating that the agents are running. There will be two lines drawn between the agents, these lines represent communication between the agents. Your **Agency Viewer** should resemble that shown in Figure 126.

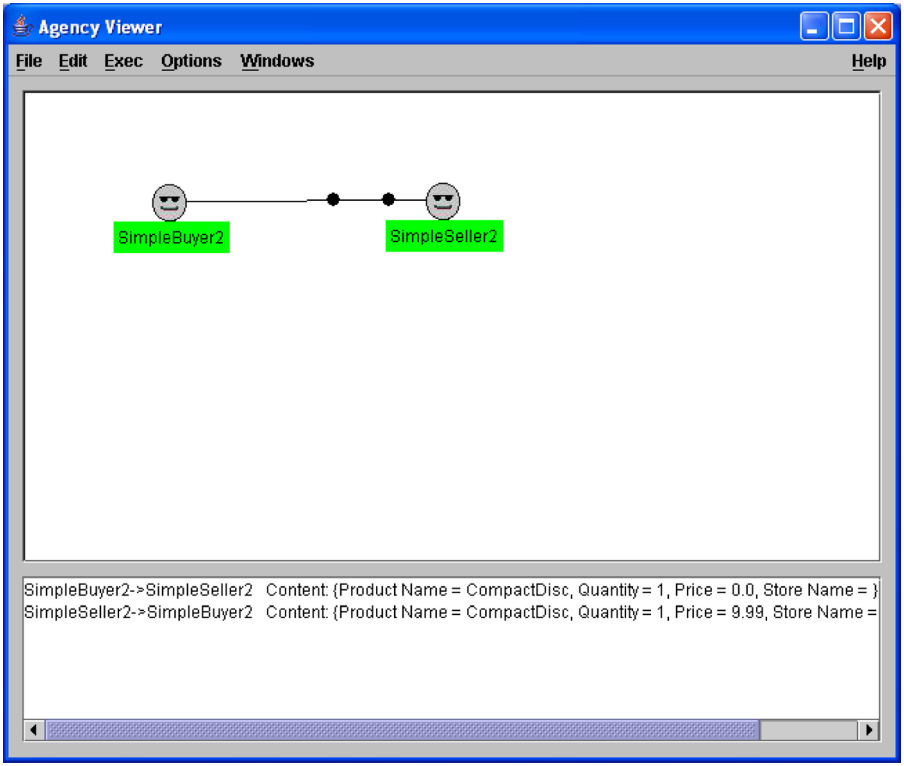

Figure 126. Agency Viewer with Buyer Seller Agents

If you look at each of the **Agent Engine Consoles** you will see the output generated by the rules you created.

The **SimpleBuyer2** console should look like Figure 127. The **SimpleSeller2** console should look like that in Figure 128. You should see a warning generated by **SimpleSeller2** in the error window in the agent console. This warning indicates that a modified mental state element has been retracted. Normally, this is very useful to the developer. This warning is typically generated when two rules are operating on the same object with one modifying the object and the other retracting it. This can often lead to errors in the control and logic of the agent. In this particular case, it is correct to retract the KqmlMessage object. The reason is that even though the object was modified, we are sending it in a message and will not need to reason with it in the future.

The Agency Viewer tool is useful for debugging agent communication during the development cycle. It provides a way to examine messages sent between agents and a way to save the results of the run. To examine the messages sent from the **SimpleBuyer2** to the **SimpleSeller2**, right-click on **SimpleBuyer2** icon and select the **Message History** popup menu item. This will display a dialog showing the messages sent and received by this agent. To examine a particular message, click on it in the **Received** or **Sent** lists. The **Message History** dialog is shown in Figure 129.

Examine the menu items in the Agency Viewer. Under the **File** menu, you will find items that will allow you to save the results of a *run (***Open Run***,* **Save Run***,* **Save Run As…**). (A run is a session in which all message traffic between agents is monitored and captured). This menu item is also used to turn the message log on or off (**Message Log**) using a check box next to the menu item.

You can use the **Edit** menu to edit the properties of the agents and agencies. Use the **Edit → Properties…** and **Edit → Agent Properties…** items.

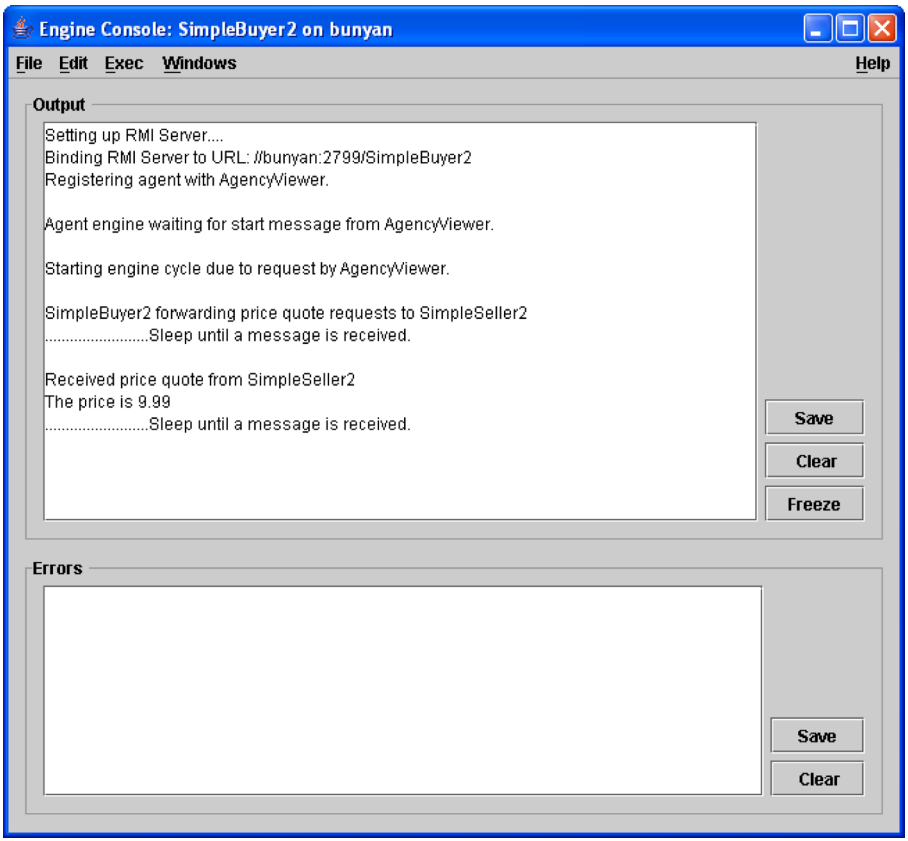

Figure 127. Simple Buyer Console

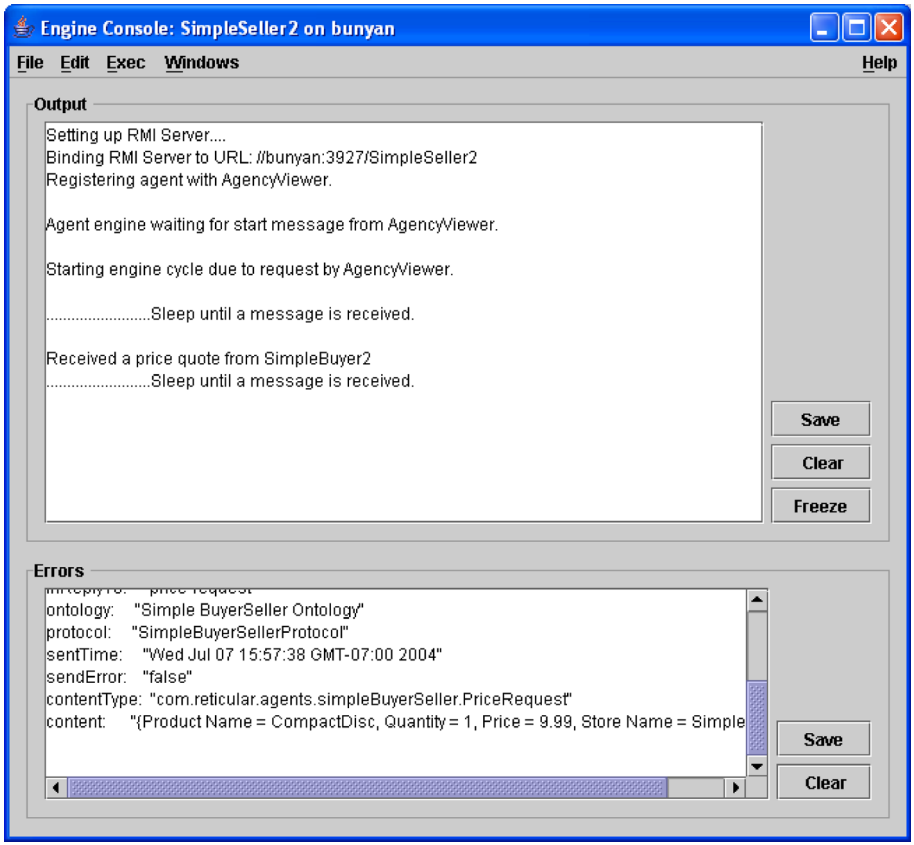

Figure 128. Simple Seller Console

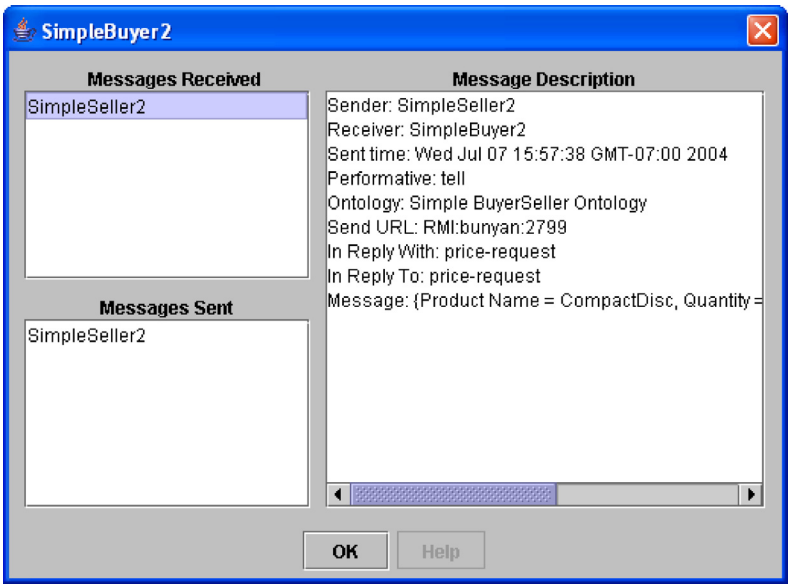

Figure 129. SimpleBuyer Messages

The **Exec** menu item controls execution of agents in the agency. This menu item includes provision for registering agents (**Register Mode**), running all agents (**Run All**), starting (**Begin**) and stopping (**Stop**) agents, pausing the agents (**Pause**), and resetting the agents (**Reset**). AgentBuilder allows you to control individual agents or all agents in an agency. Use the **Exec** menu for global control of the agents. You can control each agent individually by right-clicking on it. The popup menu provides controls for running, starting, stopping and pausing the agents.

The **Options** menu item allows you to turn on and off the agent message display. Do this using the check box in the **Options** → **Show Message** menu item. You can also use the **Options** menu to

examine the status of an agent (**Options → Agent Status)** and to change the message buffer size (**Options → Buffer Size…**).

You can also save runs, edit the agent and agency properties and specify the size of the message buffer. You can also disable the message trace window located at the bottom of the window.

Another useful and interesting feature of the tool is the pause/ unpause and reset feature. You can globally or individually use the pause feature to step through a protocol and examine the messages being sent, by whom and when. You can also reset the entire agency when necessary. Try clicking on **Exec → Reset**. This will rerun the agents without forcing you to restart them. Notice that the color of the agent icons changes.

When you are finished with a development session, select **Exec → Stop t**o stop all agents. Then choose the **File → Close** menu item. You will be asked whether or not you want to save the run. If you elect to save the run, then you can retrieve all of the message information at a later time.

Appendices

The purpose of this section is to demonstrate how to use AgentBuilder to construct several example agents by providing step-by-step instructions. We recommend that you work through these examples before starting your own agent development work. Prior to starting this section, read the previous overview of the AgentBuilder tools. If you need more information about the tools and their operation as you work through this section, refer to the remaining chapters in this User's Guide.

We have also included an **Example Project** and several example agnecies in the System Repository. The agents in this example project are the same as the ones we will develop in this guided, step-by-step introduction to AgentBuilder. You can examine these pre-constructed agents at your leisure. However, we strongly recommend that you work through the examples described in this section.

While all of the agents described in this section are simple, these step-by-step exercises for constructing them will provide you with the hand's-on experience and training you need to get started building your own agents. This section covers most of the processes involved in building agents including construction of object models, construction of basic agent behavioral rules, and execution and debugging your completed agent. This section shows you how to construct a number of related **Hello World** agents. Each successive agent has increased functionality and uses more advanced tools in the AgentBuilder toolkit.

In this step-by-step introduction you will build:

- **Example Agent 1** — agent with single rule that prints "Hello World" to the built-in console window.
- **Example Agent 2** — agent that prints 'Hello World' to the console once every 10 seconds, and variations on this agent that demonstrate other aspects of AgentBuilder.

- **Example Agent 3** — agent that demonstrates the incorporation of a PAC into rules and the run-time creation of objects.
- **Example Agent 4** — agent that utilizes a GUI-based PAC and demonstrates message passing between the user interface and an agent.
- **SimpleBuyer** — agent that utilizes a GUI-based PAC and demonstrates how a buyer can purchase a product from the least expensive store agent.
- **SimpleSeller** — agent that utilizes a GUI-based PAC and demonstrates how a store can respond to a buyer agent's request for a product price.

Note that this section is not intended to completely describe all the built-in functions, PACs, objects, tools, etc. in AgentBuilder, nor is it intended to provide a tutorial on rule-based programming. You should consult other sections of the User's Guide and other references to increase your understanding in these areas. This section provides a starting point for acquainting you with many of the features of AgentBuilder and shows you how to get started building agents with this toolkit.

Chapter 13

Running Agents Outside the AgentBuilder Environment

This chapter provides detailed information on executing agents outside the AgentBuilder environment. This chapter describes:

- Required directories and files
- Running agents in the Windows environment
- Running Agents int he UNIX environment
- Debugging Agencies

This chapter provides details for exporting and running agents outside the AgentBuilder development environment. One you have completed you agent development, you will likely want to move the agents to one or more machines for their actual execution. Remember, that once you have completed the development of an agent, you can easily run it in a stand-alone mode without requiring AgentBuilder.

You will need the following directories and files in order to execute the AgentBuilder agent runtime engine:

- JRE directory
- `lib/agentBuilder.jar`
- Engine batch file OR engine script file
- Agent's RADL file
- any class files being used in the Agent's PACs

The procedure for exporting agents for execution in a Microsoft Windows environment is slightly different from that required for execution in a UNIX environment. The following paragraphs provide instructions for exporting to each of these environments.

A. Running Agents in the Windows Environment

You must first create the directory structure that will contain your agent files. Now follow the steps described below.

Step 1. Create File Folder

Create the file folder where the new agent files will be copied to. From the Windows Explorer, select the hard drive or folder and choose **File → New → Folder**.

Step 2. Copy AgentBuilder JRE Directory

Copy the AgentBuilder JRE directory to the new agent directory. Again, from the Windows Explorer, select the folder where AgentBuilder is installed, right-click the Jre folder and select **Copy**. Now, right-click the new agent's folder and select **Paste**.

Step 3. Copy AgentBuilder lib Folder

Using the same method described above, copy the AgentBuilder lib folder and the `engine.bat` file, and paste them in the new agent's folder.

Step 4. Copy Agent RADL File(s)

Copy the RADL file and paste them in the new agent's folder. (If you plan on running more than one agent in the same JVM, make sure to copy over their RADL files as well). The default location for the RADL files is in the AgentBuilder's `Users/<username>/RADL` directory. Copy the RADL file over to the new agent's folder under the "radl" folder (You will need to create the new radl folder).

Step 5. Copy Class Files

The agent will need all class files it uses in it's PACs. You can copy the class files into the agent's lib directory. The directory hierarchy should be similar to that shown in Figure 130.

Step 6. Modify Engine Batch File

Modify the engine batch file to startup the agent automatically.

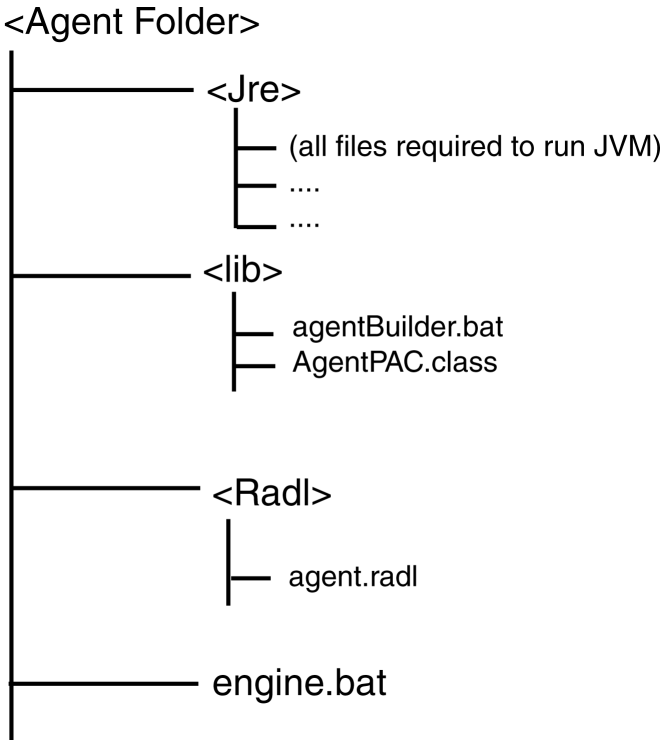

Figure 130. Windows Directory Structure

A. Edit the Batch File

Open the `engine.batch` file with the Windows Notepad application. Right-click the `engine.bat` file and select **Edit**.

B. Modify the Classpath

Add the agent's lib folder to the classpath. You will need to append the `;.\lib` to the end of the classpath.

C. Replace the Main Class File

Replace the main class file

`com.reticular.agentBuilder.agent.engine.EngineLauncher`

with `com.reticular.agentBuilder.agent.engine.AgentEngine` .

D. Append RADL File Name

Append the `radl` file name extension to the end of the line. (e.g. `Radl\Agent.radl`).

E. Add AgentEngine Options

Add any AgentEngine options to the right of the radl files. The following is an example of how the command line should look like after making the above changes. These AgentEngine options specify no display of the agent's console window, log all output to the file `AgentOutput.txt` and log all error messages to the file `AgentErrors.txt`.

Make sure this is on a single line:

```
.\jre\bin\java -ms5m -mx32m -classpath ".\lib\
agentBuilder.bat;.\lib" com.reticular.
agentBuilder.agent.engine.AgentEngine radl\Agent.radl -no-
console -no-system-out -no-system-err -o AgentOutput.txt -e
AgentErrors.txt
```

Step 7. Test the Agent

To test the agent and make sure it starts up, double click the modified batch file. You can now move the agent's folder to a new machine and run the agent.

B. Running Agents in the UNIX Environment

The first step in preparing your agents to run in the UNIX environment is to create the directory structure that will contain your files. Then follow the steps outlined below.

Step 1. Create Directory

Create the directory where the new agent files will be copied to. For example:

```
cd ~
mkdir agent
```

Step 2. Copy Directories and Files

Copy the following directories and files (located in the AgentBuilder directory) to the new agent directory: jre and lib directories. For example:

```
cp -r /usr/local/agentBuilder/jre ~/agent/
cp -r /usr/local/agentBuilder/lib ~/agent/
```

Step 3. Copy RADL File

We now need to copy the RADL file (If you plan on running more than one agent in the same JVM, make sure to copy over their RADL files as well). The default location for the radl files is in

```
.AgentBuilder/RADL
```

located in your home directory. Copy the RADL file to the new agent's radl directory (You will need to create the radl directory). For example:

```
mkdir ~/agent/radl
cp ~/.AgentBuilder/RADL/agent.radl ~/agent/radl
```

Step 4. Copy PAC Class Files

The agent will need all class files it uses in it's PACs. You can copy the class files into the agent's lib directory. For example:

```
cp ~/javaClasses/*.class ~/agent/lib
```

The directory hierarchy should be similar to that shown in Figure 131.

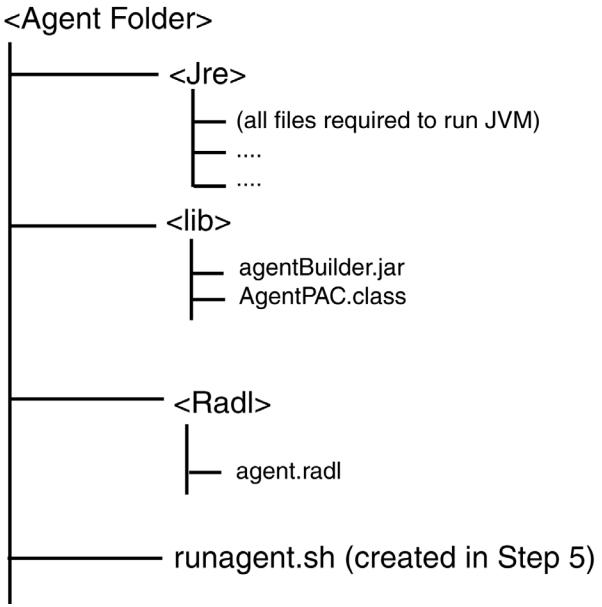

Figure 131. UNIX Directory Structure

Step 5 Create a Script File

Create a script file to run the agent. In this file, we set options needed for the agent to execute.

A. Create the Script File

Create the file `runagent.sh` under the agent directory. Open it with your favorite editor and add the command line that starts the agent. Figure 132 is an example of a runagent.sh script:

```
#!/bin/csh
#
# cd into the agent directory
cd /home/user/agent
# Run the agent's jvm
./jre/bin/java -Xms5m -Xmx32m -classpath ./lib/
agentBuilder.bat:./lib
com.reticular.agentBuilder.agent.engine.AgentEngine
radl\Agent.radl
-no-console -no-system-out -no-system-err -o AgentOutput.txt
-e AgentErrors.txt
```

Figure 132. UNIX Agent Script

For this script:

`./jre/bin/java` - Specifies the path to the java executable

`-Xms5m -Xmx32m` - Java options for the JVM

`-classpath ...` - The classpath for the agent's JVM

`com...AgentEngine` - Main class for starting the agent

`radl/Agent.radl` - The agent's RADL file

`-no-console` - AgentEngine option. Specifies do not display the console window.

`-no-system-out -no-system-err` - AgentEngine option. Specifies do not print any output or error messages to the console window.

`-o AgentOutput.txt` - AgentEngine option. Specifies log all output to file `AgentOutput.txt`

`-e AgentErrors.txt` - AgentEngine option. Specifies log all error output to file `AgentErrors.txt`

Step 6. Test the Agent Script

Test the agent script to make sure it starts correctly. You can now move the agent's directory to a new machine and run the agent there.

Appendix A. KQML Performatives

Basic Informative Performatives

```
tell
     :content <expression>
     :language <word>
     :ontology <word>
     :in-reply-to <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

Performatives of the type noted above indicate that the `:content` sentence is in the sender's VKB.

```
deny
     :content <performative>
     :language KQML
     :ontology <word>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

Performatives of this type indicate that the meaning of the embedded <performative> is not true of the sender. A deny of a deny cancels out.

```
untell
     :content <expression>
     :language <word>
     :ontology <word>
     :in-reply-to <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

A performative of this type is equivalent to a deny of a tell. Untell is weaker than telling the negation of the sentence. The sender may not have the negation in its VKB either. Also, the inclusion of the untell performative is obviously redundant, however this is preferred to deficiency.

Database Performatives

These performatives, INSERT, DELETE-ONE, etc. provide an ability for one agent to request another agent to insert or delete sentences in its VKB.

```
insert
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

The sender requests the receiver to add the `:content` sentence to its VKB. The performative can either fail or succeed. Possible errors and warning conditions include:

- content duplicates sentence already in VKB.
- content contradicts sentence already in VKB.
- sender is not authorized to INSERT content.

```
uninsert
             :sender      <word>
             :receiver    <word>
             :in-reply-to <word>
             :reply-with  <word>
             :language    <word>
             :ontology    <word>
             :content     <expression>
```

This performative is a request to reverse an *insert* that took place in the past by deleting the inserted expression. Performatives like *insert* and *delete* can only be used when an agent has advertised that it is going to accept them. Such an advertisement implies the acceptance of the corresponding *uninsert* and *undelete* messages. Although it is tempting to view *insert* and *delete* as complementary and use *delete* in the place of *uninsert*, and *insert* instead of *undelete*, it is preferable to use performatives of the *un-* variety because (a) an agent might advertise only an *insert* or only a *delete* for a particular `:content`, and (b) to emphasize that the intent of the *un-* performative is to reverse an action that has taken place rather than negate its effects. An *uninsert* can only be used after a corresponding *insert*.

```
delete-one
             :content <performative>
             :aspect <expression>
             :order {first | last | undefined}
             :language KQML
             :ontology <word>
```

```
:reply-with <expression>
:in-reply-to <expression>
:sender <word>
:receiver <word>
```

The sender requests the receiver`delete-one` sentence from its VKB which matches `:content`. Note that performatives of this type make most sense with languages that define schema variables.

The `:aspect` parameter describes the form of the desired reply. For the match of the deleted `:content` in the recipient's VKB, the reply will be the `:aspect` with all of its schema variables replaced by the values bound to the corresponding schema variables in the deleted sentence. The value of the `:aspect` parameter defaults to the value of the `:content` parameter. If the `:aspect` is `NIL`, then no response will be given for a successful deletion.

The optional `:order` parameter specifies whether the sentence to be deleted should be the first or last one found in the VKB. This will only make sense to some agents, e.g., Prolog-based ones. The default value for the `:order` parameter is "`undefined`." In addition, the performative can either fail or succeed. Possible errors and warning conditions are:

- no sentence matching content in VKB.
- content necessarily true in VKB.
- sender is not authorized to DELETE content.

```
delete-all
     :content <performative>
     :aspect <expression>
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

This performative is like `delete-one` except that the reply should be a collection of instantiated aspects corresponding to all deleted sentences matching the `:content`. Also, the performative can either fail or succeed. Possible errors and warning conditions are:

- no sentence matching content in VKB.
- all content necessarily true in VKB.
- sender is not authorized to `DELETE` content.

```
undelete
     :sender       <word>
     :receiver     <word>
     :in-reply-to  <word>
     :reply-with   <word>
     :language     <word>
     :ontology     <word>
     :content      <expression>
```

This performative is a request to reverse a *delete* that took place in the past by inserting the deleted expression(s). An *undelete* can only be used after a corresponding *delete-one* or *delete-all*. In either case, it undeletes whatever was deleted in the first place, assuming of course that the original *delete* action was executed successfully (i.e., no *error* or *sorry* was sent in response).

Basic Responses

```
error
      :in-reply-to <expression>
      :sender <word>
      :receiver <word>
      :comment <string>
      :code <integer>
```

A performative of this type indicates that the sender can not understand, or considers to be illegal, the message referenced by the `:in-reply-to` parameter. The `:code` parameter gives a numeric code to

classify the error type. The `:comment` parameter can be used to return a string, further describing how the sender considers the message to be ill-formed.

```
sorry
      :in-reply-to <expression>
      :sender <word>
      :receiver <word>
      :comment <string>
```

This type of performative indicates that the sender understands, but is not able to provide any (more) response(s) to, the message referenced by the `:in-reply-to` parameter. A performative of this type may be used in response to an `evaluate` or `ask-one` query when no other reply is appropriate. The optional `:comment` parameter can be used to pass a string which describes the situation leading to the refusal to provide a response or additional responses.

Basic Query Performatives

```
ask-if
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type is the same as `evaluate` except that the `:content` must be a sentence schema in the `:language`. In other words, the sender wishes to know if the `:content` matches any sentence in the recipient's VKB.

```
ask-one
     :content <expression>
     :aspect <expression>
     :language <word>
     :ontology <word>
```

```
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type is like an `ask-if` except that the `:aspect` parameter describes the form of the desired reply. For some match of the `:content` in the recipient's VKB, the reply will be the `:aspect` with all of its schema variables replaced by the values bound to the corresponding schema variables in `:content`. The value of the :aspect parameter defaults to the value of the `:conten`t parameter. Note that performatives of this type make most sense with languages that define schema variables.

```
ask-all
     :content <expression>
     :aspect <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type is like `ask-one` except that the reply should be a collection of instantiated aspects corresponding to all matches of the `:content` sentences on the recipient's VKB.

Multi-Response Query Performatives

```
stream-all
     :content <expression>
     :aspect <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type is like `ask-all` except that rather than replying with the collection of instantiated aspects, the responder should send a series of performatives. When taken together, these should identify the members of that collection.

```
eos
      :in-reply-to <expression>
      :sender <word>
      :receiver <word>
```

The "`end of stream`" performative indicates that the sequence of responses to an earlier multi-response message (e.g., `stream-all`) `:in-reply-to` has terminated successfully. No more responses will be sent.

Basic Effector Performatives

```
achieve
     :content <expression>
     :language <word>
     :ontology <word>
     :force <word>
     :sender <word>
     :receiver <word>
```

Performatives of this type are requests that the recipient try to make the sentence in :content true of the system .

```
unachieve
     :content <expression>
     :language <word>
     :ontology <word>
     :sender <word>
     :receiver <word>
```

A performative of this type is the same as a `deny` of an `achieve`.

Generator Performatives

The following performatives comprise a generator mechanism for the delivery of responses to a KQML performative. This mechanism allows an agent to explicitly retrieve responses in a series. This is especially useful when there are a large number of responses and the agent is not able to efficiently buffer incoming responses.

```
standby
     :content <performative>
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to take the `would-be` response(s) from the performative in `:content` and announce its readiness to accept requests for the responses.

```
ready
     :reply-with <expression>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender will answer requests for the responses to the performative contained in some performative with the `:in-reply-to` label. The `:reply-with` parameter is, in function, the returned generator.

```
next
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wishes to receive the next response from those promised by the performative identified by the `:in-reply-to` parameter.

```
rest
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wishes to receive the remaining responses, in a stream, from those promised by the `ready` performative identified by the `:in-reply-to` parameter. This performative does not have a `:reply-with` parameter because the `:in-reply-to` parameter of the next response should match the `:reply-with` parameter of the performative embedded in the original standby message.

```
discard
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender will issue no more replies to the ready performative identified by the `:in-reply-to` parameter.

Capability-Definition Performatives

```
advertise
     :content <performative>
     :language KQML
     :ontology <word>
     :force <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender is particularly suited to process the class of KQML performatives described by the `:content`

parameter. If the embedded performative is missing any parameters (defined for the embedded performative), then those parameters may take any otherwise legal values.

Notification Performatives

```
subscribe
     :content <performative>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wishes the recipient to tell it about future changes to what would be the response(s) to the KQML performative in the :content parameter.

Networking Performatives

```
register
     :name <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender can deliver performatives to the agent named by the :name parameter. This subsumes the case when the sender calls itself by this name.

```
unregister
     :name <word>
     :sender <word>
     :receiver <word>
```

This type is the same as a deny of a register.

```
forward
     :to <word>
     :from <word>
     :content <performative>
     :language KQML
     :ontology <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the `:to agent` to process the performative in the `:content` parameter as if it came from the `:from agent` directly. It is important that the `:to agent` receive the package, not just the performative, or it will think that the performative is from the next-to-last step in the path.

This will normally entail that the response(s) are also wrapped in forward(s), since the responder will want to deliver the response(s) to the requesting agent. Achieving this may involve the use of a package or other networking performatives. However, it is possible that agent `A` must use a package to send a performative to `B`, but `B` can send a performative to `A` directly.

Previously, we defined three levels of KQML syntax: the communication (package) layer, the message layer, and the content layer. The current approach is a proper generalization, since the layers arise from the embedding of performatives.

```
broadcast
     :from <word>
     :content <expression>
     :ontology <word>
     :language <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender would like the recipient to route the `broadcast` performative to each of its outgoing connections, unless the recipient has already received a `broadcast` performative with this `:reply-with`. This serves cycle detection.

```
transport-address
     :name <word>
     :content <expression>
     :language <word>
     :ontology <word>
```

The `transport-address` performative is a way to define an association between a symbolic name for a KQML agent and a transport address.

Facilitation Performatives

```
broker-one
     :content <expression>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to process the embedded performative through the help of a single agent that is particularly suited to processing the embedded performative. Presumably, such suitability was established using the `:advertise` performatives.

```
broker-all
     :content <expression>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :sender <word>
```

```
     :receiver <word>
```

This type is similar to `broker-one` except that the sender wants the recipient to enlist the help of all agents particularly suited to processing the embedded performative. The recipient of the `broker-all` replies with a list of all responses.

```
recommend-one
     :content <expression>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to reply with the name of a single agent that is particularly suited to processing the embedded performative.

```
recommend-all
     :content <expression>
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to reply with a list of names of agents that are particularly suited to processing the embedded performative.

```
recruit-one
     :from <word>
     :content <expression>
     :language KQML
     :ontology <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to forward the embedded performative to a single agent that is particularly suited to processing the embedded performative. This differs from `broker-one` because the recruited agent will forward its response directly to the original sender.

```
recruit-all
     :from <word>
     :content <expression>
     :ontology <word>
     :language KQML
     :sender <word>
     :receiver <word>
```

This type is similar to `recruit-one` except that the sender wants the recipient to forward the embedded performative to all agents particularly suited to processing the embedded performative. The recruited agents individually forward their responses to the original sender.

Appendix B. Bibliography

Cheong, F.-C. (1996). Internet Agents: Spiders, Wanderers, Brokers, and Bots. Indianapolis, IN: New Riders.

Finin, T., Fritzon, R., McKay, D., & McEntire, R. (1994/1994a). KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*, ACM Press.

Finin, T., Fritzson, R., McKay, D., & McEntire, R. (1994b). KQML - A Language and Protocol for Knowledge and Information Exchange (Technical Report No. CS-94-02). University of Maryland, Department of Computer Science.

Finin, T., Weber, J., Wiederhold, G., Genesereth, M., Fritzon, R., McGuire, J., Shapiro, S., McKay, D., Pelavin, R., & Beck, C. (1994c). Specification of the KQML Agent-Communication Language plus example agent policies and architectures (DRAFT Report), DARPA Knowledge Sharing Initiative External Interface Working Group.

FIPA Foundation for Intelligent Physical Agents, "FIPA 97 Specification Part 1 Agent Management," Specification Oct 10, 1997.

Foner, L. N. (1993). What's An Agent, Anyway? A Sociological Case Study. (Agents Memo 93-01), Massachusetts Institute of Technology.

Franklin, S., & Graesser, A. (1996). Is it an agent, or just a program?: A taxonomy for autonomous agents.

Giaratanno, J. and G. Riley (1989), Expert Systems PWS-Kent.

Gilbert, D., & et al (1996). The role of intelligent agents in the information infrastructure.

Hayes-Roth, B. (1995). An architecture for adaptive intelligent systems. Artificial Intelligence, 72, 329 - 365.

Labrou, Y. and T. Finin, (1994). "A semantics approach for KQML - a general purpose communication language for software agents," University of Maryland.

Labrou, Y., (1996). "Semantics for an agent communication language," in Computer Science and Electrical Engineering Department. Baltimore, MD: University of Maryland Graduate School, pp. 116.

Maes, P. (1995). Intelligent Software. Scientific American, 273(3).

Minsky, M. (1985). The Society of Mind. New York, NY: Simon and Schuster.

Newell, A. (1988). Putting It All Together. In D. Klahr & K. Kotovsky (Eds.), Complex Information Processing: The Impact of Herbert Simon. Hillsdale, NJ: Lawrence Erlbaum.

Nwana, H. S. (1996). Software Agents: An Overview. Knowledge Engineering Review.

Rich, E. (1983). Artificial Intelligence. New York: McGraw Hill.

Rumbaugh, J., *et al* (1991). Object-Oriented Modeling and Design, Englewood Cliffs, NJ: Prentice Hall.

Russell, S. J., & Norvig, P. (1995). Artificial Intelligence: A Modern Approach. Englewood Cliffs, NJ: Prentice Hall.

Shoham, Y. (1990). Agent-Oriented Programming (Technical Report No. TR STAN-CS-90-1335). Stanford University.

Shoham, Y. (1991). AGENT-0: A simple agent language and its interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, Vol II. (pp. 704 - 709). Anaheim, CA: MIT Press.

Shoham, Y. (1993). Agent-oriented programming. Artificial Intelligence, 60(1), 51 - 92.

Shoham, Y. (1995). CSLI Agent-oriented Programming Project: Applying software agents to software communication, integration, and HCI (CSLI

Web page). Stanford University, Center for the Study of Language and Information.

Smith, D. C., Cypher, A., & Spherer, J. (1994). KidSim: Programming agents without a programming language. Communications of the ACM, 37(7), 55 - 67.

Sowa, J., (1984). Conceptual Structures: Information Processing in Mind and Machine. Reading, MA: Addison-Wesley.

Thomas, S. R. (1993) PLACA, An Agent Oriented Programming Language. PhD Thesis, Stanford University.

Thomas, S. R. (1994). The PLACA Agent Programming Language. In M. J. Wooldridge & N. R. Jennings (Eds.), Lecture Notes in Artificial Intelligence (pp. 355 - 370). Berlin: Springer-Verlag.

White, J. E. (1995). Telescript Technology: Mobile Agents (White Paper). General Magic.

Wooldridge, M. J., & Jennings, N. R. (Ed.). (1995). Intelligent Agents: ECAI-94 Workshop on Agent Theories, Architectures, and Languages. Berlin: Springer-Verlag.

Index

Symbols

A

B

C

Index

D

E

F

G

H

I

Index

J

K

Index

L

M

N

O

P

Index

Q

R

S

Index

T

U

V

W