Engineering Machine Learning Techniques into Multi-Agent Systems

José Alberto Rodrigues Pereira Sardinha
sardinha@inf.puc-rio.br

Ricardo Choren
choren@les.inf.puc-rio.br

Ruy L. Milidiú
milidiu@inf.puc-rio.br

Carlos José Pereira de Lucena
lucena@inf.puc-rio.br

PUC-RioInf. MCC12/04, May 2004

Abstract: Agent technology is a Distributed Artificial Intelligence (DAI) approach to implement autonomous entities driven by beliefs, goals, capabilities, plans, and agency properties: adaptation, interaction, learning, etc. Software agents are the focus of considerable research by the artificial intelligence community, but there is still much to be done in the field of software engineering in order to systematically create large scale multi-agent systems. In this paper, we present an object-oriented framework for building a distributed multi-agent system. We explore in this framework the intersection of DAI with machine learning techniques, and propose a methodology for introducing intelligence in a multi-agent system. A multi-agent system created for the Trading Agent Competition is presented as a case study. Our engineering approach provides a performance gain of 97,3% due to the introduction of machine learning techniques.

Keywords: OO Frameworks, Software Agents, Anote, Machine Learning.

Resumo: A tecnologia de Agentes é uma abordagem da Inteligência Artificial Distribuída para implementar entidades autônomas movidos por crenças, objetivos, capacidades, planos e propriedades de agência: adaptação, interação, aprendizado, etc. Agentes de Software é o foco de vários trabalhos da comunidade de inteligência artificial, mas ainda há muito trabalho a ser feito no campo da engenharia de software para se criar sistemas multi-agentes de larga escala. Nesse artigo, apresentamos um *framework* orientado a objetos que permite a construção de um sistema multi-agentes distribuído. Exploramos nesse *framework* a interseção da Inteligência Artificial Distribuída com técnicas de *machine learning*, e apresentamos uma metodologia para incluir inteligência em um sistema multi-agentes. Uma agência criada para o *Trading Agent Competition* (TAC) é utilizada como um estudo de caso. Essa técnica de engenharia permite um ganho de 97,3% quando aplicada a agência do TAC.

Palavras-Chave: *Frameworks* orientado a objetos, Agentes de Software, Anote, *Machine Learning*.

1. Introduction

Multi-Agent Systems [1] [2] is a new technology that has been recently used in many simulators and intelligent systems to help humans perform several tasks. To achieve the system's goal, agents have to react to events, define strategies, interact, and participate in organizations. However, Software Agents [3] have gained greater importance for both academic and commercial applications with the advent of the Internet. Applications for the Internet are easily modeled with agents, mainly because of their distributed nature. We believe that many other applications based on this technology are still to be built to help leverage the use of the Internet.

We define Software Agents [3] as autonomous entities driven by beliefs, goals, capabilities, plans and a number of agency properties, such as autonomy, adaptation, interaction, learning and mobility. Although we recognize that the object-oriented paradigm has some flaws [3][4][5] related to design and implementation of multi-agent systems, we also believe that it is still the most practical paradigm to implement the agent technology. Our object-oriented framework [6] implements a communication infrastructure for agents over a network, and uses some hot spots [6] in order to implement the agent's beliefs, goals, capabilities, plans and some agency properties, such as autonomy, adaptation, interaction, and learning. Our agent-based system can be seen as an artificial society or organization, where every software agent has one or more roles, and can interact in order to achieve a common goal. `

We are exploring in this framework the intersection of distributed artificial intelligence with machine learning techniques, as described in [7]. Our framework and methodology have the goal of building a decentralized learning system, where several agents are engaged in the same learning process. However, each agent can be an expert in a particular task and have different machine learning algorithms to perform a specific activity.

Kendall et all [8] also propose an agent framework that is an architectural pattern organized in layers. In this architecture, an agent is composed of seven layers, such as the layer of sensors that are responsible for detecting changes in the environment. This pattern permits the modeling of both simple and complex agents. In [9], this agent design pattern is criticized for being too general. In fact, some difficulties arise in the maintenance and evolution process of the system. Also, the process of modeling in layers does not permit an easy removal of a layer when changes are needed. Adjacent layers to the removed layer normally have to suffer changes in order to adapt the system.

JADE (Java Agent Development Framework) [10] is a software development framework aimed at developing multi-agent systems and applications conforming to FIPA standards for intelligent agents. The java.core package of JADE includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. JADE has two main components: Agent and Behavior. The Agent component is the class that must be extended by application programmers to create agents, very alike to our proposed framework. The Behavior components implement the tasks, or intentions, of an agent. However, JADE provides a limited set of predefined behaviors (simple, composite, sequential, parallel, etc), none of which are related to cognition aspects.

We believe that a good engineering practice for a large scale environment requires two key elements: a modeling language for requirements and design phase; and a framework to help the development phase. This framework needs a good communication infrastructure, such as a blackboard and message-passing communication, and an easy mechanism to introduce learning properties in software agents.

Anote [11] is a modeling language for multi-agent system analysis that provides a notation based on views to express the structure and behavior of the system. In Anote, the multi-agent system can be specified and designed in two different levels: the inter-agent and the intra-agent levels. The inter-agent level deals with the structure of the system, such as the environment and its resources. The intra-agent level is related to the behavior of an individual agent.

Anote provides a set of models for multi-agent system specification since no single model is sufficient. Every non-trivial, large-scale, system is best approached through a small set of nearly independent models, or views. A view enables the software designer to concentrate on a single set of properties each time. Anote offers seven views that can be seen as an abstract image of a slice of a multi-agent system which neglects certain aspects that may be represented in other views. Thus, each Anote viewpoint has a specific representation that supplies some properties. The combination of these properties provides an extended knowledge about a system specification.

The domain of applications covered by the MAS Framework is restricted to systems with agents in a distributed environment and the following agency properties: autonomy, adaptation, interaction and learning. The framework deals with the intra-agent level, and focuses on the construction of agents that will become part of a large scale multi-agent system. The inter-agent level is modeled with the Anote modeling language and developed through the instantiation of the application.

These instantiated agents use some learning properties to accomplish their goals and help the system to increase the overall performance. In section 4, we describe a methodology to introduce these properties in a large scale multi-agent system with limited computational resources. Our methodology is unique because it guides the introduction of learning properties in software agents through design, implementation and testing stages. In frameworks such as JADE [10] and Kendall et all [8], this easy guide is not presented in a structured format. Consequently, our framework and methodology build systems that are easier to implement and reuse.

We have used IBM's TSpace [12] software to implement the communication infrastructure. IBM TSpace is a reflective tuple space architecture [13] that provides support to all basic associative blackboard [2] operations (read, write, and take). TSpace can also be programmed to react to specific stimuli, and we have used this feature to enable the exchange of messages between software agents. In fact, our communication infrastructure is a layer over TSpace that provides blackboard and message passing communication for agents.

The agent framework is called the MAS Framework, and it has been used in four projects. The first development [14][15] uses agents to build a tool for automatically creating offerings in a retail market. The second development [16] uses software agents to encourage people to participate in Consuming Groups, and the third project [17] uses agents for negotiation purposes in a virtual marketplace. We are now using the framework [18] to develop a multi-agent system for the Trading Agent Competition [19]. All projects were able to re-use the same code and consequently reduce the development time and effort.

In Section 2, the MAS Framework is presented in detail and in Section 3 we describe how to instantiate an application that uses this framework. In Section 4, we describe a methodology for introducing learning techniques in a multi-agent system, and in section 5, we present an instantiated application for the Trading Agent Competition that also uses the framework. In section 6, we give our conclusions and future works.

2. The MAS Framework

The main goal of the MAS Framework is to reduce development time and complexity of implementing Software Agents. This work is an evolution of the framework presented in [20]. The design of this framework allows for an easy mapping to the implementation level of models created using the Anote language in the design phase. Learning properties are built by using decision making algorithms and machine learning techniques. Other agency properties, such as mobility, are not included in this version of the framework.

2.1 Engineering a Multi-Agent System using Anote

The first step in the engineering process of a multi-agent system is to define the main goal and its decomposition into sub-goals. In Anote this is done with the Goal Diagram, and every goal in this model defines a service/functionality that must be offered by one or more agents in the system. In [11], some heuristics are presented to relate views presented in the Goal Diagram, Agent Class Diagram, Organization Diagram, Ontology Diagram, Scenario Diagram, Planning Diagram and Collaboration Diagram.

To specify the environment of the multi-agent system in Anote we use the Ontology Diagram. With this diagram we are able to specify the resources available together with their interfaces and functions. The next step for modeling our multi-agent system can take us to build the Agent Class Diagram and the Scenario Diagram. The Agent Class Diagram depicts the entities that will carry on the goals elicited in the Goal Diagram, and the Scenario Diagram specifies the behavioral descriptions of one or more agents. An important heuristic in [11] determines that functional goals in the Goal Diagram must be described in one or more Scenario Diagrams. To structure the entities in the Agent Class Diagram, an Organization Diagram can be modeled. In this diagram, we group entities that have related goals and scenarios.

Anote has two more diagrams to model the agent's dynamics: Planning Diagram and Collaboration Diagram. The Planning Diagram describes a set of actions as a state chart. It illustrates the sequential or branching of actions and the flow of emerging

actions. The Collaboration Diagram shows the structural organization of the agents that send and receive messages while executing an action plan. It helps the specification of the communication layer and interaction.

2.2 The Classes and Interfaces of the MAS Framework

The MAS Framework is composed of three abstract classes – *Agent*, *SearchAlgorithm, LearningProperty,* three final classes – *ProcessMessageThread, AgentIntelligence* and *AgentCommunicationLayer* and five interfaces – *AgentMessage, AgentBlackBoardInfo, InteractionProtocols, AgentInterface and DecisionSearchTree* . All of these classes have been developed in Java, and in the following paragraphs we will describe in detail each class and interface.

Agent is an abstract class, and the subclass that inherits it implements the private actions or activities of the software agent. These private actions do not depend on any interaction with other agents in order to accomplish specific tasks. The developer is obliged to implement methods related to the startup code (method *initialize*), ending code (method *terminate*), and display of messages (method *trace*). The methods *process* and *stopAgent* are to start and stop the agent. The attribute *name* specifies the agent's name in the system, and due to implementation details it has to be unique.

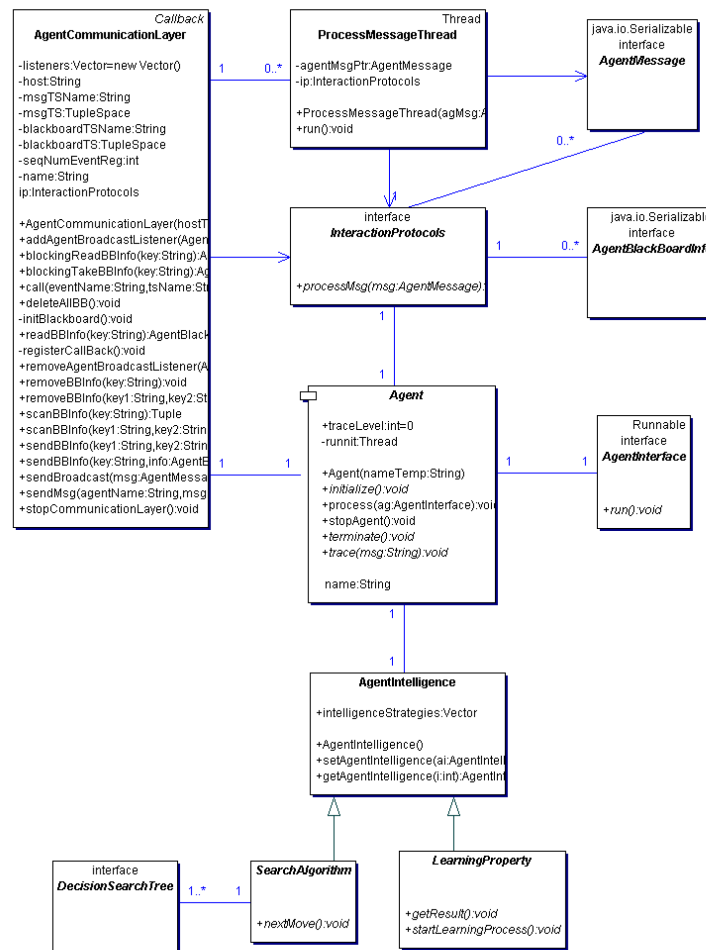

Figure 1 – MAS Framework

The *AgentInterface* is responsible for making the subclass inherited by *Agent* into a thread. This subclass implements a method called *run*, and this method is responsible for starting the agent's private activities.

The *InteraactionProtocols* is an interface of a class that will define the way a software agent can interact with other agents in the society. All the code related with interaction is placed in this class. The implemented class also requires the implementation of a method called *processMsg*. This method is called every time a new message is received from another agent.

The *ProcessMessageThread* is in charge of processing messages received by agents. In fact, it creates a new thread for every incoming message, which will automatically call the abstract method *processMsg*. The *AgentMessage* is an interface used by the class that specifies the message format, and *AgentBlackBoardInfo* is an interface used by the class that specifies the blackboard message format. Thus, all blackboard information and messages in the system must implement these interfaces.

The *AgentCommunicationLayer* is a class that implements the entire communication infrastructure needed for agents to interact in a distributed system over a network. This infrastructure is a layer over IBM Tspaces. A summarized description of every method is displayed below:

addAgentBroadcastListener - Method that adds an agent to the broadcast listener group;

blockingReadBBInfo - This method is used for reading information in the MAS blackboard. When the agent issues a *blockingReadBBInfo* call, and the data is not yet there on the MAS blackboard, the application blocks the call until an answer is returned. When the information arrives on the MAS blackboard that matches the *blockingReadBBInfo* query, it is sent to the Agent and it resumes;

blockingTakeBBInfo - This method is used for removing information in the MAS blackboard. When the agent issues a *blockingTakeBBInfo* call, and the data is not yet there on the MAS blackboard, the application blocks the call until an answer is returned. When the information arrives on the MAS blackboard that matches the *blockingTakeBBInfo* query, it is sent to the Agent and it resumes;

deleteAllBB - Method used for deleting all information in the MAS blackboard;

readBBInfo - This method is used for reading information in the MAS blackboard;

removeAgentBroadcastListener - Method that removes an agent from the broadcast listener group;

removeBBInfo - This method is used for removing information in the MAS blackboard. No blocking service is offered in this call;

scanBBInfo - This method is used for querying information in the MAS blackboard;

sendBBInfo - This method is used for posting information in the MAS blackboard;

sendBroadcast - This method is used for sending a broadcast message for the registered agents;
sendMsg - This method is used for sending a message to an agent;
stopCommunicationLayer - This method is used for ending the communication layer.

The *AgentIntelligence* is a class that implements the *Strategy* [21] design pattern, and it represents a family of different algorithms that implement learning and intelligence techniques. The subclasses of *AgentIntelligence* can vary independently from agents that use it. The subclasses *LearningProperty* and *SearchAlgorithm* are two examples of intelligence abstractions in agents, but more subclasses can also implement other strategies.

As our framework is only concerned with intelligence built through machine learning techniques, the *LearningProperty* is the abstract class that represents this abstraction. The subclasses of *LearningProperty* implements the machine learning algorithms [22] such as neural networks, support vector machines, and reinforcement learning.

The abstract classes *SearchAlgorithm* and the interface *DecisionSearchTree* implement the *Iterator* [21] design pattern. These entities separate the searching mechanism from the data structure. The subclass of *SearchAlgorithm* can implement problem solving algorithms [22] such as breadth-first-search, uniform cost search, depth-first search and minimax. The class that implements *DecisionSearchTree* represents the data structure of the decisions.

Object oriented framework design can be divided into two parts [24]: the kernel subsystem and the hot spot subsystem. The kernel subsystem design is common to all instantiated applications, and in the MAS Framework the classes *AgentCommunicationLayer, ProcessMessageThread* and *AgentIntelligence* represent it. Hot spot design describes the different characteristic of each instantiated application. In our framework the hot spots are the classes *Agent, InteractionProtocols, AgentMessage, AgentBlackBoardInfo*, *AgentInterface, LearningProperty, SearchAlgorithm* and *DecisionSearchTree*.

2.3 The MAS Framework and the code mapping of the models of Anote

The Agent view in Anote has a direct mapping with the MAS Framework. Consequently, every software agent modeled in the Agent Class Diagram has to instantiate an Agent from the MAS Framework. This instantiation includes the specialization of the abstract class *Agent*, the implementation of the interfaces (*AgentMessage, AgentBlackBoardInfo, InteractionProtocols and AgentInterface*), and the use of the objects *ProcessMessageThread* and *AgentCommunicationLayer*.

The Scenario Diagram in Anote captures the agent's behavior in a specific context, and it depicts internal actions and interaction protocols. Internal actions are plans that an agent can execute by himself, and interaction protocols are plans executed with the help of other agents. Every scenario description in the Scenario Diagram of Anote will be implemented as a method. The scenarios that represent internal actions of the agent will be coded as methods in the specialized class of *Agent*. Analogously, the scenarios that are derived from interaction protocols are coded in the class that

implements the interface *InteractionProtocols*. These interaction protocols are easily depicted with the Collaboration Diagram.

The Collaboration Diagram in Anote is very useful because it represents the interactions between agents. There is a direct mapping between the Collaboration Diagram and the sequence diagrams in UML. The sequence diagrams will use the methods coded in the specialized class of *Agent* and the class that implements the interface *InteractionProtocols*.

In section 5, we present a case study of a multi-agent system for the Trading Agent Competition. The system is modeled with Anote and implemented using the MAS Framework. All the code mappings are presented in the following section to help elicit the transformation of the models to code.

3. How to Instantiate the MAS Framework

In this section, we will describe the instantiation process of the framework, and show the interdependence of the classes. It is important to have an IBM TSpace server running in your network or local machine. The configuration of the IBM TSpace server can be found in [12].

The first class to be extended is *Agent*, and the subclass that inherits it will implement the private actions or activities of the software agent. This subclass shall also implement the interface *AgentInterface*, and write code for the methods *initialize*, *run*, *terminate*, and *trace*. When an activity ends up in an interaction protocol, it is necessary to have a reference to the class that implements the *InteractionProtocols*.

A class has to implement the interface *InteractionProtocols*, and all the code related with interaction is placed here. The method *processMsg* needs to have code that interprets an incoming message, and a reference to *AgentCommunicationLayer* is required in order to implement interaction through the communication infrastructure. In Figure 2, we present a simple instantiation of the MAS Framework.

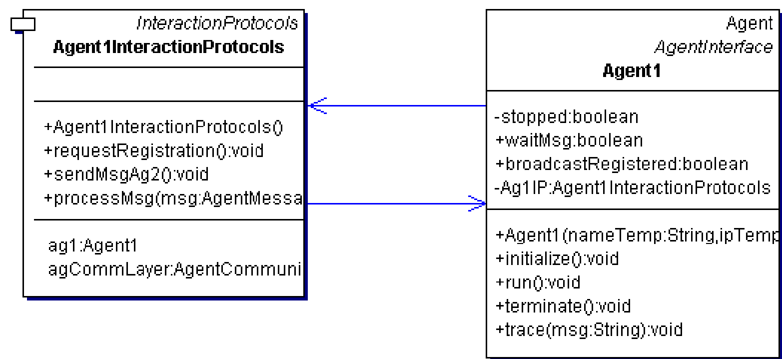

Figure 2 – A reactive agent

If the implemented software agent uses message passing in order to communicate, a class that specifies the message format shall implement the interface *AgentMessage*. If

the agent also uses a blackboard for communication, another class that specifies the message format shall implement the interface *AgentBlackBoardInfo*.

A reactive agent [2] is an agent that reacts over the environment with no prior knowledge of its history. If two events of the same kind are received by the reactive agent, an identical behavior will be executed. The instantiated agents that only extend the classes *Agent* and *InteractionProtocols* can be considered as reactive.

A cognitive agent [2] reacts over the environment using knowledge based on the agent's history, and executes complex plans to accomplish established goals. To instantiate a cognitive agent using the MAS Framework, the subclasses of *Agent* and *InteractionProtocols* are implemented together with one of the subclasses of *LearningProperty* (if learning properties are required), or *SearchAlgorithm* and *DecisionSearchTree* (if complex decision making is required).

In figure 3, we depict an example of an instantiated Software Agent that uses a Prediction Algorithm as a Machine Learning technique for the execution of plans. The class *PricePredictorAgent* extends the class *Agent* and implements the interface *AgentInterface*. The class *PricePredictorAgentIP* implements the interface *InteractionProtocols*, and the class *PricePredictorAlgorithm* extends the class *LearningProperty*.

The *PricePredictorAgent* is a software agent of the TAC Agency presented in section 5. The main goal of this agent is to predict prices of auctions in the Trading Agent Competition environment. The machine learning strategy is presented in detail in sections 4 and 5.

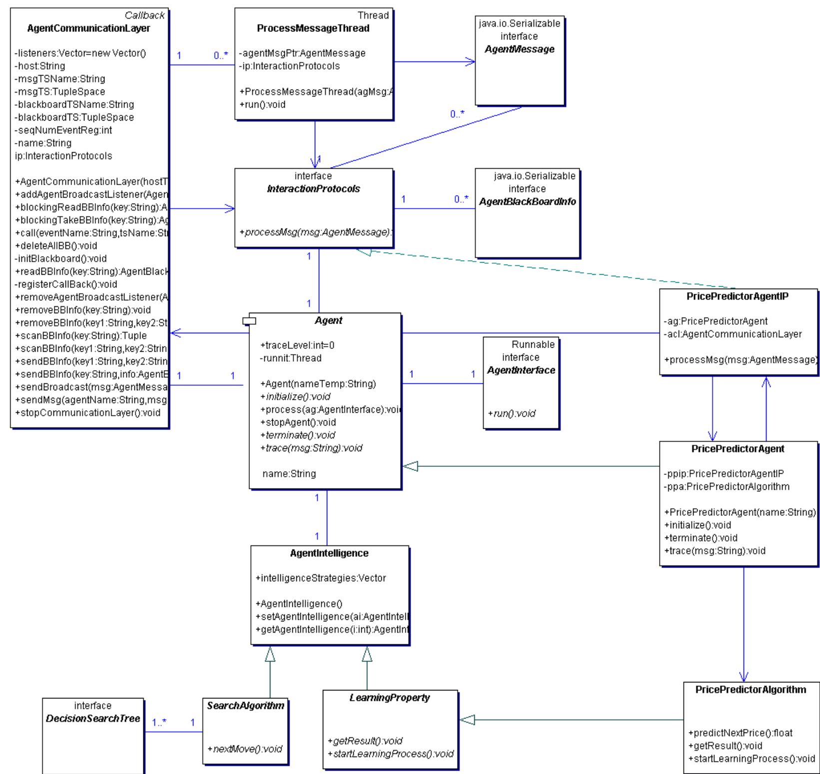

Figure 3 – A cognitive agent

4. Introducing Learning Techniques in Multi-Agent Systems

An important issue that arises when we introduce learning properties in large scale Multi-Agent Systems is how to introduce intelligence and knowledge acquisition in a society of agents. Machine learning is time consuming and it is unfeasible to allow every agent in the organization to use the computational resources at the same time. We present a methodology to introduce learning techniques in multi-agent systems built over a distributed environment and with limited resources.

4.1 A Methodology for Introducing Machine Learning Techniques

The main goal of this methodology is to introduce cognition in agents through machine learning techniques that can leverage the performance of the system. We assume that an intelligent agent can improve the performance of the society when compared to a same agent role implemented as a reactive agent. This methodology has six phases: Agent Selection, Problem Domain Analysis, Machine Learning Design, Implementation, Training, and Testing & Evaluation. In figure 4, we depict all the phases of the methodology.

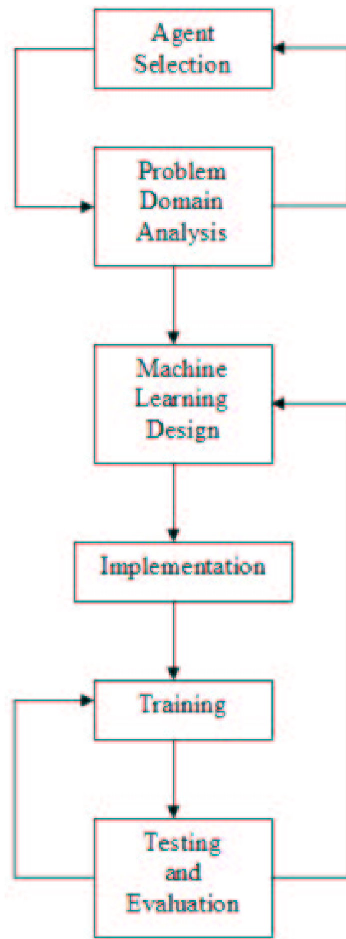

Figure 4 – The methodology for introducing Learning Techniques

4.2 Agent Selection and Problem Domain Analysis

After the definition of agent types or roles and the description of the agent behavior with a modeling language such as Anote, we shall identify the best agents that will receive a learning and intelligence behavior. An important definition in this phase is the organization's learning problem. This learning problem is defined as the Organization's Goal, *OG*, and the Organization's Performance Measure, *OP*. Normally in a design of a Multi Agent System, each agent is concerned with a sub problem, which can be solved by applying a specific machine learning technique. The combination of these solutions must achieve the organization's goal and leverage the organization's performance measure.

In the Agent Selection phase, the designer of the system notices that a specific agent has a complex plan to perform and needs a machine learning technique in order to improve the performance of the system. Moreover, for every selected agent, a problem domain analysis is performed to identify important learning issues. This phase is the Problem Domain Analysis and has the goal to establish a well-defined learning problem. In [22], this learning problem is defined with three features: a Task *T*, a Performance measure *P*, and a Training experience *E*.

4.3 Machine Learning Design

In the Machine Learning Design Phase, we make the following questions: *How am I going to get my agent to learn this?* and *What kind of evaluation function should I use to measure the performance of the agent after the learning process?* We propose a Machine Learning Design similar to [22] composed of four phases: 1- Determine a Type of Training Experience; 2 – Determine a Target Function; 3 - Determine a Representation of Learned Function; and 4 - Determine a Learning Algorithm. These four phases illustrate some basic issues and approaches to machine learning systems.

In the first phase, we deal with the design choice of which type of training experience the system will have to learn. A key attribute of the training experience is whether it is direct or indirect from the training examples. The second phase determines exactly what type of knowledge will be learned and how it will be used by the performance program. It may be very difficult in general to perfectly learn this target function, and normally we reduce the complexity and transform the problem to learn only some approximation of the target function.

In the third phase, we must choose a representation that the agent will use to describe the approximate target function. This representation can be described as a linear weighted function, a collection of rules, a neural network, or a quadratic polynomial function. In general, this design choice involves an important tradeoff because we would like to pick a representation that is as close as possible to the ideal target function. However, an expressive target function requires more training data in the training phase.

In order to learn the approximate target function we will need a training set. These training examples are obtained through a direct or indirect experience. In the direct experience, the designer can carefully select the best training that leads to the

approximate target function. However, the indirect experience requires a design that suggests actions that will lead to already known states that improve the performance of the system, and unknown states that guide to new experiences. As mentioned above, exploration is important for indirect learning agents that are willing to discover much better actions for the long run.

4.4 Implementation, Training, Testing and Evaluation

The implementation phase transforms the models into code, and leads to the next phase that is training. The training experience selected in the Problem Domain Analysis and Machine Learning Design is presented to the agent through training sessions. This training session consists of one or more Agents that learn based on a selected training set, or through interaction with other agents in a controlled training environment. Some adjustments in learning parameters are made at this time.

Testing phase starts with unit testing of the implementation and learning property. After the unit testing, integration with the Multi-Agent System is done and the testing of the performance is evaluated. A performance improvement indicates that the learning property is a successful strategy. Normally, some modifications in the machine learning design and learning parameters are made to improve the performance of the system.

In some systems, our selected agents might need to learn from the interaction with other agents or the environment through all the life cycle of the system. Consequently, the agents have to undergo training sessions from time to time. It is important to remember that machine learning techniques are time consuming and that computational resources are always limited. A careful policy is required to organize which agents are submitted to these training sessions and at what time. Moreover, a performance measure has to be monitored at all time in order to detect flaws of the system.

5. The Trading Agent Competition MAS

The Trading Agent Competition (TAC) [19] is an international forum designed to encourage high quality research on competitive trading agents. The multi-agent system in TAC operates in a shopping scenario of goods for traveling purposes. The artificial agents are travel agents that buy and sell airplane tickets, hotel rooms, and entertainment tickets for clients. TAC scores are based on the client's preferences for trips, and net expenditures in the travel auctions.

In TAC, each agent has a goal of assembling travel packages. Every package is from TACtown to Tampa, for a 5-day period. Each agent is acting on behalf of eight clients, who express their preferences for various aspects of the trip. The objective of the travel agent is to maximize the total satisfaction of its clients, with the minimum net expenditure in the travel auctions. The satisfaction is defined as the sum of all client utilities.

A run of the game is called an instance. Several instances of the game are played during each round of the competition in order to evaluate each agent's average

performance and to smooth the random variations in clients' preferences. Each game instance takes twelve minutes.

Our travel agent is modeled as a multi-agent system that trades in the related auctions of an instance. Trading problems were identified in the goal view of Anote, such as: calculate best allocations, predict auction prices and calculate demand segmentation. Each agent in our TAC Agency is concerned with one or more of these trading sub-problems. This allows us to apply different computational techniques to solve the sub-problems separately and then combine the solutions. This is an evolution of the previous work conducted by Milidiu et al. [18].

5.1 – The Goal Diagram of the TAC Agency

Our TAC Agency has the main goal of acquiring travel packages that maximize the customer's utility with the minimum net expenditure. Instead of building a single agent to achieve the goals specified in figure 5, we use a multi-agent system composed of experts that tackle each individual goal. This design strategy helps to reduce the complexity of developing an artificial travel agent for the TAC environment, and also to scale up the solution to more complex e-commerce environments. In more complex environment, we are able to re-use the implemented code and only change or include specific expert agents to match the new goals. There are two key elements for decomposing an e-commerce multi-agent system in experts: (1) Select expert agents in the system that are responsible for building a knowledge database for marketing and sales; (2) Assign goals for other agents in the procurement process.

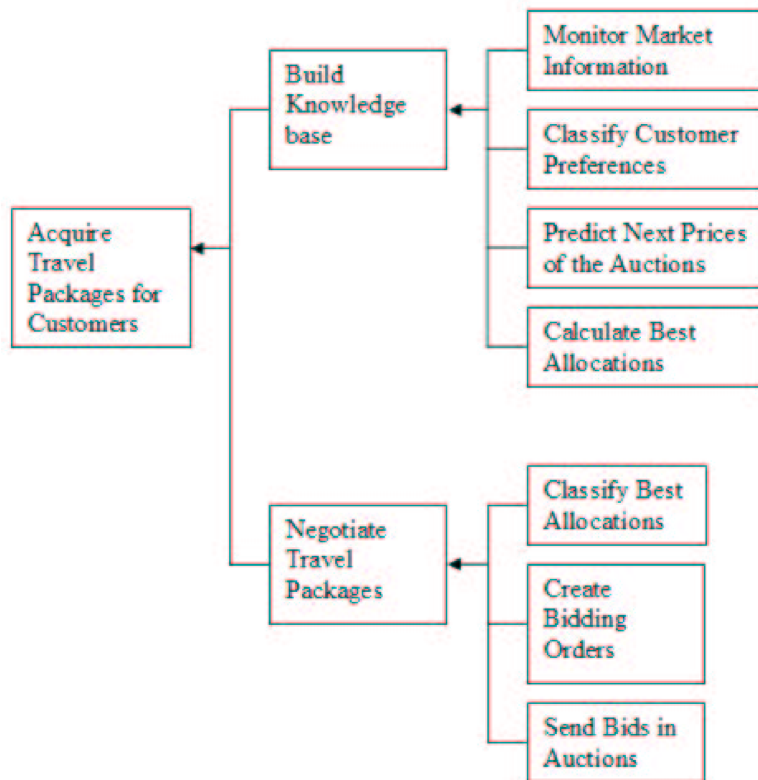

Figure 5 - The TAC agency goal diagram

5.2 – The Environment of the TAC Agency

The TAC Environment is composed of an instance of the game and several events that are sent to our Agency through a communication infrastructure. The auctions of flights, hotel rooms and entertainment tickets are part of this instance. When the instance of the game starts, the client's preferences are sent to our agency and stored in the Client Demand Information. Thus, our agency can build a strategy to obtain the maximum utility with the minimum net expenditure.

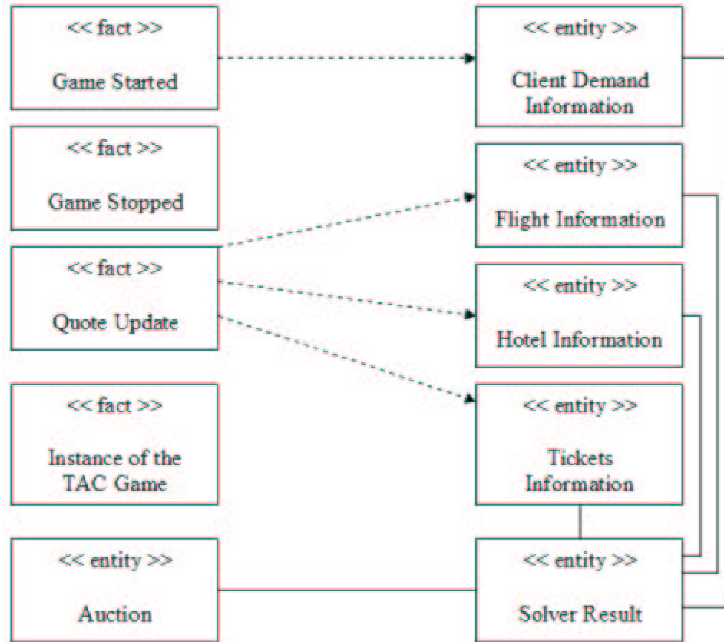

Figure 6 - The Ontology Diagram

Every moment that a quote is updated in one of the auctions, an event is fired to our agency. Consequently, the Flight, Hotel and Ticket Information are updated in our knowledge base. Our agency uses the Client Demand, Flight, Hotel, Tickets and Auction Information to execute plans that maximize our profit in the instance. This profit is calculated as the sum of the utilities obtained by the clients, minus the net expenditure in the auctions.

Entity	Classes
Client Demand Information	*DemandTable*
Flight Information	*FlightInformation*, *PredictedFlightInformation*
Hotel Information	*HotelInformation*, *PredictedHotelInformation*
Tickets Information	*TicketsInformation*, *PredictedTicketsInformation*
Solver Results	*SolverResults*, *SolverResult*
Auction	*Auction*

Table 1 - The code mapping of the Ontology Diagram to the Class Diagram

Every entity in the Ontology Diagram is mapped as one or more classes in our TAC Agency Environment. The facts are events fired by the TAC Instance to our TAC Agency. Moreover, this method call sends events to agents in our agency. In figure 7, we present the class diagram of the entities in the Ontology Diagram. The code mapping is presented in table 1. The class *DynamicMap* is implemented as the Façade and Singleton design pattern [21]. This class is the main interface to all of the TAC Agency Environment.

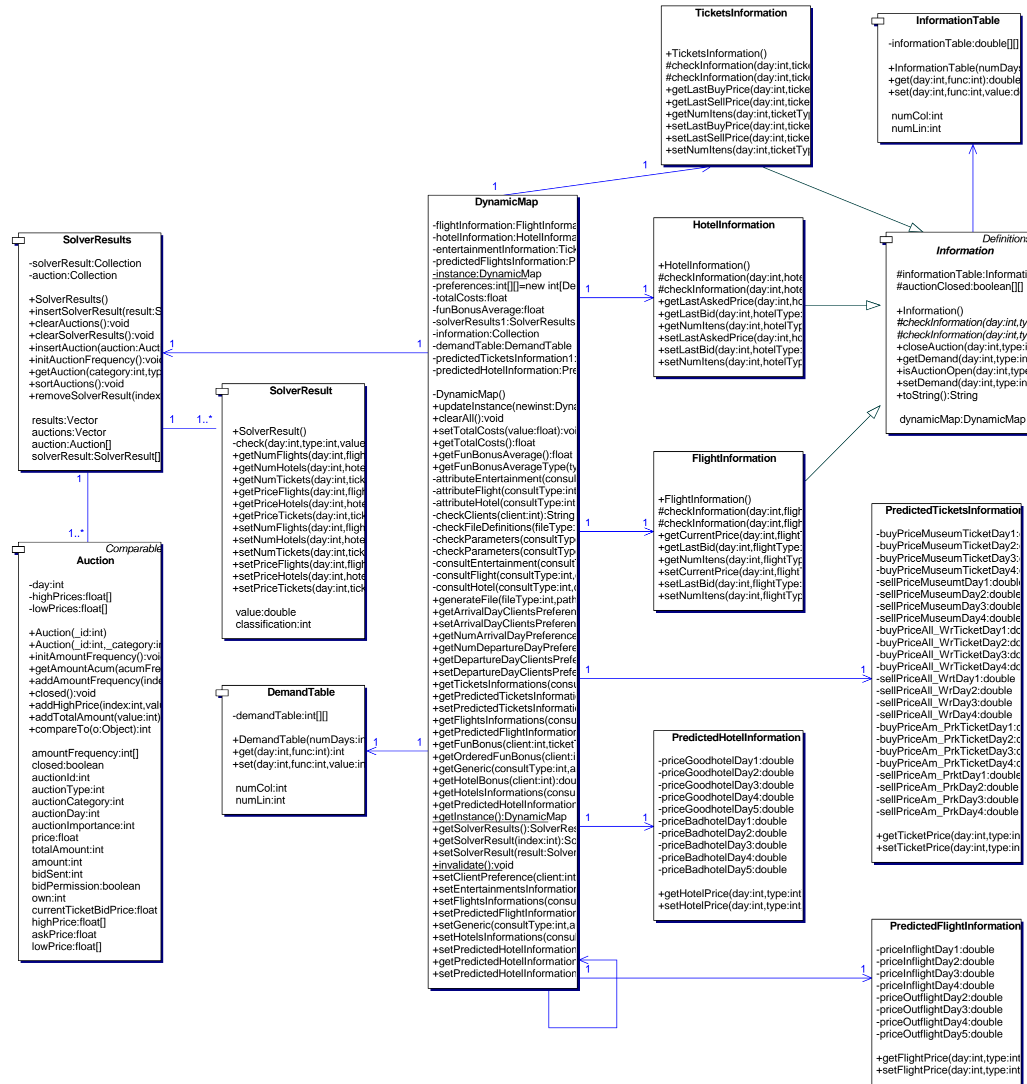

Figure 7 - The Class Diagram of the Ontology

5.3 – The TAC Agency

Every sub-goal modeled in the Goal Diagram is conducted by one or more agents in the TAC Agency. These goals also affect the environment modeled in the Ontology Diagram by the agents. In figure 8, we present the Agent Class Diagram of the TAC Agency. Every agent in this diagram was implemented with the MAS Framework. An

example of the code mapping is depicted in figure 3 of section 3, where the Price Predictor Agent is instantiated with the MAS Framework.

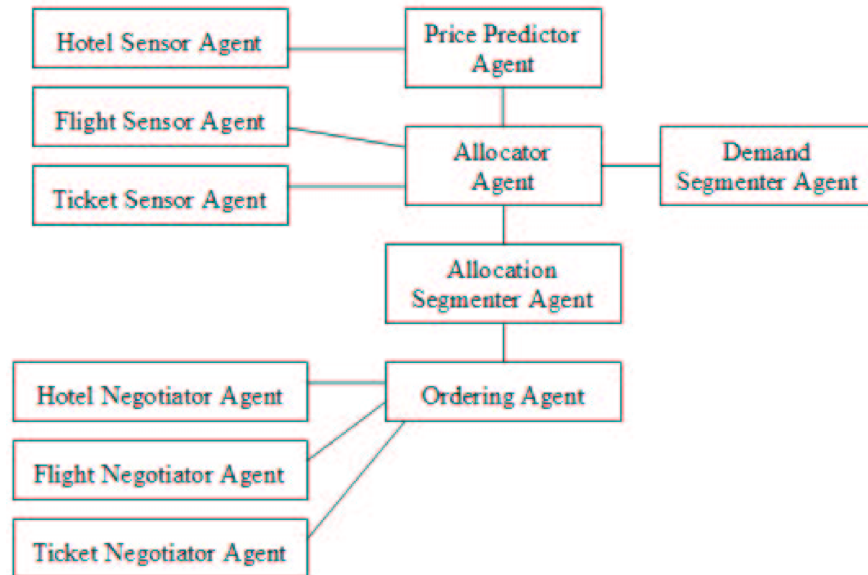

Figure 8 - The Agent Class Diagram

The Hotel Sensor Agent, Flight Sensor Agent and Ticket Sensor Agent are responsible for collecting data and monitoring the market. The Price Predictor Agent has the duty of predicting hotel auction prices for the knowledge base. In the first minute, the agent predicts the closing ask price (ask price - calculated as the 16th highest price among all bid units) and in the following minutes of the game instance, it predicts the ask price of the next minute. The strategy of predicting the closing asks price in the first minute of the game is used by the Allocator Agent, Allocation Segmenter Agent, and Ordering Agent. Together they cooperate to buy flight tickets in the first minute, and reduce the net expenditure.

The Demand Segmenter Agent classifies the customer preferences in order to minimize the risk of buying flight tickets and not purchasing hotel rooms afterwards. The segmentation of the demand transforms customer preferences that are considered to by very risky to preferences of goods that are easier to buy. These preferences are also saved in the knowledge base of the agency.

The Allocator Agent uses a service of an integer programming model to obtain the optimal allocation of available goods for the customers. The service is set in motion not only to search for the optimal allocation, but as many best allocations as the solver can optimize in 25 seconds. Such model is executed during the game instances to indicate which goods the agents must acquire. The model receives price quotes, predicted prices, goods already bought and preferences of the clients. All this information is obtained through the knowledge base of the agency.

The Allocation Segmenter Agent receives all of the optimal allocations and classifies them. Based on these segmentation results, the Ordering Agent calculates the amounts of required goods, maximum and minimum prices of the bids and importance of

goods. Moreover, the Ordering Agent sends purchase orders to the negotiating agents. These orders are received and converted to bids for the auctions.

5.4 – The Scenario Diagram of the TAC Agency

The Class Diagram depicts the entities that will carry on the goals elicited in the Goal Diagram, and the Scenario Diagram specifies the behavioral descriptions of one or more agents. In figure 9, we illustrate the dynamics of the Hotel Negotiator Agent.

<table>
<tr><th colspan="2">Send Hotel Bids in Auctions</th></tr>
<tr><td>Main Agent</td><td>Hotel Negotiator Agent</td></tr>
<tr><td>Preconditions</td><td></td></tr>
<tr><td>Main Action Plan</td><td><pre>WHILE (aution open)
    Obtain Current Ask Price
    Obtain Number of Goods Needed and High
    Price from Ordering Agent
    IF (Goods are Needed) AND
       (Actual Bid < High Price) THEN
            Negotiate using Intelligence
    END IF
    Wait for next auctions ask price
    refresh from the Game Instance
END WHILE
IF (Agent participates in Auction) THEN
    Train intelligence
END IF</pre></td></tr>
<tr><td>Interaction</td><td>Ordering Agent</td></tr>
<tr><td>Variants</td><td></td></tr>
</table>

Figure 9 - The Scenario Diagram

In this diagram, we specify the action plans and interactions that are performed by the agent. The Hotel Negotiator Agent only sends bids to the Game Instance if purchase orders are received. This communication with the Ordering Agent is made in an asynchronous form through a blackboard.

Every agent in the system has at least one Scenario Diagram, and all of these scenarios were mapped into methods of the subclasses of *Agent* and *InteractionProtocols* in the MAS Framework.

5.5 – The Collaboration Diagram of the TAC Agency

Interaction in the system generates message protocols that depict the communication dynamics of the agents. Although this interaction is specified in the Scenario Diagram, the Collaboration diagram clearly guides the specification of the interactions into the implementation phase.

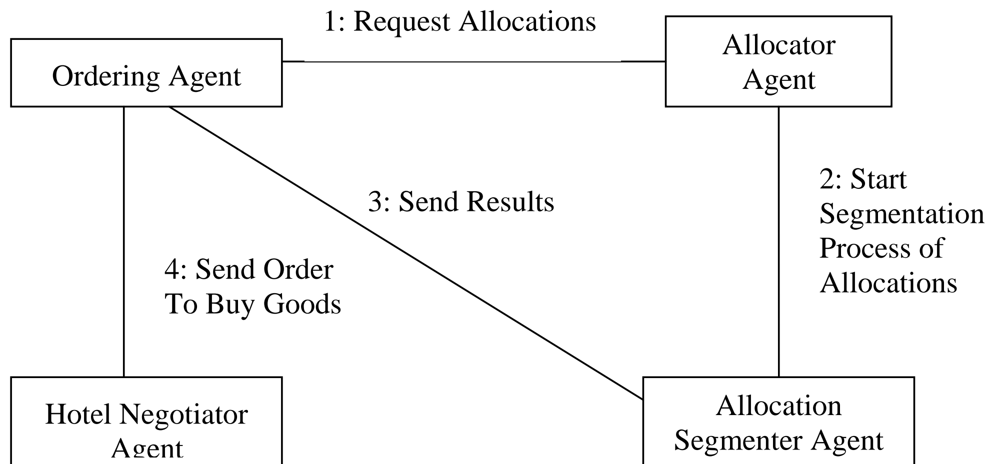

Figure 10 - The Collaboration Diagram (High Level)

In figure 10, we present the Collaboration Diagram of the Ordering Agent, Allocator Agent, Allocation Segmenter Agent, and Hotel Negotiator Agent. Whenever a Hotel Quote Update event is received, the Ordering Agent requests the calculation of new allocations and starts the segmentation process. With these results, an order is sent to the Hotel Negotiator Agent to buy goods. Every interaction protocol in the TAC Agency is depicted as a Collaboration Diagram and is an important tool for the implementation and maintenance phase.

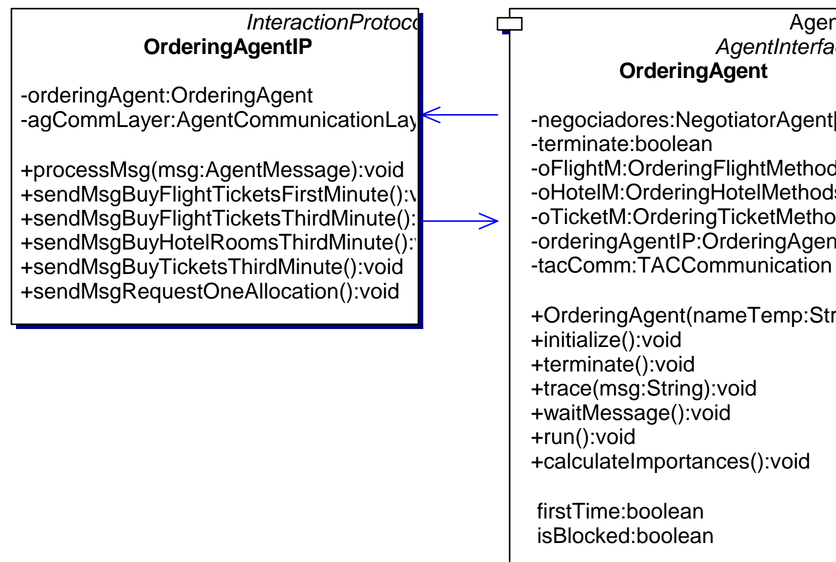

Figure 11 - The Class Diagram of the Ordering Agent

The code mapping of the Collaboration Diagram to the MAS Framework is very easy. In figure 11, we present the classes that implement the code of the Ordering Agent (*OrderingAgent* and *OrderingAgentIP*). The method *sendMsgRequestOneAllocation*

in the class *OrderingAgentIP* has the code that sends the message *1: Request Allocation* in figure 10. In this method, we have calls to *AgentCommunicationLayer* that is responsible for sending the message through the communication infrastructure.

The method *processMsg* of the class *OrderingAgentIP* is called whenever a Message Received Event is fired by the communication infrastructure. Consequently, the message *3: Send Results* (received from the Allocation Segmenter Agent) is processed in this method. Moreover, the message *4: Send Order To Buy Goods* is also implemented by the method *sendMsgBuyHotelRoomsThirdMinute* in the *OrderingAgentIP* class.

5.6 – Introducing Machine Learning Techniques in the TAC Agency

A similar version of the agency modeled above was implemented by Milidiu et al. [18]. The agency is called SIMPLE and is composed of reactive agents with a service of an integer programming model to obtain the optimal allocation of available goods for the customers.

The result presented in table 2 was obtained when playing against 7 DummyAgents [19]. SIMPLE obtained first place in 87,5% of the game instances and in 96,8% it achieved either first or second place.

Average Postion	Average Score	Games
1,18	1372	32

Table 2 – Experimental results of SIMPLES against 7 dummies

An agent with similar features (LA-clone) of the LivingAgents [25], the winner of the 2001 competition, was implemented to enhance the testing environment of SIMPLE. The best strategy of SIMPLE provided a 56,2% winning rate, when competing against one LA-clone and six Dummies. Table 3 presents the results of SIMPLE and LA-Clone.

Agent	Average Score	Games
SIMPLE	1740	32
LA-Clone	1390	32

Table 3 – Experimental results of SIMPLES against LA-Clone and 6 dummies

Our goal is to improve the performance of the agency by using the methodology and framework presented in section 2 and 4. Therefore, our first step in the methodology is to define the organization's learning problem:

Organizations's Goal, *OG* : Obtain first place in the competition
Organization's Performance Measure, *OP* : Average Score

5.6.1 Agent Selection and Problem Domain Analysis

We notice that two agents have complex plans to perform and need a machine learning technique in order to improve the performance of the system. The first agent is the Hotel Negotiator Agent and the second is the Price Predictor Agent. A problem domain analysis is performed to identify the type of problem, performance measure and training experience.

The learning problem is defined with three features: a Task T, a Performance measure P, and a Training experience E. The Hotel Negotiator Agent has the following training features:

T : negotiating hotel rooms;
P : percentage of goods purchased in the purchase order;
E : interaction in the game instance.

The Price Predictor Agent has the following training features:

T : predict future ask prices for hotel rooms;
P : error between predicted price and actual price;
E : history of ask prices.

5.6.2 Machine Learning Design

In the Machine Learning Design Phase, we use the four phases presented in [22]: 1- Determine a Type of Training Experience; 2 – Determine a Target Function; 3 - Determine a Representation of Learned Function; and 4 - Determine a Learning Algorithm.

5.6.2.1 Training Experience Type

In the first phase, we deal with the design choice of which type of training experience the system will have to learn. The training experience of the Hotel Negotiator Agent is indirect from the training examples, because the agent will build its knowledge through the final results of the negotiations. To be exact, it compares the number of acquired goods with the number of goods to be purchased from the Ordering Agent. The Price Predictor Agent uses a direct knowledge building, since it uses the ask prices in the past to predict the next one.

5.6.2.2 Target Function

The second phase determines exactly what type of knowledge will be learned and how it will be used by the performance program. The target function for the Hotel Negotiation Agent is called *NextBid*, and it is modeled as a function that accepts a state S of the environment modeled in the Ontology Diagram and produces a value B of the next bid. Therefore, the target function is $NextBid: S \rightarrow B$. The target function for the Price Predictor Agent is called *NextAskPrice*, and accepts the ask price A of the last game instances and produces the next ask price N. Consequently, the target function for the Price Predictor Agents is $NextAskPrice: A \rightarrow N$.

5.6.2.3 Learned Function Representation

In the third phase, we must choose a representation that the agent will use to describe the approximate target function. In the Hotel Negotiator Agent, we use a min-max procedure [23] and reinforcement learning technique as the evaluation function for the min-max algorithm.

The min-max procedure is a search technique for a two player game that decides the next move. In this game there are two players: MAX and MIN. A depth-first search tree is generated, where the current game position is the root. The final game position is evaluated from MAX's point of view, and the inner node values of the tree are filled bottom-up with the evaluated values. The nodes that belong to the MAX player receive the maximum value of the children. The nodes for the MIN player will select the minimum value of the children. The min-max procedure is also combined with a pruning technique called Alpha-Beta [23].

Reinforcement learning [22] is different from supervised learning [22], since the agent is not presented with a learning set. Instead, the agent must discover which actions yield the most reward by trying them. Consequently, the agent must be able to learn from the experience obtained from the interaction with the environment and other agents. A challenge that arises in reinforcement learning is the tradeoff between exploration and exploitation. Most rewards are obtained from actions that have been experienced in the past. But to discover such actions and to earn better selections, the agent must explore new paths and eventually fall in to pitfalls.

Every state of the min-max search tree is modeled with the following information obtained from the environment:

AskPrice – current Ask Price
LastAskPrice – Ask Price of the last minute
deltaAskPrice – AskPrice - LastAskPrice
Gama – A Constant
Bid – Current Bid
LastBid – Bid sent in the last minute

The calculation of the next bid sent to the game instance is based on the decision taken with the min-max procedure as shown in figure 12. Three decisions can be taken by the decision tree:

1 – Bid = LastBid + 0.5*Gama*deltaAskPrice
2 – Bid = LastBid + 2*Gama*deltaAskPrice
3 – Bid = LastBid + 5*Gama*deltaAskPrice

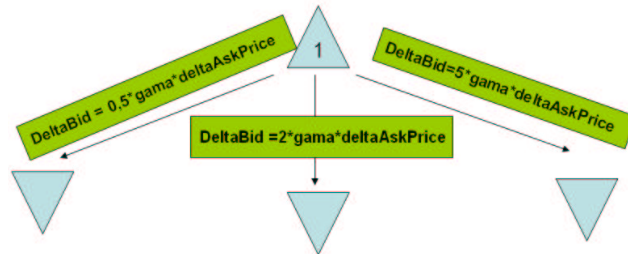

Figure 12 - The Maximizing Level of the min-max procedure

The MIN player is modeled as the market response to the bid sent by our player (representing the MAX player). The market response is modeled as three possible results as shown in figure 13:

1 – AskPrice = AskPrice + 0.5*Gama*deltaAskPrice
2 – AskPrice = AskPrice + 2*Gama*deltaAskPrice
3 – AskPrice = AskPrice + 5*Gama*deltaAskPrice

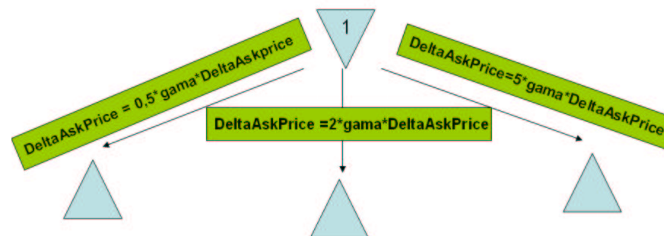

Figure 13 - The Minimizing Level of the min-max procedure

For the evaluation function, we use a single-layer perceptron with only one artificial neuron. This perceptron receives a state of the min-max search tree as an input and scores the state between 0 and 1. The main goal of this agent is to adapt the weights of the perceptron to encounter an evaluation function that leads to good final states. These good final states represent environment states where bids are sent to the Game Instance and are able to purchase all ordered goods with the least net expenditure.

The Price Predictor Agent uses a simple representation to describe the approximate target function. The following formula is used to predict the next price:

$$PredictedAskPrice(n) = \alpha * AskPrice(n-1) + (\alpha - 1) * PredictedAskPrice(n-1)$$

where:

α is a number between 0 and 1;
n is the n-th game instance.

5.6.2.4 Determine a Learning Algorithm

In order to learn the approximate target function we will need a training set. These training examples are obtained through a direct or indirect experience. The Hotel Negotiator Agent uses an indirect experience because it uses data from the environment to adapt the weights of the perceptron rule [22]:

$w_i=w_i+ \eta \cdot (y(t) - d(t)) \cdot x_i$, where

w_I is the i-th weight of the perceptron

x_i is the i-th data input of the state in the min-max search tree

$y(t)$ is the actual result of the evaluation function

$d(t)$ is the expected result of the evaluation function

η is the learning rate.

Whenever the game instance ends up, a Perceptron rule is computed for all of the traversed intermediate and final states. Moreover, the perceptron adapts its weight using the Perceptron rule above and a TD-Learning strategy [22]:

$d(t-1)=d(t) + ß\ (reward(t) + (d(t)-d(t-1)))$, where:

$d(t)$ is the expected output of the decision point at time t,

ß is the Reinforcement learning rate,

$reward(t)$ is the reward obtained in the decision point at time t.

The value of d(t) that represents a final states is computed with a reward value of 1 when it accomplishes the goal of purchasing all the goods in the purchase order, or a reward value of 0 in all other states. For final states d(t) is calculated using the following formula:

$d(t) = reward(t)$

The Price Predictor Agent uses a Least Mean Squares (LMS) [22] strategy to compute the value of α:

$\alpha(n) = \alpha(n-1) + ß * (AskPrice(n-1)-PredictedAskPrice(n-1))$

where ß is a learning rate.

In order to learn the approximate target function we will need a training set. The Price Predictor Agent uses a direct experience. We selected the 50 last ask prices and predicted ask price to train the approximate target function.

5.6.3 Testing and Evaluation

To evaluate the performance of each learning strategy, we first tested the TAC Agency only with the Hotel Negotiator Agent. Consequently, we excluded the Price Predictor Agent and the Ordering Agent only sent purchase orders for flight at the end of the Game Instance. This strategy is used because the Allocation Agent is using the

current ask prices of the environment, and it is safer to buy the flight tickets after the hotel rooms are acquired.

The result presented in table 4 was obtained when playing against 7 DummyAgents. The TAC Agency obtained first place in 93.4% of the game instances and in 98.5% it achieved either first or second place.

Average Postion	Average Score	Games
1,08	1752	32

Table 4 – Experimental results of the TAC Agency against 7 dummies

The agent with similar features (LA-clone) of the LivingAgents was tested with the TAC Agency with only the Hotel Negotiator Agent. The best strategy of the TAC Agency provided a 62.3% winning rate, when competing against one LA-clone and six Dummies. Table 5 presents the results of TAC Agency and LA-Clone.

Agent	Average Score	Games
TAC Agency	1920	32
LA-Clone	1355	32

Table 5 – Experimental results of TAC Agency against LA-Clone and 6 dummies

The same tests were performed with the Hotel Negotiator Agent and the Price Predictor Agent. The result presented in table 6 was obtained when playing against 7 DummyAgents. The TAC Agency now obtains first place in 99.3% of the game instances and in 100% it achieved either first or second place.

Postion	Average Score	Games
1.01	2214	32

Table 6 – Experimental results of the TAC Agency against 7 dummies

The agent with similar features of the LivingAgents was again tested with the TAC Agency with both the Hotel Negotiator Agent and the Price Predictor Agent. The best strategy of the TAC Agency provided a 73.4% winning rate, when competing against one LA-clone and six Dummies. Table 7 presents the results of TAC Agency and LA-Clone.

Agent	Average Score	Games
TAC Agency	2592	32
LA-Clone	1323	32

Table 7 – Experimental results of TAC Agency against LA-Clone and 6 dummies

A final test was performed where our TAC Agency played against the SIMPLE agency to evaluate the performance obtained with the introduction of learning

strategies. The best strategy of the TAC Agency provided a 85,6% winning rate, when competing against one SIMPLE and six Dummies. Table 8 presents the results of TAC Agency and SIMPLE.

Agent	Average Score	Games
TAC Agency	1977	32
SIMPLE	1002	32

Table 8 – Experiment results of TAC Agency against SIMPLE and 6 dummies

6. Final Comments

The paper presents a multi-agent system framework for building intelligent agents. The intelligence of an agent relies on machine learning techniques in order to perform complex plans, adapt the beliefs and achieve the predefined goals. In complex and open environments with many cooperating agents, it is important to have a system that is able to adapt to unknown situations. Learning techniques are crucial to the development of open multi-agent systems since they provide well-known strategies to support the construction of adaptable agents.

We present an easy mapping of the models of Anote to the framework, because we believe that a good engineering of the system requires a useful language for the analysis and design phase of a multi-agent system and a simple transformation of this language to code. We decided to use Sun´s Java [26] as the generated code in order to enable our software agents to run in almost any operating system.

Our framework is being used in many implementations, and has proved to reduce time and complexity of the development stage. The framework uses IBM´s TSpace as the communication infrastructure to enable the distribution of agents in the system. Moreover, a multi-agent system can only be scaled up if distribution policies are implemented. IBM´s TSpace is a reliable and easy to use reflective tuple space environment that enables these distribution policies.

A software engineering methodology is also presented for the disciplined introduction of learning properties in software agents through different development stages. This systematic approach helps the development team of an open multi-agent system to include machine learning techniques in adaptive environments, and consequently, leverage the performance of the system. The methodology also reduces the complexity of modeling, implementing and evaluating learning techniques in large scale multi-agent systems. The machine learning design phase of the methodology relies on a simple but general enough model presented by [22]. This model permits an easy modeling of machine learning techniques in a single agent, and we present a methodology that generalizes the inclusion of these learning techniques in multi-agent systems.

Our results in the TAC environment indicate that learning techniques can leverage the performance of a system. In the results, a 97,3% improvement of performance was observed, when the TAC Agency played against the SIMPLE Agency. This result is

due to the engineering of machine learning techniques in the TAC Agency using our framework and methodology.

In future works, we would like to instantiate other applications that use different machine learning techniques such as rule based systems and learning decisions trees. To permit adaptive policies for distribution, our framework will need to enable mobility. This property could be extremely useful for agents with learning techniques, and systems that require agents to learn during all of its lifecycle. The training process is normally time consuming and requires computational availability, and it would be interesting to implement policies that move agents to other machines during this learning stage.

References

[1] Wooldridge, M. *Intelligent Agents*. In: Weiss, G. Multiagent systems: a modern approach to distributed artificial intelligence. The MIT Press, Second printing, 2000.

[2] Ferber, J. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Pub Co, 1999.

[3] Garcia, A.; Silva, V.; Lucena, C.; Milidiú, R. *An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems*. Simpósio Brasileiro de Engenharia de Software, Rio de Janeiro, Brasil, Outubro 2001.

[4] Garcia, A.; Lucena, C.; Cowan, D. *Agents in Object-Oriented Software Engineering*. Software: Practice and Experience, Elsevier, May 2004, pp. 1-32.

[5] Garcia, A.; Lucena, C. J. *An Aspect-Based Object-Oriented Model for Multi-Agent Systems*. 2nd Advanced Separation of Concerns Workshop at ICSE'2001, May 2001.

[6] M. Fayad, D. Schmidt. *Building Application Frameworks: Object-Oriented Foundations of Design*. First Edition, John Wiley & Sons, 1999.

[7] Sen, S.; Weiss, G. *Learning in Multiagent Systems*. In: Weiss, G. Multiagent systems: a modern approach to distributed artificial intelligence. The MIT Press, Second printing, 2000.

[8] Kendall, E.; Krishna, P.; Pathak, C.; Suresh, C. *A Framework for Agent Systems.* In: Implementing Application Frameworks – Object-Oriented Frameworks at Work, M. Fayad et al. (editors), John Wiley & Sons, 1999.

[9] Silva, V.T.; Lucena, C.J.P. *Um Modelo Orientado a Objetos para Sistemas Multi-Agentes.* MCC30/01. Departamento de Informática. PUC-Rio. October 2001.

[10] Telecom Italia Lab. *JADE Programmer's Guide*. http://sharon.cselt.it/ projects/ jade/ doc/ programmersguide.pdf, Feb. 2003.

[11] Noya, R. C. *Uma Linguagem de Modelagem para Sistemas Baseados em Agentes*. PhD Thesis. Departamento de Informática, PUC-Rio. Dezember 2002.

[12] IBM TSpaces Web Site. *http://www.almaden.ibm.com/cs/TSpaces/.*

[13] Silva, O; Garcia, A; Lucena, C.J. T-Rex: *A Reflective Tuple Space Environment for Dependable Mobile Agent Systems*. III WCSF at IEEE MWCN 2001, Recife, Brasil, August 2001.

[14] Sardinha, J. A. R. P. *VGroups – Um framework para grupos virtuais de consumo*. Master's dissertation – Departamento de Informática – PUC-Rio. March 2001.

[15] Milidiu, R.L.; Lucena, C.J.; Sardinha, J.A.R.P. *An object-oriented framework for creating offerings*. 2001 International Conference on Internet Computing (IC'2001) June 2001.

[16] Bevilacqua, F.; Sardinha, J. A. R. P. *Estruturas dinâmicas de incentivos para grupos de consumo*. Multi-agent system workshop. Departamento de Informática. PUC-Rio. July 2001. ISBN 85-7493-155-1.

[17] Ribeiro, P.C. *Modelagem e Implementação OO de Sistemas Multi-Agentes*. Master's Dissertations, Departamento de Informática, PUC-Rio, 2001.

[18] Milidiú, R. L.; Melcop, T.; Liporace, F.; Lucena, C. *SIMPLE – A Multi-Agent System for Simultaneous and Related Auctions*. IV Encontro Nacional de Inteligência Artificial 2003. SBC 2003.
[19] TAC web site. *http://www.sics.se/tac*.
[20] Sardinha, J.A.R.P.; Ribeiro, P.C.; Lucena, C.J.P.; Milidiú, R.L. *An Object-Oriented Framework for Building Software Agents.* Journal of Object Technology. January - February 2003, Vol. 2, No. 1.
[21] Gamma, E. et al. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995. ISBN 0201633612.
[22] Mitchell, T. M. *Machine Learning*. McGraw-Hill, 1997. ISBN 0070428077.
[23] Russell, S. et al. *Artificial Intelligence*. Prentice Hall, 1995. ISBN 0-13-103805-2.
[24] Fontoura, M.F.; Haeusler, E.H.; Lucena, C.J.P. *The Hot-Spot Relationship in OO Framework Design*. MCC33/98, Computer Science Department, PUC-Rio, 1998.
[25] Fritschi, C.; Dorer, K. *Agent-oriented software engineering for successful TAC participation.* Proceedings of the first international joint conference on Autonomous agents and multiagent systems. 2002.
[26] Java Web Site. *http://java.sun.com*.