AGENTBUILDER®

An Integrated Toolkit for Constructing Intelligent Software Agents

Revision 1.3
Febrary 18, 1999

Reticular Systems, Inc.
4715 Viewridge Avenue,
Suite #200
San Diego, California 92123
(619) 279-9723

This documents describes products currently under development at Reticular Systems, Inc. Product functional and performance specifications are subject to change without notice. Contact Reticular Systems, Inc. to determine current and planned availability of products and available product features.

Reticular Systems, Inc.
4715 Viewridge Avenue, Suite #200
San Diego, CA 92123

Telephone: (619) 279-9723
FAX: (619) 279-9697
AgentBuilder Sales (800) 309-1342
email: info@reticular.com
web site: http://www.agentbuilder.com

List of Figures

List of Tables

SECTION 1

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 new books from an online store, find out about the latest song from a favorite musician, or monitor stock portfolios. Software agents can roam the Internet finding 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 implement-

ing software systems. Intelligent software agents provide a powerful problem-solving paradigm that is well-suited to developing complex enterprise applications.

An intelligent software agent is a high-level software abstraction. Building a complex system requires that the developer specify a solution in terms of these high-level abstractions. Implementation of these complex systems simply requires identifying the requisite agents and specifying their behaviors.

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 and this 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 that 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. Reticular Systems, Inc. has developed 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.

SECTION 2

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 for perceiving and affecting 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 research 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].

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].

Table 1 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

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 hype 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 1 depicts how these four types of agents utilize the capabilities described above.

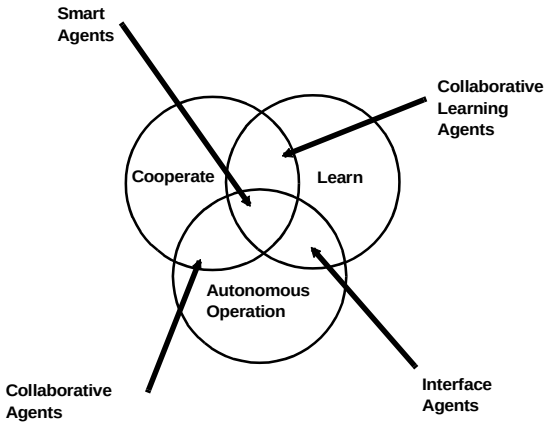


Figure 1. Agent Typology

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.

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 aids, 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 paper 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.

Agents, Objects, and Software Engineering

There is some confusion about the relationship between agents, object, beans, and other software components. Indeed, some argue that "an agent is just an object." While this is true, it is not a particularly useful observation. To argue that an agent is just an object is equivalent to arguing that you are just a collection of molecules. This misses the point that you are a complex entity with unique characteristics and behaviors.

An agent is a unique form of software *abstraction*. Abstraction means focusing on the essential, inherent aspects of an entity and ignoring its incidental properties. This means considering what an entity *is* and *does* rather than how it is implemented (Rumbaugh, 1991). For a software agent, this means focusing on the behaviors of the agent. Agents are unique abstractions because of their *inherent* capabilities. These inherent capabilities (which we've discussed previously) include:

- communicativity - agents inherently know how to communicate with other entities and other agents
- control - agents embody control; i.e., the behavior of the agent determines the sequence of actions to be performed.

- decision making - agents can draw inferences and make decisions
- autonomous operation - agents can operate without external assistance
- persistence - agents maintain information about their own state

Agents are a *high-level* abstraction. This means that we are focusing on the *behaviors* of complex entities rather than the low-level objects that these entities manipulate and reason about[1]. An agent's behavior is the response of the agent to changes in the agent's environment and changes in the agent's own internal mental state. Focusing on these behaviors is particularly important in building complex, distributed problem-solving systems. Analysis of behaviors reveals the complex interrelationships and interactions between subsystems.

Agents address many other problems that software developers must solve in building complex systems. In addition to providing mechanisms for high-level abstraction agents provide:

Encapsulation. Encapsulation or information hiding consists of separating the external aspects of an entity from the internal implementation details of the entity. Agents completely hide their internal mechanisms and external entities affect the agents behavior only by sending well-defined messages to the agent.

Modularity and Reusability. Agents are inherently modular. They operate autonomously and can thus easily be moved, repackaged or reused. Agents can be easily replicated. If the agent's behavioral program is separated from the agent's execution vehicle (the agent engine) then it is very easy to build new agents with different behaviors by simply creating a new agent program.

Concurrency. Since each agent inherently operates autonomously, makes its own decisions and maintains its own state information, it is easy to implement concurrent processes using multiple agents.

Distributed Operations. Agents provide an ideal mechanism for implementing distributed computing systems. Each agent has the inherent capability to stand alone and perform it processing task and communicate with other agents involved in the problem-solving process.

1. This doesn't remove the requirement to code and implement these low-level objects. However, it does delay the implementation of these objects until all of the complexity of the objects and their relationships is fully understood and completely characterized.

Summary

Agents provide good solutions to many of the problems software developer face in building complex systems. Agents provide high-levels of abstraction, encapsulation and information hiding, and support concurrent and distributed computation. Agents are not an appropriate implementation technique for all applications and developers need to take care in deciding to implement an agent-based solution. However, for a wide variety of applications and problem domains, agents provide an ideal system implementation paradigm.

SECTION 3

Agent Applications

New technologies are required for building these complex systems. Intelligent software agents provide a powerful problem-solving abstraction for implementing complex information processing systems.

Building Complex Information Systems

Unfortunately, most existing technologies are inadequate for building large, complex, distributed applications and systems. New tools such as Reticular's Agent-Builder are required for building these complex systems. Intelligent software agents provide a powerful problem-solving abstraction for implementing complex information-processing systems.

However, designers of complex information-processing applications must decide whether intelligent agents are the best approach to implementing a new system. Although, software agents are a powerful abstraction, they are not appropriate for all kinds of applications. The system designer should use agents only in those applications where they can provide significant benefit. An agent-based solution makes the most sense when the application can exploit the inherent strengths and capabilities of the basic agent architecture. That is, agents should be used in applications where one or more of the following conditions are satisfied:

- the application must operate autonomously
- the application must communicate with other software or human agents
- the application must monitor the state of its environment and make decisions about its own activities based on the state of that environment

Autonomous Operation. Often a software designer must build a system that requires little or no human intervention and is intended for unmanned operation. Sometimes this kind of autonomous operation can be obtained using simple scripts. However, solutions based on simple scripts often do not have the powerful reasoning or inferencing capabilities required in complex applications. An intelligent agent has the built-in reasoning capability required for automating complex tasks.

Communicating. Many modern application programs must be able to communicate with other, remote application programs; many of these programs communicate using the Internet. Therefore, software developers need effective ways of quickly and easily providing a communications capability for an application. Intelligent agents are inherently communicative; i.e, they know how to communicate with other agents, the user, or other kinds of programs. Thus, it is much easier to build an application that requires extensive communications using intelligent agents. Since the agents already know how to communicate, the developer is freed from developing extensive communications software and protocols.

Sensing the environment. Often, an application program needs to make decisions based on the state of its operating environment. That is, the application must be able to sense the environment; make an assessment of that environment; select an appropriate course of action based on the plans, goals and intentions of the application; and then execute the necessary actions. An intelligent software agent provides a reasoning mechanism that can monitor the environment, draw inferences, manipulate symbols, and determine an appropriate course of action for a given situation. Thus, application programs that need to make sense of the environment and perform complex activities can readily be implemented using intelligent agents.

An agent architecture should be selected such that each agent can be specialized for performing specific tasks. Then networks or communities of agents can be used to communicate with each other and cooperatively solve complex problems.

Agent Applications

Intelligent software agents can be used in a variety of applications. Intelligent agents can be used in business applications in the manufacturing, financial, travel, retail and e-commerce industries, as well as sales force productivity, help desk, and other applications. In addition, intelligent agents can be a key element in implementing systems for finding and using information on the Internet. The following paragraphs describe two applications using multiple intelligent agents. The first

Intelligent Agents can be used in business applications in the manufacturing, financial, travel, retail and e-commerce industries, as well as sales force productivity, help desk, and other applications. In addition, intelligent agents can be a key element in implementing systems for finding information on and using the Internet.

application uses multiple agents distributed across a wide area network and multiple organizations. The second application uses multiple agents to implement an intelligent information-seeking program for use on the Internet.

Agents in Enterprise Applications

Figure 2 illustrates a typical banking application where communicating agents are used to perform the various roles involved in processing a small business loan. In this example of a distributed application, each participant in the loan approval process is either augmented by or replaced with an intelligent agent.

The domain for this particular application consists of a branch bank, a main bank, a loan underwriter and a credit reporting agency. Note that the participants can be geographically separated from each other. Participants communicate with each other over a data network (for example, the Internet). The customer (a small business owner) typically deals with a branch bank; this is where the loan approval pro-

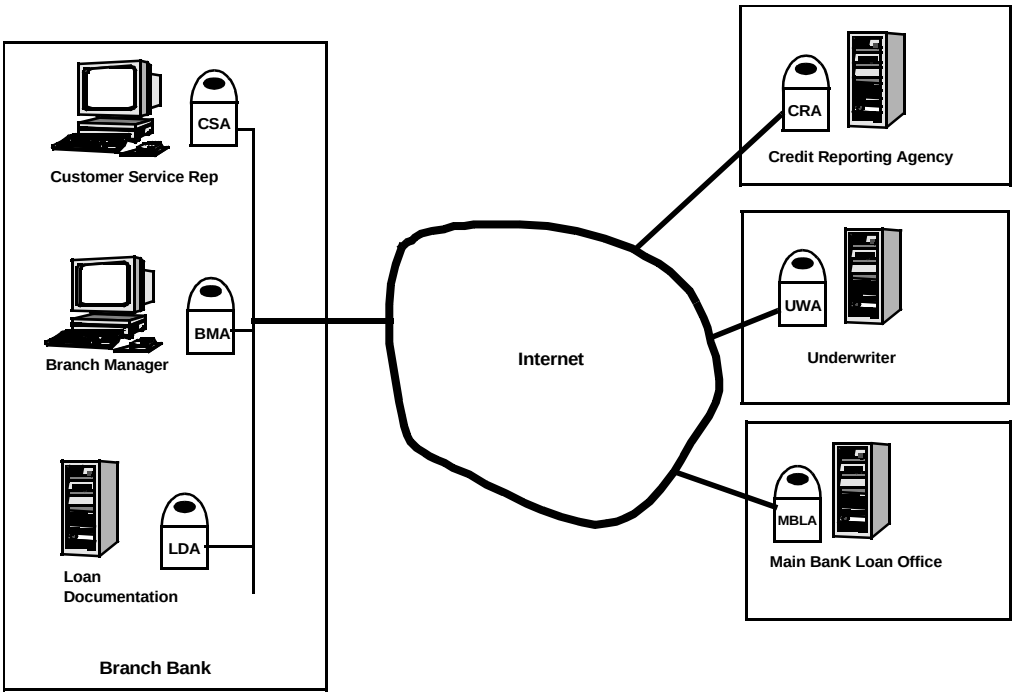


Figure 2. Intelligent Software Agents Processing Loan Applications

cess is initiated. The customer works with a customer service representative at the branch bank. This representative collects loan information and credit information from the customer and creates a loan application. The completed application is then forwarded to the branch manager for preliminary review. The manager then contacts a credit reporting agency to obtain a credit report on the prospective borrower. The credit report is returned to the branch bank where the branch manager reviews the report and application. The branch manager may make a preliminary approval of the package and then forward it to the main bank for review. If the loan is tentatively approved by the main bank, the branch manager sends the loan package to a loan underwriter for review. Depending on the underwriters response, the branch manager may grant preliminary approval for the loan and then send it to the main bank for final approval. After the main bank notifies the branch, the customer service representative contacts the customer and makes provision for funding the loan.

All of the participants in this loan processing scenario can benefit from the use of agent technology. An intelligent interface agent in the Customer Service Agent (CSA) can help the customer service representative collect all needed information from the customer and even help with a preliminary screening of the loan request. Further, the CSA can transmit the information in the loan application to the Branch Manager Agent (BMA). The branch manager's agent can review the loan application using the same business process rules as the human manager. An agent at the main bank (the Main Bank Loan Agent or MBLA) can communicate and share information with the branch manager's agent.

Note that some of the agents involved in this process don't belong to the bank. For example, the credit reporting agency can utilize a Credit Reporting Agent (CRA) to research credit history in response to a request from a remote agent (the branch manager's agent). The loan underwriter can utilize an Underwriter Agent (UWA) to communicate with the bank and control the transfer of loan application information.

In the past, the bank might have faxed or mailed documents to the loan underwriter or the credit reporting agency; now, intelligent software agents can automatically send the required information. A Loan Documentation Agent (LDA) capable of archiving and retrieving documents can be used to store the loan documents in either the main or branch bank. Further, software agents in the branch bank, main bank, credit agency and loan underwriter can utilize their own business rules for making decisions about a loan application. This example shows that many of the steps in the loan approval process can be automated by using a collection of intelli-

gent software agents. The flow of paper both within and between organizations is reduced and the bank is able to significantly speed up the loan approval process.

Using Agents in a Desktop Application

Intelligent agents are not restricted to use in enterprise applications. It is possible to construct high-performance desktop applications using intelligent agents for performing the various functions of a desktop system. Figure 3 illustrates the architecture for a intelligent desktop assistant that is designed to help users find and use information from the Internet.

This example shows four separate agents with each agent specialized to a particular task used to create an intelligent application. An *interface* agent that handles all interaction with the user. A *monitor* agent monitors Internet Web sites of interest to the user and then informs the user (through the interface agent) when new information appears on one of its sites of interest. A *domain* agent is an intelligent agent with a large amount of knowledge about the user's area of interest (e.g., a stock market domain agent). Finally, a *search/evaluation* agent specializes in locating and evaluating information at remote Web sites and determining whether this information satisfies the needs of the user. The monitor and search/evaluation agents are examples of the information agent defined previously.

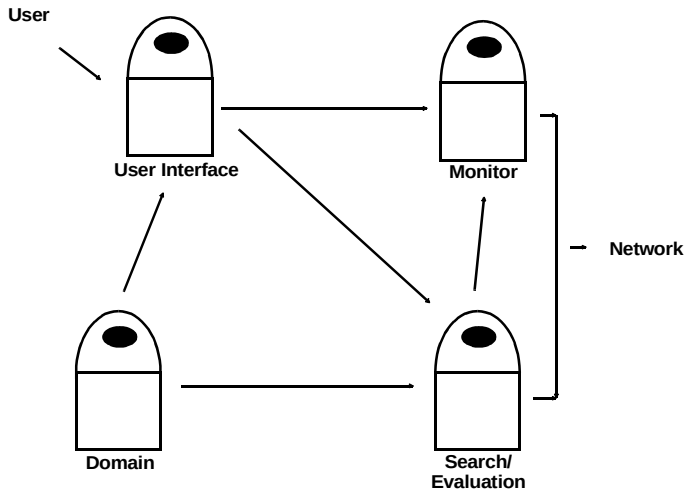
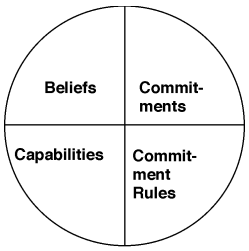


Figure 3. Using Intelligent Agents in a Desktop Application

This example shows how using the intelligent agent abstraction facilitates good system design. Each agent is designed to be an expert in performing some particular function (interface, domain knowledge expert, search, etc.). Further, each agent communicates with the other agents, the user, and the communications network using well-defined messages. In this example, all of the agents will execute on the same platform and do not need to communicate with each other over a network. Using intelligent agents as a high-level software abstraction significantly simplifies the software development process.

SECTION 4

Intelligent Agent Architectures



Shoham's Mental Models

Background: Mentalistic Agents

This section provides a detailed description of an intelligent agent implementation. Reticular Systems has been conducting research and development of intelligent agents for a number of years. The motivation for our 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

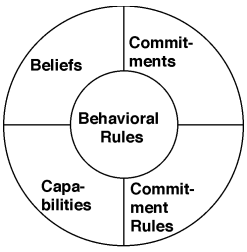
Shoham describes two primitive modalities of mental state – belief and commitment. Commitment is treated as a decision to act rather than a decision to pursue a goal. A strong condition is placed on belief; namely that agents have perfect memory of, and faith in, their beliefs and only relinquish a belief 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.

Reticular Agent Mental Models



Reticular's Mental Model

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. Reticular has further extended Shoham's and Thomas' work and developed a new agent programming language. This object-oriented language is called the Reticular Agent Definition Language (RADL). AgentBuilder provides graphical tools for creating RADL programs that execute in Reticular's run-time system. The following paragraphs describe Reticular's 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. All of the beliefs in the agent's mental model are instantiations of Java classes.

Mental model integrity is improved because beliefs 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 beliefs 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 which 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 online. 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 inter-

action 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 particular 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

Behavioral rules determine the course of action an agent takes at every point in the agent's execution.

In Shoham's model, all actions were performed only as the result of commitments. Reticular 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 environment, 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 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)
```

```
 NAME “Greenlight Move Forward Rule”
 WHEN
     ?KQMLMessage.Performative EQUALS TELL
     ?KQMLMessage.Sender EQUALS “stoplight-agent”
     ?KQMLMessage.Content EQUALS String
     ?KQMLMessage.Ontology EQUALS “Stoplight”
 IF
     ?KQMLMessage.Content EQUALS “stoplight-green”
     currentMotion.Status EQUALS stoppedAtRedLight
     currentLocation EQUALS nextIntersection
     NOT (currentLocation EQUALS destination)
     FOR_ALL (?BlockedIntersection,
           NOT(?BlockedIntersection.Location EQUALS currentLoca-
tion))
 THEN
     DO (Go(trafficSpeed))
     DO ($nextIntersection =  getNextIntersection(currentLoca-
tion,
                                        currentMotion.Direction))
     ASSERT (SET_VALUE_OF currentMotion.Status TO moving)
     ASSERT (SET_VALUE_OF nextIntersection TO $nextIntersection)
     SEND (performative = REPLY, receiver = “stoplight-agent”,
            content = “acknowledged”,
            in-reply-to = ?KQMLMessage.Reply-with)
```

Figure 4. The `Greenlight Move Forward` Rule

The following paragraphs provide an example and description of behavioral rules. The example agent is a chauffeur agent which is used to control an automobile in a simulated city environment. Several beliefs, belief instances and a rule are shown below. The behavioral rule is one of several rules used to control the car in a simulated grid of streets with traffic lights at intersections. The behavioral rule named `Greenlight Move Forward` is used to cause the car to begin moving forward after it has stopped for a red light. The `WHEN` portion is used for testing an incoming message from another agent and the `IF` portion is used to test conditions in the mental model. The rule is activated only if all conditions in the `WHEN` and `IF` portions are satisfied. If the rule is activated, the `THEN` portion makes the appropriate mental changes and executes the private and communicative actions. The `Greenlight Move Forward` rule is shown in Figure 4.

This example rule shows the object-oriented nature of RADL. Variables (prefixed with `?` or `$`) in the rule bind to belief objects in the mental model, then the objects or their subobjects can be used in comparisons. Subobjects within belief objects are accessed by name, following a dot used as a separator. For example, the `?BlockedIntersection` variable will bind to any belief object of the `Blocked-Intersection` type, and `?BlockedIntersection.Location` will access the `Location` subobject within the `BlockedIntersection` object. And subobjects of subobjects can also be accessed, using the same notation: `?BlockedInter-section.Location.Street` would extract the street number of the location of a blocked intersection. Comparisons between objects can be made at the object level or at any subobject level, as long as the objects are comparable. A pattern such as

```
?BlockedIntersection.Location EQUALS currentLocation
```

is evaluated using the equality comparison method defined for `Location` objects. The developer can specify patterns at this relatively high level of abstraction; the run-time system converts the RADL pattern into the `Location` equality test, which performs all the low-level comparisons needed to determine equality between two `Location` objects.

Several beliefs in the `Greenlight Move Forward` rule are shown below. The `Location` is composed of two `Streets`. The `Motion` is composed of `Status`, `Direction` and `Speed` fields. The `BlockedIntersection` contains a `Loca-tion`. The remaining beliefs are composed of primitive types. `Location`, `BlockedIntersection` and `Motion` are examples of beliefs that are composed of other beliefs and are shown in Figure 5 below.

Figure 6 illustrates the named belief instances that exist in the mental model. These are instances of the types defined above.

```
Location            (Street, Street)
BlockedIntersection (Location)
Motion              (Status, Direction, Speed)
Speed               (Integer)
Status              (String)
Street              (String)
Direction           (String)
```

Figure 5. Beliefs for Chauffeur Agent

```
nextIntersection    (Location)
currentLocation     (Location)
destination         (Location)
currentMotion       (Motion)
trafficSpeed        (Speed)
```

Figure 6. Named Belief Instances for Chauffeur Agent

The message conditions in the `WHEN` portion are tested against the new messages that are sent to the agent by other agents. The `?KQMLMessage` variable in these conditions is a variable of type `KQMLMessage`; it can bind to instances of the `KQMLMessage` belief. (`KQMLMessage` is predefined and represents all messages received from other agents. KQML is described later in this section.) For each new message, the variable will be bound to the new message and the message conditions will be evaluated. If all message conditions evaluate to true then the evaluation continues with the mental conditions. In this example rule, the message conditions specify that the message must be from a `stoplight-agent`; the KQML command is `TELL`; the message content type is a string; and the expected ontology is the `Stoplight` ontology. The `Content` message condition tests the *type* of the information in the `Content` field of the message, and not the information itself. Any testing of the information in the `Content` field is done in the mental conditions portion.

Next, the `IF` portion evaluates mental conditions by comparing the mental condition patterns against the message contents and the agent's beliefs, commitments, capabilities, and intentions. In this example rule, the first mental condition specifies that the contents of the message must equal the string `stoplight-green` and the second mental condition specifies that the `Status` field of the `currentMotion` belief must be `stoppedAtRedLight`. The third mental condition specifies that the `currentLocation` belief must have the same values as the `nextIntersection` belief, which means the agent believes the car is at that particular intersection. The fourth condition uses a `NOT`, and specifies that the `currentLocation` must not be the same as the `destination`. The last condition uses a variable of type `BlockedIntersection`, which will match against any `BlockedIntersection` belief instance in the belief base. For the last condition to be satisfied the `currentLocation` must not be the same as the location contained in any `BlockedIntersection` belief.

If any condition is not satisfied then the rule is not applicable to the situation. If, however, all conditions are satisfied, then the rule will be activated and the right-

hand-side (RHS) will be executed. In this example rule, the RHS causes the car to begin moving.

The RHS of the rule is composed of three fundamental types of statements: private actions (`DO`), mental changes (`ASSERT` or `RETRACT`), and message transmission actions (`SEND`). These three statement types provide the agent with the ability to modify the agent's environment, update the agent's mental model, and communicate with other agents. When the RHS of a rule is executed, the actions are performed first and their return values are available for use in mental changes.

The first line of the `THEN` portion is an action statement which causes the private action `Go` to be performed using the value of the `trafficSpeed` belief as a parameter. This action causes the car to move at the speed of the surrounding traffic, according to the agent's belief (perhaps acquired from a speed sensor). The second action, `getNextIntersection`, is a private action the agent uses to determine which intersection will be reached next, given the current location and current direction of travel. This action returns a value which is stored in the `$nextIntersection` variable, which can then be used in an assertion.

The first `ASSERT` statement updates the value of the `Status` field in the `currentMotion` belief to `moving`. The second `ASSERT` statement updates the `nextIntersection` belief with the value returned from the `getNextIntersection` private action. Finally, the last RHS statement builds and sends a reply to the message received from the `stoplight-agent`. The reply is an acknowledgment that the chauffeur agent received the `stoplight-green` message and understood its content.

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 7 illustrates Reticular's 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 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

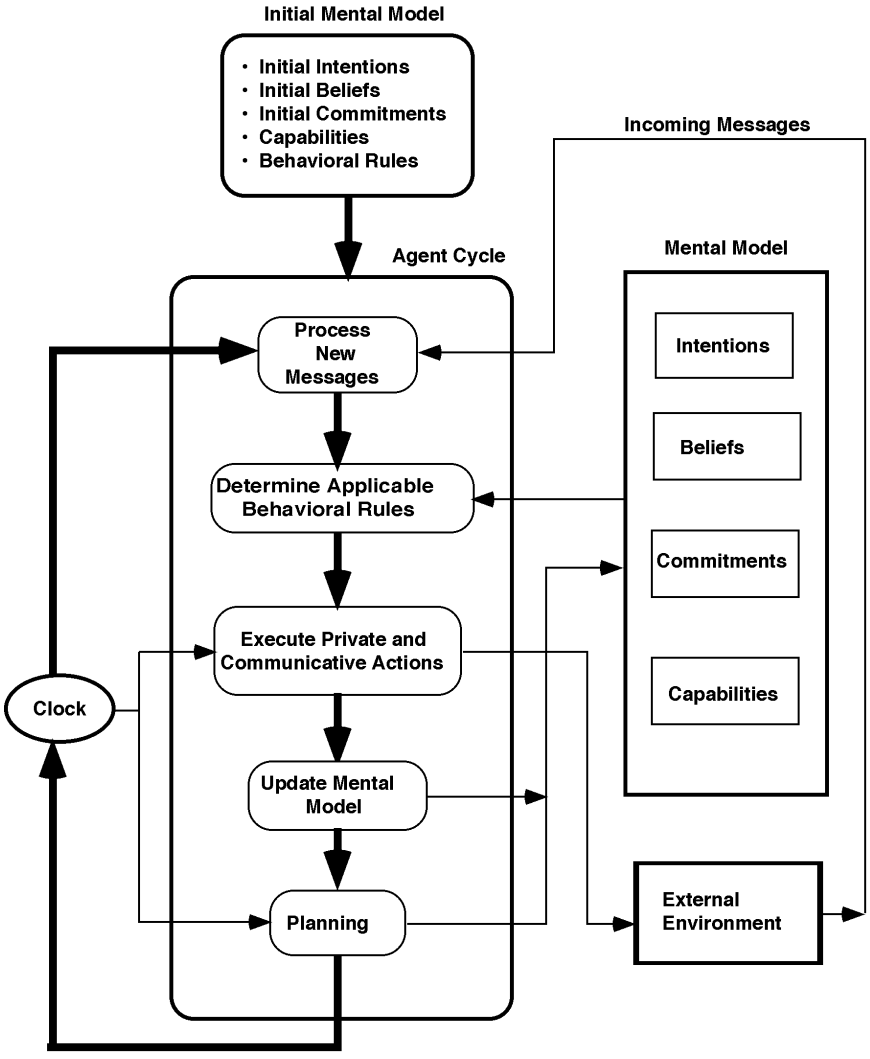


Figure 7. Agent Execution Process

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. Table 2 provides a listing of the defined performatives for KQML

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 who also "speaks" KQML. Thus, AgentBuilder agents can communicate with existing

Table 2. 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
sorry	S understands R's message but cannot provide a more informative reply

Table 2. Summary of Reserved Performatives

Name	Meaning for Sender S and Recipient R with Virtual Knowledge Base (VKB)
`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

agents or agents constructed using other methods, tools and architectures other than those supported by AgentBuilder.

SECTION 5

Agent Development Process

The Process

As noted in the previous discussion, specifying an agent's mental model requires defining its initial beliefs, initial commitments, 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.[1]

1. This section describes the overall capability of AgentBuilder. Not all of the capabilities described herein are available in all versions of AgentBuilder. Note also, that some capabilities may not be present in existing versions of the product but will be available in future versions. Check the latest AgentBuilder product brochures for a detailed description of the capabilities and features of the currently shipping version of AgentBuilder.

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 push button in a graphical user interface).

Agent-oriented programming simply requires that the programmer define the roles and functions of various agents and then specify each agents behavior

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. After this high-level analysis and design is complete, standard object-oriented analysis and design techniques can be used to develop the agents and the lower-level objects that the agent's manipulate and reason about.

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 (Reticular 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
- defining agent roles and communications protocols

- importing Java code and using it for implementing agent user interfaces and accessory classes
- specifying agent behavior
- executing the agents
- 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 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.

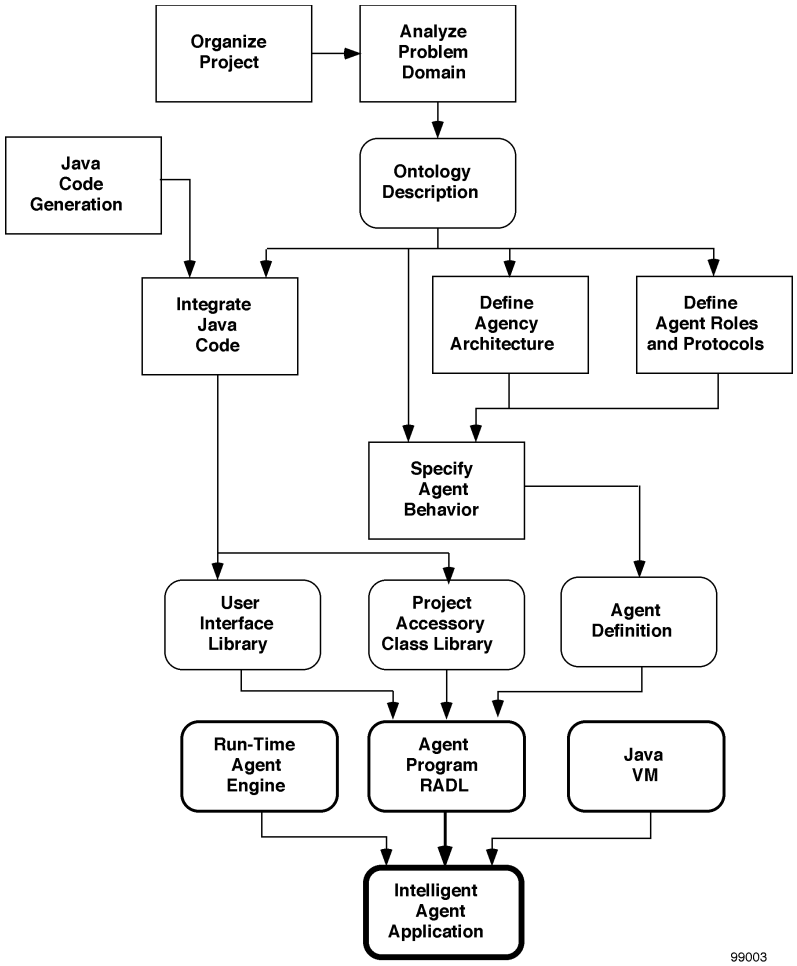


Figure 8. Constructing Intelligent Agents

Define Agent Roles and Communications Protocols

A *role* defines the characteristics or expected social behavior of an agent. For example, in an e-commerce application, an agent might take on the role of buyer or seller. Often, agents take on multiple roles. For example a broker agent might take on the roles of both buyer and seller. It is also important to separate the physical implementation of an agent from the roles of that agent. This provides significant flexibility in designing and implementing an agent-based system. AgentBuilder provides a role editor for defining roles and assigning roles to a specific agent.

After identifying the agents and their roles, the agent developer defines the inter-agent communication protocols. AgentBuilder provides a Protocol Editor 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 toolkit aids the user in developing the requisite code for specifying agent behavior. For example, AgentBuilder Pro automatically converts the agent protocols (defined using the Protocol Editor) into skeletal behavior rules that implement the agents conversational protocol.

The AgentBuilder toolkit produces an agent definition file that contains a detailed specification of the agent's initial mental model and behavior. This file fully specifies agent activities and behavior and is executed by the agent engine. This file is a flat text file called a RADL file (Reticular's Agent Definition Language).

Import Java Code for User Interfaces and Accessory Classes

AgentBuilder makes it easy to add a user interface to an agent. While many agents may not require interaction with a user, in many cases it is desirable to add a graphical interface to control agent operations, display agent status, and enter and display data. You can use Sun's JDK or any of a variety of third party development kits for creating your user interface.

In many cases it is desirable to have the agent execute procedural code. Project Accessory classes (PACs) are user-defined classes that the developer can use for this processing. Developers may use this code to implement a variety of different functions and operations. Reticular provides a number of PACs for performing common functions such as FTP, HTTP, e-mail, NNTP and document analysis. PACs can be coded in Java or you can also utilize C and C++ code through the Java Native Interface (JNI). PACs provide an ideal mechanism for allowing agents to utilize legacy code.

Create Agent Application

The final step in the agent construction process consists of loading the agent program into Reticular's 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. A Reticular agent is composed of the agent program and the run-time engine. The Run-Time Engine is described in the next section of this paper.

Agent and Agency Debugging

AgentBuilder provides tools that support all phases of the software agent development process. AgentBuilder provides an environment for integrating agents to form agencies and then controlling the execution of the agents. 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 the agent's interactions is required. AgentBuilder allows the developer to examine the contents of all messages transmitted between agents and start, stop and pause agent execution.

SECTION 6

Toolkit and Run-Time System

Introduction to AgentBuilder

All components of the AgentBuilder Toolkit and the Run-Time System are implemented in Java. AgentBuilder agents are simply Java programs.

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[1].

All components of both the AgentBuilder Toolkit and the Run-Time System are implemented in Java. This means that agent development can be accomplished on

1. This section describes the overall capability of AgentBuilder. Not all of the capabilities described herein are available in all versions of AgentBuilder. Note also, that some capabilities may not be present in existing versions of the product but will be available in future versions. Check the latest AgentBuilder product brochures for a detailed description of the capabilities and features of the currently shipping version of AgentBuilder.

any platform that supports Java and has a Java development environment. Likewise, the agents created with the AgentBuilder toolkit are Java programs so they can be executed on any Java virtual machine. Software developers can create powerful intelligent agents in Java that execute on a wide variety of computer platforms. The toolkit and the run-time system are described in the following paragraphs. Figure 9 shows the major components of AgentBuilder and their relation to each other.

AgentBuilder Toolkit

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

Project Control Tools

The Project Control Tools are provided to help the agent developer manage the overall agent development process. These tools include the Project Manager and the Project Repository Manager.

The Project Manager allows the developer to identify and manage multiple agent development efforts. For example, a developer can concurrently develop a travel reservations system and a banking application. Although these are very different applications with different agents and agent architectures, the developer needs a way to share information between these two projects. The project manager allows the developer to assign particular agents and agencies to a project and to share information between projects.

All information entered into the various AgentBuilder tools is maintained in a persistent store called the AgentBuilder Repository. Tools are provided for accessing and modifying this repository. The tools provide the capability to connect and utilize any repository in accordance with defined access permissions. The Project Repository Manager controls access to all of the data created using the AgentBuilder toolkit. This data includes domain knowledge, agent definitions, agency specifications, etc. These tools allow information to be stored and used by multiple projects.

Figure 10 illustrates the control panel for the Project Manager. Note that the panel provides a convenient display of all projects, their associated agencies, and agents.

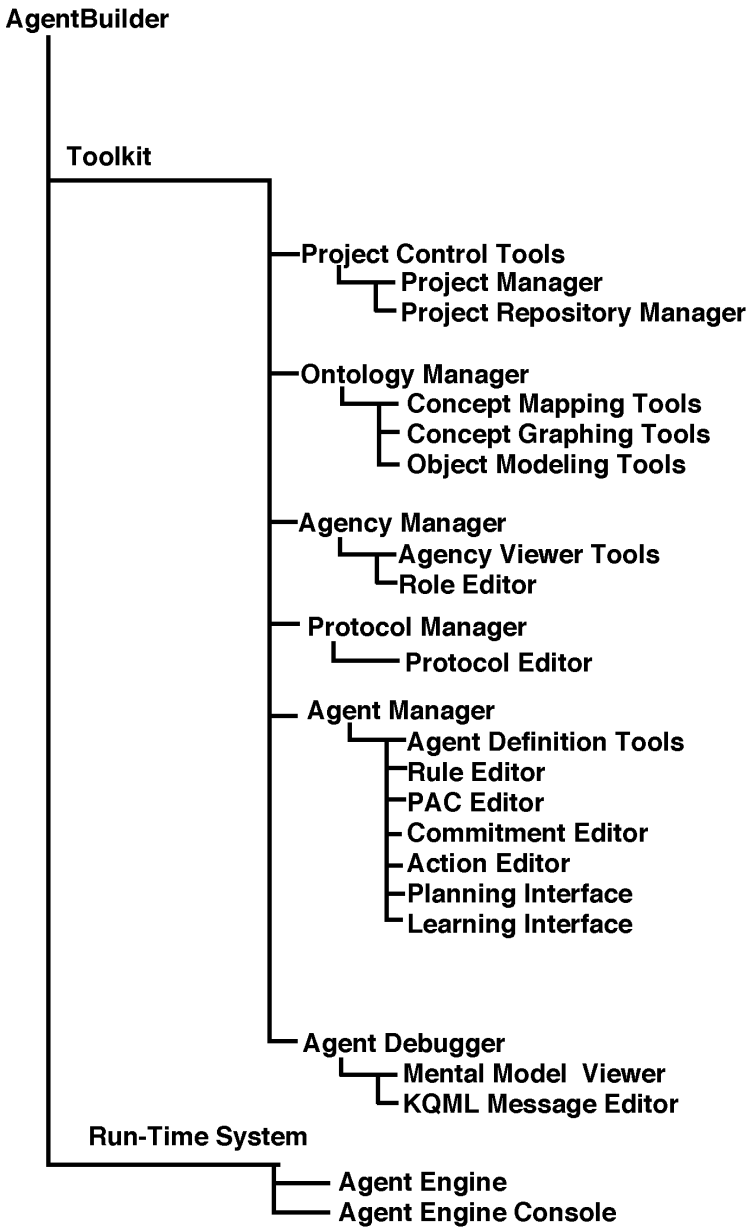


Figure 9. AgentBuilder Components

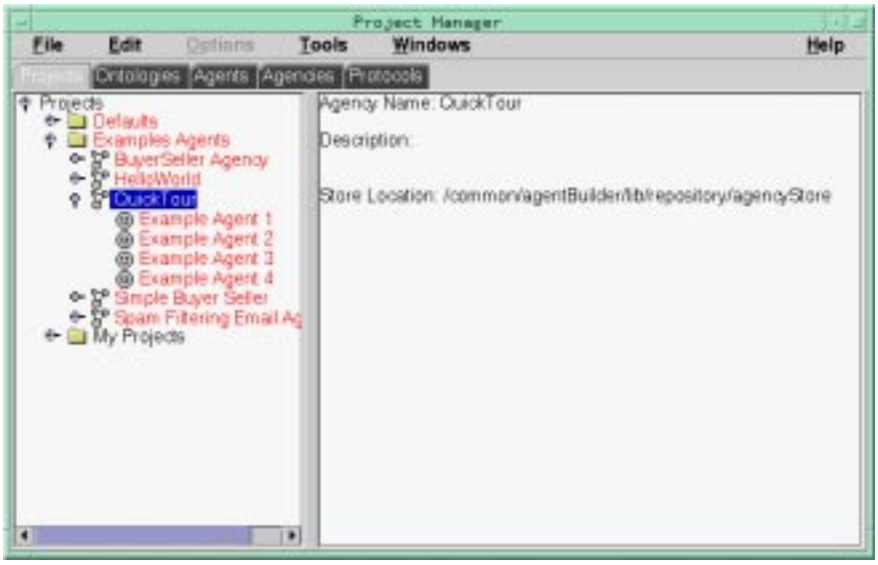


Figure 10. Project Control Tool's Project Status Window

Ontology Manager

The Ontology Manager assists the developer in analyzing the agent application problem domain and in identifying and defining the concepts relevant to the agent's operation in that domain. An *ontology* is simply a (formal or informal) specification of a conceptualization. For example, the software developer may want to develop an ontology that describes the e-mail domain or perhaps an electronic commerce domain. The Ontology Manger provides tools for visualizing domain concepts and relationships, graphically representing these concepts and relationships, and codifying them for storage and subsequent retrieval.

The Ontology Manager provides the developer with a graphical mechanism for creating, defining, and refining concepts and relations in the agent's application domain. AgentBuilder provides a number of tools for ontology development. These include a *conceptual mapping tool* that graphically shows the major domain concepts and their relationships. An *object modeling tool* provides a mechanism for the developer to take the concepts identified using the conceptual mapping tool and express these concepts in terms of objects and relationships between objects defined in the domain.

The Ontology Manager also produces a knowledge and data representation that can subsequently be used by the Agent Tool during agent construction. For example, an ontology for an airline reservations systems might include concepts such as *passenger*, *ticket*, *reservation*, *price*, *arrival time*, *departure time*, etc. The travel ontology provides a formal, consistent representation of an airline travel domain which the agent developer can use in defining agent behavior.

Ontologies can be reused; ontologies developed on one project can be reused on a second project. An ontology library is a collection of related ontologies. These libraries provide a foundation for building intelligent software agents and predefine many of the concepts and relations that are encountered in agent applications development.

AgentBuilder fully supports object-oriented analysis and design methodologies. AgentBuilder agents can reason about objects and agents can communicate by sending and receiving complex objects (not just simple strings). Figure 11 illustrates the Object Modeling tool, one of the AgentBuilder tools provided for analyzing and characterizing the agent domain. In the figure, the object modeling tool is used to display the default classes provided with the AgentBuilder tools. Note that the object modeler uses UML notation. The developer can use these graphical tools to show the project's classes and their relationships. You can graphically establish associations, aggregation (whole-part relationships) and generalization (inheritance).

AgentBuilder fully supports object-oriented analysis and design methodologies. AgentBuilder agents can reason about objects and agents communicate by sending and receiving complex objects.

Agency Manager

The Agency Manager is designed to help the developer construct an *agency.* An agency consists of two or more agents that communicate and cooperate with each other to perform some task. The agents may be identical or specialized for performing different functions. The Agency Manger allows the developer to identify and characterize all of the agents and agent types in the system under development.

The Agency Manager provides a run-time window for viewing the operation of a system of agents. Thus, the developer can monitor interagent communications, control the agents, or run the agent debugger to examine the state of any or all agents.

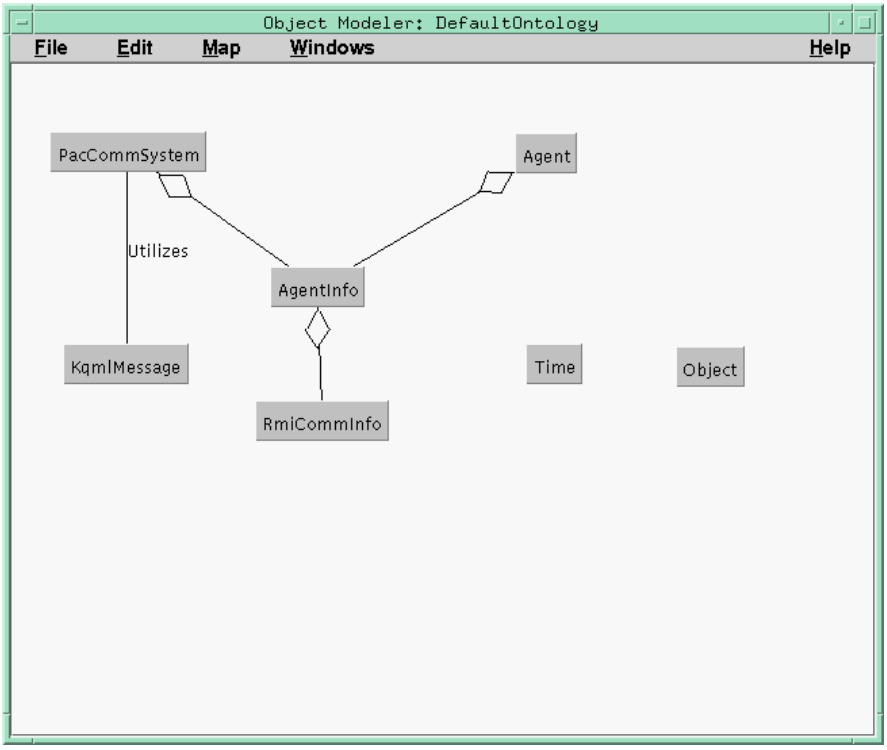


Figure 11. Analysis Using the Object Modeling Tool

Figure 12 shows run-time window viewing a pair of agents used in a shopping application. This tool provides controls for starting, stopping each agent and examining the messages transmitted between agents.

Protocol Manager

The Protocol Manager provides tools to specify the messages and conversational protocols between agents. This allows the developer to dictate the high-level information flow that occurs as part of agent-to-agent transactions. These tools are indispensable when the agent's conversational protocols are complex. The specification of interagent communication messages and conversational protocols provides the basis for specifying an individual agent's behavior. The AgentBuilder Pro

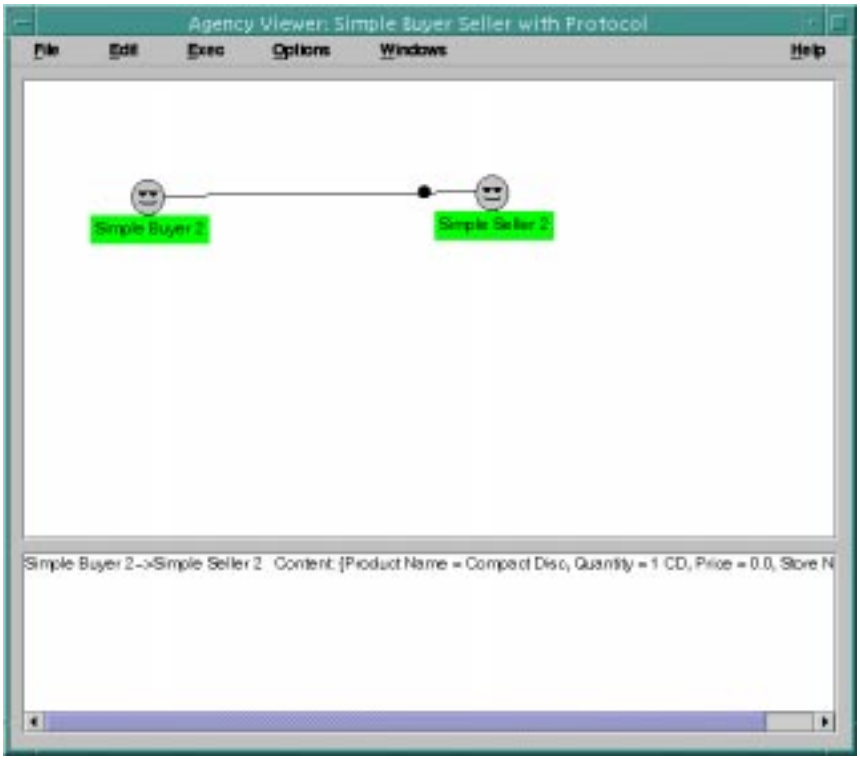


Figure 12. The Agency Viewer

version will automatically generate the skeletal behavioral rules for implementing the agent-to-agent conversations. This saves the developer significant amounts of time because he no longer has to generate the code for handling incoming messages and transmitting outgoing messages. The developer now only needs to specify the conditions necessary for sending a particular message and the actions to be taken and mental model changes needed on receiving an incoming message. Figure 13 illustrates the use of the protocol editor in diagramming the conversations involved in solving a producer consumer problem.

Agent Manager

The Agent Manager provides tools for defining an individual agent's initial mental model and behavior. The agent definition tools include graphical editors for defining the various mental constructs that make up the agent: initial beliefs, initial com-

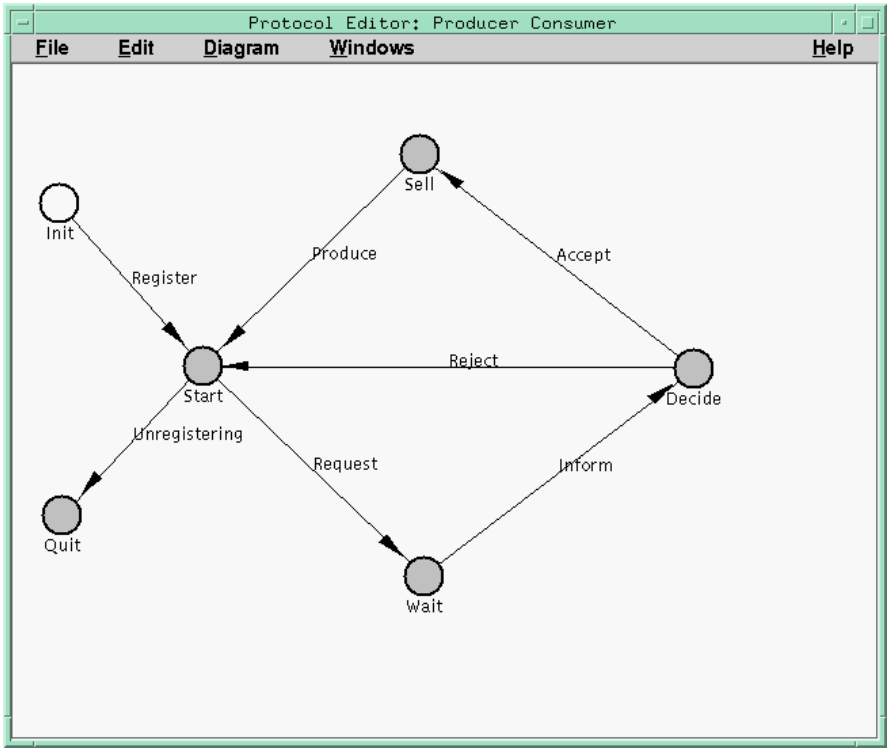


Figure 13. The Protocol Editor

mitments, capabilities and behavioral rules. The Agent Manager control panel is shown in Figure 14. The tabbed display panel allows the developer to easily examine the properties, PACs, PAC Instances, Java Instances, Actions, Commitments and Rules for a particular agent.

The Agent Manager is used to create an agent definition file written in the Reticular Agent Definition Language (RADL). The RADL file is a complete specification of the agent's behavior and initial mental model; this file is interpreted and executed by the Run-Time System.

Commitment Editor

The commitment editor portion of the Agent Manager is used to specify the agent's commitments. By specifying commitments, the developer instructs the agent to

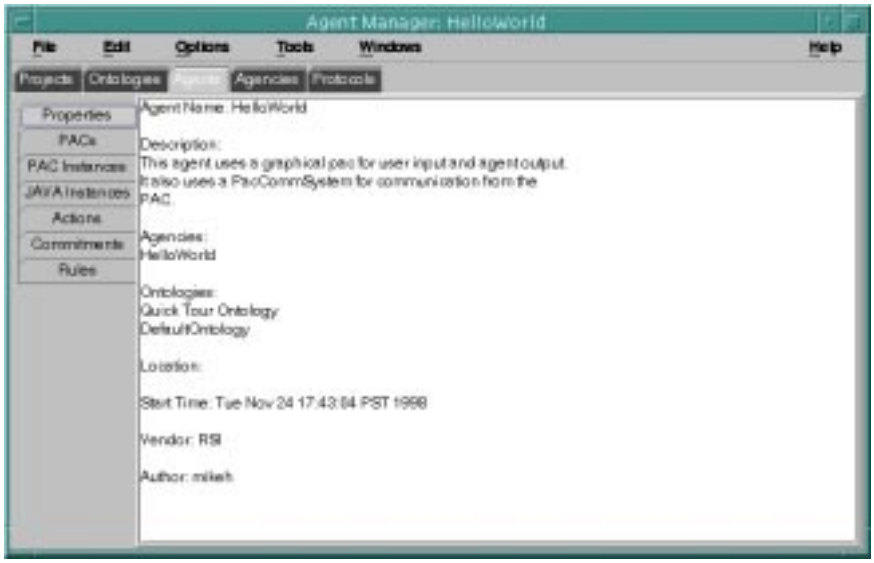


Figure 14The Agent Manager Control Panel

pursue certain actions at certain times. All commitments, including initial commitments, have a time field that specifies when the agent will perform the action specified by the commitment.

Action Editor

The action editor in the Agent Manager allows the developer to associate agent actions with methods in software libraries. The developer can browse the previously defined object model and choose objects and methods to associate with agent actions.

The action editor allows the developer to specify that an action will execute on its own thread, so the agent can implement long-running actions. For example, a control panel might require a thread that lasts for the lifetime of the agent.

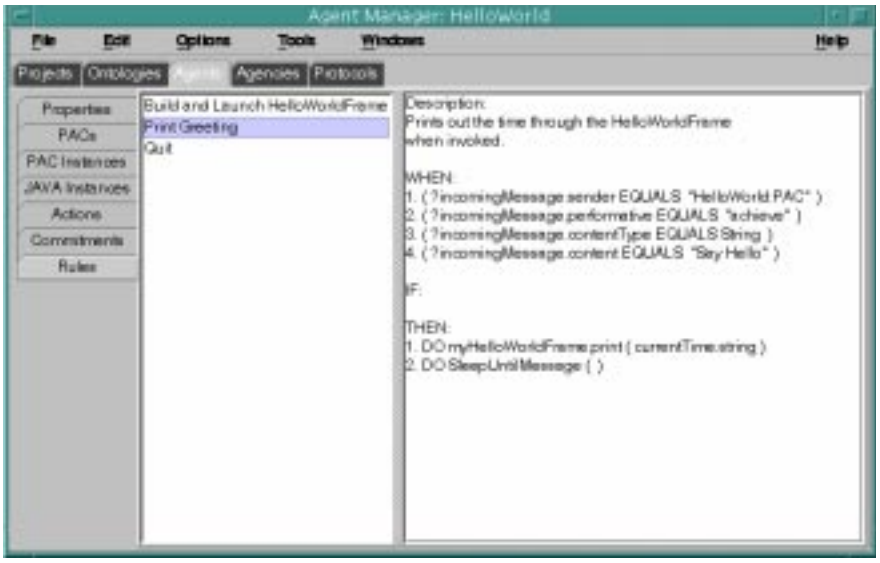


Figure 15. Agent Manager Displaying a Behavioral Rule

Rule Editor

The Agent Manger also includes the behavioral rule editor. The behavioral rule editor defines how the agent operates given a particular internal and external environmental state. Figure 15 shows the Agent Manager with the Rules tab selected.

The behavioral rules are made up of message conditions, mental conditions, private actions, communicative actions and mental model changes. The rule editor allows the developer to graphically build up the WHEN-IF-THEN portions of a rule. Figure 16 illustrates use of the Rule Editor. This figure illustrates the left-hand-side (LHS) editor. Figure 17 illustrates the right-hand-side (RHS) editor. The LHS editor builds the `WHEN` and `IF` portions and the RHS editor builds the `THEN` portions of the rule.

The LHS panel of the Rule Editor allows the developer to specify operators (e.g., EQUALS), Instances, New Variables, Defined Variables and Values for use in the LHS of the behavioral rule. The Message Conditions section of the LHS editor is used by the developer to build the `WHEN` portion of the rule. Each message condition consists of a KQML property and an associated value. The developer can build up arbitrarily complex patterns using the provided operators such as `AND`, `OR`, or `NOT`.

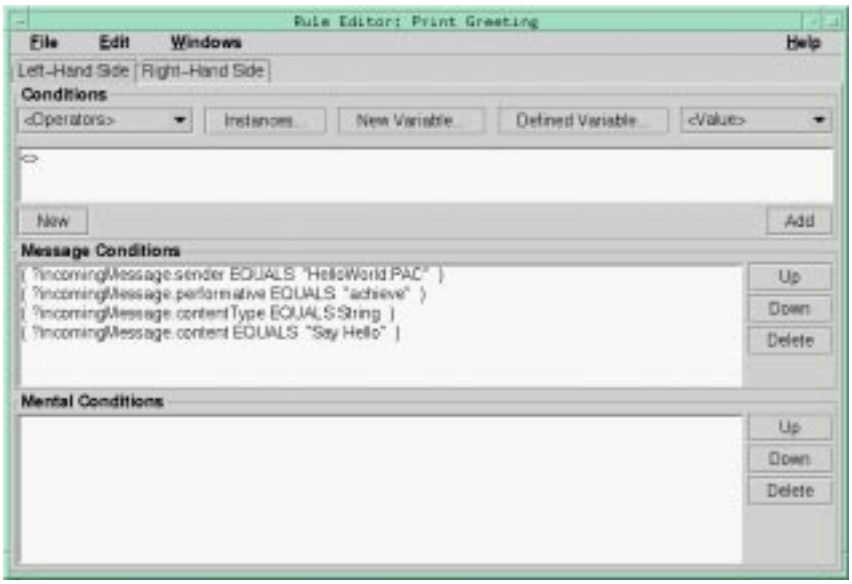


Figure 16. AgentBuilder LHS Rule Editor

The `IF` portion of a rule is constructed in the Mental Conditions section of the LHS editor. The developer constructs a list of mental conditions using any combination of beliefs, intentions, commitments, messages, and variables. The developer is able to build up arbitrarily complex patterns using the provided operators:

- Boolean (`AND, OR, NOT`)
- Relational (`>=, <=, >, <, !=, =`)
- Quantifiers (`FOR-ALL, ELEMENT-OF, EXISTS`)
- Math (`+, -, *, /, SQRT, LOG`, etc.)
- String (`CONCATENATE`, `SUBSTRING`, etc.)

The editor for constructing the RHS of the rule allows the developer to specify Operators, Actions, Built-in Actions, Defined Variables, Return Variables, Instances and Values. The Built-In Actions button allows the developer to select the private actions that will execute when the rule fires. The available actions are selected from the list of capabilities constructed earlier in the action editor. The developer selects the action, selects parameter values, and selects return variables

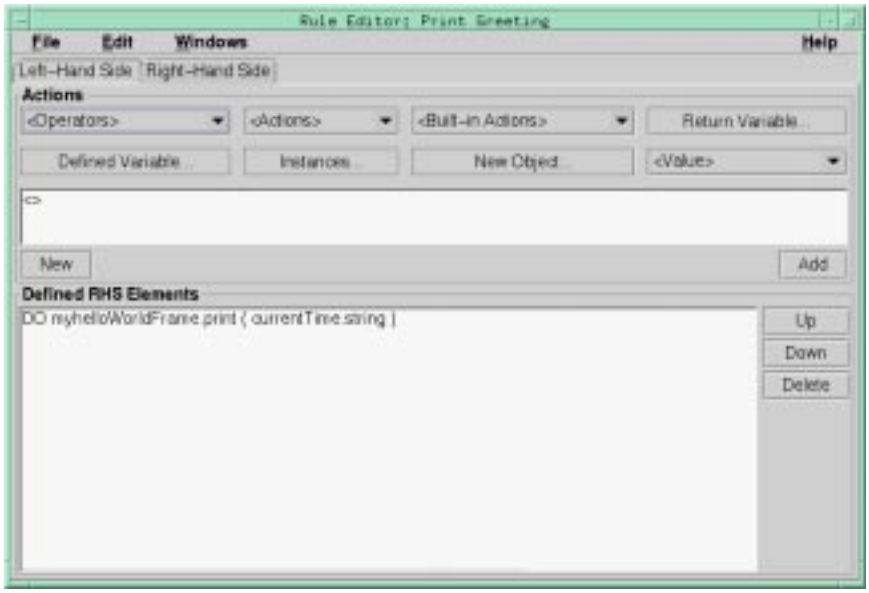


Figure 17. AgentBuilder RHS Rule Editor

from the provided pull-down menus. The return variable stores the value returned by the private action for subsequent use.

The Build Message section of the editor is used for constructing KQML messages. The developer uses the menus to construct outgoing messages using received messages and mental elements.

The Mental Change section allows the developer to specify changes to the mental model which should occur in response to the rule firing. The developer can choose from the previously specified pattern variables, beliefs, and message contents; these elements specify the changes that will be made to the mental model at run time.

Planning and Learning

There is no single learning algorithm or planning algorithm that is best suited for use in all agent applications. Therefore, AgentBuilder is designed to make it easy for the agent developer to add the learning or planning algorithm that is most appropriate for the problem domain.

Run-Time System

Agent Engine

The Run-Time System consists of the Agent Program and Reticular's run-time agent engine. The Agent Program is a combination of the Agent Definition defined in the RADL file and the classes defined in the agent's Project Accessory Class (PAC) library (i.e., Agent Actions and User Interface libraries). The Agent Program is executed by Reticular's run-time agent engine; the combination of the Agent Program and the agent engine produces an executable agent.

At start-up, the Run-Time System initializes the agent engine using information stored in the Agent Definition file and links the required components from the PAC Library. Both the agent definition and the private actions are needed: the agent definition supplies the agent with a reasoning capability and an initial mental model; the private actions give the agent the ability to interact with its environment.

The Run-Time system allows an agent to be created in the development environment and then be deployed as a stand-alone entity executing in the run-time environment. It is the agent executing in the Run-Time System that performs useful work. In the loan-processing example described in Section 3, the Credit Reporting Agent performs the work of compiling the credit history of a loan applicant and communicating with a bank's agent. The Run-Time System is distinct from the AgentBuilder Toolkit. After the agent development process is completed, an agent can be executed on any platform with a Java Virtual Machine (version 1.1 or later). The Run-Time system does not require access to the Java Development Environment.

Reticular's agent engine is a proprietary inferencing engine implemented in Java. This engine utilizes an efficient and robust inferencing procedure to match the agent's behavioral rules with the agent's beliefs and incoming messages. The agent engine performs the reasoning defined in the Agent Definition file and executes the specified actions, by invoking routines in the Private Action library. The agent engine also monitors the execution of the private actions and returns execution results to the agent.

Project Accessory Class Libraries

An agent's PACs contains all of the domain-specific code an agent requires for operation in its domain. The PAC is the mechanism the agent uses to interact with its environment; by calling methods the agent executes its private actions. Generally, agents operate in unique domains, and every domain requires its own PACs.

A PAC can be made up of classes and packages from a variety of sources. The developer may write part of the library, other classes may come from commercial off-the-shelf (COTS) packages, and other classes might be freeware downloaded from the Internet. Although the developer can write PACs entirely in Java, there are situations where Java alone does not meet the needs of an application. Developers can use the Java Native Interface (JNI) to create non-Java methods to handle those situations when an action cannot be implemented in Java; C/C++ functions that conform to the JNI can be used to implement private actions. The developer is free to create new methods and/or integrate existing code (Java, C or C++) into the PAC library. In the bank loan example cited earlier, one private action invokes a method that queries a database of credit histories. This database-query method can either be written by the agent developer (in Java, C, or C++) or obtained as part of a COTS database package. Either way, the method is invoked as the agent's private action and executed in the Run-Time System.

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

The private actions in the PAC are able to communicate information back to the agent in two ways. For non-threaded actions, information is usually returned to the agent via a return value. For threaded actions, information can only be returned to the agent in a message. For example, a GUI element can send messages back to an agent as the user enters information.

AgentBuilder and Run-Time System Requirements

Developers will usually want access to some kind of Java development environment. Example environments include Sun's JKD, Symantec's Cafe or Inprise (Borland) JBuilder. A developer can easily use these general purpose Java development tools to create user interfaces (GUIs) and PACs. AgentBuilder makes it easy to import the classes created with these tools into the AgentBuilder agent construction environment.

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.

SECTION 7

AgentBuilder Product Family

The AgentBuilder product family[1] includes three different products. The AgentBuilder Lite product 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 are building their first agent applications. The AgentBuilder Pro product supports development of agencies of communicating agents. AgentBuilder Pro is a full-featured product and is ideal for use by a small or medium-sized development team. AgentBuilder Pro can automatically generate the code required for implementing conversations between agents. Academic licenses are available for AgentBuilder Lite and AgentBuilder Pro. AgentBuilder Enterprise provides advanced tools and is designed for creating complex enterprise-wide applications. AgentBuilder Enterprise provides support for large and/or distributed development teams and distributed object technology (CORBA/DCOM).

1. Check the latest AgentBuilder product brochures for a detailed description of the capabilities and features of the currently shipping version of AgentBuilder.

Reticular Systems, Inc. also provides special-purpose ontological libraries and Project Accessory Classes (PACs). These additional components, planning, and learning modules. Reticular Systems, 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.
- Ontology Manager including concept mapping and object modeling tools
- Agent Manager tools for creating agent programs using the Reticular Agent Definition Language (RADL)
- Run-Time Engine†
- Support provided through AgentBuilder Web site access and mailing list membership
- 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 that includes tools for creating and managing multiple software agents.
- Agency Viewer tools allows real-time examination of remote agent operation
- Role Editor allows definition of roles for agents
- Protocol Editors for specifying inter-agent protocols; i.e., inter-agent conversations.
- Agency debugging support
- Support for optional learning and planning modules

- Single-user license with annual maintenance
- E-mail, FAX, and telephone support

Academic versions of AgentBuilder Pro are available to accredited universities.

AgentBuilder Enterprise

AgentBuilder Enterprise includes all of the tools in AgentBuilder Pro with the addition of the following:

- Conceptual graphing tools
- Enhanced Ontology development tools
- CORBA/DCOM support
- Mobile agent support
- Enhanced Application Development Support
- Support for Business Rules and Logic
- Enhanced repository manager with support for remote agent development and revision control
- Ten-user license with annual maintenance
- Subscription to new ontology libraries (as released)

Other Products

Reticular Systems, Inc. provides a broad line of products and services for intelligent agent developers. These include:

- Ontological Libraries
- Project Accessory Classes including:

    - HTTP PAC - for building agents that access web pages
    - E-mail PAC - for building agents that can read and send E-mail messages
    - FTP - for building agents that can perform FTP file transfers

- NNTP - for building agents that can read and post to USENET news groups

- Document Analysis - for building agents that can analyze and compare documents using a vector-space analysis technique.

- Consulting Services
- Maintenance and Support Contracts
- Training

SECTION 8

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.

For Additional Information Contact:

Reticular Systems, Inc.
4715 Viewridge Avenue, Suite #200
San Diego, CA 92123
Phone (619) 279-9723
FAX (619) 279-9697