An OO Framework for building Intelligence and Learning properties in Software Agents

José A. R. P. Sardinha, Ruy L. Milidiú, Carlos J. P. Lucena, Patrick Paranhos

Abstract

Software agents are defined as autonomous entities driven by beliefs, goals, capabilities and plans, and other behavioral properties such as adaptation, learning, interaction, and mobility. Software agents are the focus of considerable research in the artificial intelligence community, but there is still much to be done in the field of software engineering in order to systematically create large MAS (Multi-Agents Systems). In practice, they are often built in an ad-hoc manner and are error-prone. In this paper, we present an object-oriented framework that introduces intelligence and learning properties in agents for competition environments. We use Artificial Intelligence techniques and OO Frameworks to help the development of such complex properties in a systematic way. We present an instantiated application that uses this framework to illustrate an implementation.

1. Introduction

We define Software Agents [1][2][3] as an autonomous entity driven by beliefs, goals, capabilities, plans and a number of behavioral 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 programming language to implement the agent technology. 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 believe that this common goal can be accomplished through learning techniques and decision making algorithms. In [6], we developed an object oriented framework [7] for building software agents, and encouraged the use of an organizational process in the analysis and design phase with the Gaia Methodology [8]. We present here an extension to such framework in order to introduce intelligence and learning properties in a systematic way.

Another issue that arises when we introduce learning properties in large scale Multi-Agent Systems is how to introduce skill management and other policies related to knowledge acquisition. Machine learning is time consuming and it is unfeasible to allow every agent in the society to use the computational resources at the same time. Instead, we can introduce learning capabilities only in a few agents, and restrict all the others to a non learning behavior. However, this naïve policy will not build a dynamic and flexible organization. Several approaches to Skill Management are presented in [9] [10] [11]. In [9], the system is used in e-Learning software, and it presents a multi-agent system which assist managers and employees in the Skill Management process. In [10], intelligent agents are presented to produce better teamwork simulation, to work with humans as virtual team members, and to facilitate team training. In [11], a system is presented to support capability management in organizations, and to align skills of present and future employees with strategic business objectives. The main difference is that we want to introduce Skill Management in a totally artificial environment. In an organizational design, we can introduce agents with different roles

that are responsible for managing the skill management of the entire artificial organization. These roles have the following responsibilities: the allocation of an agent team to solve a given task; the measurement of agent resources and capabilities; the decision to point out capability gaps of each agent; the indication of agents that must undergo personalized training sessions. This paper does not present learning policies as part of the framework, but we consider this feature an extremely important issue that has to be considered in our future work.

The learning framework is called MAS-L, and it is used in two experiments that also use the MAS Framework [6]. The first development uses agents that play matches of TIC-TAC-TOE, and are able to learn a strategy to win or draw. The second development uses software agents to participate in the Trading Agent Competition (TAC) [12]. Our Negotiator Agent uses this framework to adapt the bids and win the desired good. 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 Section 3, the MAS-L is presented in detail and in Section 3 we describe how to instantiate an application that uses this framework. In Section 4 and 5, we present a case study of a project that uses the framework.

2. The MAS Framework

The main goal of the MAS Framework is to diminish the development time and reduce the complexity of implementing Software Agents. The design of this framework is inspired on the Gaia Methodology, and it can introduce an easy implementation of all the models developed by the analysis and design phase. Gaia is a methodology for building models in the analysis and design phase. It is applicable to a wide range of multi-agent systems, and also deals with both the macro-level (societal) and the micro-level (agent) aspects of the system. It considers the system as a computational organization consisting of various interacting roles.

Gaia allows an analyst to go systematically from a statement of requirements to a design of models that can be coded, for example, with the MAS Framework. The first two models that are built in Gaia are the Roles model and the Interaction Model. The Roles model identifies the key roles in the system or an entity's expected function. However, there are many dependencies and relationships between the various roles in an agent organization. These dependencies and relationships originate the interaction model, which is responsible for describing the various interaction situations between the agent's roles. Both the Roles model and the Interaction model derive three other models: Agent model, Services model, and Acquaintance model. The Agent model describes an agent type with set of roles (as identified in the Roles model). The Services model is to identify the services associated with each agent role, or we can understand as the agent's functions. The Acquaintance model defines the communication links that exist between agent types.

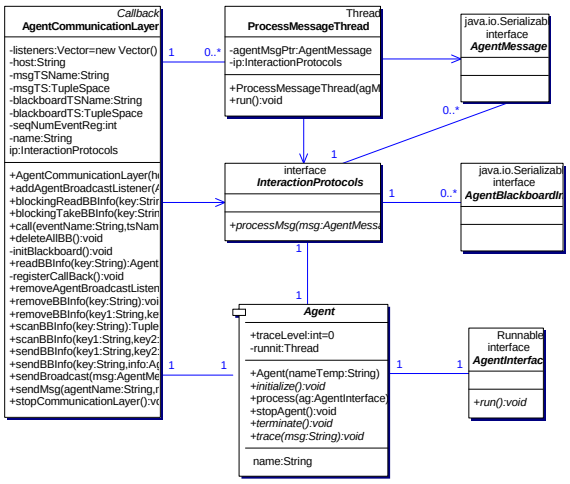

Figure 1 – MAS Framework

The MAS Framework is composed of one abstract class – *Agent*, two final classes – *ProcessMessageThread* and *AgentCommunicationLayer* and four interfaces – *AgentMessage, AgentBlackBoardInfo, InteractionProtocol and AgentInterface*. All of these classes have been developed in Java [13], and are presented in Figure 1. The Agent model in Gaia has a direct mapping with the MAS Framework. Consequently, every software agent modeled in the Agent model 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, InteractionProtocol and AgentInterface*), and the use of the objects *ProcessMessageThread* and *AgentCommunicationLayer*.

Every service in the Services model will be implemented as a method. The services that are derived from activities in the Roles model will be coded as methods in the specialized class of *Agent*. Analogously, the services that are derived from protocols are coded in the class that implements the interface *InteractionProtocol*. Furthermore, the inputs and outputs of the Services model and the permissions of the Roles model are coded as attributes of the specialized class of *Agent*. The pre-conditions and post-conditions of each service are also coded in the specialized class of *Agent*, and it is possible that some of these pre-conditions and post-conditions have to be implemented using monitors. The Interaction model in Gaia is very useful because it represents the interactions between the roles in the Roles model. As the roles are mapped to agents in the Agents model, the interactions also represent interactions between agents. Consequently, there is a direct mapping between the interaction models 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 *InteractionProtocol*.

Object oriented framework design can be divided into two parts [14]: the kernel sub-system and the hot spot subsystem. The kernel subsystem design is common to all instantiated applications, and in the MAS Framework the classes *AgentCommunicationLayer* and *ProcessMessageThread* represent it. Hot spot design describes the different characteristic of each instantiated application. In our framework the hot spots are the classes *Agent, InteractionProtocol, AgentMessage, AgentBlackboardInfo* and *AgentInterface*.

2. The MAS-L Framework

The MAS-L Framework is combined with the MAS Framework to implement intelligence and learning properties in Software Agents. The MAS-L Framework is designed for Multi-Agent Systems in a competition environment, and it is inspired on two techniques of Artificial Intelligence: a decision making search tree and an evaluation function based on machine learning.

The decision making search tree helps to organize in a systematic way all the possible decisions a software agent can take based on a current state. We also introduce customized search algorithms to encounter this best decision. The nodes of this tree are intermediate decision point states and the leaves are final states where it is possible to evaluate if all the decisions taken were successful. The search algorithm indicates the best move that leads to a successful final state. Normally, it is impossible to search this entire tree in a reasonable amount of time and to reduce computation we introduce several pruning techniques. Another strategy used to reduce processing time is to define a maximum depth for the search algorithm. This limits the algorithm to only search up to intermediate nodes. Consequently, an evaluation function is introduced for these nodes. The search algorithm will now indicate the best move that leads to the highest value of the evaluation function.

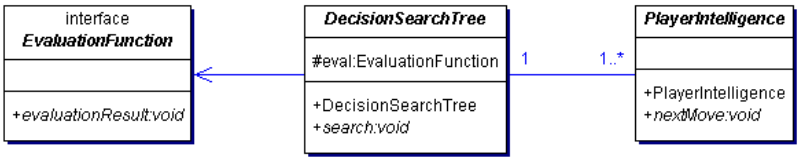

Figure 2 – MAS-L Framework

The MAS-L Framework is composed of two abstract classes – *DecisionSearchTree and PlayerIntelligence*, and an interface – *EvaluationFunction*. All of these classes have been developed in Java [13], and are presented in Figure 2. *PlayerIntelligence* is an abstract class, and the subclass that inherits it implements the main interface with the Software Agent. This subclass shall place some code in the abstract method *nextMove()*, which is responsible for making decisions for the Agent. The *DecisionSearchTree* is responsible for storing and searching the decision tree. This subclass implements a method called *search()*, and it is responsible for deciding the best move based on a search technique. The pruning strategy shall also be defined and implemented in this class. The class that implements *EvaluationFunction* should code the function that evaluates an intermediate node. This function should use a machine learning technique such as a neural network. This builds a more dynamic and flexible intelligence for a software agent. Any change that occurs in the MAS environment, can adapt the function through learning process. This adaptation permits the agent to make decisions for totally new events.

In our framework the hot spots are the classes – *DecisionSearchTree*, *PlayerIntelligence*, and *EvaluationFunction*. This design demonstrates that MAS-L is closer to a conceptual framework, although we do reuse some code in the abstract

classes. However, the process of implementing intelligence and learning properties is reused for all applications instantiated from MAS-L.

3. How to Instantiate the MAS-L Framework

In this section, we describe the instantiation process of the framework, and show the interdependence of the classes. The first class to be extended is *EvaluationFunction*, and the subclass that inherits it implements the function that evaluates an intermediate node of the decision tree. Prior to the implementation, it is important to design the structure and properties of the decision tree. This class shall choose a machine learning technique to implement this function. The second class to be extended is *DecisionSearchTree*, and the subclass that inherits it implements the data structure of the decision tree and the searching process. In order to reduce computational time, we can also introduce in this class several pruning techniques. The strategy of defining a maximum depth for the search algorithm is implemented in this subclass, and to code such strategy we shall instantiate the class that implements *EvaluationFunction* to perform the evaluation of intermediate nodes.

The class that extends *PlayerIntelligence* will implement the main interface with the Software Agent. This subclass shall instantiate the class that extends *DecisionSearchTree*, which is responsible for deciding the next best decision. Some simple reactive decisions are also implemented in this class, and the software agent uses *nextMove* method to request a decision. The instantiation process of the MAS Framework is presented in [6], and the class that inherits from *Agent* (in the MAS Framework) is responsible for instantiating the class that extends *PlayerIntelligence*. Consequently, the Software Agent will now use the method *nextMove* to make decisions.

4. The LAP application – A case study

The Learn Agent Player (LAP) is a software agent that plays Tic-Tac-Toe, and learns how to improve its strategy solely by playing against other artificial opponents. Our system has an environment with an artificial player that challenges LAP until it is capable of winning or making draws in all the games. The LAP combines a search technique for games called Min-max and a Perceptron with Reinforcement Learning as a board evaluator [15]. Our results show that LAP is able to learn how to defeat or draw any opponent that uses a fixed policy.

The min-max procedure [16] 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 [16].

A single-layer perceptron [15] network consists of one or more artificial neurons in parallel. Each neuron in the layer provides one network output, and is usually

connected to all of the environmental inputs. The perceptron learning rule was originally developed by Frank Rosenblatt in 1958. A training set is presented to the network's inputs, and the weights w_i are modified when the expected output d does not match with the output y. The perceptron rule adapts the weight using the following formula $w_i=w_i+ \eta(y - d)x_i$, where η is the learning rate. This kind of learning is called supervised learning, where the agent learns from examples provided by some knowledgable external supervisor.

Reinforcement learning [15] is different from supervised learning, 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.

In our environment we have two agent players: The LAP player and the Perfect Tic-Tac-Toe player. The Perfect Tic-Tac-Toe player is a software agent that uses the Min-Max search to decide the next move. This player does not use an evaluation function because it searches the whole decision tree. If the leaves of this tree correspond to a win, it evaluates it with the value 100. Otherwise, the loose receives the value -100. It is feasible to search the entire decision tree, because Tic-tac-toe has at most 9^9 nodes. LAP is the second player and also uses the Min-max search technique, but establishes a maximum depth for the search. Consequently, an evaluation function is introduced to evaluate intermediate nodes. For this evaluation, we use a single-layer perceptron with only one artificial neuron. This perceptron receives a board configuration as an input and scores the board 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 wins or draws in the final states (leaves).

Instead of presenting a learning set to the perceptron, we use a Reinforcement learning strategy. Our agent player starts with random initial weights in the perceptron. Therefore, the LAP player starts loosing most of the games he duels the Perfect Tic-Tac-Toe Player. At the end of every match, the perceptron adapts its weight using the Perceptron rule and two learning strategies. The first learning strategy only presents final configurations to the Perceptron rule and establishes that the expected output d is 0 in case of a loose. With this strategy the LAP agent player is able to generalize the evaluation of intermediate nodes based on final state nodes. The second learning strategy uses Reinforcement learning to adapt the weights. For every match that ends up in a loose for LAP, the Perceptron rule is computed for all of the traversed intermediate states that correspond to a decision point. The expected output d for every board state is now calculated using the following formula: $d(n)=d(n-1) + ß\ (reward(n) - d(n-1))$, where $d(n)$ is the expected output of the n^{th} decision point, $ß$ is the Reinforcement learning rate, and $reward(n)$ is the reward obtained in the n^{th} decision point. The variable n can assume values from 1 to 9, where $n = 1$ represents root of the decision search tree. With this strategy the LAP agent player is able to learn how to evaluate intermediate nodes based on all the traversed sates of the matches against the Perfect Tic-tac-toe player.

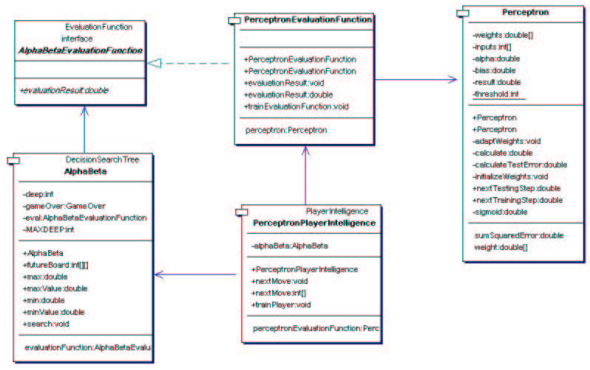

Figure 3 – The LAP Player Intelligence

In Figure 3, we present only the class diagram of the LAP intelligence. The LAP player intelligence uses a Perceptron neural network with Reinforcement Learning as the machine learning technique for the evaluation function. The class *PerceptronEvaluationFunction* implements the interface *AlphaBetaEvaluationFunction*, which implements the *EvaluationFunction* interface from the MAS-L framework. This design is adopted because we intend to create other evaluation functions for the min-max search procedure with alpha-beta pruning. The class *AlphaBeta* extends the abstract class *DecisionSearchTree*, and in this class we place code for the min-max search procedure with alpha-beta pruning technique. The attribute MAXDEEP holds the value of the maximum depth the search algorithm shall go for a given decision point. This class also uses the *PerceptronEvaluationFunction* through the method *evaluationResult* to compute the evaluation of a intermediate board. The class *PerceptronPlayerIntelligence* extends the abstract class *PlayerIntelligence* from the MAS-L framework. This class implements the interface with the software agent. We also code in this class the two learning strategies described above. At the end of each match against the Perfect Tic-tac-toe agent player, the method *trainPlayer* is executed to start the learning techniques.

5. The LAP application – Experimental Results

To evaluate the learning performance of the LAP player, we scheduled several matches against the Perfect Tic-tac-toe Player. The LAP player starts with random values for the weight in his perceptron, which represents no knowledge or game strategy. At the end of each match, the LAP player undergoes a training session using one of the two learning strategies presented above. We stop LAP from dueling Perfect when LAP is able to win or draw 10 matches in a row.

For each fixed depth of the min-max search procedure, we test how many matches are needed for LAP to learn a strategy to win or draw Perfect. In Figure 4, we present a graph that compares the two learning strategies tested in LAP. The graph also illustrates the average number of matches that were needed for the LAP to learn the win or draw strategy. The results demonstrate the better performance of the Perceptron Rule combined with the Reinforcement Learning technique.

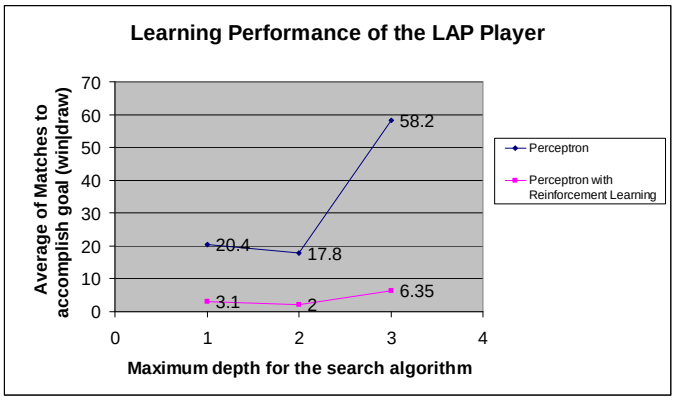

Figure 4 – Learning Performance of LAP Player Intelligence

6. Final Comments

The MAS-L Framework is being used in two experiments, and in both of them we were able to reduce the implementation time of a complex behavioral property in software agents. This framework helps the introduction of intelligence and learning properties in large scale multi-agent system due to its simplicity confirmed in experimental results. However, we believe that learning policies and skill management are also extremely important to systematically develop an "organizational intelligence". We want to extend this framework to introduce such ideas that are extremely important for building large scale multi-agent systems.

References

[1] 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. J.; Cowan, D.D. *Engineering Multi-Agent Object-Oriented Software with Aspect-Oriented Programming*. Submitted to Practice & Experience, Elsevier, May 2001.

[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] 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.

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

[8] Wooldridge, M; Jennings, N. R.; Kinny, D. *The Gaia Methodology for Agent-Oriented Analysis and Design*. Kluwer Academic Publishers.

[9] Garro, A.; Palopoli, L. *An XML Multi-Agent System for e-Learning and Skill Management*. Third International Symposium on Multi-Agent Systems, Large Complex Systems, and E-Businesses (MALCEB'2002), Erfurt, Thuringia (Germany), 8-10 October 2002

[10] Ioerger, T. R.; He, L.; Lord, D.; Tsang, P. *Modeling Capabilities and Workload in Intelligent Agents for Simulating Teamwork*. Proceedings of the Twenty-Fourth Annual Conference of the Cognitive Science Society (CogSci'02), 482-487

[11] Stader, J.; Macintosh, A. *Capability Modeling and Knowledge Management.* In Applications and Innovations in Expert Systems VII, Proceedings of ES 99 the 19th International Conference of the BCS Specialist Group on Knowledge-Based Systems and Applied Artificial Intelligence, Cambridge, December, 1999; Springer-Verlag; ISBN 1-85233-230-1; pp 33 - 50.

[12] Trading Agent Competition Web Site - http://auction2.eecs.umich.edu/

[13] Java Web Site - http://java.sun.com/

[14] 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.

[15] Russell, S.; Norvig, P.. *Artificial Intelligence, A Modern Approach*. Prentice-Hall, 1995.

[16] Winston, P.H. *Artificial Intelligence*. Addison Wesley, 1992.