Agent toolkit for distributed knowledge networks

by

Tarkeshwari Sharma

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Major Professor: Vasant G. Honavar

Iowa State University

Ames, Iowa

2000

Graduate College
Iowa State University

This is to certify that the Master's thesis of

Tarkeshwari Sharma

has met the thesis requirements of Iowa State University

Major Professor

For the Major Program

For the Graduate College

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Vasant G. Honavar for his guidance, patience and support throughout this research and the writing of this thesis. I am very much thankful to him for encouraging me to do research and helping me with my ideas. His insights have been very helpful to me during my research.

I would also like to thank my committee members for their efforts and contributions to this work: Dr. Leslie Miller and Dr. Drena Dobbs.

I am specially thankful to Adrian Silvescu, my colleague, for helping me with the proofs for theorems related to distributed learning.

I would like to thank the computer science department for giving me teaching assistantship for the year 1998-1999. My research assistantship for the year 1999-2000 was funded by a grant from John Deere Foundation, Distributed Knowledge Networks, PI: Vasant Honavar, $30,000.

I am very much thankful to my friends, Melanie Elkhart, Lisbeth Meum and Rushi Bhatt who patiently read my thesis and helped me to correct the errors.

ABSTRACT

This thesis focuses on the design and development of an agent toolkit which can provide developers with agent models that are useful for rapid design and prototyping of applications in a given domain. Most of the existing agent systems belong to either of the two categories - agent systems specific to a problem or generic agent architectures that describe functions and properties of an agent at a very abstract level. There is a need to devise a toolkit that can provide agent models that can be reused for applications belonging to a specific domains.These agent models are generic for a domain of applications and are more specific than the models provided by the generic agent architecture. The application which motivated us to build such a toolkit is distributed knowledge networks.

Advances in sensor, high throughput data acquisition, and digital storage techniques have made it possible to acquire and store large volumes of data. Distributed Knowledge Networks are proposed as a solution for selective, reactive, proactive information retrieval, knowledge discovery, distributed problem solving to translate these advances into fundamental scientific advances and technological advances. We have designed a toolkit which gives agent prototypes for such networks. We propose formal specifications of agent models which are useful in domain of distributed knowledge networks. Agent toolkit implements some of these prototypes. This would help developers to concentrate more on agent related aspects of the system rather than generic features. It also helps to develop the generic domain specific features that can be used across applications.To demonstrate utility of the agent toolkit, design and implementation of applications built using the toolkit is described.

Recent advances in data storage and acquisition technologies made possible to produce increasingly large data repositories. Moreover most of these data sources are physically dis-

tributed and assembling them together at a central site is an expensive process in terms of network bandwidth. Many organizations are not willing to provide the actual data for security reasons but they are willing to give some summary of data. This generates a need for Learning Algorithms that are able to learn from distributed data without actually collecting data. In this thesis, we explore a general technique involving information extraction to enable the current learning algorithms to learn from the distributed data. We propose distributed learning techniques for horizontally and vertically distributed data. These techniques are illustrated with the help of decision tree learning algorithm. The results obtained in case of distributed learning is same as in the centralized case where the complete data is at a central location. We give time, space and communication cost analysis for the distributed algorithms.

CHAPTER 1. INTRODUCTION TO AGENTS

In today's world, use of computers is increasing day by day for handling and processing vast data sources. They are proving very efficient in accurate processing of data. They use very little time for processing data compared to humans, and they are replacing humans to some extent - at least in the field of repetitive tasks involving mundane calculations. But they can do only what they are "told" to do and this seems to be sufficient for a variety of applications, e.g., payroll processing. Nothing could happen in case of running a program without user's wish, i.e., the computer program do not take *initiative*. If the program ever encounters a situation which was not predicted by the programmer, results can be disastrous.

There is an increasing need to change the role of computers from being obedient and unimaginative servants to that of a creative, intelligent and proactive helper. We need programs that can act on behalf of the user, give suggestions to him and make selections for him after knowing his preferences. Such programs have come to be known as "Agents". Agents are viewed as software programs that can *take decisions* regarding steps they need to take in order to accomplish their goal. They are *autonomous* and can act in a dynamic environment using their "intelligence" to respond according to changes in the environment.

An example of an agent is the Internet search agent. This agent helps users to search the Internet for a particular kind of information. The user can specify his criteria and the agent will search various Internet sources to furnish the user with the required information. The agent can search for news item, airline tickets, research articles etc. Over time, the agent can learn a user's preferences and get information for him without his guidance or intervention. This agent would fail in the case when there are no resources on Internet on a particular topic.

This chapter gives an overview of agents, what they are composed of, how they function

and why are they more efficient than other techniques.

1.1 What is an Agent?

There is no universal agreement on the definition of an agent. People involved in the area of agent related research have given a variety of definitions ranging from very simple to complex ones. Below we try to give a view of agent by examining some of these definitions (FG96).

The MuBot Agent[http://www.crystalize.com/logicware/mubot.com]: "The term agent is used to represent two orthogonal concepts. The first is the agent's ability for autonomous execution. The second is the agent's ability to perform domain oriented reasoning."

The AIMA Agent (RN95a): "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." AIMA is an acronym for "Artificial Intelligence: a Modern Approach," a book which is used as text book for Artificial Intelligence (AI) in many universities. This definition depends to a great extent on the meaning of terms environment, sensing and acting.

The Maes Agent (Mae95): Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed. Pattie Maes, of MIT's Media Lab, is one of the pioneers of agent research.

The KidSim Agent (SCS94): "Let us define an agent as a persistent software entity dedicated to a specific purpose. 'Persistent' distinguishes agents from subroutines; agents have their own ideas about how to accomplish tasks, their own agendas. 'Special purpose' distinguishes them from entire multi function applications; agents are typically much smaller." The authors are working at Apple. This definition requires agents to be persistent which is a new requirement. It says that agents are special purpose which might not always be true. Agents could be serving more than one purpose also.

The Hayes-Roth Agent (HR95): "Intelligent agents continuously perform three functions: perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences, and determine

actions." Barbara Hayes-Roth is working with Stanford's Knowledge Systems Laboratory.

The IBM Agent [http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm]: "Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in doing so, employ some knowledge or representation of the user's goals or desires." This definition is from IBM's Intelligent Agent Strategy white paper.

The Wooldridge-Jennings Agent (WJ95): "... a hardware or (more usually) software-based computer system that has the following properties:

1. autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;

2. social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;

3. reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;

4. pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative."

The Wooldridge and Jennings definition spells out terms like environment, acting and sensing very clearly. It also introduces the need for *communication* among agents.

The SodaBot Agent [Michael Coen http://www.ai.mit.edu/people/sodabot/slideshow /total/P001.html]: "Software agents are programs that engage in dialogs [and] negotiate and coordinate transfer of information." SodaBot is a development environment for software agent being constructed by Michael Coen at the MIT AI Lab. This definition looks very different from what we have seen so far. By using the term dialogue, the author stresses the need for communication but it is not explicit here.

The Brustoloni Agent (Bru91): "Autonomous agents are systems capable of autonomous, purposeful action in the real world."

The Brustoloni agent, unlike the prior agents, must live and act "in the real world." Real world contains dynamic elements and is non deterministic, e.g., internet.

While there is no unanimity on definition of agents, the general consensus is that autonomy is an essential notion of agency. By *autonomy*, it means that agents can function without human intervention or other external agencies. They have control over their internal state and behavior.

1.2 Components of an Agent

For our purpose, we adopt the definition given by Wooldridge-Jennings. An agent is situated in the *environment* which is a collection of states. The agent gets information about relevant things in the environment by observing states of the environment. It interacts with the environment through its actions. Fig 1.1 shows an agent.

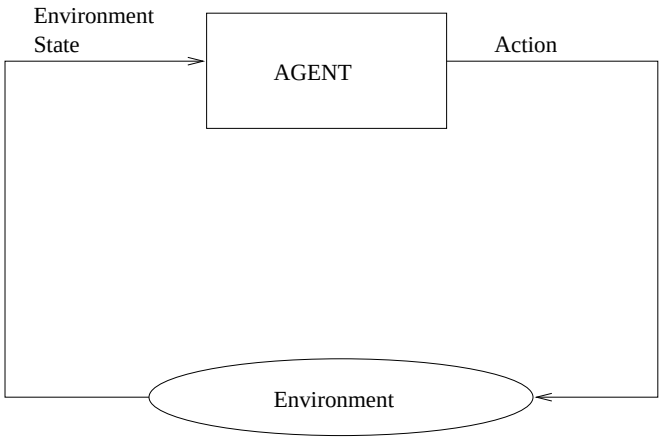

Figure 1.1 Components of Agent

Agent can be situated in various types of environment. Its functioning is affected to a great degree by the type of environment it is situated in. A constantly changing environment will need the agent to monitor changes in environment continuously where as in a static environment, agent can plan its activities without with the assumption that environment states would not change during the planning process. Following is the classification of environment

properties: (RN95b)

1. Accessible vs inaccessible. An accessible environment is one in which an agent can get complete, accurate and up-to-date information about the environment's state. But most of the real world environments, e.g., Internet are not so accessible. Agents have incomplete information about the environment.

2. Deterministic vs Non-deterministic. An environment is deterministic if it can be known with certainty the result of performing an action. There is a single guaranteed effect for every action. Real world is highly non-deterministic. It is difficult to design agents that work in non-deterministic environments.

3. Episodic vs Non-episodic. In an episodic environment, an agent's performance is dependent on a number of discrete episodes with no connection between any of those episodes. Again, it is easy to program agents for an episodic environment compared to a non-episodic environment as agents have to consider only the present episode and they are not affected by previous or future episodes.

4. Static vs Dynamic. A dynamic environment is one that is continuously changing due to processes other than the agent. A static environment is one that is not affected by anything other than the actions of agent. For most practical purposes, environment is dynamic and it is difficult to program agents for dynamic environment as it has to take changes in the environment into account while executing its actions.

5. Discrete vs Continuous. An environment is discrete if there is a finite number of actions and percept in it. For example, the environment for game of chess is a discrete one and for taxi driving is continuous.

The definition says that agents are autonomous i.e., they can function without intervention from humans or other systems.

An abstract, top-level view of agent is shown in fig 1.1. It shows that the action generated by an agent affects its environment. However, an agent never has complete control over the

environment. It has either no control or at best has a partial control over the environment.

In its simplest form, an agent has a knowledge of actions available to it. These actions represent the *effectoric capability* (WJ95) - its ability to modify its environment. All actions have *pre conditions* associated with them and their success depends on fulfillment of these preconditions, e.g., the action "lift an object" is possible only when the weight of the object is sufficiently small for agent to handle. Agents have been categorized into various architectures depending on the way they decide about their actions, e.g., reactive agents, learning agents etc. The agent's environment plays an important role in this decision making process.

Real world environments are dynamic consisting of other agents and other entities which themselves are dynamic. The environment is also non deterministic. In this situation, agents need "intelligence" to function autonomously and efficiently. Intelligent agents are discussed in depth in the following section.

1.3 Intelligent Agents

As per our definition, for an agent to be intelligent, it should exhibit flexible autonomous behavior. Following are the desired characteristics of intelligent agent:

1. Reactivity: Intelligent agents are able to perceive their environment and react to changes in it in a timely manner. They are constantly aware of the changes in environment.

2. Pro-activeness: Intelligent agents exhibit *goal-directed* behavior and are able to take initiative to achieve their goal. They plan a set of actions to be executed to accomplish their goal.

3. Social-ability: The most important thing is that intelligent agents are able to interact with other agents in their organization to co-operate, co-ordinate or negotiate. They are capable of working in teams and this allows for possibility of having multi-agent systems where agents can engage themselves in various forms of co-operation and co-ordination.

Each of these properties have a significant role to play in an agent's intelligent behavior. Pro-active behavior is most appropriate when the environment is not dynamic so that it remains

unchanged during the time the agent is planning its actions to accomplish a goal. In a static environment, the goal also remains valid during the period of the execution of plan. If the environment is dynamic and non deterministic, agents need to be reactive so that they can execute actions in accordance with the change in environmental states. In most of the real world cases, a combination of reactive and pro-active behavior is desirable. It is easy to build a purely reactive or pro-active system but it is difficult to build a system with right combination of reactive and pro-active behavior. Social ability opens avenues for cooperation and coordination among agents. In a multi-agent environment, it is necessary that agents interact with each other and have knowledge of other agent's capacities and their commitments.

1.4 Motivation for using Agent Based Systems

By an agent based system, we mean a system that is designed and implemented using agents. The question is that why agents should be used when there are other technologies available such as expert systems and object-oriented programming? What do agents offer which make them better than any of existing ways to solve a problem? Agent based systems provide an attractive paradigm for developing software applications. Designing and building agents exploit advances in various fields in AI such as planning, learning etc. In this section, we discuss characteristics of domains where agent based systems provide an effective solution.

Systems can range from being very simple to highly complex ones requiring communication, networking, continuous monitoring of processes etc. Building complex systems is a relatively difficult task and the difficulty stems from the fact that complex systems are made up of several sub-systems which interact with each other. *Reactive Systems* are examples of complex systems that need an on-going interaction with their environment. Some of these complex systems are process control systems, network management systems and distributed database systems. The systems where agent technology can be of significant use are of three types (WJ98):

1. Complex systems

2. Open systems

3. Ubiquitous Computing systems

1.4.1 Complex Systems

Building high quality complex systems is one of the most challenging tasks in computer science. In order to handle this complexity that arises because of various subsystems and interactions among them, the solution needs to be *modular and abstract.* According to Booch (Boo93), tools for tackling complexity in software are:

1. Decomposition: Decomposition leads to dividing a complex problem into smaller modules which can be dealt with more efficiently in relative isolation. It helps to focus on different parts of the problem which can be solved and put together to form solution for the complex problem.

2. Abstraction: Abstraction allows one to get a simpler view of the system emphasising some details while hiding others.

3. Organization: Organization helps to put together various sub-parts of the solution in an efficient and useful manner.

Complex systems are made of several subsystems organized in a hierarchical manner which in turn are composed of components working together. There is a pattern of interacting systems at all levels. Agents represent a very powerful and efficient tool for making systems modular. To solve the problem of interaction and inter dependency, agents provide a means to communicate among each other and maintain a social model - knowledge about other agents in the organization. Different agents can concentrate on different objectives to be achieved by system. Complex systems also have multiple loci of control, and hence, it is needed that individual components should localize and encapsulate their own control. Each point of control could be autonomously managed by an agent and these agents could work together as a system. Different types of agents exist in a multi-agent system, performing different functions. By using an agent based approach, the problem can be divided into smaller components which can be developed individually and be specialized at solving sub problems. Agents have the ability to

initiate and respond to interactions in a flexible manner, and hence they are appropriate in situations where the nature of interactions is not known in advance. Complex systems have a constantly changing web of interactions between their components. Collection of various sub-components in a complex system are required to be viewed at different levels of abstraction. Agent based systems provide a good level of abstraction here. Different functions can be assigned to different agents and they can be put together to form the solution. For managing organizational relationships, a rich set of structures are available. For managing interactions among various components, protocols are available along with communication languages. Also structures are available for modeling collectives. Agent based systems provide the flexibility to arrange sub systems in different types of organizations according to need.

1.4.2 Open Systems

An open system is characterized by the capability of the system structure itself continuously changing. In such a system, there is frequent addition of new components and elimination of old ones. Components themselves are dynamic, non-deterministic and heterogeneous in nature. Here a system is required that can deal with the dynamic nature of open systems. An agent based system can be reactive, they can sense their environment and respond to every change that affect their functionality without human or any other kind of intervention. An example of open systems is the Internet- a loosely coupled network of ever expanding size and complexity. It contains thousands of servers of different types created by different organizations or individuals, having various types of information. The information on the Internet is very dynamic and heterogeneous in nature. For a system to be useful in this kind of scenario, it should posses the ability to react, be autonomous and communicate with other agents involved to share information and knowledge.

1.4.3 Ubiquitous Systems

Ubiquitous computing [http://www.ubiq.com/hypertext/weiser/UbiHome.html] means one computer serving many people. The computers are made in such a way that they are almost

invisible. They have many displays–audio, video and environmental. Ubiquitous computing devices are tabs–tiny wireless handhelds, pads–booksized X terminals, boards–whiteboard computers. Ubiquitous computing system involves internet as backbone and short-range unlicensed wireless as leaves. Individual PCs are used as servers and variety of light weight devices as clients. It provides means to integrate new devices with existing PCs using wireless technology. This permits easy use of the installed local infrastructure. Aim of ubiquitous systems is to connect existing computing infrastructure with each other without using network wires. These systems are required not to behave just as "obedient" servants but as helpful assistants. It is needed that systems behave autonomously, respond according to the changing environment, help users in making decisions. For this they should be able to act on their own without human intervention to a great extent and should be able to "think" of a plan to achieve its goals. Intelligent agents can be of great use in ubiquitous computing sice they can function as autonomous agents.

1.4.4 Improvement in the Efficiency of Software using Agent Based Systems

Existing systems can be improved by using agent based approach. Agent based systems can provide better solutions when the domain involves entities which are distributed physically or logically and involves interaction of one or more of these entities. In such cases, agents provide a natural way of solving problems. These entities and their associated interactions can be mapped to autonomous agents which have knowledge and resources. Agents can move from one location to another and synthesize information from data sources. Data can be synthesized by each agent locally and then only high level information needs to be sent to relevant places or agents. This eliminates the need to collect all data at a central processor and reduces network traffic considerably. If the control is distributed, each agent can be given limited control and they can have a supervisor agent that monitors the functions of all subordinate agents. When expertise is distributed, agents provide an effective solution because of their ability to interact with each other. Agents with various skills can communicate and cooperate with each other to solve problems which require more than one kind of skill.

CHAPTER 2. MOTIVATING APPLICATION FOR THE AGENT TOOLKIT - DISTRIBUTED KNOWLEDGE NETWORKS

Agents provide the means to build effective, modular and extensible solutions for complex systems. Design and development of agent based systems is a time consuming process. There is on-going research on providing systems for agent development which capture as much abstract information as possible about agent architecture. This abstraction would help developers concentrate on agent-related features of the multi-agent systems such as reactivity, intelligence and pro-activity. One way to provide this abstraction for agent related programming is an Agent Toolkit with agent models that can be used for designing multi-agent systems. In this chapter, we will discuss need for an agent toolkit for a set of closely-related problems in a domain. This toolkit provides various types of agents with basic functionalities relevant to a specific domain.

2.1 Distributed Knowledge Networks

Recent advances in high throughput, data acquisition and digital information technologies have resulted in acquisition and storage of large volumes of data. Decision makers have, at least in principle, large volumes of data as well as analysis and decision support tools. Data and tools reside on multiple, geographically distributed heterogeneous hardware and software platforms around the globe that are connected to internet. In order to acquire, store, translate and analyze this data into gains in our understanding of respective domains and effective decision making, we need sophisticated tools for information retrieval, knowledge discovery, distribution problem solving and decision making. In any organization, people at different level of authority need to see different forms and quantities of same data, e.g., computer operators

in a department need to deal with data in that department only where an executive manager would want to have a report on data across departments and locations in a concise manner. Distributed Knowledge Networks (DKN) have been presented as a solution to meet challenges in ability to gather, store and analyze a wide variety of data on multiple, geographically distributed, heterogeneous and often autonomously owned and operated on different software and hardware platforms (HMW98).

2.1.1 Components of DKN

Multi-agent systems, because of their modularity, offer an attractive approach to the design of DKN. Appropriateness of using agent-based systems for DKN has been summarized below.

Data sources autonomously owned and operated by agencies reside on different hardware and software platforms. This gives rise to the need of sufficient inter-operability among different data sources. For example, military applications have to use data from various sensors, intelligent sources, etc. Hence, DKN makes use of intelligent agents described in section 1.3 to provide access to these data sources. Various types of agents are needed to perform different functionalities. Reactive agents respond to the changes they perceive in their environment. Proactive agents exhibit a goal-directed behavior and decide upon a plan to accomplish their goal. Utility-driven agents act in ways designed to maximize a suitable utility function and learning agents modify their behavior as a function of their experience. Details about these agents can be found in (RN95c). DKN systems implemented by Honavar et al includes prototypes for customized information retrieval, information assimilation and knowledge discovery functions.

The data sources are geographically distributed in many cases. This calls for the use of intelligent agents for intelligent, selective and context-sensitive data gathering and data assimilation prior to large scale data analysis. Hence, DKN includes tools for monitoring these data sources, extracting the relevant information from them and transmitting this information to the relevant location.

Because of the nature of the location of data sources, it is desirable to perform as much

data analysis as possible at the location where data is stored rather than collecting all data at a centralized location. Transmitting only the results of data analysis saves a lot of network bandwidth compared to complete data transmission. For this purpose, DKN uses mobile agents (Whi97). A mobile agent is a named object which contains code, persistent state, data and a set of attributes and can transport itself from one host to another as needed for accomplishing a task. Mobile agents provide effective solution for DKN by performing data processing at sites where data is located and by carrying only the context-sensitive information with them.

The data sources contain multiple types of data, e.g., text, figures, relational tables etc. DKN provide tools for extracting, transforming and assimilating relevant information from heterogeneous data sources into *data warehouses* (Gup97),(Inm96) where it can be further analyzed to give context sensitive information.

Data to be analyzed is large in volume, diverse and in various forms. A range of scientifically relevant, but complex in nature relations, are likely to exist among the data. Hence, DKN uses *Machine Learning* for data-driven knowledge discovery. A variety of machine learning approaches include artificial neural networks, decision tree learning, rule induction and evolutionary techniques (Hon98) , (Mit97).

The data sources are dynamic, i.e., they keep changing very rapidly with time. Hence, DKN needs agents which sense change in data and trigger update in related data sources and sites.

Design of complex information systems require a modular approach where the over all task is divided into more manageable sub tasks. Multi-agent systems offer a very effective solution for these kind of systems since there can be autonomous, individual agents responsible for different assignments and these agents can coordinate to give a working system. Each of these agents could be responsible for individual data sources and their analysis. Modular design can be easily extended to a broader class of applications. Hence, multi-agent systems have been used for design of DKN (HMW98). In such multi-agent systems, it is very necessary to have effective communication and coordination among agents to complete any task. Since the agents are autonomous, it is necessary to have a notion of adequate control over their behavior. This

notion includes functions such as coordination among agents, synchronization among agents, activation and deactivation of individual or group of agents, selection of agents, creation and elimination of agents, learning from experience, etc.

2.2 Motivation for Toolkit for DKN

DKNs can significantly differ from each other in terms of data storage and analysis. For example, different applications can have different methods to learn from data sources or they can differ in the communication language they use. Therefore, abstraction is needed at a level where agent models with generic capabilities for a system are provided, i.e., provision of a tool which would help developers design agent systems from a certain level. For applications belonging to a domain, agents perform some generic functions that are common in the domain and are not application specific, e.g., all agents in DKN need communication capability but the language to be used can be different from application to application.

We propose an agent toolkit to support various levels of abstraction and modular approach to design. This toolkit needs to provide the following facilities:

1. Designs of generic agent types which are analogous to design patterns in object oriented design which can be reused or can be used as a part of a complex design.

2. Support for creation of multi-agent organizations with different types of structures. It should give the flexibility to arrange agents in a vertical or horizontal hierarchy or in a hybrid manner.

3. Support for reuse of the components so that complex agent types can be created by refining and composing from existing types. This is a very important function of the toolkit since it can save work involved in designing complex agent types from scratch.

4. Support for creation of agent models with more than one type of functionality by combining different models.

5. Support for designing modular agent based application. Agents have the ability to act autonomously for a specific function as well as act together to provide solutions for a

larger system. It is necessary that the toolkit should provide models which have well defined interfaces to coordinate with each other.

6. Support for multiple means of coordination and interaction among agents.

At present, agent-based systems are designed and implemented according to specific needs of an application. Generally they are not generic and cannot be reused. On the other hand, there are architectures defined for agents which specify very generic properties of agents in a multi-agent system. Next section summarizes both kind of agent based systems.

2.3 Agent Based Systems

Existing agent based systems can be grouped into two categories:

1. First category contains systems that are specific to a particular problem like Electronic Market Place (AJT99), Load balance for electricity (BCG+98), Letizia for web browsing [http://agents.www.media.mit.edu/groups/agents/projects/] etc. These problem-specific systems are designed keeping an application in mind and, more often than not, they cannot be used for similar applications.

2. Second category contains generic agent architectures such as Generic Agent Models (GAM) (BJT99),(BKJT97). This architecture defines functions and interfaces for agents that are generic for any agent irrespective of the domain of the application.

Both kinds of systems are discussed in more detail below.

2.3.1 Application Specific Agent Systems

This category contains agent-based systems which are built keeping in mind a certain problem. These applications cannot be used for similar problems. They are designed after analyzing only one problem, and hence, are very specific to that problem.

Agent based system supporting negotiation for load balance in electricity use is an example of this kind of system (BCG+98). This system uses models of negotiation for load management

to make better and more cost-efficient use of electricity production capabilities and to increase customer satisfaction. Methods of negotiation used are as follows:

1. The Offer Method: The offer is made to customer agents that if they use a certain percentage of electricity, they will be given the same at a lower price.

2. Request for Bids Method: When a peak in electricity load is expected, the customers are requested to respond by saying how much electricity they really need when a reward is promised. If a customer makes a bid that is awarded, it pays low price for that amount of electricity and higher price for extra electricity. If the customer does not make a bid, it pays the normal price.

3. Announce Reward Table Method: This method can be seen as a structured combination of the two methods described above. But here, the customer cannot make a bid of their choice, instead they have to select from a set of discrete values.

This system uses two main types of agents - Utility agent and Customer agent. Utility agent always starts the model of negotiation and works for the electricity manufacturing company. Customer agent represents the customers using electricity. Utility agent communicates an announcement to all customer agents to which they respond by making a bid. The utility agent might need to make further announcements depending on the response received from customer agents and this goes on until an agreement has been established. The main functions of utility agent are: to acquire information about availability of electricity, cost and external world, to determine which negotiation strategy is most appropriate and to monitor the negotiation process as it progresses, to assess when to negotiate with customer agents, to determine content of negotiation and to interact with customer agents. Similarly the main functions of customer agent are: to determine which negotiation strategy is most appropriate for negotiation with its own Resource Customer Agents and to monitor negotiations with them, to determine which negotiation strategy is most appropriate for negotiation with a Utility agent and to monitor this negotiation process.

This is a very effective solution for balancing electricity load using negotiation but it is restricted to this application. The agents are designed according to needs of the electricity company for which the system is made. They follow the corporate rules followed by the company. The design lacks abstraction of models such that the models can be used in similar applications. There is a variety of agent based application which follows the similar pattern of design for a specific problem only.

Another example of a problem specific multi-agent system is Environmental Emergency Management(EEM) (Wei99). It is concerned with events that alter the usual mode of operation of an installation, having a negative impact on its environment. Examples of such events are forest fires, river floods or gas dispersions from chemical plants. Emergency management involves assessing potentially risky situations ahead of time and proposing solutions for them. An agent architecture for EEM is described here. Following are the types of agents involved in this system:

- The Local Emergency Management Agent (LEMA), is responsible for understanding problematic situations in a predefined area and proposing a local solution, respecting initial management plans. The LEMA receives data in terms of variables of rainfall and water level and uses pattern matching to detect if the current situation is problematic. It uses method of simulation and a rule base proposing potential causes to diagnose the situation. Once a collection of causes has been identified, it tries to eliminate the cause, or if that is not possible, it tries to take protective actions. It builds a task plan and for the tasks, that cannot be accomplished on its own, it takes help from other agents.

- The Dam Management Agent (DMA), is responsible for making decisions about dam control. It takes into consideration other agent's needs and the situation of the dam and its resources. The DMA receives data about the state of the water levels and spill flows in every dam. DMA sends limitations of the outflows from dams proposed by the LEMA according with its flooding problems. It uses pattern matching to detect situations like a dam break or major flood. The main task of this agent is to decide the outflows from every dam including the case of spill null.

- The Fire Brigade Management Agent (FBMA), is responsible for population evacuation and providing resources for protective works. It receives data about the state of its resources in terms of machinery, vehicles and manpower. LEMA sends messages to it when it needs manpower or machine resources. It also uses pattern matching to diagnose trouble in an area. If there is excess demand from any particular area, it means there is trouble in that area. FBMA tries to fulfill all demands but if it is not possible to do so, it uses reasoning to give a solution with adequate delays. To decide these delays, the knowledge about control and social knowledge will be applied to reformulate the constraint base until some solutions are found. Control knowledge will select the options to guide the process of constraint reformulation.

- Transport and Ambulance Management (TAMA), is responsible for the viability in the road transport network and ambulance resources management. It functions in the same way as FBMA.

Each LEMA maintains a two-way communication with any of the other three types of agents. Strategic knowledge for this system is based on the local choice rules at the LEMA where the tasks to be provided by every general agent are defined and local choice rules at the level of general agents where the preference between different LEMAs' are established. Every agent gets tasks according to its specialty defined by rules. LEMA and other agents engage in dialogue whenever a task needs to be modified.

A LEMA monitors *what is happening* by event detection method using pattern matching. For *what to do*, it uses two approaches-*damage suppression* and *damage limitation*. In damage suppression, it is possible to eliminate causes of damage with help of actions from other agents. For example, if LEMA needs help from DMA, it is possible that the DMA actions might be sufficient to solve the problems or else DMA proposes actions which can partially solve the problem. In damage limitation, an estimate of damages and needs to support its effects is provided by LEMA to TAMA and FBMA agents. A dialogue is established between these agents depending on the available sources in FBMA and TAMA together with norms of coexistence with LEMA.

2.3.2 Generic Agent Architectures

Generic models are proposed with the goal of providing abstraction in agent programming. These agent models propose some abstract structure on the basis of which specific agent models can be developed. These systems belong to another extreme of the spectrum when compared to agent-based systems for specific problems. They provide pre-defined agent models and tools with generic agent functionalities. Nevertheless, they help in designing agent-based applications which save programmers considerable amount of time and effort. DESIRE (framework for DEsign and Specification of Interacting REasoning components) proposed by Brazier, Jonker and Truer (BJT99), is described here briefly.

2.3.2.1 DESIRE

DESIRE is built with an intention to structure the design process - the acquisition of a conceptual model for an application is based on generic structures used in the model. This generic agent model stems from specific application domains and by refinement it can be used for designing a variety of agent based systems. This model is generic in two ways.

1. With respect to processes or tasks: The model abstracts from processes at lower levels. A more specific model will have an increased number of processes specific to it.

2. With respect to knowledge: The model abstracts from knowledge structures used in various domains.

2.3.2.2 Process Level Abstraction

The generic model lays out some processes within agents at the highest level of abstraction. Following are the processes identified by the generic model:

1. Own process control: This includes processes to decide and evaluate its goals and plans along with maintaining a self model.

2. World interaction management: This involves interacting with the outside world which makes agent's environment.

3. Agent interaction management: This involves managing communication with other agents.

4. Maintenance of world information: This involves maintaining knowledge of the external world.

5. Maintenance of agent information: This involves maintaining knowledge about other agents.

6. Cooperation management: This involves tasks related to social processes such as cooperation and negotiation etc.

7. Agent specific task: This involves tasks which are specific to a particular agent.

In this model, perception of the environment is done by *world interaction management*, *maintenance of world information* and *maintenance of agent information*. Agents manage actions in the world using *world interaction management*. Social actions are managed by *agent interaction management* and *cooperation management*. Thus, it complies with the definition of the intelligent agent in section 1.3.

2.3.2.3 Knowledge Level Abstraction

The generic model also provides information types for input and output of the generic agent model. For an agent, input-information type is incoming communication information and observation result information. The outgoing information is outgoing communication information, observation information and action information. Various types of interface information types are provided for components.

Generic model defines links for information exchange among agents. At the highest level of knowledge abstraction, information types are distinguished to represent concepts such as belief information about world and the other agents, communication information, information on observation and action, information on agent's characteristics and information to be remembered. These information types are in turn made of information types at a lower level of abstraction. This model proposes some generic knowledge bases which may be reused in specific applications depending upon their relevance.

GAM enables different agent architectures to be compared at a conceptual but formally defined level. Different processes and structures of GAM can be refined and reused by agent developers for developing their application.

Both types of systems fail to suit the needs of agent toolkit described above. DESIRE gives abstraction of processes and knowledge structures at a very generic level. It definitely helps in the design of agents and saves the need to design from scratch, but it does not provide agent models which could be used for making agent systems. For example, an agent model in DESIRE has *own process control* which manages tasks regarding its process management but it is not clear as what are the different categories of agent models based on various types of *own process control* tasks. Developer of an agent based system has to do all design for the specific functions as well as knowledge structures of agents. Multi-agent systems that are built for a specific problem provide efficient solution for that problem. These systems are unable to provide any kind of abstraction to be used for other similar applications.

We propose an agent toolkit which can give agent prototypes which can be directly reused or refined and used for applications. This toolkit is more specific than a generic agent system in the sense that it will give prototypes of agents specific to a set of closely related applications, e.g. DKN. It is more abstract than a specific agent-based application since its components need to be refined to suit the needs of a particular application. This toolkit will provide functional models for agents which can be used by way of refining or as composition for design of agent based solutions.

In following chapters, we present the design and the implementation of the agent toolkit proposed here.

CHAPTER 3. AGENT ARCHITECTURES

In order to design a toolkit with agents, it is necessary to understand agent components and functionality in detail. So far, we have seen an informal description of what agents are composed of and how they function. Agents have been described using sound formal specifications and these specifications clearly explain agent mechanism. It is not possible to develop any agent-based system without clearly understanding these specifications. In this chapter, we introduce the formal specifications and definitions of the agent and its components.

3.1 Abstract Architecture for Agents

3.1.1 Definitions

An agent is a complex unit made of various components that work in cooperation with each other in an orderly manner. The components may or may not be autonomous. Intelligent behavior in an agent is a result of interaction of these components with each other. It is necessary to understand behavior of these components and their interaction with each other to understand how agents function. In this section, we will focus on formalization of the agent architecture.

Environment is a component of an agent. Agents are situated in the environment and their functionality is modeled to a great extent by the kind of environment. Different types of environment and their significance has been discussed in section 1.2. We represent a specific stage in the environment during a time frame by its *state*. An agent's information about its state of environment is typically represented symbolically. We will assume that states of environment can be characterized as a set S = $\{s_1, s_2, s_3, ...\}$ of *environment states* and at any given time, the agent is supposed to be in one of these states. An agent's environment may

contain other agents also.

Capability of an agent to affect its environment can be modeled by a set of actions $A = \{a_1, a_2, ...\}$. We define these actions to be $effector$ actions if they directly affect the environment where they are executed. Agents can have different ways of choosing the actions to be executed. Some agents perform a great deal of deliberation in deciding which action to be executed while others react instantaneously to input received without going through any deliberation or "thinking".

Now, by using formal representation of actions and environments, we can define agent as a function:

$$agent : S^* \longrightarrow A$$

that maps sequences of environmental states to actions. The non deterministic behavior of an environment is modeled as a function:

$$env : S \times A \longrightarrow \wp(S)$$

which takes the current state of environment $s \in S$ and an action $a \in A$ and maps them to a set of environment states $env(s, a)$ those that could result from performing action a in state s.

Interaction of an agent and its environment is recorded in the form of *history*. History stores agent's actions in various states of environment in a chronological order. It is useful when there is a need for the agent to refer to its previous experience for executing an action. History also provides the agent with data to learn on the basis of which it can make better action choices. It consists of actions and states resulting from those actions over a period of time. A *history* h is a sequence

$$h : s_0 \longrightarrow^{a_0} s_1 \longrightarrow^{a_1} s_2 \longrightarrow^{a_2} \ldots$$

where s_0 is the initial state of the environment, a_u is the u^{th} action that the agent chose to perform and s_u is the u^{th} environment state which results from performing action a_{u-1} in state s_{u-1}.

If $action : S^* \longrightarrow A$ is an agent, $env : S \times A \longrightarrow \wp(S)$ is an environment, and s_0 is the initial state of the environment, then the sequence

$$h : s_0 \longrightarrow^{a_0} s_1 \longrightarrow^{a_1} s_2 \longrightarrow^{a_2} \ldots$$

represents a possible history of the agent in the environment iff the following two conditions hold:

$$\forall u \in N, a_u = action((s_0, s_1, \ldots, s_u)\ and\ \forall u \in N\ such\ that\ u > 0, s_u \in env(s_{u-1}, a_{u-1})$$

The *characteristic behavior* of an agent

$action : S^* \longrightarrow A$

in an environment

$env : S \times A \longrightarrow \wp(S)$

is the set of all the histories that satisfy these properties. If a property ϕ holds of all these histories, then *phi* can be regarded as an invariant property of the agent in the environment.

Let $hist(agent, environment)$ denote the set of all histories of agent in environment. Two agents, ag_1 and ag_2, are said to be *behaviorally equivalent* with respect to environment env iff $hist(ag_1, env) = hist(ag_2, env)$ and simply behaviorally equivalent iff they are behaviorally equivalent with respect to all environments.

Agents are said to be *non terminating* if their interaction with the environment does not end, that is, if their histories are infinite.

There are some problems associated with this method of defining agent architecture. The environment is passive, i.e., we assume that changes in environment are caused only by agent actions and not by any other entity. Simultaneous actions from different agents are not allowed. While considering effect of action on the environment, it is assumed that it is the only action being executed. In order to overcome these problems, definition of action can be modified as follows. Any agent's action has some *influence* on the environment and hence, agent's behavior is changed to:

$$action : S \times A \longrightarrow I$$

where I is union of all influences on the environment.

$$I = I_1 \cup I_2 \cup ... \cup I_n$$

State of environment is collectively changed by all influences instead of being affected by a single action.

$$env : S \times I \longrightarrow S$$

For the purpose of our implementation of toolkit, we have used the traditional definitions of agent, action and environment. It will be interesting to developagent models in the toolkit using concept of *influence*.

3.2 Basic Agent Types

An agent gets its input from the environment and performs some actions but it is not clear what the form of input is or what kind of action is executed and how. It is also not clear as to how an agent interprets its input from the environment and if its actions affect environment always. Depending on answers to these questions, there are different types of agents. In this section, we will discuss formal specifications of various types of agents using formal definitions of the agent and its components.

Our purpose is to give an idea of the various types of agents needed to solve the problem of dynamic and distributed data sources. Hence, we will refine the abstract architecture to derive models which form components of a multi-agent system. An *agent architecture* is essentially a map of internals of an agent - its data structures, the operations that may be performed on these agents and the control flow between these data structures.

We focus on high-level design issues of the subsystems that are essential part of any agent. In the later part of this thesis, we will see the design of agents having more than one kind of functionality.

The agent's environment is composed of a multitude of elements in different forms. Although agents take inputs in a variety of ways from the environment but they are not able to interpret environmental state to filter information relevant to it. Consider a robot situated in

a room. Its environment consists of the floor and objects on the floor. But it cannot get an idea about its environment until it is able to "see" the floor and objects. It needs to view the floor using a camera or get the coordinates of objects and the room. The environmental state should be converted into a form which the agent is able to realize as input. For this purpose, we define a function *see* which captures agent's ability to observe the environment:

$$see : S \longrightarrow P$$

where P is a non-empty set of percept. Here, a percept is in a form that the agent can use as input. This function ensures that the agent gets only relevant information from the elements of the environment. For example, if we want to capture information about activities going on in a network (environmental state), we should provide the agent with the logs of activities involving communication (percept). Concept of percept gives a very clear and formal representation to input to the agent. This function *see* could be performed by a hardware component for agents situated in the physical world, e.g., robots. If the agent is a software agent, percept is usually in the form of some input to the program.

The agent takes its action depending on the percept it receives and we can define agent to be a function:

$$action : P^* \longrightarrow A$$

which maps sequences of percept to actions.

3.2.1 Purely Reactive Agents

The term "reactive" is used to denote the capacity to react to something instantaneously. Purely Reactive agents do not involve the reasoning process. They do not make use of any previous experience either. They have a predefined process to select actions for the percept. Fig 3.1 shows a purely reactive agent. Percepts and their corresponding actions are stored in various forms, e.g., a lookup table or a database. Action function for these agents is defined as:

$$action : S \longrightarrow A$$

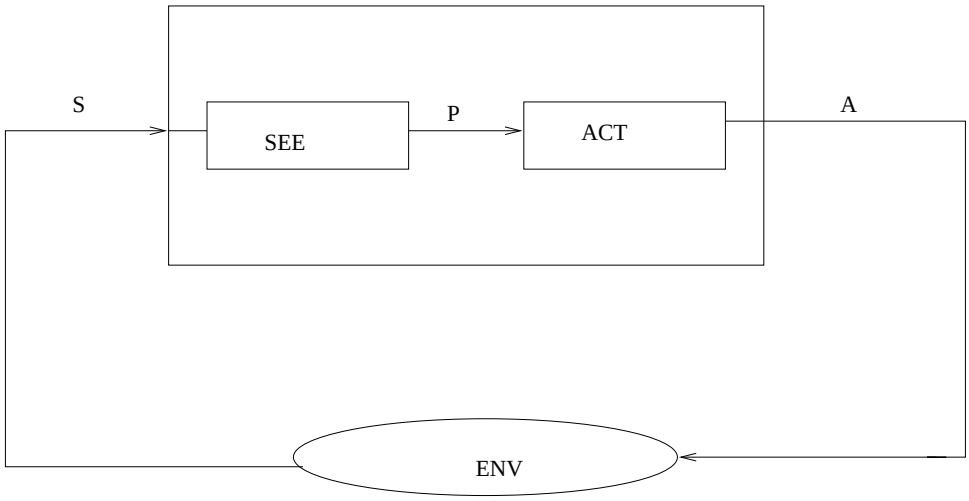

Figure 3.1 Purely Reactive Agent

An obvious requirement is to find out about all possible incoming percepts and provide agents with their corresponding actions. If there is any state of the environment for which no matching action has been given to the agent, it will not be able to react. However, this architecture is very robust as agents do not consume time in the thinking process. It is simple and economic since there is not a time consuming process of deciding upon an action after going through some kind of planning. Programming these agents is relatively simple. A very simple example of reactive agent is a thermostat.

A thermostat's environment can be in one of the two states - either too cold or correct temperature. Then thermostat's action function is:

$action(s)$ = heater off, if temperature OK

and

$action(s)$ = heater on otherwise

Neural networks are examples of reactive systems.

Intelligent and rational behavior in these agents is a result of interaction between the agent and its environment. However, there are certain problems associated with this architecture as well:

• Agents need to have sufficient knowledge of their local environment in order to be able to take any action as they do not employ any model of environment.

• This results in agents having a very short term view as they do not take into account the whole environment while making a decision.

• They cannot learn from their experience since they do not store any kind of information.

• Behavior of these agents is supposed to emerge from the interaction between agent and its environment. This makes it very difficult to build agents with a large number of behaviors. The dynamics of the interactions between the different behaviors become too complex to understand.

3.2.2 Agents with State

Purely reactive agent does not use history or refer to its previous experience. To allow for agents to be able to archive their interactions with environment, agents that *maintain state* are proposed. Ability to maintain state gives them the capacity to remember states of environment and their significant internal states. These agents have some internal data structure, which might be a vector of objects or a set of records etc. It is used to store information about history and environment state. State information enhances the intelligent behavior of agent and gives it access to the global environment so that the actions taken by the agent are not limited to its local environment alone.

As we had in the purely reactive model, we have the *see* function which maps percept to actions.

Each agent chooses its action based on the state it is in. The is no direct relation between percept and action. Percept affects the internal state of the agent and for this purpose we define the next function:

$$next : I \times P \longrightarrow I$$

The action selection function now is defined as:

$$action : I \longrightarrow A$$

which maps internal state to action. Fig 3.2 shows an agent with state.

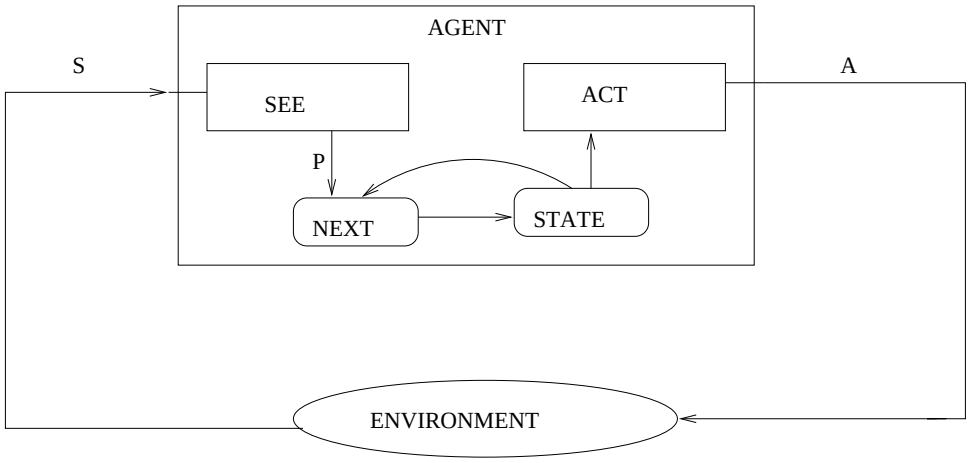

Figure 3.2 Agent with State

The agent starts in some initial internal state i_0. It then observes its environment state s and generates a percept $see(s)$. The internal state of the agent is updated via the next function, becoming set to $next(i_0, see(s))$. The action selected by the agent is then $action(next(i_0, see(s)))$. This action is then performed and the agent enters another cycle, perceiving the world via see, updating its state via next and choosing an action to perform via action.

3.3 More Sophisticated Agent Types

We have defined basic agent types in the previous section. Agents differ in their ways of receiving percepts, maintaining states and deciding upon actions. In this section, various subsystems of agents are refined and categorized further to derive agent models that can perform more than one functions. Their internal structures and operations being performed on these structures are discussed in depth. Focus is on various components of agents as different modules and how these modules can be combined together to get different kind of functionalities. Our purpose is to show that different agent functionalities can work in coordination with each other to provide multi-agent solutions to problems.

We will consider following types of agents:

1. Communicative agent – the agent can interact with other agents via communicative actions such as message passing.

2. Learning agent – the agent learns from its environment and use this knowledge in its activities.

3. Reactive Learning agent – the agent learns while responding to the environment.

4. Reactive Communicative agent without state – the agent performs communicative actions in addition to its normal functions.

5. Reactive Communicative agents with state - the agents perform communicative actions and also maintain an internal state.

3.3.1 Communicative Agents

The purpose of communicative agents is to use their actions to interact with other agents in their environment. They accomplish this by sending messages to other agents and interpret incoming message to understand what the other agent is trying to communicate. Communication can serve various purposes. Roman Jackobson (Jak95) distinguishes six modes of communication:

1. Expressive function: It characterizes the sender's state, beliefs, the state it is in and how it sees things.

2. Conative function: It represents the concept of order or request which a sender sends to addressee.

3. Referential function: This function represents the state of the world or the environment or third party.

4. Phatic function: It is used to establish, prolong or interrupt a communication. Its most important use is to verify that communication channel is working.

5. Poetic function: It is connected with the highlighting of message.

6. Metalinguistic function: It is related to messages, language and communication situation. It is the most important function in multi-agent systems as it allows messages to give information about other messages.

A message performs several functions at the same time. Agents could send any of these messages depending on their need. Agent communication is formally defined by means of Speech-Act theory of communication (Sea69). Speech acts describe all intentional actions which are carried out during communication. It describes different kinds of acts which represent the modes of communication discussed earlier. However, it takes into account only isolated acts and has no relevance to series of interaction or conversation which might occur in the course of a communication between two agents. Conversations are more formally represented by finite state automatons and Petrinets (Fer99). Finite state automatons (FSA) describe conversation as a series of states linked by transitions which are the messages exchanged by agents. FSA are good for describing conversations when they occur as a single process. In a multi-agent system, often agents are involved in more than one conversations and have to maintain them. Petrinets are used to describe these multiple conversations.

Petrinets proposed by Charles Petri in 1962, are a class of formal methods which have been used to describe and analyze distributed systems, programs, protocols, language, etc. A petrinet is represented by an oriented graph consisting of two types of modes- *places* represented by circles and *transitions* represented by bars. Petrinets represent dynamic aspects of a process by moving *tokens* from place to place based on "firing rules". A transition T is *enabled* if all the input places p, for which there exists an arc from p to T possess a token. Firing a transition involves removal of a token from each input place and the addition of a token to each output place. When multiple transitions are enabled, it is assumed that exactly one of the tokens fire and choice of token is random.

While describing agent conversation, each agent is described by a Petri subnet, where the positions correspond to internal state in agent or task in hand. The transitions correspond either to synchronizations due to receipt of messages or to conditions of the application of actions. Messages being routed are represented by supplementary locations which link agents in a way to form a single net. To model several simultaneous communications with several correspondents, colored petrinets (Jen92) are used. In colored petrinets, tokens can be distinguished based on their color. To model communication, structures that represent the conversations are

circulated in place of tokens. Each message is given a number in sequence, corresponding to the conversation to which it belongs to.

Performatives are various types of messages that agents can send and interpret. While modeling communication language in a multi-agent system, a careful decision has to be made regarding the set of performatives and propositional content of the language. The bigger the set of performatives, the more the propositional content of the language can be simplified. Some of the examples of performatives of speech-act theory are as follows:

- Request: requesting other agents to carry out a task, e.g., Request
- Interrogation: asking question, e.g., Question
- Affirmation: asserting that some information is correct, e.g., Assert
- Indication of skills: telling others about the skills agent posses, e.g., KnowHowDo

These agents execute *communicative* actions which are different from the effector actions in the sense that effector actions do not need a specific language, and hence, a way to interpret the language and understand the message content. For an agent to understand the meaning of another's communicative actions, they must either share a common information model or have the ability to transform information between their respective models. Here we refine the set of actions to contain communicative and effector actions:

$$A = A_e \times A_c$$

where $A_e = \{a_{e1}, a_{e2}, ...\}$ is a set of effector actions and $A_c = \{a_{c1}, a_{c2}, ...\}$ is a set of communicative actions. Effector actions affect and change the environment directly where as communicative actions try to contact some element of environment and deliver some message to it which might or might not change the environment. Hence, communicative actions involve coding, decoding and channeling of messages using some language which needs to be specified formally. Fig 3.3 shows a communicative agent.

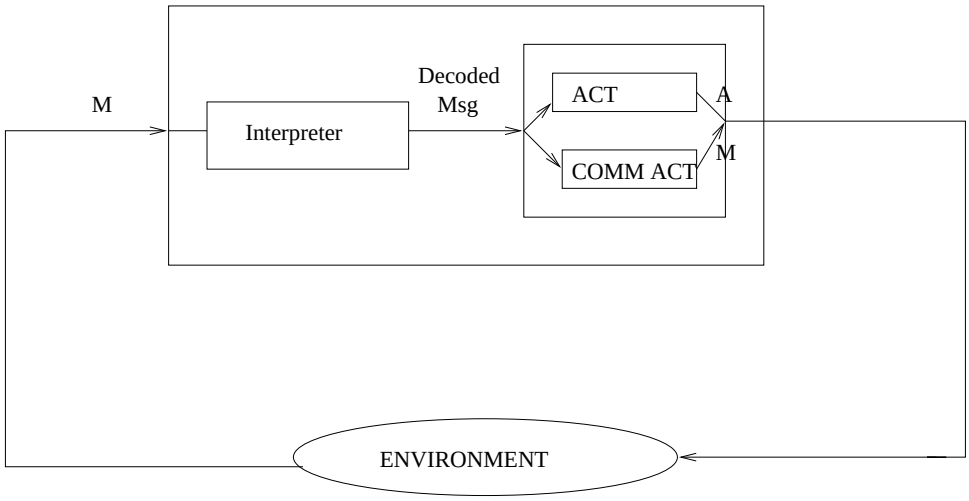

Figure 3.3 Communicative Agent

Communicative agents need to agree upon some language and common semantic interpretation of symbols used in the message which can be understood by the sender as well as receiver. This enables agents within an organization to:

- *Understand* the motive of sender agent (e.g., if the agent wants to "ask" something, "reply" to an already asked question, "inform" something, etc.)

- *Parse and interpret* the contents of a message through a common interpreter or compiler, e.g., Knowledge Query and Manipulation Language (KQML) and Knowledge Interchange Format (KIF) are widely used as agent communication languages and parsers for these are readily available (JAT98) .

  KQML is a language for programs to use to communicate attitudes about information, such as querying, stating, believing, requiring, achieving, subscribing, and offering. KQML is indifferent to the format of the information itself, thus KQML expressions will often contain subexpressions in other so-called "content languages." It gives a wide set of performatives for different purposes such as database, basic query, basic response, multi-response, basic effector, networking etc. A basic performative is "tell" where an agent gives some information to other agent. Semantics of this performative are:

  tell

:content < *expression* > /*information*/

:language < *word* > /*language used*/

:ontology < *word* >

:in-reply-to < *expression* > /*content for which this is a reply*/

:force < *word* > /*if word is "permanent", then sender never denies the meaning of this performative or else it can deny in future

:sender < *word* > /*sender of message*/

:receiver < *word* > /*receiver of message*/

Database performatives allow agents to insert or delete sentences in their belief system. An example is the "insert" performative.

insert

:content < *expression* > /*information*/

:language < *word* > /*language used*/

:ontology < *word* >

:reply-with < *expression* > /*if expression is true, then sender expects a reply*/

:in-reply-to < *expression* > /*content for which this is a reply*/

:force < *word* > /*if word is "permanent", then sender never denies the meaning of this performative or else it can deny in future

:sender < *word* > /*sender of message*/

:receiver < *word* > /*receiver of message*/

There are various types of performatives which help to facilitate communication in a multi-agent system. Detailed specification of KQML can be found at http://www.cs.umbc.edu/kqml.

- Understand the *meaning* of message and take action based upon the content.
- Send message to communicate their *needs, preferences and information* to other agents.

Communication leads to another important issue - that of coordination. By communicating their beliefs, goals and commitments to each other, agents can coordinate among themselves effectively to achieve a common goal. This leads to the possibility of team work among agents.

Language used in communication can be very simple, requiring less effort in parsing and decoding or it can be very complex requiring the use of a parser and interpreter. More complex the language is, the more effort is required in understanding the semantics. Languages like KQML have been developed which are considered very effective for agent communication. Communicative agents can be with or without state, reactive or deliberative. We are going to focus on the communicative part of the agent. It might have *see* function as described above if it is a reactive communicative agent. It can have *next* function depending on whether it has state or not.

Agents need to interpret the language and understand the content of the messages received. For this purpose, we introduce the "interpreter". The message coming from the outside environment as well as from internal components of the agent is interpreted by the interpreter according to the parsing and decoding rules which must be predefined. We define this function as:

$$interpreter : \alpha(M) \longrightarrow M_d$$

Interpreter maps message M to decoded message M_d. Action to be taken depends on the message received. Hence, action is defined as:

$$action : M_d \longrightarrow A$$

Here action can be communicative or effector depending on the content of message. For example, the message could be asking the agent to send some message to another agent, in which case, the agent will execute a communicative action. These messages could also be a result of some internal function. The agent might get a message to execute some function and send the result to the other agent. As a result of getting a message, an agent could be executing some internal function, in which case, it is an effector action.

3.3.2 Learning Agents

By learning, we mean a process by which a system *improves* its performance on a *set of tasks* as a result of *experience*. The idea is that agent should be able to improve its performance as a result of feedback on its actions from the environment. Various types of possible learning methods are:

- Rote Learning: In this type of learning, almost no inference is involved. Knowledge is stored in same form as it is provided, e.g., for game of chess, chess board configurations and their backed up evaluations are stored.
- Learning by Instruction: In this learning, some transformation of knowledge is necessary before it is stored for future use.
- Learning by Induction: Here, given a set of examples, the learner has to induce a set of general problem solving procedures.
- Learning from Reinforcement: This type of learning is done by exploration. Here, use of rewards is made to learn a successful agent function. The agent learns a utility function on states and uses it to select actions that maximize the expected utility of their outcomes.

Learning could affect various components of agent, for example the agent could be learning its actions or it could be learning to solve some problem using its experience as in chess. In the first case, it gets result of performing action from the critic and uses this feedback to learn its actions. In the second case, the agent learns from the reward that it gets for selecting its moves. Learning agents must assimilate the instances from which they learn and the hypothesis learned. Here we use the idea of agents with state. Agents can maintain state to store history which is used for learning. In an environment of distributed data sources, learning data patterns is essential in order to do the data mining.

While designing a learning agent, the most important questions to be considered are

1. How is the agent going to learn some thing?
2. How will the agent know if it is improving its performance or not?

The *learning* element is the component in learning agent which is responsible for learning from instances and thereby, improving efficiency of agent in the area where it is utilizing the knowledge which it has learned. Using its *action* function, agent knows if it is improving its performance or not. The *critic* is the component which gives feedback to agent about its performance as well as makes it aware of the environmental states. In our model, critic is an identity function over percept as percept is what comes from external environment.

Information from critic is fed to the learning element which does learning with percept and builds a set of hypothesis. It is defined as a function:

$$learn : P \longrightarrow \mathcal{H}$$

Fig 3.4 shows a learning agent. Action function in this agent is a communicative action. The agent sends information about the hypotheses it has learned to other agents. It is defined as:

$$action : \mathcal{H} \longrightarrow \mathcal{A}_{\rfloor}$$

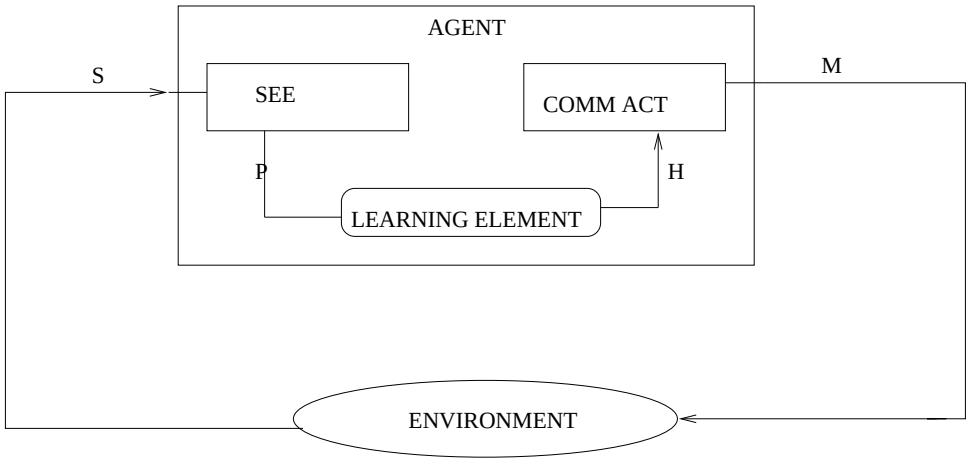

Figure 3.4 Learning Agent

An example of a simple learning agent is an agent that receives images and their explanation in form of percepts and relating these two. This agent is shown in fig 3.5. Each image has a an explanation associated with it. Over time, the agent is capable of constructing the image given the explanation and vice versa. Here, the *see* function gets these percepts and the learning module learns to relate the images and explanations. If needed, the agent transmits

hypotheses it has learned to the outside world.

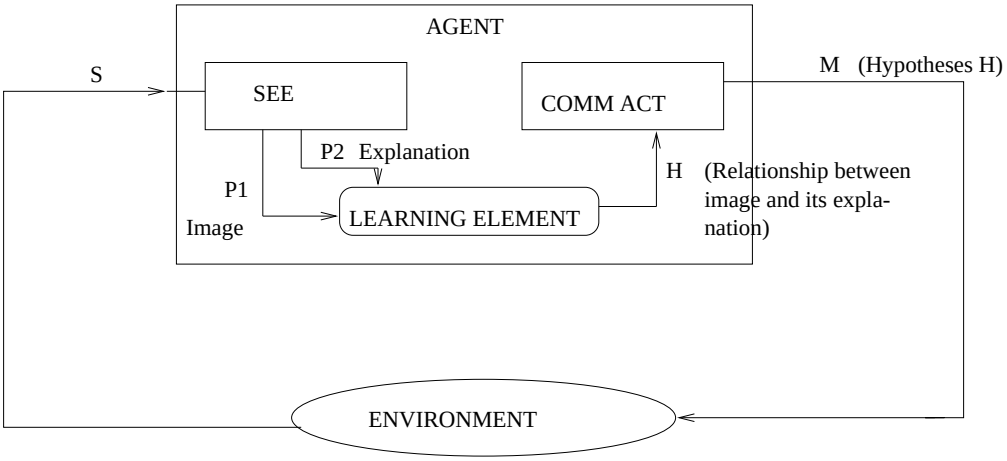

Figure 3.5 Sample Learning Agent

CHAPTER 4. REFINEMENTS OF AGENT ARCHITECTURES

In the previous chapter, we described abstract architecture for agents in general and concrete architecture for some of the basic types of agents. Each model described so far has the capacity to perform a certain kind of function but in real life situations, agents often need to perform more than one type of function. Consider a multi-agent system that collects information from various data sources, assemble the information into some form required by user and also learn from it. The system also answers user queries based on information stored in a local database. In this case, we need an agent architecture that has the capability to react to user's queries along with learning function. If we consider a system where agent has to cooperate with other agents in carrying out some plan, it needs to communicate with other agents. But this is just one function and it does not define the agent completely. The agent also needs to react to its environment in order to carry out its plans. Hence, it is necessary that the toolkit should provide agent models which can be created using different components such as learning and communication.

In this chapter, we describe the need for an abstract architecture for DKN, the motivating application for toolkit in the first part. In the later part, we discuss the general architecture of toolkit with design and implementation of agent models which make the toolkit.

4.1 Need for an Abstract Architecture for DKN

Complex systems are being modeled using powerful abstractions such as procedural abstractions, abstract data types and object-oriented programming. More and more emphasis is laid on re-usability of components. Most suitable abstractions are usually the ones that minimize the semantic gap between units of analysis that are intuitively used to conceptualize the

problem and the elements of solution. This concept is becoming obvious in agent programming also. A generic model is needed that can provide a well-laid foundation for development of multi-agent systems. Instead of designing each and every system from scratch, an abstract model could be used to define basic functionalities of agents. The generic model has to be modular enough so that different components can be put together to serve as complex agents. Re-usability of components enables a developer to expend less effort for designing systems since the same component can be used for different types of agents. However, reuse is difficult to attain unless design and development of components is done while taking into consideration some close-knit range of problems with similar characteristics (WJ98). Hence, we propose our model keeping DKN in mind. Interaction between various components plays a very important role in complex multi-agent systems. Our model provides flexible ways of arranging interactions within the system and agent. In a multi-agent architecture, it is necessary to view components at a single conceptual level and an efficient multi-agent architecture should be able to provide this abstraction.

We propose the design and implementation of an agent toolkit which helps an analyst to go from a set of requirements to a working solution. The purpose of toolkit is to bridge the gap between very abstract agent architectures and very specific implementations of multi agent systems by giving a set of specific agent models which could be reused as they are or can be composed into more complex models. The toolkit has been designed with the following objectives:

(i) keep basic building blocks not too specific for a particular requirement. They are not fine grained and can be refined to suit needs of various applications. They can be composed to form complex individual components whose behavior is context dependent.

(ii) design basic building blocks so they can be used for concurrent and distributed problem solving.

(iii) define interactions in a very flexible way. Different agent models can have different interactions with the environment or within their components, e.g., communicative actions

and effector actions are defined separately and models can use both of these or any of these without requiring a change in their design.

(iv) define non-organizational restrictions. Analysts can arrange agents in any manner they feel appropriate.

The toolkit consists of various agent models which are the result of refinement of general agent architecture discussed in the previous chapter and abstraction of specific multi-agent solutions used in DKN.

Various components of the model are described in the following sections. We describe refinement of the basic agent model used to derive agents which are part of our toolkit. Every agent has a functional model where we describe its interaction with its environment and its internal functioning.

4.2 Reactive Learning Agent

Certain types of agents are required to learn from their history. Their actions are not a direct response to percept but to some extent are based upon the hypothesis they have learned so far. This gives a more natural way of representing agents as entities that can react in accordance with the *experience* they have had of the world and this is the way we see human beings also behaving. Agents are reactive as they do not plan their activities, but act in response to a particular percept according to some pre-defined rule. The basic characteristic of reactive agents is to react to the environment without reasoning about it. If the environment is dynamic and non deterministic, it is difficult to predict every possible state it could be in. Thus, it will be really useful for agents to be able to learn from their experience so that they can respond accordingly, without any kind of external intervention, if the same situation arises again. Learning also provides a way to build high performance systems. Agents could be learning from the reward they get for their actions, they could be learning new actions depending on their previous experience or they could be learning a new hypothesis which they can use in classification function. The learning function may vary from agent to agent in the same environment.

On the other hand, if the agent is only capable of learning and not responding to the changes in environment, it is less useful in a dynamic environment. There is no defined time period for the learning agent to stop learning and use the knowledge it has learned. Instead, it has to be a parallel process of responding to the environmental state changes and learning from it. We need agents which are constantly aware of changes in environmental states, are reacting accordingly to serve the needs of the system, are able to *memorize* the significant states, learn from them and use this knowledge for future.

We will now refine the existing architecture of reactive and learning agents to be able to perform the functionality described above. As explained in section 3.2.1, a *purely reactive* agent is of the form:

$$agent : P^* \longrightarrow A$$

which maps a sequence of percepts to the action. In order for the agent to be able to learn, it needs the *critic* to determine how it is doing and the *learning element* which learns and the *performance element* which selects the action. The *Reactive Learning* agent receives a set of percept from its environment and implements the *see* function.

It has an internal state that maintains the history H which is a set of percept and actions resulting from them. Pair of state and action to be included in history depends on the environment of agent and its functionality. If the agent is learning to classify some data, it will store the instance (state) and the method to classify it (action). If the agent is learning its actions from the utility function, it will store the action and the reward associated with each action.

The internal state of agent is updated in accordance-with a new percept and action resulting from it. We introduce the *history update function* which maps a percept, resulting action and existing history to updated history:

$$historyupdate : P \times A \times H \longrightarrow H$$

Fig 4.1 shows a reactive learning agent. The *critic* in our model is the percept itself which the agent uses to get a feedback about its performance. It is an identity function over percept.

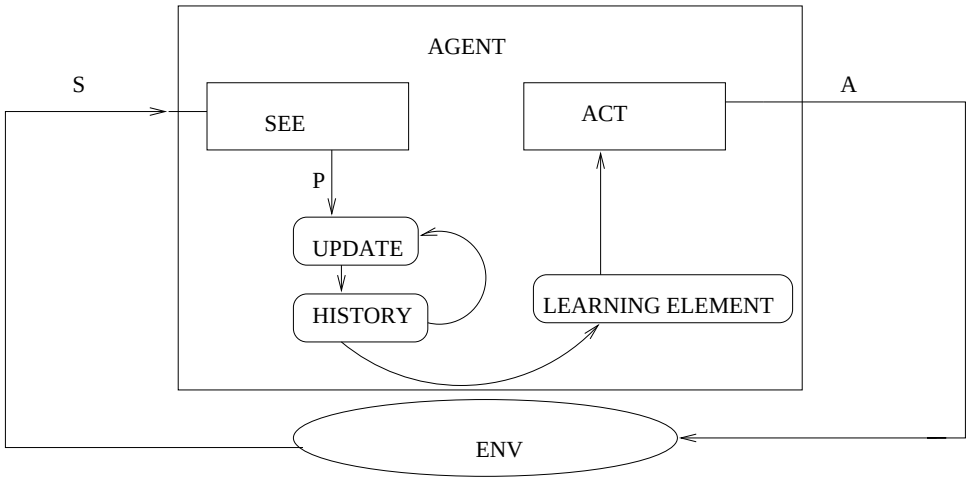

Figure 4.1 Reactive Learning Agent

A hypothesis is an estimate of target function that the agent is trying to learn. An agent may be learning a single hypothesis or a set of hypotheses. We denote a set of hypotheses by $\mathcal{H}$ and an element of that set by h.

We introduce the *learning element* as a function which takes history as input, applies a learning function to it and outputs a hypothesis:

$$learn : \gamma(H) \longrightarrow h \in \mathcal{H}$$

Learning functions can be of various types and depend on the need of multi-agent system. The action function is defined as:

$$action : P \times \mathcal{H} \longrightarrow \mathcal{A}$$

which maps a percept and the hypothesis it has learned to an action. Thus the agent is reactive but uses its knowledge which it has already learned to decide which action to be used.

Here, we have described agents which learn from their history, which is just one instance of learning. Agents learn in different ways, e.g., from set of instances for a classification problem or they could be using reinforcement learning (RN95c) to learn their actions. Analytical learning (Mit97) requires agents to use prior knowledge to reduce the complexity of a hypothesis. More refined models can be developed for allowing different types of learning.

4.3 Reactive Communicative Agent

Reactive agents by themselves do not have the capacity to communicate with other agents. Here, we describe architecture of an agent type that can communicate with other agents while performing an action according to the percept it receives from the environment.

In applications like e-commerce, agents need to exchange messages with each other in order to bid, negotiate, buy or sell goods. They also need to constantly keep a watch on the state of the environment which is in the present case, the market place. The *Reactive Communicative* agent can be utilized for these applications.

We need to refine the way the agent looks at its environment to distinguish between effector percept and communicative messages. The see function is changed to:

$$see : S \longrightarrow P \times M$$

Environmental state is mapped to a combination of Percept and Message. Fig 4.2 shows a reactive communicative agent.

The interpreter function remains the same as in the communicative agent. Actions performed by reactive communicative agents depend on the percept and decoded messages. The agent could be executing effector action or sending out a message. Messages to be sent could be the result of internal functions of agent, e.g., agent has to transmit some value after doing some kind of calculation at regular intervals. The agent could be executing effector action in response to a percept as well as to a message, e.g., it is doing data validation in response to a message received. We define the action function to be:

$$action : P \times M_d \longrightarrow A_e \times A_c$$

The agent uses both percept and message received to decide its actions.

4.4 Reactive Communicative Agent with State

We defined an agent architecture that has functionality to react and communicate. The above architecture is good if the agent is not in a very dynamic environment. It can not

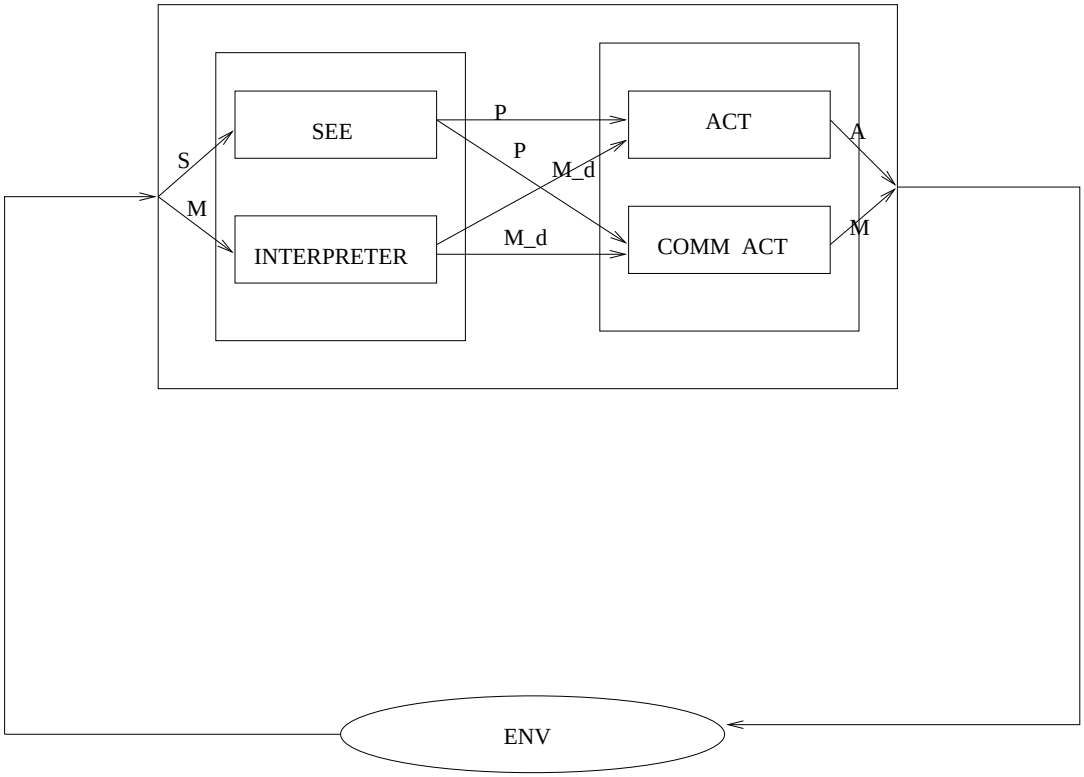

Figure 4.2 Reactive Communicative Agent

possibly remember anything or learn anything from its previous experience. To give more robustness to this agent, we introduce reactive communicative agents with state. As seen in agents with state, any kind of information necessary can be maintained in the internal state of agent. In addition to the functions described in reactive communicative agents, this agent has state update function to maintain and update the state:

$$next : P \times M_d \times I \longrightarrow I$$

Unlike the next function described previously, state gets updated not only due to percept but messages are also taken into account. Fig 4.3 shows a reactive communicative agent with state.

The action function also changes as the agent looks at its internal state and current percept (message) to decide what action it should execute. We define action function as:

$$action : P \times M_d \times I \longrightarrow A_e \times A_c$$

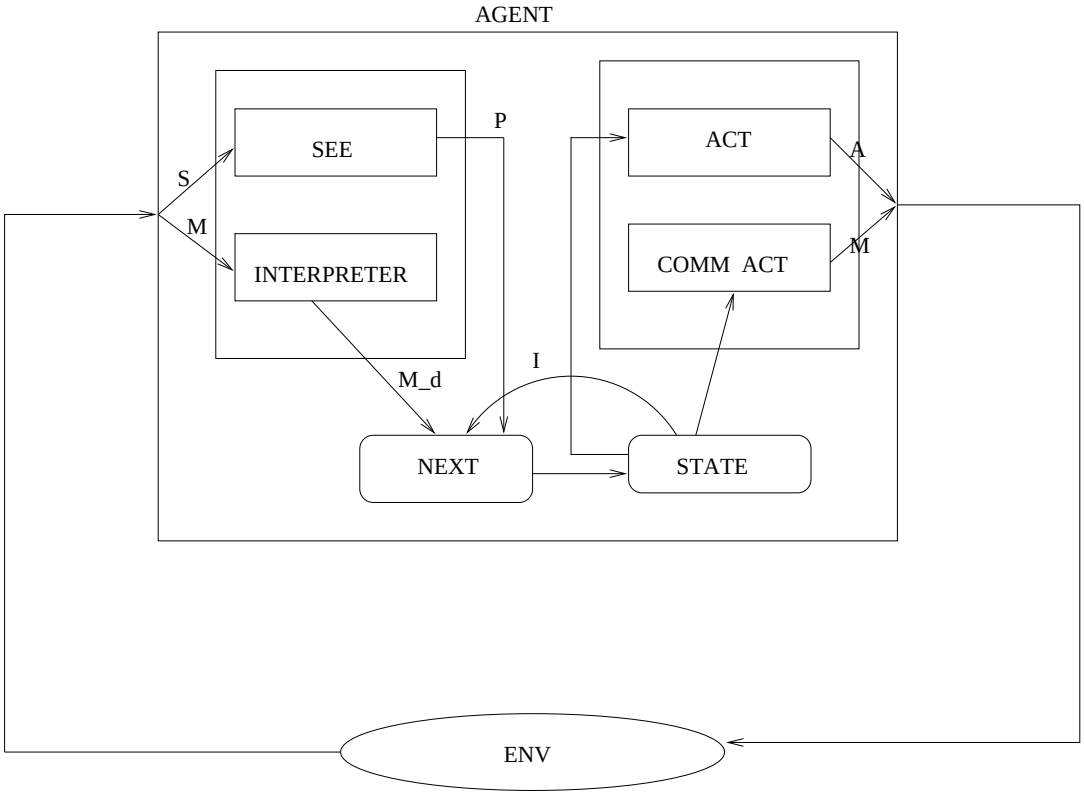

Figure 4.3 Reactive Communicative Agent with State

4.5 Reactive Communicative Learning Agent

This agent is a simple composition of reactive, communicative and learning agents. Here, specifically we refine the internal state of agent. The function *see* and the component *interpreter* remain same as in reactive communicative agent. History update function updates the history based on percept and messages. It is defined as:

$$historyupdate : P \times M_d \times H \longrightarrow H$$

Fig 4.4 shows a reactive communicative learning agent.

Learning function remains the same as in the learning agent described in section 3.3.2. Action function is a mapping from percept, message and hypothesis learned to action.

$$action : P \times M_d \times \mathcal{H} \longrightarrow \mathcal{A}_\urcorner \times \mathcal{A}_\lrcorner$$

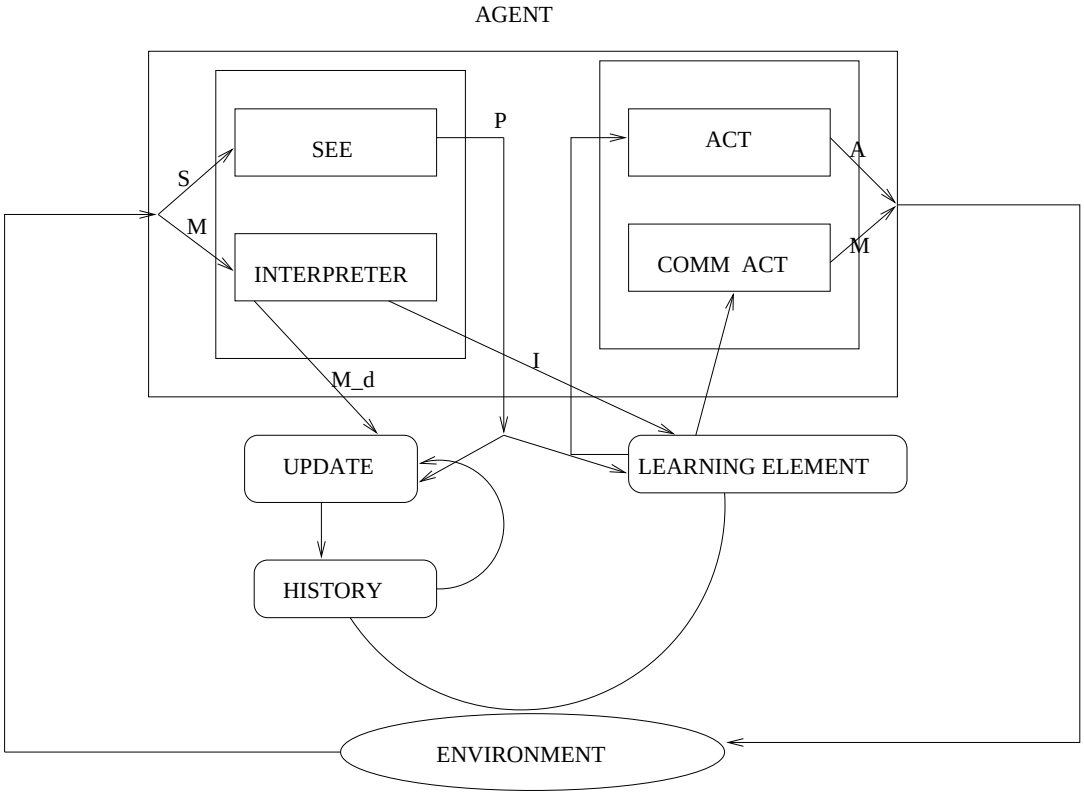

Figure 4.4 Reactive Communicative Learning Agent

4.6 The Agent Tool Kit

In this section, we will describe the general architecture and the implementation aspects of the Agent Toolkit discussed in the previous section.

The toolkit is developed using a mobile agent infrastructure. A mobile agent is defined as a named object which contains code, persistent state and a set of attributes (e.g., movement history, authentication keys) (Whi97) and can move about or transport itself from one host to another as needed for accomplishing its tasks. They support ongoing interactions between programs without the need for ongoing communications by creating proxies for the programs. Thus, mobile agents provide an efficient framework for performing computation in a distributed fashion at sites where the relevant data and computing resources are available instead of expensive transport of large volumes of data across networks. Mobile agent infrastructures facilitate creation and use of mobile agents by providing: a host independent execution platform

for mobile agent programs; communication facilities so that agent and servers can engage in dialogue; and support for creation, deployment, transportation, management and registration of mobile agents and other book keeping functions; support for secure and authenticated access to server resources, secure auditing and error recovery mechanisms. Several mobile agent infrastructures have become available over the recent years.

Voyager is a mobile agent infrastructure implemented in Java (Voy). It supports both mobile and stationary agents. A mobile agent in Voyager is simply a special kind of Java object that can move from host to host and execute its code on different hosts as it moves. Voyager provides a compiler for transforming a Java program into a mobile agent. Mobile agents can be executed on different hardware and software platforms that can run the Voyager mobile agent server. It also supports directory services and several forms of messaging services. Keeping the needs of the agent toolkit in mind, Voyager is used as the infrastructure for developing our agent toolkit. Section 4.7 describes all the features of the toolkit and explains with illustrations how various features can be used to construct agent systems. It also describes creation of complex agents using toolkit and interactions among them.

4.7 Design of AgentToolKit

So far, we have seen the theoretical basis for various agent models in our toolkit. In this section, implementation aspects of the toolkit have been described. We explain general architecture of toolkit and organization of agents followed by explanation of various classes that implements the functionalities of various agents.

Development of the toolkit is done in an object-oriented fashion. This gives a modular approach to design. Various components can be reused or composed into complex components, and hierarchy can be established among components. The toolkit consists of various classes which correspond to architecture of agents described in previous section. There are some basic classes which provide primitive functionalities and complex classes with added functionality are derived from them.

4.7.1 Architecture of Agent Toolkit

The basic purpose of providing the toolkit is to make agent models described in previous sections available for use in development of multi-agent systems. Models with complex functionalities are constructed by refinement of simpler models and by composition of functionalities from different models. All agents are registered with a directory as they come into existence in an organized manner. They have the capacity to find other agents with specific capabilities if the latter are registered with the directory. The simplest model of the agent implements the functionalities of contacting and using a directory system and setting expertise for itself. All agent models are mobile. Fig 4.5 shows the directory system in the agent toolkit.

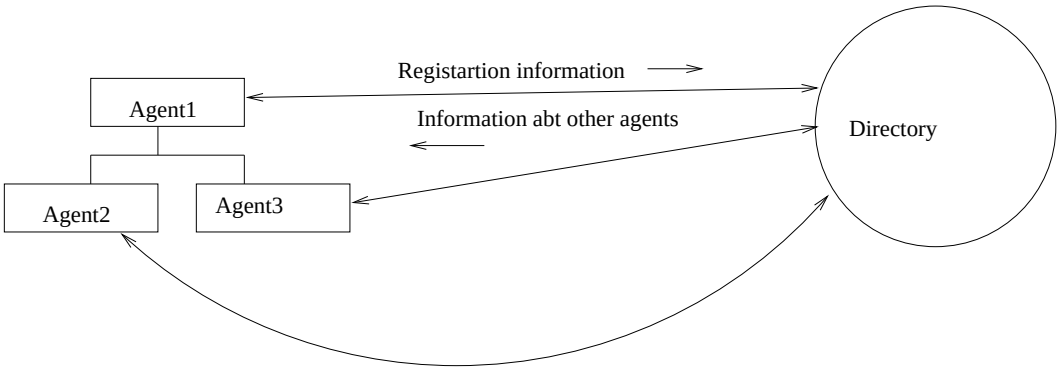

Figure 4.5 Directory System in Agent Toolkit

From this simple agent called the basic agent, models are derived for reactive agent, learning agent and communicative agents. These models make the building blocks of our toolkit. They provide the generic functionalities associated with their class. For example, the communicative agent provides the facility to send and receive messages. Message content depends on the requirements of specific application. The learning agent provides the functionality to read and load data from a text file, normalize the data and create variable classes for various attributes that describe instances in data. Basic models in the agent toolkit are shown in fig 4.6.

Furthermore, agent types are developed by refining these models. The *Reactive-Communicative*

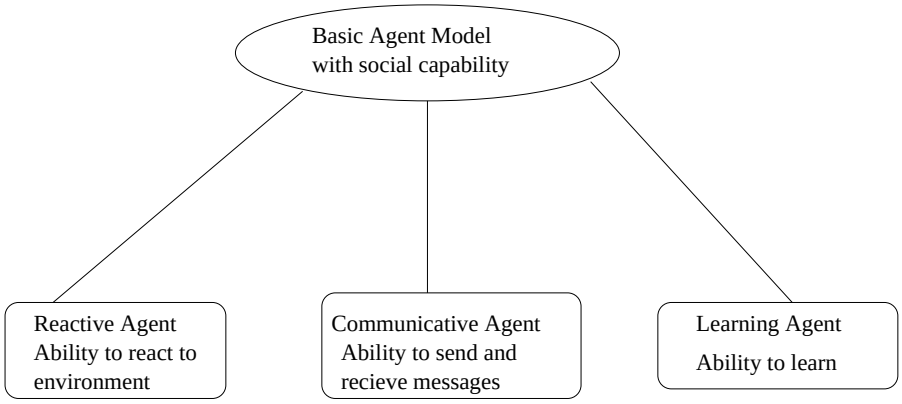

Figure 4.6 Basic Models in Agent Toolkit

agent is built using composition of reactive and communicative models. In this model, as a result of response to some percept, the agent should be able to send messages which is different from being only a reactive agent. Hence, the reactive function is refined so that the agent can send a message, if needed. The *Reactive Communicative agent with state* is developed by refining the Reactive-Communicative agent to have state information. Agents maintain various types of state information and in various forms. In this case, the internal state information needs to be updated and hence, the *see* function is refined to direct the percept to state update function. The *Reactive-Communicative-Learning agent* is developed by composing functionalities of the Reactive-Communicative agent with state and the Learning agent. The agent needs to take into account the current percept and hypotheses learned to execute action. Hence, in this case, the action function of the Reactive-Communicative agent is refined to take into consideration the hypotheses learned while deciding the action. Hierarchy of various models in the toolkit is shown in fig 4.7.

The *Learning agent with state* maintains state information and percept is redirected to state. It is derived from the Learning agent. The *Communicative Learning agent with state* is built from the Communicative agent with state and Learning agent. Here again, the agent needs to maintain and update the state information, and therefore, messages are used for this purpose and actions are decided based on current percept and hypotheses learned. For all

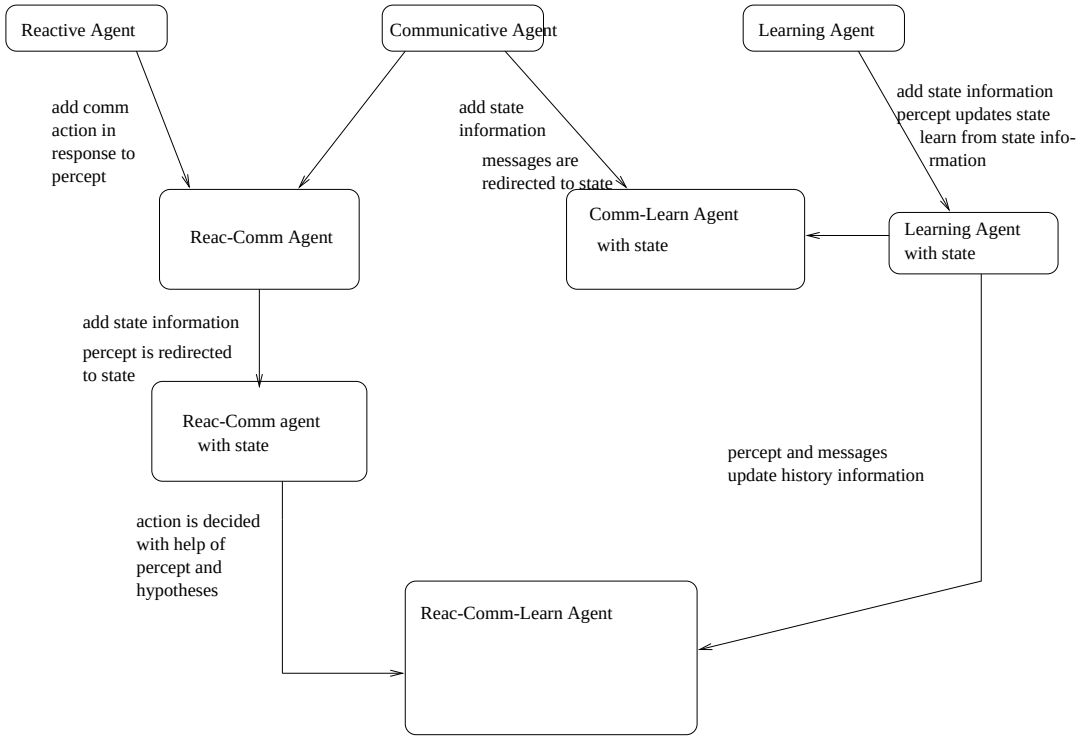

Figure 4.7 Agent Toolkit

agent types involving learning agents, we have provided a learning agent model with generic functionalities to load data and normalize data from text file where data is stored in a certain format. Functions of this class can be used in combination with any other class to affix learning capacity to it. For example, for a Communicative-Learning agent, the communicative agent with state uses functionalities of the Learning agent model.

Thus, the toolkit provides agent models with generic capabilities for their class. These models can be then refined according to the need of a multi-agent system. Next section describes specific functions of these models.

4.8 Agent Models in Toolkit

In this section, the implementation level details of the various agent models discussed in the previous section are provided. Implementation is done using JDK 1.1.7.

4.8.1 Basic Agent

This class encapsulates functions of the basic agent discussed in the previous section. It has functions for registering the agent with the directory, maintaining information about other agents and mobility. Various functions used by this class are:

- Register with the directory: All agents register themselves with a universal directory when they come into existence. This directory is a repository of all agents and description of their expertise. The function is defined as:

  public void registerAgent(String name, String desc, IDirectory dir)

  Input:

  name: Identity of agent

  desc: Description of agent's expertise

  dir: The directory with which agent is getting registered.

  This forms the social model which has details of other elements that the agent can use when needed.

- Agent can find out from the directory if there is an agent with a particular expertise or can find the agent with a particular name using:

  public String findAgent(String agentName, IDirectory directory)

  Input:

  agentName: Name of agent

  directory: The directory where the agent is registered.

  Output:

  String: Identity of the agent if it exists or else null

  It gets information about an agent having identity as *agentName* from the *directory*.

- Maintenance of information about agents: Each agent maintains a list of agents it frequently contacts. This function reduces the need to contact the directory every time an

agent needs to communicate with another agent. Whenever an agent is looked up in the directory for the first time, it is added to this list. For this purpose, we have defined another class called *AgentListElement* which provides functions for maintaining the list such as updating description of agent, adding and removing agents from list etc.

Functions defined for AgentList maintenance are:

public void updateAgentInfo(String agentName, String description)

Input:

agentName: Identity of agent

description: Description of agent

This function updates information about the agent with identity as *agentName*. This function is used by an agent to update its information about other agents.

public void changeAgentInfo(String agentName, String desc)

agentName: Identity of agent

description: Description of agent

This function changes the information maintained by an agent about other agents. It finds the agent with identity as *agentName* and changes its description to *desc*.

- Expertise of agent: All agents have their expertise defined by the user who creates them. It is stored as an attribute of an agent when the agent is created for the first time.

- Biography of agent: All agents maintain an internal biography where they store the information about activities which reflect their performance over a period of time. For this purpose, the programmer should overwrite the function:

  public void maintainBiogr()

  This is different from internal state maintenance in agents as it does not affect the action which agents take.

4.8.2 Reactive Agent

The Reactive agent model is inherited from the Basic Agent model. It has an action function that is over-written by the programmer according to their specific needs. As we described in section 4.3, the *see* function is defined as:

$$see : S \longrightarrow P \times M$$

The agent has to be able to detect regular percept and messages. Percept will be different for different applications. For example, for an application that is meant to serve user requests and collect data, new data arrival is percept and a user request is message. For a flood management system, increase in water level is a percept and informing users about possible danger is the message. Reception of percept and decoding it is done according to specific needs of the multi-agent application.

4.8.3 Communicative Agent

The communicative agent encompasses all functionalities of the basic agent. This agent implements communicative functions to send and receive messages. As seen in section 3.3.1, the agent receives messages from environment and requires to parse and interpret it. Depending on the content of this message, it takes an effector or communicative action. In this class, messages are sent in the form of a vector of objects. Messages are decoded by getting the contents of vector. Selection of objects depend on the language being used. For example, in a simple language, the first object can be a message number or a count of a number of objects in vector, the second object can be a string related to the message, etc. If messages are sent in KQML format, these objects can be a string containing the KQML performatives.

For communication, we made use of remote messaging provided by Voyager. Voyager allows creation of proxy for a particular class and invokes its method by some other class. Method calls made to a proxy are forwarded to its object. If the object is in a remote program, the arguments are serialized using the standard Java serialization mechanism and are de-serialized at the destination. The morphology of the arguments is maintained. By default, parameters are passed by value (Voy). Using this, function for communication is

public void getMessage(Vector input, String identity).

Message is contained in input and identity is for identification of the sender. Sender of the message calls this function on receiver, i.e., if class A wants to send a message to class B, class A invokes the call, "B.getMessage(param1, param2)". Class A sends its identity with the message and thus, class B knows whom to return the message to. Vector contains the message object.

The agent is able to concurrently handle the message communication and sense the percept. While it is carrying out some action and if a message comes, the ongoing action is completed first and the message is handled. These messages are synchronous message, i.e., when the sender sends any message, it blocks until the message completes and the return value, if any, is received.

In order to send a one-way message, where the sender does not block and waits for a reply, users can use **OneWay.invoke()** which is a function provided by Voyager. It returns a Result object which holds an exception if any error occurs.

In order to execute the action part, we need an *interpreter* to interpret the message. If the users prefer to use their own communication language, they have to provide their interpreter. We have given the facility to use KQML messages and for that we have provided the KQML parser used by JATLite (JAT98). All the functions provided by this parser can be seen in the documentation of the toolkit.

Depending on the message and percept, action is executed. Communicative actions are coded by the user in the **getMessage()** function where message is decoded and the action is decided based upon the message.

4.8.4 Learning Agent

The learning agent need to learn depending on the requirements of application. We try to give some generic functions which are useful in the case of most learning algorithms. Some of these functions are adopted from (BB98). For example, functions to read the data set from the text file, to create variables corresponding to various attributes, normalize data etc. For

this purpose, we provide some classes and their description is given below.

4.8.4.1 Variable Class

This class creates variables corresponding to attributes in a dataset. Attributes can have different values and this class stores all those values in the form of a vector. It provides functions for setting and retrieving labels for categorical data, i.e., for storing values that a particular attribute can have. Classes for attributes taking different kinds of values like continuous values, discrete values and categorical values are derived from this class.

4.8.4.2 Discrete Variable Class

Attributes having discrete values are stored using instances of this class. While reading data, it makes a list of values that the attribute can have using function:

public void computeStatistics(String inValue)

Input:

inValue: value of the attribute for which the variable is created

It checks if the value is already present in the list of values and adds if it is not there.

Once all the data has been read, function for normalizing the data is:

public int normalize(String inValue, float[] outArray, int inx)

Input:

inValue: value of the attribute for which the variable is created

outArray: Array to store the normalized value

inx: counter indicating place in array to store the normalized value

Output:

int: counter inx incremented by 1

This function finds the index of *inValue* in a list of values maintained for the variable and translates it into one of N value code where N is the number of values that the attribute can possibly have.

4.8.4.3 Continuous Variable class

This is also derived from Variable class. Variables for attributes having continuous values are stored using instances of this class. For normalizing data, the function is:

public int normalize(String inValue, float[] outArray, int inx)

Input:

inValue: value of the attribute for which the variable is created

outArray: Array to store the normalized value

inx: counter indicating place in array to store the normalized value

Output:

int: counter inx incremented by 1

Function **public void computeStatistics(String inValue)** sets the minimum and maximum value of data. In this case, data normalization is done by finding the minimum and maximum value in data and dividing invalue by difference of these two values.

4.8.4.4 Dataset class

Instance of this class stores a complete dataset. The dataset class reads a dataset from a text file having data in the form of instances associated with their classifications. Values of attributes in instances are separated using a single space and instances are separated using line break character. This class stores records in data in the form of vector and maintains variable list in the form of hashtable. Datafile definition has to be loaded prior to loading data.

Data definition file contains rows of attribute type and attribute name. Variables corresponding to these attributes are created using this file.

Function provided to read data file definition is:

public void loadDataFileDefinition()

This function reads data line by line from the input file and checks for the type of attribute. Depending on attribute type, it creates a variable, e.g., if the attribute type is continuous, it creates a continuous variable.

Function provided to read data file is:

public void loadDataFile()

This function calls loadDataFileDefinition() to create variables for various attributes. It reads each line from the data file and adds a record to vector corresponding to each line. It uses the **computeStatistics()** function to record all values that an attribute can have in case of categorical data and to find minimum and maximum value of data in case of continuous data. It normalizes data after loading it completely.

4.8.5 Reactive-Communicative Agent

This agent is derived from the Reactive and Communicative agent. It has functions from both the classes. However, the enhancement that needs to be done is the provision for communicative actions in response to percept has to be given. This enhancement is done while designing the action function for Reactive agent.

4.8.6 Communicative-Learning Agent

This agent is derived from Communicative and Learning agent with state. Here, the incoming messages are used to update state information. In the getMessage() function, decoded messages are used for state maintenance purpose. Learning is done using state information and in the action function, the learned hypotheses and incoming messages are used to decide about the action to be executed.

4.8.7 Reactive-Communicative-Learning agent

This agent is derived from Reactive-Communicative agent with state and Learning agent with state. It implements functionality of both these classes. In this case, percept is used to update state information in agent. In order to use this agent, it needs to be derived from the above said classes, to use incoming messages and percept to update the internal state information. In the getMessage() function, after decoding the message, it should be used to update the internal state if needed. Learning algorithm can be used for learning from the state information. Actions to be executed are decided after taking into consideration the percept,

message and learned hypotheses. This agent can have a monitoring function as it is reactive as well as learning. Monitoring functions are a part of the reactive and communicative agent's action. As a response to a percept or message, the agent can detect if there is any emergency situation and send a message to the appropriate authority.

In this chapter, we have described various models used in toolkit. Most of the agent models that are part of the toolkit are reactive. It will be interesting to design agent models that can use deliberation to decide their actions. These agents can use planning to acheive their goal. It will also be useful to use the concept of *influence* in designing agent models.

CHAPTER 5. APPLICATION OF THE AGENT TOOLKIT

In this chapter, some practical multi agent solutions are described which have been implemented using Agent Toolkit. These applications were developed for problems which belong to the domain described in chapter 1. The first application is a multi-agent system for a twenty-four hour call center.

5.1 Twenty-four Hour Call Center

There is an increasing need for twenty-four hour customer service in the consumer industry. More and more companies are competing to provide the best customer service possible and a twenty-four hour call center is a valuable service which can enhance the level of customer satisfaction and hence, the company's dependability. All calls and requests made by people are not complex, some of them are fairly simple and can be answered by some automated means instead of using expensive man power. Some of the calls cannot be answered by automated means, e.g., if someone wants to know what kind of financing will be most appropriate for his needs. Every person has different sources of income and variety of expenditures. Such calls need human expertise. One possible solution to increase the level of service is to automate the process of answering users queries. We present a multi agent solution to automate the call centers. This solution helps to manage human resources economically, that are more expensive than machine resource. Humans can not continuously work round the clock while machines can. The Call Center problem and its solution has been addressed in (BJJT98). We have developed an application in similar way for this problem using the Agent Toolkit described in section 4.6 . The purpose is to demonstrate use of toolkit for designing and developing multi-agent systems. From now on, we will refer the multi agent system developed for the call

center using Agent Toolkit as Call Center System(CCS). CCS is developed for use in banks.

5.1.1 Organizational Solution

This section gives a description of the organizational solution for call center. It is necessary to understand the organizational solution in order to develop software based solution.

Clients' requests and questions can be divided into three categories:

(i) simple questions that can be answered directly by a person with limited knowledge of processes used in the industry, i.e., in a bank, request about rates of savings account and checking account or loan rates.

(ii) simple queries which do not require user interaction but need further processing, i.e., request for checks.

(iii) complex requests which can be solved only with human expertise.

Simple requests form 70 percent of total calls. For these, trained operators are used who are not an expert in the field. This helps reduce need for client-advisors who are knowledgeable about terms and the procedures used in the industry and they can focus more on complex requests from clients.

Most of the banks have this process automated and the procedure employed is:

1. The client makes a call to the call center. If the call is relatively simple, it is answered by the computer system at the call center.

2. If the request is complex, it is forwarded to the computer system at the local bank with a request for service.

3. The computer system at the local bank determines if serving the request is feasible taking into account various things such as availability of client advisors and gets back to the operator who forwarded the request with an appropriate response.

4. The operator informs the client and conveys client response to the local bank.

The process flow in CCS is based on the organizational solution described above.

5.2 Architecture of Call Center System

The problem is clearly a distributed problem. One call center serves several clients and employees at a local bank. Client-advisors are free to change their schedules according to their priority. Each of these entities, the call center, the client-advisor, the local bank are autonomous and distributed. They are completely responsible for their own process management and their interaction with the world. They are running as independent processes interacting with each other through communication channels whenever needed.

We introduce three types of agents to form CCS. Corresponding to the operator at the call center, there is a Call Center Agent(CCA). It is not possible for an employee to be present 24 hours a day and hence, we introduce a Personal Agent(PA) to represent the employee. Personal agent maintains schedules of an employee. An employee can change its schedule using the personal agent. Local bank is implemented in form of a Work Manager(WM).

5.2.1 Description of Components of CCS

Client: A client is a user who wants to get some information from the call center. In this case, we are dealing with clients who wants to know about policies of banks and want to take a loan from the bank. We are not providing the facility for accessing account of any customer in the bank from the call center. Clients deal with CCA regarding all their request and have no other access to any other component in the callcenter.

Call Center Agent: Requests which are relatively simple and straightforward are answered by CCA. It is provided with a knowledge base of simple queries and their answers. Complex requests which require some kind of processing or human intervention is forwarded to WM, which in turn, returns appropriate response to CCA which is conveyed to the client. The client makes final decision about things like appointment with a client-advisor. Before this the client has to give details about his preference for the appointment.

Work Manager: WM interprets the requests send by CCA. It sends the request to appropriate department for further processing if needed. If the request is regarding making an appointment with a client-advisor, WM contacts the Personal Agent of employees with rele-

vant expertise and tries to schedule an appointment. It informs CCA if it is able to schedule the appointment or not.

Personal Assistant: PA is model of assistant for an employee. It maintains the employee's schedule. The PA informs the WM to what extent it is possible for the employee to reschedule his agenda. An employee can change his schedule according to his preference and the PA informs the WM of any such change. Depending on the change, the WM may have to reschedule an appointment for a client. If the employee becomes sick and is unable to keep an appointment, the WM tries to reschedule it and informs client through the CCA.

Employees: Employees regularly check their agenda and make changes to it according to their needs and preferences. They execute activities as described in their schedule.

5.2.2 Interaction Among Agents

In this section, we describe each of the agents and their interaction with their environment in detail.

There is a strong need for communication among agents for this multi-agent system to work. Various communication channels in CCS are:

(i) between PA and WM

(ii) between CCA and WM

(iii) between CCA and Client

(iv) between PA and Employee

As discussed in section 3.3.1, a common language and protocol is needed in organizational context. Here, we have used a language in form of objects. Agents communicate with each other using java objects. A vector of objects is passed between agents. The first object of the vector is always an integer. This number indicates the request type which can be a request for a loan appointment when message is passed from the CCA to the WM. When the WM sends result back to CCA, number corresponds to request sent by the CCA, i.e., if request type from the CCA was 3, the corresponding number in reply is 3. Depending on the request

type, each agent expects a set of objects in the vector and decode those objects accordingly. For each request type, the agent behavior is defined. For example, if the request is for an appointment with a client-advisor expert about house loan, other objects in the vector consist of username, date on which he wants the appointment and time slot. The vector also contains the identity of the CCA. Communication between an employee and its PA is relatively simple as human intervention is involved here. Employee can change the schedule according to his preference and view his current schedule with help of PA. Similar is the case between CCA and client. CCA takes information from client regarding his requirement, preference and identity. It informs the client of the appointments scheduled. Interactions among components of the call center are shown in fig 5.1.

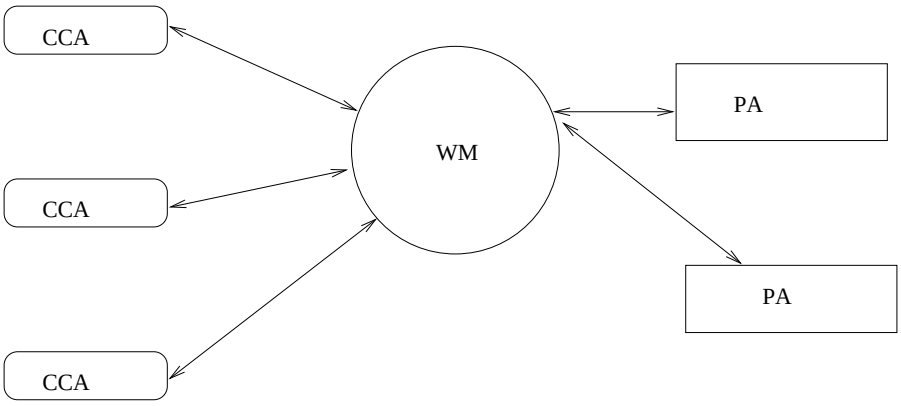

Figure 5.1 Interactions Among Components of Call Center

In order to provide the best service level to clients, the WM needs to cooperate with Personal Agents to come up with a schedule that satisfies the client's requests and efficiently uses the employees time as well. Various models of cooperation are possible between agents using contract net, negotiation, etc. We are using Jenning's (Jen95) model of cooperation. In this model, agents are able to organize and monitor projects to achieve given goals. There is an organizing agent which first identifies a set of activities needed to reach a given goal. Then it contacts agents that are capable of doing those activities. Taking into account the capability of other agents who are available and willing to participate into these activities, it comes up

with a project team and schedule. Then it distributes the schedule to the team members. The creation of this schedule is an iterative process involving interaction between the organizing agent and the team members. Once the schedule is distributed, all agents are committed to the time schedule. They also monitor the progress of the project and report to the organizing agent if they are unable to fulfill some commitment. Based on this model, the WM is the organizing agent. When a request comes up for an appointment with a client advisor, the WM contacts the agent or agents with that particular expertise, i.e., if the appointment is for house loan, it contacts client-advisor expert with mortgages. The WM gets schedules from all the personal agents and tries to schedule the new appointment according to availability of agents. Once an appointment is fixed, information is send to that personal agent. Personal agents inform the WM if there is any change in the pre-decided schedule of employees for any reason. The WM tries to reschedule the appointment in this situation.

In this section, we described the interaction among different components of CCS and now we will see design and implementation of each component in detail resulting from refinement of components in toolkit.

5.3 Implementation of CCS Components

5.3.1 Call Center Agent

The main functionality of CCA is to respond to clients' queries and communicate with WM regarding queries it can not handle. Essentially it is a reactive communicative agent and hence, it is implemented as refinement of the Reactive Communicative Agent from the toolkit. Its environment consists of clients and WM. Its knowledge base consists of answers corresponding to simple queries and procedures corresponding to simple queries which need input from the user. Refinement of the agent from the toolkit is done in following ways to implement a functional CCA:

1. For CCA, percept is user's query. In order for CCA to sense its percept, we designed a GUI called CallCenter-GUI (CCGUI) that allows a client to interact with it.

2. Reactive actions of CCA are defined.

3. An interpreter is provided for the language used by agents in the system.

5.3.1.1 CCGUI

Main interfaces provided by CCGUI are for loan rates and making appointment with the client-advisor. These interfaces are provided in the form of menu options. Client can see different loan rates by clicking on the menu options for the rates. When the client wants to have an appointment with a client advisor, by clicking on the appropriate menu option, a dialog box appears where the user provides details about his identity, preference for the time slot and the date on which he wants the appointment. The CCGUI displays the time, date, and client-advisor name if the appointment is scheduled or else it asks the client to choose another day or time. This refinement is for percept only. Dealing with communicative actions is a function of the CCA and is described in the next section. Fig 5.2 shows the interaction between the CCA and the WM.

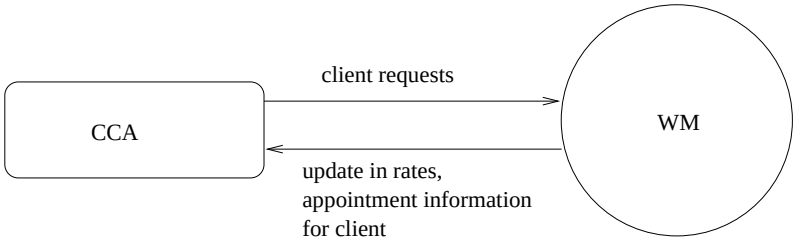

Figure 5.2 Interaction between the CCA and the WM

5.3.1.2 CCA

The CCGUI provides interface for Percept reception of the CCA. Since, the CCA is a reactive communicative agent, it receives communicative input also. Communicative actions are implemented by overwriting the **getMessage()** function of the basic agent. Agents send a vector of objects as their message. The CCA gets messages from the WM which are either in response to its request for some client appointment or about changes in rates. The first object in the vector is for request type. Request types 1, 2 and 3 indicate response to client

appointment requests and in such cases, subsequent objects in vector are identity of the WM and name of the personal agent with whose employee the appointment is scheduled. For request type 4, the subsequent objects are strings describing various loan rates. Responding to user queries is the reactive part of the CCA. When the user enquires about rates, the CCA fetches information from its local database and displays.

In this case, corresponding to the action function defined in section 4.3, percept is the user query, decoded message is message received from WM, communicative action is passing messages to client and WM and reactive action is to respond to user query.

5.3.2 Work Manager

WM's main functions are:

(i) Receive requests from CCA regarding appointment with client-advisors - Communicative action

(ii) Receive schedules from PAs and fix new appointment -action in response for request of schedule

(iii) Maintain a record of PAs and CCAs in the organization -state maintenance action

(iv) Reschedule the appointment if a PA fails to keep its commitment -action in response of communication

(v) Record any change in rate of loan or interest rate - reactive action

(vi) Inform all the CCAs of any change in loan or interest rates - communicative action

These functions indicate that the WM is a reactive communicative agent with state. The WM is derived from the Reactive Communicative Agent with state keeping the reactive part non-functional. The language used for communication is same as in the CCA. The messages are passed in the form of a vector of objects. The message received from the CCA has request type as the first object. It corresponds to different types of requests, e.g., request for appointment with a student loan advisor, request for appointment with a house loan advisor

etc. Subsequent objects establish the identity of the user and his preferences for time and date of the appointment. For any of these requests, the WM checks if the PA with relevant expertise is having a free slot. If it does, an appointment is scheduled. A message is send to the CCA with details about the appointment and if an appointment is not fixed, another message indicating failure is send.

The WM reacts to changes in loan rates and interest rates by sending messages to all CCAs regarding them. Percept is the change in rate of interest. The WM receives this percept using a GUI.

The WM maintains its state in the form of a list of PAs and CCAs in the organization. The WM updates this state whenever a new PA or CCA comes up. The WM uses *next* functions defined in section 4.4 for state maintenance. The *next* function uses the percept which is null in this case, decoded message M_d which is the message send by PA or CCA when it comes into existence, and state which is a record of PAs and CCAs.

The action function is deciding an appointment(effector action), sending updates to CCAs (communicative action) and sending client request to PAs to match with their schedule (communicative action). It uses the incoming message M_d, percept P which is change in rates, and state S which is list of agents for its input and generates an action.

5.3.2.1 WMGUI

WMGUI is a graphical user interface (GUI) to provide interface for user to inform the WM of any changes in bank policies like rates of loan or interest rates for savings/checking accounts. A dialog box pops up where all the new rates should be entered. The WMGUI uses the WM to send these updated rates to all CCAs in its list.

5.3.2.2 WM Agent

Communicative actions of the WM Agent are implemented by overwriting the **getMessage()** function. The first object in the vector is an integer which is the request type 1, 2 or 3 if the message is send by the CCA for a client request. Subsequent objects are the CCA's

identity and client information. Upon getting this message, the WM contacts PAs depending upon the request type. Each request type needs a different expertise. The WM sends client information to each of these PAs. If any PA returns the message that it has a free time slot, the WM allots that time slot for the client. If the message is send by a PA or a CCA upon their coming in existence, the request type is 4 and 5 followed by the identity of the sender. The WM adds the PA or the CCA to its list of agents in the organization.

5.3.3 Personal Agent

The personal agent's main functions are:

(i) Maintain employee schedule - state maintenance function

(ii) Display schedule for a particular day - reactive function

(iii) Schedule an appointment for a particular client if there is free slot - effector action executed as a result of communication from WM

(iv) Inform the WM whether it was able to schedule the appointment or not - communicative action

The PA's environment consists of the WM and the employee whom it is assisting. The personal agent maintains schedule for the employee along with other functions. Therefore, it is necessary for it to store this information and keep updating it continuously, depending on the environment. It executes reactive and communicative actions as described above. Clearly, the PA is a reactive communicative agent with state. To receive percept, the PA has a GUI.

5.3.3.1 PA GUI

A PA works as an assistant for the employee and to make the interaction easy, a GUI is provided for it. The employee can view his schedule for a particular day and add or remove entries to the already existing schedule. It has text fields for date, employee name, time slot and description of appointment. Fig 5.3 shows the interaction between the PA and the WM.

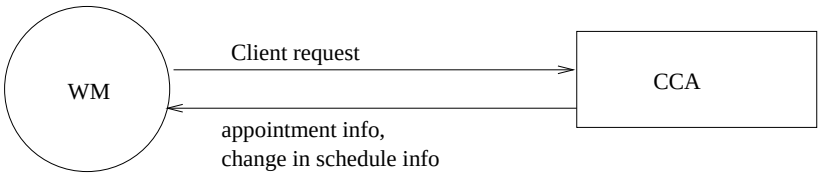

Figure 5.3 Interaction between the PA and the WM

5.3.3.2 PAgent

The PAgent class encodes functionality of the PA. It gets percept through the PAGUI. The reactive function for the PA is to display the schedule for a particular day for a particular employee. It is a simple lookup function that fetches information from the local database depending on the date parameter. Another effector action is to find out if a particular appointment can be fixed or not. This is in response to communication from the WM. The state of environment can be a message or a percept as defined in section 4.3.

The PA uses the function **updateSchedule()** to add or remove entries to the schedule. The internal state of PA consists of information about the schedule. Continuous updates to the schedule happen by execution of the *next* function as defined in section 4.4. The percept is change in schedule from the employee and the message M_d is message from the WM regarding an appointment.

On getting a message from the WM, the PA executes the **getMessage()** function. The message is in the form of a vector of objects. The information is contained in the form of username, his time preference and a description of his needs. After decoding the message, the PA checks its schedule to see if it can fix the appointment. Both in the case of success and failure, the PA sends message back to WM. It uses the *action* function as defined in section 4.3. It uses schedule along with the percept and the decoded message as described in previous section, to decide upon its next action.

This completes our description of the call center application and its components. This application can be extended to include facilities for customer account information. The purpose of designing and implementing this system is to demonstrate pragmatic application of agent toolkit. This application does not make use of mobile agents and in next chapter, we describe

another application build with help of toolkit that uses mobility.

CHAPTER 6. DISTRIBUTED LEARNING APPLICATION

In this chapter, we describe the second application made using the agent toolkit. This application is based on distributed learning. The data is distributed at different sites in different manners and learning is done without collecting all the data in a central place. For this purpose, a mobile agent visits different sites and collects the relevant information to learn from distributed datasets.

6.1 Distributed Learning

Recent advances in high throughput data acquisition and data storage technologies have made it possible to gather and store large amounts of data at increasing rates. For example, biologists are generating and collecting gigabytes of protein sequencing and genome data at a steady rate. Many organizations are collecting data related to their products, consumer interests and market status. These data repositories are very large and they keep growing. Many application domains, (e.g., miliary command and control, law enforcement etc) require the use of multiple, geographically distributed, heterogeneous data and knowledge sources (HMW98), (YHH+98). Data and knowledge sources often reside on different hardware and software platforms and are independently owned and operated by different authorities. Data is being gathered in different forms at different sites, though some of this data may be related to each other but because of the form in which data is stored, it might not be possible to do any analysis without changing its format. Since there is no control of a central authority, data is being gathered by organizations, research labs, and individuals according to their need and convenience.

Many of the existing mining algorithms do not scale up to extremely large data sets. One

way to solve the problem is to partition data into many subsets which can be managed as autonomous data sets. All the above mentioned reasons are resulting into large data sources across various hardware and software platforms.

Although it is convenient for different authorities to maintain data in accordance to their requirement, it is not easy to learn from this data. One solution is to collect all the data in a central location in order to learn from it. The sheer volume of the data and the rate of accumulation, often prohibits the use of *batch learning* algorithms which would require processing the entire data set whenever new data is added to the data repository. Collecting all the data at a central location in order to do batch learning is an expensive process in terms of network bandwidth and communication overhead. It is also possible that organizations are not willing to provide access to raw data for various security and privacy reasons but they allow access to some kind of summary or statistics of data, e.g., count of people having a particular disease as opposed to revealing list of names of people. Hence, another approach to learn from these datasets is to learn from each of them and then somehow combine the resulting hypotheses. This is one of the key problems in the area of learning. In order to facilitate learning from distributed datasets, there is a need to devise *distributed learning* algorithms that can incorporate new data as they become available across space (*distributed learning*). Data-driven knowledge acquisition algorithms that can incrementally process large amounts of data that is distributed in space are called distributed learning algorithms. In this chapter, we present the design and implementation of a distributed learning algorithm using Decision tree.

Some of the basic notions of a database are entity, records and attributes. *Entity* is defined as some "object" or "thing" that is distinguishable from other objects, e.g., each *employee* in an organization is an entity (SHS96). An entity is represented by a set of *attributes*. Attributes are descriptive properties possessed by an entity, e.g., attributes for an *employee* are *employee id*, *name*, *department*, etc. An entity set is a set of entities of the same type that share the same properties or attributes. A *database* is a collection of entity sets each of which contains entities of the same type. Each entity is described by attributes and it can be represented as a table where each column is an attribute and each row is a collection of values that these

attributes can take. Each row or *tuple* has the same number of fields, or *attributes*. A *dataset* is a collection of tuples and it can have duplicate tuples.

In a distributed database, datasets are stored across all sites in form of fragments. These fragments contain enough information to reconstitute the complete dataset. Datasets can be distributed or *fragmented* in two ways:

- Horizontal Distribution: In this case, the data is distributed in such a manner that each site contains a multiset of tuples. The (multiset) union of all these multisets constitute the complete dataset. If the multisets of tuples of data are indicated by $D_1, D_2, ..., D_n$ etc., each site contains one or more of these sets. Let D denote the complete data set, then *Horizontally Distributed Data* (HDD) is specified as:

$$D_1 \bar{\cup} D_2 ... \bar{\cup} D_n = D$$

A complete data set containing student data is shown in table 6.1. Horizontal fragments of this dataset are shown in table 6.2 and table 6.3.

Table 6.1 Student Dataset (Complete)

Student Id	Name	Department	GPA
1121	Taru Trivedi	Computer Science	3.5
1122	Adrian Silvescu	Animal Science	3.7
1123	John Silver	Animal Science	3.8
1124	Doina Caragea	Computer Science	3.6

Table 6.2 Student Dataset (Horizontal Fragment I)

Student Id	Name	Department	GPA
1121	Taru Trivedi	Computer Science	3.5
1122	Adrian Silvescu	Animal Science	3.7

- Vertical Distribution: In this case, each data tuple is fragmented into several subtuples each of which shares a unique key or index. Thus, different sites store *vertical*, possibly overlapping, fragments of the data set. Each fragment corresponds to a subset of the attributes that describe the complete data set.

Table 6.3 Student Dataset (Horizontal Fragment II)

Student Id	Name	Department	GPA
1123	John Silver	Animal Science	3.8
1124	Doina Caragea	Computer Science	3.6

Let $A_1, A_2, ..., A_n$ indicate set of attributes whose values are stored at sites $1 \cdots n$ and A denote the set of attributes that are used to describe the data tuples of the complete data set. Then *Vertically Distributed Data* (VDD) has the property:

$$A_1 \cup A_2 ... \cup A_n = A$$

Let $D_1, D_2, .., D_n$ respectively denote fragments of the dataset stored at sites $1 \cdots n$ and let D denote the complete data set. Let a tuple at i^{th} index in a data fragment D_j be denoted as $t^i_{D_j}$. Let $t^i_{D_j}.unique\_index$ denote the unique index associated with tuple $t^i_{D_j}$. The VDD has the property:

$$D_1 \times D_2 \times ... \times D_n = D$$

$$\forall\ D_j, D_k,\ t^i_{D_j}.unique\_index = t^i_{D_k}.unique\_index$$

Thus, the subtuples from the vertical data fragments stored at different can be put together using their unique index to form the corresponding data tuples of the complete dataset.

Vertical fragments of the student dataset shown in table 6.1 are shown in table 6.4 and table 6.5. In this case, the student id is used as the unique index associated with subtuples.

Table 6.4 Student Dataset (Vertical Fragment I)

Student Id	Name	Department
1121	Taru Trivedi	Computer Science
1122	Adrian Silvescu	Animal Science
1123	John Silver	Animal Science
1124	Doina Caragea	Computer Science

Table 6.5 Student Dataset (Vertical Fragment II)

Student Id	Department	GPA
1121	Computer Science	3.5
1122	Animal Science	3.7
1123	Animal Science	3.8
1124	Computer Science	3.6

6.2 Learning from Distributed Datasets

By learning we mean a process by which a system *improves* its performance on a *set of tasks* as a result of *experience*. The idea is that percept are used not only for action but also for improving an agent's ability to act in the future. Various types of learning methods are:

- Rote Learning: In this type of learning, almost no inference is involved. Knowledge is stored in same form as it is provided, e.g., for the game of chess, the chess board configurations and their backed up evaluations are stored.
- Learning by Instruction: In this type of learning, some transformation of knowledge is necessary before it is stored for future use.
- Learning by Induction: Here, given a set of examples, the learner has to induce a set of general problem solving procedures.
- Learning from Reinforcement: This type of learning is done by exploration. Here, rewards are used to learn a successful agent function. The agent learns a utility function on states and uses it to select actions that maximize the expected utility of their outcomes.

The *distributed learning* problem can be summarized as follows: The data is distributed among different sites and the learner's task is to discover some useful knowledge (e.g., a hypothesis in the form of a decision tree that classifies instances). This can be accomplished by a learning agent that visits the different sites to gather the information it needs to generate a suitable hypothesis from the data. There are different ways to learn from distributed datasets. In *Serial Distributed Learning*, the information is collected by an agent that visits the sites where the distributed datasets are located, collects relevant information from them and learns

from this collected information. Fig 6.1 shows serial distributed learning.

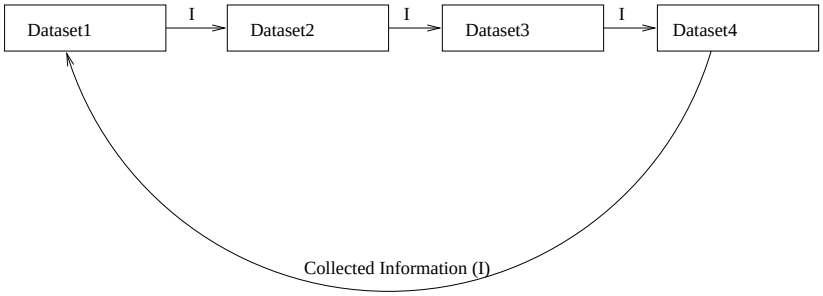

Figure 6.1 Serial Distributed Learning

In *Parallel Distributed Learning*, learning is done at a central location where the information is collected from all sites. Fig 6.2 shows parallel distributed learning.

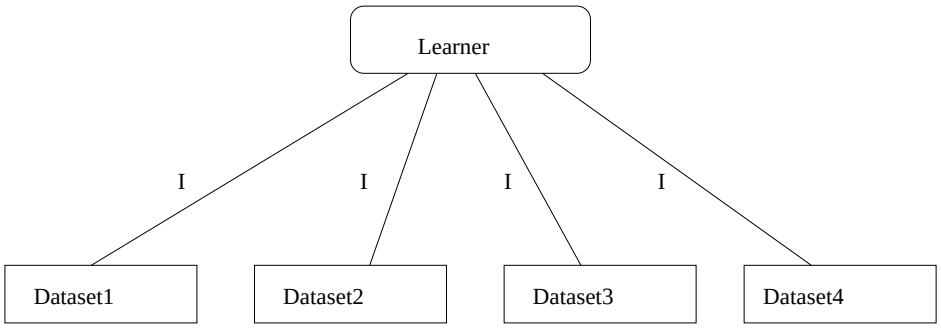

Figure 6.2 Parallel Distributed Learning

We assume that it is not feasible in the distributed learning setting to transmit raw data from different sites. Consequently, the learner has to rely on the information extracted from the sites. Thus, identification the information required by different learning algorithms, and design of efficient means for providing such information to the learner are central questions that need to addressed in devising distributed learning algorithms.

A *Distributed Learning* algorithm $L_{Distributed}$ is said to be *exact* with respect to the hypothesis inferred by a learning algorithm L if the hypothesis produced by $L_{Distributed}$ using distributed data sets $D_1 \cdots D_n$ stored at sites $1 \cdots n$ (respectively), is the *same* as that ob-

tained by L from the complete data set D. That is, $L_{Distributed}$ is an exact distributed learning algorithm with respect to the hypothesis inferred by a learning algorithm L if it is the case that:

$$L_{distributed}(D_1, D_2, ..., D_n) = L(D_1 \bar{\cup} D_2 \bar{\cup} ... \bar{\cup} D_n)$$

where $\bar{\cup}$ denotes multiset union. For example, the multiset union of the multisets $\{a\ b\ a\}$ and $\{b\ c\ a\ d\}$ is $\{a\ a\ a\ b\ b\ c\ d\}$.

Similarly, we can define exact distributed learning with respect to other criteria of interest (e.g., expected accuracy of the learned hypothesis). More generally, it might be useful to consider *approximate* distributed learning in similar settings. However, we restrict the discussion that follows to an approach to exact distributed learning using *information extraction.*

This approach to exact distributed learning involves extracting from distributed datasets, the information necessary for inferring the appropriate hypothesis. We introduce the information extraction operator $I(D_i)$ that extracts from each data set D_i, the information necessary for $L_{Distributed}$ to learn from $D_1 \cdots D_n$. If the information extracted from the distributed datasets is same as that used by L to infer a hypothesis from the complete dataset D (that is, $C[I(D_1),\ I(D_2),\ ...,I(D_n)] = I(D)$), $L_{Distributed}$ will be exact with respect to L.

If information is extracted at lowest level, complete data needs to be collected at the central place. On the other hand, if information is being extracted at the highest level, it means that result of learning is being transmitted. The purpose is to find an optimum amount of information that needs to be carried for doing learning. Usually, this is at a level in between the highest and lowest level of information extraction, where some statistics about the data is collected from a dataset. The information to be collected depends on the algorithm being used. In the next section, we prove that the tree constructed using information extraction with decision tree in case of distributed datasets is the same tree as of datasets at a central location.

Another way to learn from distributed datasets is to change the existing algorithm itself. However, using the existing algorithm and providing it with required information is a simpler way to do the learning.

6.2.1 Decision Tree Learning

Distributed Learning presented in this chapter uses the Decision tree as the learning algorithm L.

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. The ID3 (Iterative Dichotomizer 3) algorithm proposed by Quinlan (Quinlan, 1986) and its variants offer a simple, and practically rather effective approach to inferring decision trees from labeled examples. It is a class of inductive learning which is learning from examples and counterexamples. An *example* is an ordered pair (X_k, C_k) where X_k is a pattern (e.g., a vector of *attribute values*) and C_k is a desired classification. An example is same as tuple with C_k as an attribute.

An inductive learner tries to a learn a "concept description" or a function that

(i) correctly classifies all (or most of) the training examples

(ii) generalizes well on examples not used in training

Decision trees make use of Information Theory (SW48). It defines the information content of a message m with probability P as $I_m = I_p = -\log_2 P_m$ bits where P_m is the probability of the message m. For example, information provided by the outcome of a fair coin toss is $-\log_2 \frac{1}{2} = 1$ bit. Consider a set of instances S which is partitioned into M disjoint subsets (classes) $C_1, C_2, ..., C_M$ such that

- $S = \bigcup_{i=1}^{M} C_i$
- $C_i \bigcap C_j = \emptyset \; \forall \; i \neq j$

The probability of a randomly chosen instance $s \in S$ belonging to class C_j is $\frac{|C_j|}{|S|}$, where $|X|$ denotes the cardinality of the set X. So, the *information content* of the knowledge of membership of a randomly chosen instance in class C_j is $-\log_2 \left(\frac{|C_j|}{|S|}\right)$ bits. The expected information content of knowledge of class membership of a random instance $s \in S$ is

$$-\sum_j \frac{|C_j|}{|S|} \cdot \log_2 \left(\frac{|C_j|}{|S|}\right)$$

This quantity, which is also known as the *entropy* of set S and measures the expected information needed to identify the class membership of instances in S. The decision tree learning algorithm searches in a greedy fashion, for attributes that yield the maximum amount of information for determining the class membership of instances in a training given training set S of labeled instances. The result is a decision tree that correctly assigns each instances in S to its respective class. The construction of the decision tree is accomplished by recursively partitioning S into subsets based on values of the chosen attribute until each resulting subset has instances that belong to exactly one of the M classes. The selection of attribute at each stage of construction of the decision tree maximizes the estimated expected information gained from knowing the value of the attribute in question.

The *information gain* for an attribute a, relative to a collection of instances S is defined as:

$$Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where Values(A) is the set of all possible values for attribute A, and S_v is the subset of S for which attribute a has value v.

The information gain associated with an attribute can be calculated using the count of examples from different classes that have specific values for the attribute in question. ID3 algorithm is described in fig 6.3.

6.2.2 Distributed Learning Algorithm using the Decision Tree

As mentioned in previous section, we use the method of information extraction to learn from distributed data sources. Information required to construct a decision tree is the information gain associated with the attributes which can be calculated using count of examples of different classes for each attribute and its value. Suppose we want to partition the set of instances at a particular node in a partially constructed decision tree. Let a_j denote the attribute at the jth node (starting from the root node) leading up to the node in question. Let $v(a_j)$ denote the value of the attribute a_j attribute along the path leading up to the node in question.

ID3(Examples,Target-attribute,Attributes)

Input: Training examples *Examples*, *Target-attribute*is the attribute whose value is to be predicted by the tree. *Attributes* is a list of other attributes that may be tested by the learned decision tree.
Output: A decision tree that correctly classifies the given *Examples*

begin
Create a **Root** Node for the tree
If all Examples are of same class c, **Return** the single-node tree Root , with label = c
If Attributes is empty, **Return** the single node tree Root,
with label = most common value of Target-Attribute in Examples
else begin
A $\leftarrow$ the **Attribute** that best (with maximum information gain) classifies Examples
The decision attribute for **Root** $\leftarrow$ **A**
for each possible **value**, v_i, of **A**
Add a new branch below **Root**, corresponding to the test **A** = v_i
Let $Examples_{vi}$ be the subset of examples that have **value** v_i for **A**
If $Examples_{vi}$ is empty
then below this new branch , **add** a leaf node
with label = most common value of Target-attribute in Examples
else below this new branch add the subtree
ID3 (Examples,Target-attribute,Attributes-[A])
end for
end else
end

Figure 6.3 Decision Tree Algorithm.

For adding a node below any branch of tree, the set of examples being considered satisfy the constraints on values of attributes specified by:

$$L_l = [a_1 = v(a_j), a_2 = v(a_2), ..., a_l = v(a_l)]$$

where l is the depth of the node in question.

We need to obtain the relevant counts from the set of examples that satisfy this constraint. This calculation has to be performed once for each node that is added to the tree starting with the root node.

Using the following two theorems, we prove that count of examples collected from distributed data sets is same as that which would be collected from the complete data set. This suffices to prove that the decision tree constructed from a given data set in the two scenarios is exactly same.

When data is horizontally distributed, examples for a particular value of a particular attribute are scattered at different locations. For finding the count of examples for a particular node in tree, all the sites are visited and count is accumulated. The learner uses this count to find the best attribute to further partition the set of examples being considered. We use f_l to denote the count of examples that satisfy the constraints specified by L_l and f_l^i denotes the the count of examples that satisfy the constraints given by L_l at site i.

Algorithm ComputeCountHDD

Input: L_l (attributes and their values along the path leading upto the node under which a subtree is being added)

Output: count of examples across all sites that satisfy the constraints specified by L_l.

```
    f:= 0
    for i = 1 to m      //there are m sites
        f := f + f_l^i
    return f
end
```

Figure 6.4 Algorithm Compute Count for HDD.

Theorem 6.1 *For every input* $L_l = [a_1 = v(a_1), a_2 = v(a_2), ..., a_l = v_(a_l)]$, *ComputeCountHDD(l) computes* f_l.

Proof:

$f = \sum_{i=1}^{m} f_l^i$

= number of examples that satisfy constraints given by L_l $in(D_1 \bar{\cup} D_2 \bar{\cup} ... \bar{\cup} D_m)$ // by definition of HDD

$= f_l$ □

In this algorithm, for each node in tree, f_l is calculated for a set of attributes and their values.

Time Complexity: Time required to calculate f_l^i at each site i, is proportional to $\mid D_i \mid$. Hence, the time required for calculating f_l is therefore proportional to $\mid (D_1 \bar{\cup} D_2 \bar{\cup} ... \bar{\cup} D_m \mid = \mid D \mid$. Let A denote set of all attributes and V denote the largest set of values that an attribute can take. For each node, the maximum time required to calculate f_l is bounded by $\mid A \mid\mid V \mid\mid D \mid$. Therefore, the run time of the distributed decision tree learning algorithm for horizontally distributed data sets is proportional to the the product of the time required per node and the number of nodes in the tree. Let $treesize(D)$ denote the number of nodes in the decision tree constructed from the dataset D. Then the run time of the algorithm is bounded by of nodes and is $\mid A \mid\mid V \mid\mid D \mid treesize(D)$.

Space Complexity: Information gathered to facilitate learning is simply the counts of examples for each class along each outgoing branch of the node in question. A loose upper bound for this is given by $M \mid A \mid\mid V \mid$ where M is the number of classes.

Communication cost: The upper bound for information carried from one location to another is $M \mid A \mid\mid V \mid$ for one node in the tree. Hence, the communication cost involved in constructing the complete tree is $M \mid A \mid\mid V \mid treesize(D)$.

In vertically distributed datasets, we assume that each example has a unique indices asso-

ciated with it. Subtuples of an example are distributed across different sites. However, they can be related to each other using their unique index. In order to select the attribute to partition the instances at a node in a partially constructed tree, the relevant counts are gathered using the unique indices. To find the best attribute, a pass is made through all data sites to compute the count of examples. As before, let L_l =$[a_1 = v(a_1), a_2 = v(a_2), ..., a_l = v(a_l)]$ denote constraints on the values of attributes satisfied by the instances at the tip of the node in question. Let $I_{L_{l-1}}$ denote the set of indices for tuples satisfying L_{l-1}. We use f_{L_l} to denote the count of examples that satisfy the constraints given by L_l among the set of tuples with indices in $I_{L_{l-l}}$.

Algorithm for computing the counts in case of VDD is as follows:

Algorithm ComputeCountVDD

Inputs: L_l (attributes and their values along the path leading upto the node under which a subtree is being added), $I_{L_{l-1}}$ (the set of indices of instances that satisfy the constraint L_{l-1}.
Output: count of examples whose indices are in I_{Ll-1} and satisfy the constraints specified by L_l.
begin
visit the site that has subtuples that has values for the attribute a_l
Compute I_{L_l} by selecting tuples which satisfy the constraint L_l and whose indices appear in $I_{L_{l-1}}$
return $\mid I_{L_l} \mid$
end

Figure 6.5 Algorithm Compute Count for VDD.

Theorem 6.2 *For every input specified by* L_l *and* $I_{L_{l-1}}$, *ComputeCountVDD(*L_l,$I_{L_{l-1}}$*) computes* f_{L_l}.

Proof:

Provided that I_L is correctly computed, then $\mid I_{L_l} \mid = f_{L_l}$

We prove this by induction. Base case:

I_{L_0} = the set of all indices.

Induction step:

IL_l = The set of indices in IL_{l-1} that satisfy $a_l = v(a_l)$ □

Suppose we denote the indices for the examples at the n^{th} node (assuming left to right numbering of nodes) at depth k in the tree is denoted by I_{kn}. Let A denote set of all attributes, V denote the largest set of values that an attribute can hold and let h denote the depth of the tree. Since each node in the tree corresponds to a unique combination of attribute values, indices of examples at one node cannot appear in another node at the same depth in the tree. That is,

$$\forall\, k\ (\forall\, i \neq j\ I_{ki} \cap I_{kj} = \phi)$$

Time Complexity: The time required for partitioning the examples at the nth node at depth k is

$$\mid A \mid\mid V \mid\mid I_{kn} \mid$$

. Therefore, the time required for partitioning examples at all of the nodes at depth k in the tree bounded by

$$\begin{aligned}
&\sum_{n \in Level k} \mid A \mid\mid V \mid\mid I_{kn} \mid \\
=&\mid A \mid\mid V \mid \sum_{n} \mid I_{kn} \mid \\
\leq&\mid A \mid\mid V \mid\mid D \mid
\end{aligned}$$

for all levels in tree,

$$\begin{aligned}
\leq &\sum_{k \in levels} \mid A \mid\mid V \mid\mid D \mid \\
\leq&\mid A \mid\mid V \mid\mid D \mid h
\end{aligned}$$

Space Complexity: Information gathered to facilitate learning is simply the counts of examples for each class along each outgoing branch of the node in question along with the relevant

indices. Let $indexsize$ denote the space required to store an index. A loose upper bound for this is given by $M \mid A \mid\mid V \mid + \mid D \mid indexsize$ where M is the number of classes and $\mid D \mid indexsize$ provides an upper bound on the number of indices.

Communication Complexity: The upper bound for the information carried from one site to another is $M \mid A \mid\mid V \mid + \mid D \mid indexsize$ for one node in the tree. Hence, the communication cost involved in constructing the complete tree is $(M \mid A \mid\mid V \mid + \mid D \mid indexsize)treesize$. This decomposition gives a relatively standardized way of solving problems where the data is distributed. This technique of doing distributed learning by supplying algorithm with the extracted information from data sources is quite independent of the algorithm and does not require the understanding of the algorithm in full detail. Therefore, this technique is a consistent source of upper bounds for solutions of the distributed problem. However, "smarter", i.e., faster algorithms can presumable be designed by changing the algorithm to account for the fact that data is distributed.

6.2.3 Distributed Learning Algorithm using the Decision Tree for HDD

In this case, all the attributes are at all sites but it is not possible to know information gain of a particular attribute as the possible values that it can take are scattered across all sites. For building the decision tree, we need to gather information about the number of examples for all attributes and their values. We use a mobile agent that visits different sites and extracts information from these sites. The agent gathers information about the number of different classes of examples for each attribute and their possible values. Thus, the agent carries a very small amount of information in comparison to carrying the complete data. This reduces the required network bandwidth drastically. Every time the agent loads all the data available at site and then it filters it according to the attribute and its value for which a node in the tree is being made. The agent returns to a central location and another agent, called the decision tree agent, makes use of the information about counts to build the tree. For constructing a subtree for a particular attribute a, with value v, the agent carries a and v with itself to be

able to select examples at all locations having these values. At each site, the agent loads all examples and filters them according to the values of a and v. Pseudo code for the algorithm is shown in fig 6.6.

HDDDecisionTree(mbAgent,locations, att-val, current-attr, attributes)

In this implementation,, we collect the counts for all attributes and their values at each site. This saves the need to visit a site for every node and its value. Hence, the time required for calculating f for each node is bounded by $\mid D \mid$. Hence, the runtime of the algorithm is bounded by $\mid D \mid treesize(D)$.

6.2.4 Distributed Learning Algorithm using Decision Tree for VDD

Here, the data is distributed in such a manner that each site contains tuples with values for a subset of attributes. For building the decision tree to correctly classify these examples, we need to find the information gain associated with all the attributes. A mobile agent visits a site and chooses the best attribute at that site. It visits all sites and carries with it the information about the local best attribute. It finds the best attribute across all sites and delivers this information to decision tree agent. Decision tree agent builds decision tree using this information. While trying to construct a subtree for some particular value v, of an attribute a, only the examples with that value for attribute needs to be considered. For this purpose, the mobile agent carries with itself the unique indices of examples having value v, for attribute a. It visits each site, loads all data and filters it according to the indices it is carrying. At each site, it loads the definitions for all attributes. For a particular pass, all attributes are not considered in choosing the best attribute as they might have already been chosen. The mobile agent also carries attributes to be eliminated from all attributes loaded at a site. Every time a best attribute is chosen, it is added to the list of attributes to be eliminated.

Algorithm for decision tree for a VDD is presented in fig 6.7.

In this implementation, we find the local best attribute at a site and transmit that information. This saves the need to visit a site for counts corresponding to every attribute and hence, the run time is bounded by $\mid D \mid h$. Since the information carried is only the information gain

HDDDecisionTree(mbAgent,locations, att-val, current-attr, attributes)

Input: A mobile agent *mbAgent* that visits all sites listed in *locations*, *current-attr* to be considered for building node in tree, its value *att-value*, list of attributes *attributes* that needs to be eliminated for choosing the best attribute.
Output: A decision tree that correctly classifies examples across all *locations*

```
begin
    exampleIndex = null /*contains attributes and their values for filtering data.*/
    for each location in locations
        mbAgent reads dataset
        if att-val ≠ first /*first pass*/
            filter data using exampleIndex
        else use all data
        /*get information on counts of examples for all classes*/
        get counts of examples belonging to all classes
        for all attributes a
            for all values v
                count no of examples for all classes and put in list-count
    end for
    decisiontree-agent.list-count = mbAgent.list-count
    if examples are identical /*count of only one class is positive*/
        mbAgent returns to home location /*to start a new pass*/
        remove current-attr and att-val from list of attributes to be considered
        return a node with the class with positive count
    end if
    else if attributes is empty, return the single node tree Root,
         with label = class with highest count
    else
        choose best attribute A by calculating information gain for all attributes
              using list-count
        The decision attribute for Root ← A
        for all values v of best attribute
                Add a new branch below Root, corresponding to the test A = v
                add A and v to exampleIndex
                below this branch, add a new subtree
                  HDDDecisionTree(mbAgent,locations, v, A, attributes-A)
        end for
end
```

Figure 6.6 Decision Tree Algorithm for HDD.

VDDDecisionTree(mbAgent,locations, indices, varExcluded)

Input: A mobile agent, *mbAgent* that visits all sites in *locations*, data structure to hold unique index associated with examples,*indices* and attributes *varExcluded* not to be considered for making tree.
Output: A decision tree that correctly classifies examples across all *locations*

```
begin
    best-attribute = null
    for all locations l
        mbAgent reads and filters data according to indices to create examples
        if examples are identical
            returns a node with label = class of examples
        else
            choose local-best-attr
            if (local-best-attr better than best-attribute)
                best-attribute = local-best-attr
    end for
    /* function of decisiontree-agent*/
    give best-attribute and indices-attr to decisiontree-agent
    The decision attribute for Root ← best-attribute
    for all values v of best-attribute
        Add a new branch below Root, corresponding to the test best-attribute = v
        indices = index of examples having best-attribute with value v
        below this branch, add a new subtree
                VDDDecisionTree(mbAgent,locations,indices,varExcluded+A)
    end for
end
```

Figure 6.7 Decision Tree Algorithm for VDD.

of the local best attribute, the space required reduces to $\mid D \mid indexsize$. The communication cost also reduces to $(\mid D \mid indexsize)treesize$.

6.3 Architecture of Distributed Learning Application (DLA)

The system architecture consists of two types of agents. First is a mobile agent with the capacity to load and read data in a certain format and extract the relevant information. The second agent is a learning agent with the capacity to learn from the data made available to it. It uses the information gathered by the mobile agent to do its learning. This architecture is common to HDD and VDD.

Below is a description of each agent in detail:

6.3.1 Mobile Agent

This agent is a reactive agent with the ability to move to different locations. It is derived from the reactive agent class described in section 4.8.2. The reactive action of this agent is to collect the context sensitive information at each site. Having visited all the sites, it gives this information to the decision tree agent.

For VDD, the agent loads the data and attribute definition, filters it according to the indices of examples it is carrying with it, thus resulting in a working dataset. It also removes the variables which are not needed. It carries with it the indices and variables to be excluded in form of vectors to all the locations. It makes a node in the tree if the examples are identical in that location or else it tries to find the best attribute to make tree. It gives the information about the best attribute to the Decision tree agent which in turn builds the tree. If there is a need to explore various values of best attribute chosen in a pass, the Decision tree agent recursively call mobile agent to visit all sites and return with information.

For HDD, the agent again loads all the data, but not the attribute definitions as it carries the attribute information with it. The agent carries attributes and their values in a vector. Depending on these values, it filters the data from the dataset at each site. It visits all the sites and transmits the collected information to Decision tree agent. If all selected examples

at a site have the same class, a node is made in the tree with this class. This attribute and its value for which the node is made is removed from the index of attributes and values, on which the selection of data at each site is based. If the examples have different classes, the best attribute needs to be chosen. After choosing the best attribute, it is removed from the list of attributes to be considered. For all values of this attribute, the tree is build incrementally using the process of visiting all sites.

6.3.2 Decision Tree Agent

The decision tree agent receives information from the mobile agent and builds a decision tree based on this information. This is derived from the learning agent model in the agent toolkit. It uses functions provided by the learning agent to load the data and normalize it. Unlike the mobile agent, it is stationed at one location, usually the location of origin of the mobile agent. After visiting all the sites, the mobile agent collects the relevant information depending on the fragmentation of data -horizontal or vertical. Depending on this information, this agent builds a node in tree and if a subtree needs to be constructed under this node, it again sends out the mobile agent to collect the information. In relation to the learning agent model described in section 3.3.2, here the percept is the information given by the mobile agent, and the learning function is the decision tree algorithm.

An approach to adapting decision tree learning algorithms to work with distributed databases was explored in (Bhatnagar97). The scenario that they address can be viewed as a variant of the vertical fragmentation of data discussed in this paper. However, since their approach is motivated by somewhat different considerations, it is focused on the problem of obtaining counts from *implicit tuples*. In particular, they do not assume the existence of a unique index for each tuple in the complete data set that can be used to associate the subtuples of the tuple. The resulting algorithm simulates the effect of *join* operation on the sites without enumerating the tuples.

The distributed decision tree learning algorithms discussed in this paper are designed to deal with horizontally fragmented or vertically fragmented distributed data sets. They oper-

ate by decomposing the learning task into an information extraction phase and a hypothesis generation phase. This provides a general approach to designing *provably exact* distributed learning algorithms. The approach to learning from vertically fragmented distributed data sets assumes the existence of a unique index for each tuple in the complete data set. This assumption significantly simplifies the theoretical analysis of the resulting algorithm. Furthermore, in many distributed learning scenarios that arise in practice, it is reasonable to make such an assumption. For example, in scientific datasets generated by a number of collaborating laboratories investigating molecular structure-function relationships, each sample is identified by a unique identifier. However, it is possible to relax this assumption to modify the algorithm to deal with distributed learning scenarios wherein such an assumption may not hold. It is also relatively straightforward to extend these algorithms for data that is both horizontally and vertically distributed.

CHAPTER 7. SUMMARY

In this thesis, we have addressed two research problems:

1. The need of an agent toolkit to provide abstraction in area of agent programming and algorithms for distributed learning.

2. Algorithms for learning from horizontally and vertically distributed data.

Building complex systems is a relatively difficult task and the difficulty stems from the fact that complex systems are made up of several sub-systems which interact with each other. Multi-agent systems are being used as solutions for complex systems, open systems and ubiquitous systems. Agents represent a very powerful and efficient tool for making systems modular. Each point of control in a multi-agent systems could be autonomously managed by an agent and these agents could work together as a system. Different types of agents exist in a multi-agent system, performing different functions. In an open system, components are dynamic, non-deterministic and heterogeneous in nature. An agent based system can be reactive, they can sense their environment and respond to every change that affect their functionality without human or any other kind of intervention. Agent based systems provide better solutions when the domain involves entities which are distributed physically or logically and involves interaction of one or more of these entities. Design and development of agent based systems from scratch is a time consuming process. There is on-going research in field of agent programming on providing systems for agent development which capture as much abstract information as possible about agent architecture. This abstraction would help developers concentrate on agent-related features of the multi-agent systems such as reactivity, intelligence and pro-activity. A generic model is needed that can provide a well-laid foundation for development of multi-agent systems. Instead of designing each and every system from scratch, an abstract model could be

used to define basic functionalities of agents. The generic model has to be modular enough so that different components can be put together to serve as complex agents. Re-usability of components enables a developer to expend less effort for designing systems since the same component can be used for different types of agents. One way to provide the abstraction for agent related programming is an Agent Toolkit with agent models that can be used for designing multi-agent systems.

Recent advances in high throughput data acquisition and data storage technologies have made it possible to gather and store large amounts of data at increasing rates. Data and knowledge sources often reside on different hardware and software platforms and are independently owned and operated by different authorities. The data could be distributed horizontally or vertically. In order to learn from this distributed data, it is required that complete data be accumulated at a central location. The sheer volume of the data and the rate of accumulation, often prohibits the use of *batch learning* algorithms which would require processing the entire data set whenever new data is added to the data repository. In order to facilitate learning from distributed datasets, there is a need to devise *distributed learning* algorithms that can incorporate new data as they become available across space (*distributed learning*).

We summarize the contribution of this thesis research in the areas of abstraction in agent programming and learning from distributed data sources and outline some promising directions for future.

7.1 Contributions

7.1.1 Agent Toolkit

We propose the design and implementation of an agent toolkit which helps an analyst to go from a set of requirements to a working solution. The purpose of toolkit is to bridge the gap between very abstract agent architectures and very specific implementations of multi agent systems by giving a set of specific agent models which could be reused as they are or can be composed into more complex models. We have described theoretical as well as implementation aspects for all models in the toolkit. All the models have the capability of being mobile which

allows them to move from on e location to another and execute code. Agent models that make the toolkit are:

1. Reactive agent–An agent that responds to its environment depending on the percept received from the environment.

2. Learning agent–An agent that learns from the percept through the learning element and transmits this knowledge to other agents.

3. Communicative agent–An agent that can communicate with other agents leading to team work. There needs to be a common language among agents taking part in a conversation. An interpreter is needed to parse and interpret the message. Depending on the message content, the agent can execute effector action or communicative action.

4. Reactive Communicative agent–An agent that can communicate to other agents and also respond to the percept received. Thus, the input to the agent can be in form of percept or a message. The agent can execute effector actions as well as communicative actions.

5. Reactive Learning agent–An agent that responds to its environments and learns from the percepts. The agent could be learning in different forms depending on the need of application.

6. Reactive agent with state–An agent that responds to its environment depending on the percept received and maintains an internal state also. The internal state gets updated with the percept. Actions executed by agent depend on the internal state and percept.

7. Reactive Communicative agent with state–An agent that has functionality of the reactive communicative agent and also maintains an internal state. The internal state gets updated by percepts and incoming messages. Actions to be executed by the agent depend on the percept or message received and the internal state.

8. Reactive Learning agent with state–An agent that has functionality of the reactive learning agent and also maintains an internal state. The internal state gets updated by

percepts. Actions to be executed by the agent depend on the percept and the hypotheses learned by the agent.

9. Reactive Communicative Learning agent–An agent that is derived from the reactive communicative agent and the reactive learning agent. It has the functionality to respond to and learn from percepts and incoming messages. Action executed by the agent depends on the the percept or the message and the hypothesis learned.

Multi-agent solutions can be developed using models provided by the toolkit. These agent models can be refined according to the need of application or they can be composed together to form complex models. We have demonstrated the use of the agent toolkit using two multi-agent systems–a call center application and a distributed learning application.

7.1.2 Distributed Learning

In this part of the thesis, we have proposed algorithms to learn from distributed data sources. Learning algorithms use information from the data to learn and if they can be provided with this information, they will not need the entire data. This information can be extracted at various levels from the data sources. At the lowest level, the information is complete data available at a site and amount of information reduces as we go higher. At the highest level, information extracted is the result of learning itself. We have proposed algorithms that use an optimum level of information extraction from the data sources at various sites and provide it to the learning algorithm. The information to be extracted is different for horizontal and vertical distribution of data. We have demonstrated the use of these algorithms using decision tree learning algorithm.

7.2 Future work

During the course of this research we have identified several interesting problems and avenues that merit further investigation. In this section we seek to highlight some key future research directions.

7.2.1 Agent Models in the Agent Toolkit

It is of interest to explore the possibility of including different agent models in the toolkit. We have explored a few of the agent types. Our toolkit is based on DKN and it would be interesting to study other domains and develop agent models based on the needs of those domains. Most of the agent types that we have considered are reactive and research needs to be done in the area of deliberative agents. We have followed the traditional notion for definitions of agent, action, environment etc. The traditional definition of *action* does not allow an agent to access its global environment. It does not take into consideration the effect of other entities in the environment. Hence, *action* is replaced with *influence* which solves both the problems. It will be useful to develop agent models using the concept of *influence*.

7.2.2 Distributed Learning Algorithms

The algorithms for distributed learning discussed in this thesis assume either horizontal fragmentation or vertical fragmentation of data, but not both. It is also relatively straightforward to devise distributed learning algorithms for data sets that exhibit both horizontal and vertical fragmentation.

Work in progress is aimed at the elucidation of the necessary and sufficient conditions that guarantee the existence of exact or approximate cumulative learning algorithms in general and different types of incremental and distributed learning algorithms in particular in terms of the properties of data and hypothesis representations and information extraction and learning operators. Other long term goals of this research include: design of such theoretically well-founded algorithms for incremental and distributed learning; and application of such algorithms to large-scale data-driven knowledge discovery tasks in applications such as bioinformatics.

APPENDIX A. FUNCTIONS OF THE AGENT TOOLKIT

In this appendix, we provide details of functions of various classes in agent toolkit.

Basic Agent

This class encapsulates functions of the basic agent discussed in previous section. It has functions for registering the agent with directory, maintain information about other agents and mobility. Various functions used by this class are:

- Register with the directory: All agents register themselves with a universal directory when they come into existence. This directory is a repository of all agents and description of their expertise The function is defined as:

  public void registerAgent(String name, String desc, IDirectory dir)

  Input:

  name: Identity of agent

  desc: Description of agent's expertise

  dir: The directory with which agent is getting registered.

  This forms the social model which has details of other elements which agent can use when needed.

- Agent can find out from the directory if there is an agent with a particular expertise or can find the agent with a particular name using:

  public String findAgent(String agentName, IDirectory directory)

  Input:

agentName: Name of agent

directory: The directory with which agent is getting registered.

Output:

String: Identity of agent if it exists or else null

It gets information about an agent having identity as *agentName* from *directory*.

- Maintenance of information about agents: Each agent maintains a list of agents that it frequently contacts. This function reduces the need to contact the directory every time agent needs to communicate with another agent. Whenever an agent is looked up in the directory for the first time, it is added to this list. For this purpose, we have defined another class called *AgentListElement* which provides functions for maintaining the list such as updating description of agent, adding and removing agents from list etc.

  Functions defined for AgentList maintenance are:

  public void updateAgentInfo(String agentName, String description)

  Input:

  agentName: Identity of agent

  description: Description of agent

  This function updates information about agent with identity as *agentName*. This function is used by an agent to update its information about other agents.

  public void changeAgentInfo(String agentName, String desc)

  agentName: Identity of agent

  description: Description of agent

  This function changes the information maintained by an agent about other agents. It finds the agent with identity as *agentName* and changes its description to *desc*.

- Expertise of agent: All agents have their expertise defined by the user who creates them. It is stored as an attribute of agent when agent is created for the first time.

- Biography of agent: All agents maintain an internal biography where they store the information about activities which reflect their performance over a period of time. For this purpose, programmer should overwrite the function:

  public void maintainBiogr()

  This is different from internal state maintenance in agents as it does not effect the action which agents take.

Learning Agent

Learning agent uses functions of following classes:

Discrete Variable class

Attributes having discrete values are stored using instances of this class. While reading data, it makes a list of values that the attribute can have using function:

public void computeStatistics(String inValue)

Input:

inValue: value of the attribute for which the variable is created

It checks if the value is already present in the list of values and adds if it is not there.

Once all the data has been read, function for normalizing the data is:

public int normalize(String inValue, float[] outArray, int inx)

Input:

inValue : value of the attribute for which the variable is created

outArray: Array to store the normalized value

inx: counter indicating place in array to store the normalized value

Output:

int: counter inx incremented by 1

This function finds index of *inValue* in list of values maintained for the variable and translates it into one of N value code where N is the number of values that the attribute can possibly have.

Continuous Variable class

This is also derived from Variable class. Variables for attributes having continuous vales are stored using instances of this class. For normalizing data, the function is:

public int normalize(String inValue, float[] outArray, int inx)

Input:

inValue : value of the attribute for which the variable is created

outArray: Array to store the normalized value

inx: counter indicating place in array to store the normalized value

Output:

int: counter inx incremented by 1

Function **public void computeStatistics(String inValue)** sets minimum and maximum value of data. In this case, data normalization is done by finding the minimum and maximum value in data and dividing invalue by difference of these two values.

Dataset class

Function provided to read data file definition is:

public void loadDataFileDefinition()

This function reads data line by line from the input file and checks for the type of attribute. Depending on attribute type, it creates a variable e.g. if the attribute type is continuous, it creates a continuous variable.

Function provided to read data file is:

public void loadDataFile()

This function calls loadDataFileDefinition() to create variables for various attributes. It reads each line from the data file and adds a record to vector corresponding to each line. It uses **computeStatistics()** function to record all values that an attribute can have in case of categorical data and to find minimum and maximum value of data in case of continuous data. It normalizes data after loading it completely.

APPENDIX B. FUNCTIONS OF THE CALL CENTER APPLICATION

In this appendix, we provide some important functions of classes in Call Center Application.

Call Center Agent

Functions provided by CCA are:

- **public String reqProc(int type)**

  Input:

  type: type of request from customer

  Output:

  String: result of request

  This function returns result of requests from customers depending on request type. Corresponding to each request , information is stored in a vector that gets updated by WM whenever there is a change in information.

- **public void getMessage(Vector input, String identity)**

  Input:

  input: vector containing incoming message

  identity: identity of the sender

  This function parses and interprets the message. It executes action according to content of message.

WM Agent

Functions provided by WM agent are:

- **public PagentSchedule findSchedule1(Vector input , int reqType)**

  Input:

  input: Vector containing work manager's request

  reqType: indicates the request type

  Output:

  PagentSchedule: returns identity of Personal Agent that has been assigned this schedule and details of schedule

  This function gets request for schedule from CCA and contacts all Personal Agents in its list that have the required expertise. It checks with each of the personal agents and schedules appointment with the one that has a free time slot.

- **public void getMessage(Vector input, String identity)**

  Input:

  input: vector containing incoming message

  identity: identity of the sender

  This function parses and interprets the message. It executes action according to content of message.

- **public void addPagent(String pagent)**

  Input:

  pagent: Identity of personal agent

  This function is used for building list of personal agents.

- **public void addCCagent(String ccagent)**

  Input:

ccagent: Identity of call center agent

This function is used for building list of personal agents.

Personal Agent

Following are the functions provided by Personal Agent class:

- **public Vector getSchedule1(Vector input)**

  Input:

  input: the vector that contains requestType, userName, date, time slot boundaries that user wants, and description

  Output:

  Vector: containing name of agent and slot, "none" as agent name if no slot is found

  This function checks if there is any empty slot with this Personal Agent for the requirement sent by WM.

- **public String getSchedule(String datestr)**

  Input:

  datestr: String representing date

  Output:

  String: concatenation of all schedules for a particular date

  This function returns schedule for a particular day given the date.

- **public void initiateSchedule(String fromDate)**

  Input:

  fromDate

  This function allocates dates and slots for schedule for the week starting from the given date for a personal agent.

- **boolean updateSchedule(String userName, String date, int slot , char flag , String desc , Iwmagent wma)**

  Input:

  userName: name of the user with whom appointment is *date*: date of appointment *slot*: slot for which appointment is needed *flag*: indicates removal or addition to slot *desc*: description of appointment *Iwmagent*: WMAgent who needs to be informed of change in schedule

  Output:

  boolean: indicates if schedule has been successfully updated

APPENDIX C. FUNCTIONS OF THE DISTRIBUTED LEARNING APPLICATION

In this appendix, we provide some important functions of classes in Distributed Learning Application.

Mobile Agent

Functions provided by mobile agent class are:

- **public DataSet giveData()**

  It returns the DataSet read by agent.

- **public Vector giveRealData()**

  It returns the examples contained in DataSet.

- **public void ReadData(String name, String dataFile)**

  Input:

  name: Name of data

  dataFile: Name of file containing data

  It loads the data from data file , attribute definition and their possible value from data definition file. It takes name of dataset and name of datafile as parameters.

- **public void countExamples(Hashtable variables)**

  Input:

  variables: Hashtable containing all attributes

It counts number of examples of all classes for all attributes and their values. It takes a hashtable containing all attributes as parameter.

- **public Vector subset1(Vector examples, Vector varValues)**

  Input:

  examples: Vector containing instances in dataset

  varValues: Vector containing attributes and their values depending on which examples are collected in subset

  Output:

  Vector: containing examples in subset

  It finds subset of examples matching a set of attributes and their value. It takes a vector of all examples and vector of attributes and their values as parameter. It returns a set of matching examples.

- **public int[] getHorCounts()**

  Output:

  integer array: contains counts of all classes in the data set

  It returns counts of all classes in the data set in form of an array. Array size is same as number of classes. Each index corresponds to a class and content of array at that index is number of examples of that class.

- **public void filterDataVF(Vector indices)**

  Input:

  indices: contains unique indices used for filtering tuples from a dataset

  This function is used for selecting data at a location where data is vertically fragmented. It selects data set from the data set available at site according to indices it is carrying in form of vector.

- **public int[] getCounts(Vector examples)**

  Output:

  integer array: contains counts of all classes in the data set

  This function returns counts of all classes in the data set in form of an array of integers.

- **public double computeInfo(int p, int n)**

  Input:

  p: number of positive examples

  n: number of negative examples

  Output:

  double: information gain

  This function computes information content, given number of positive and negative examples.

- **public void removeVar(Vector varExcluded)**

  This function removes attributes from attribute list loaded by reading definition file. Attributes are removed as they have already been considered in earlier passes.

- **public Variable chooseVariable(Hashtable variables, Vector examples)** Input:

  variables: hashtable containing attributes and their values

  examples: Vector containing instances in dataset

  Output:

  Variable: attribute with most information gain

  This function chooses the best attribute depending on the information gain, at a particular location.

- **public void chooseBestVar()**

  This function chooses best attribute across all locations where data is vertically fragmented.

Decision Tree Agent

Following are the main functions provided by Decision tree agent:

- **public boolean horidentical(Hashtable varListCnt)**

  Input:

  varListCnt: Hashtable containing number of positive and negative examples for all values of all attributes

  Output:

  boolean: true if examples have identical classification else false

  This function is used in case of HDD. It finds if all examples have the identical classification.

- **public String hormajority(Hashtable varListCnt)**

  Input:

  varListCnt: Hashtable containing number of positive and negative examples for all values of all attributes

  Output:

  String: Majority value of examples

  This function is used in case of HDD. It finds class of majority of examples. The mobile agent collects data from all sites in form of a vector containing number of positive and negative examples for all values of all attributes and this function uses it to find class of majority of examples.

- **public void ReadLabels(String fileName, Hashtable variables)**

  Input:

  fileName: Name of file containg labels

  variables: hashtable containing attributes and their values

  This function reads reads attribute definition and the values they take from a file.

- **Vector findIndices(Vector examples, Variable variable, String value)**

  Input:

  examples: Vector containing instances in dataset

  variable: attribute

  value: value of attribute

  Output:

  Vector: contains indices associated with examples

  This function is used when data is vertically fragmented. It return indices of examples having same attribute and value as in parameters.

- **Variable chooseHorVariable(Hashtable variables, int counts[],double numRecs,Hashtable varListCnt)**

  Input:

  variables: hashtable containing attributes and their values

  counts: an integer array containg counts of number of positive and negative examples

  numRecs: number of records

  varListCnt: Hashtable containing number of positive and negative examples for all values of all attributes

  Output:

  Variable: attribute with most information gain

  This function is used when data is in horizontally fragmented. It returns the attribute with most gain after getting information from all sites. The information is in form of a hashtable containing all attributes, an integer array containg counts of number of positive and negative examples and Vector containing number of positive and negative examples for all values of all attributes.

- **public Node buildtree(Idlagent drone,Vector locations, Vector indices, Vector varExcluded)**

  Input:

  Idlagent: mobile agent

  locations: Vector containing address of sites for data

  indices: Vector containing unique indices

  varExcluded: vector containing attributes that are not to be considered for choosing best attribute

  Output:

  Node: a node in decision tree

  This function is used to build nodes in decision tree when data is in form of VDD. It uses the algorithm defined in 6.2.4.

- **public Node buildHorTree(Idlagent drone,Vector locations, String val, Variable current, Hashtable varList)**

  Input:

  Idlagent: mobile agent

  locations: Vector containing address of sites for data

  current: attribute for which node is being constructed

  val: value of attribute

  varList: Hashtable containing attributes and their values

  Output:

  Node: a node in decision tree

  This function is used to build nodes in decision tree when data is in form of HDD. It uses the algorithm defined in 6.2.3.

BIBLIOGRAPHY

[AJT99] M. Albers, M. Jonker, C.M.and Karami, and J. Treur. An electronic market place: Generic agent models, ontologies and knowledge. In *Proceedings of the Fourth International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, PAAM'99.*, pages 211–228, Lancashire, UK, 1999. The Practical Application Company Ltd.

[BB98] Joseph Bigus and Jennifer Bigus. *Constructing intelligent agents using Java.* Wiley Computer Publishing, Chichester, UK, 1998.

[BCG+98] F.M.T. Brazier, F. Cornelissen, R. Gustavsson, C.M. Jonker, O. Lindeberg, B. Polak, and J. Treur. Negotiating for load balancing of electricity use. In M.P. Papazoglou, M. Takizawa, B. Krmer, and S. Chanson, editors, *Proceedings of the 18th International Conference on Distributed Computing Systems, ICDCS'98*, pages 622–629, 1998.

[BJJT98] F. Brazier, C. Jonker, F. Jungen, and J. Treur. Distributed scheduling to support a call center:a co-operative multi-agent approach. *Applied AI*, 13:65–90, 1998.

[BJT99] F.M.T. Brazier, C.M. Jonker, and J. Treur. Compositional design and reuse of a generic agent model. *Artificial Intelligence Journal*, 1999. To appear.

[BKJT97] F. Brazier, D. Keplicz, N. Jennings, and J. Treur. Desire: Modelling multiagents systems in a compositional framework. *International Journal of Cooperative Information Systems*, 6(1):67–94, 1997.

[Boo93] G. Booch. *Object Oriented Analysis and Design with Applications.* Addison Wesley, Reading, MA, 1993.

[Bru91] Jose C. Brustolini. Autonomous agents: Characterization and requirements. Technical Report CMU-CS-91-204, Carnegie Mellon University, Pittsburgh, PA,, 1991.

[Fer99] J. Ferber. *Multi-Agent Systems: An introduction to distributed artificial Intelligence.* Addison-Wesley, Reading, MA, 1999.

[FG96] S. Franklin and A. Graesser. Is it an agent, or just a program? In *Third International Workshop on Agent Theories,Architectures and Languuages*, New York, NY, 1996. Springer-Verlag.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the International Conference on Database Theory*, pages 113–117, 1997.

[HMW98] V. Honavar, L. Miller, and J. Wong. Distributed knowledge networks. In *IEEE Information Technology Conference*, Syracuse, NY, 1998.

[Hon98] V. Honavar. Machine learning: Principles and applications. In J. Webster, editor, *Encyclopedia of Electrical and Elextronics Engineering.* Wiley, New York, NY, 1998.

[HR95] B. Hayes-Roth. An architecture for adaptive intelligent systems. *Artificial Intelligence: Special Issue on Agents and Interactivity*, 72:329–365, 1995.

[Inm96] W. Inmon. The data warehouse and data mining. *Communications of the ACM*, 39(11):49–50, 1996.

[Jak95] R. Jakobson. *On Language.* Harward University Press, Cambridge, MA,, 1995.

[JAT98] Jatlite. 05/10/00, http://java.stanford.edu/, 1996-98.

[Jen92] K. Jensen. *Coloured Petrinets. Basic Concepts, Analysis Method and Practical Use*, volume 1. Springer-Verlag, New York, NY, 1992.

[Jen95] R. Jennings. Controlling cooperative problem solving in industrial multiagent system using joint intentions. *Artificial Intelligence Journal*, 74(2):195–240, 1995.

[Mae95] P. Maes. Artificial life meets entertainment: Life like autonomous agents. *Communications of the ACM*, 38(11):108 –114, 1995.

[Mit97] T. Mitchell. *Machine Learning*. McGraw Hill, New York, NY, 1997.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[RN95a] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, page 33. Prentice Hall, Engelwood Cliffs, NJ, 1995.

[RN95b] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, page 46. Prentice Hall, Engelwood Cliffs, NJ, 1995.

[RN95c] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Engelwood Cliffs, NJ, 1995.

[SCS94] D.C. Smith, A. Cypher, and J. Spohrer. Kidsim: Programming agents without a programming language. *Communications of the ACM*, 37(7):55–67, 1994.

[Sea69] J. R. Searle. *Speech Acts*. Cambridge University Press, Cambridge, NY, 1969.

[SHS96] A. Silberschatz, Korth H.F., and Sudarshan S. *Database Systems and Concepts*. Mc-Graw Hill Companies, New York, NY, 1996.

[SW48] C. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, IL, 1948.

[Voy] Voyager. 05/10/00, http://www.objectspace.com.

[Wei99] G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999.

[Whi97] J. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*. MIT Press, Cambridge, MA, 1997.

[WJ95] M. Wooldridge and N. Jennings, editors. *Intelligent Agents*, pages 1–22. Springer-Verlag, New York, NY, 1995.

[WJ98] M. Wooldridge and N. Jennings, editors. *Foundations, Applications and Markets*, pages 3–28. Spreinger-Verlag, New York, NY, 1998.

[YHH+98] J. Yang, R. Havaldar, V. Honavar, L. Miller, and J. Wong. Coordination of distributed knowledge networks using contract net protocol. In *IEEE Information Technology Conference*, Syracuse, NY, 1998.