# A Combined Specification Language and Development Framework for Agent-based Application Engineering

**José Alberto Rodrigues Pereira Sardinha**

**Ricardo Choren Noya**

**Viviane Torres da Silva**

**Ruy Luiz Milidiú**

**Carlos José Pereira de Lucena**

Departamento de Informática

# A Combined Specification Language and Development Framework for Agent-based Application Engineering *

José Alberto Rodrigues Pereira Sardinha, Ricardo Choren Noya[1] ,

Viviane Torres da Silva, Ruy Luiz Milidiú, Carlos José Pereira de Lucena

[1]Engenharia de Sistemas e Computação – Instituto Militar de Engenharia (IME)

sardinha@inf.puc-rio.br, choren@de9.ime.eb.br, viviane@les.inf.puc-rio.br,        milidiu@inf.puc-rio.br, lucena@inf.puc-rio.br

**Abstract.** Software agents are used to solve several complex problems. To properly build agent systems, it is necessary to create software engineering processes and tools to support all the development phases. Indeed, many modeling languages and implementation frameworks are available for system engineers. However, there are few methods that properly combine agent-oriented design and implementation initiatives. In this paper, we propose a development method that goes from the requirements elicitation to the actual implementation of agent systems using a modeling language and an implementation framework. We also present a case study to illustrate the suitability of our approach.

**Keywords**: Software agents, agent oriented software engineering, framework.

**Resumo**. Agentes de Software são utilizados para resolver vários problemas complexos. Para construir esse sistemas baseados em agentes, é preciso criar processos de engenharia de software e ferramentas para apoiar todas as fases de desenvolvimento. Existem muitas linguagens de modelagem e arquiteturas de implementação para engenheiros de sistemas. Porém, poucos métodos apresentam uma maneira de combinar o *design* orientado a agentes e iniciativas de implementação. Neste artigo, apresentamos um método de desenvolvimento que inicia com a definição de requisitos e vai até a implementação do sistema baseado em agentes utilizando uma linguagem de modelagem e uma arquitetura de implementação. Um estudo de caso é apresentado para ilustrar o método proposto.

**Palavras-chave**: Agentes de Software, engenharia de software orientado a agentes, *frameworks*.

---

# 1 Introduction

The agent technology is used to develop systems that perform several complex tasks. A software agent embodies some goals, some actions that are executed to achieve these goals, some high-level message interface, and a set of agency properties, such as autonomy, adaptation, interaction and learning. The system's overall goal is achieved through the synergetic cooperation of the agents.

An agent-oriented software engineering is currently under development and its main goal is to determine how agent qualities affect software engineering and what additional tools and concepts are needed to apply software engineering processes and structures to agent systems (Wooldridge and Ciancarini, 2001). In this sense, the engineering of agent-oriented applications require particular tools and techniques for developers to improve the design and the implementation of agent systems. For instance, methodologies to guide analysis and design phases are required; agent architectures are needed for the design of individual components, and supporting infrastructure must be integrated (Luck et al., 2004).

At present, there are a number of agent-oriented modeling languages and methodologies, such as (Castro et al., 2002; DeLoach, 1999; Padgham and Winikoff, 2002b; Zambonelli et al., 2003), and there are also some platforms or architectures for agent-based system implementation, such as (Howden et al., 2001; Jade, 2005). However, little effort has been done to create a method that properly combines agent design and implementation initiatives. Such methods are important not only to allow agent systems to be developed quicker and easier but also to foster the interest of commercial organizations in agent systems and to provide further dissemination of agent technologies (Luck et al., 2004).

In this paper, we propose a development method for building multi-agent system that goes from the requirements elicitation to the actual implementation of the agent system. The method tries to realize the benefits of a proper software engineering technique to the progress of the agent-oriented field. The proposed method is composed of an agent design phase, a migration or mapping path from the design results to an agent framework, and the actual implementation of the agent system.

The method begins with the system specification, using an agent-oriented modeling language called ANote (Choren and Lucena, 2004; Choren and Lucena, 2005). ANote is a modeling language that adopts the metaphor of agents and design views to provide natural and refined means to specify agent systems. It has a conceptual meta-model and its diagrams are able to model goals, actions, interactions and some agency characteristics.

Then, the artifacts generated in the specification phase are mapped to a framework, called Agent's SYNergistic Cooperation (ASYNC) framework (Sardinha et al., 2003b). The ASYNC object-oriented framework defines the agents and the system environment, implements a communication infrastructure, and uses some hot spots in order to implement the agent's goals, actions and interaction protocols. The agent system is implemented by instantiating the ASYNC framework and properly implementing the defined hot spots.

The proposed method is a result of the development of early case studies from which we could evaluate some important factors for the deployment of agent systems and also recommend a mapping from design to implementation. These case studies were a market simulator for creating offerings (Milidiu et al., 2001), learning agents in two-person game scenario (Sardinha et al., 2003a), a distributed system for the procurement of travel packages (Sardinha et al., 2005) and a multi-agent system for a supply chain management scenario of a PC manufacturer.

In order to demonstrate the applicability of the proposed method, the LearnAgents (Sardinha et al., 2005) case study will be presented. The LearnAgents is a multi-agent system for the procurement of travel packages with multiple and simultaneous auctions that participated in the 2004 edition of the classic Trading Agent Competition (TAC) (Tac, 2005). We will show the system specification using ANote, the mapping relations from the ANote artifacts to the ASYNC framework structure and some resulting implementation code.

The paper is organized as follows. In Section 2, we review the ANote modeling language, describing its concepts and artifacts. Section 3 shows the ASYNC framework by illustrating its structure and how it supports the development of agent systems. Section 4 presents the method, while reporting the LearnAgents system development. Section 5 describes some related work and, finally, Section 6 reviews the method contributions and suggests some future work.

## 2  The ANote

From an analysis point of view, systems including the agent technology require dedicated basic concepts and languages (Luck et al., 2004). ANote (Choren and Lucena, 2004; Choren and Lucena, 2005) is a modeling language that was developed to offer a standard way to describe concepts related to the agent-oriented modeling process. It has an underlying conceptual meta-model (figure 1) that defines, at the high level, the interactive, environmental and societal concepts such as goals, interaction protocols, environment resources and organizations. Furthermore, at the level of individual agents, it has elements to represent basic agent concepts such as actions, communication and plans.

These concepts define a variety of different aspects of concern, or views, which may complement or overlap each other during the system specification. ANote defines seven views based on its conceptual meta-model: goal, agent, scenario, planning, interaction, ontology and organizational views. Each view generates an artifact (diagram) and it is a partial specification that enables the designer to concentrate on a single set of properties each time. Therefore, the designer considers only those features that are important in a particular context.
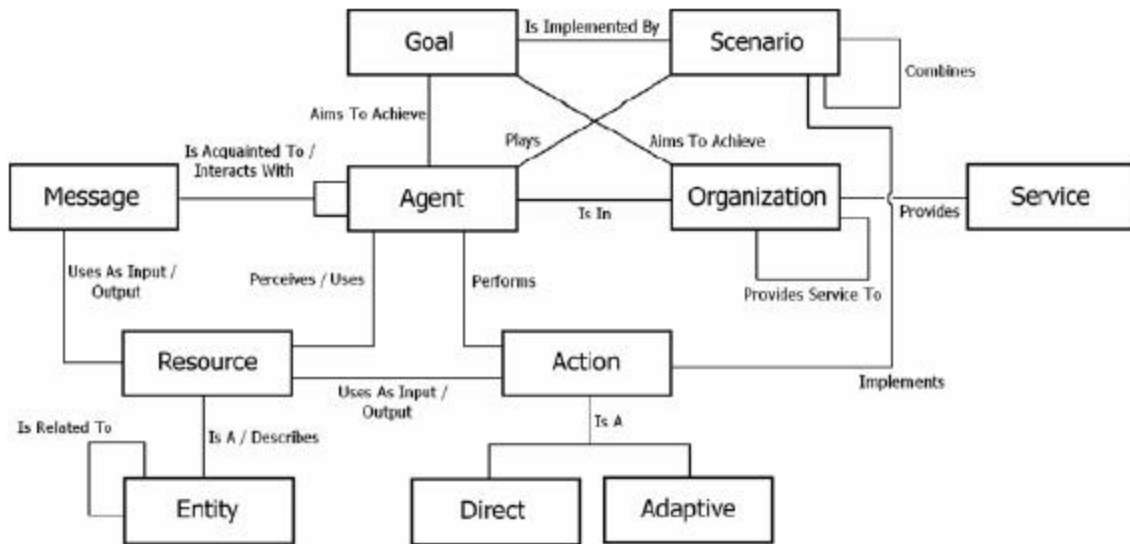
**Figure 1.** The ANote meta-model structure

ANote goal view provides an initial identification of a tree of goals that outline the system functions. Analyzing requirements in terms of goal decomposition and re-finement can be seen as teasing out many levels of requirements statements, each level addressing the demands of the next level (Choren and Lucena, 2005). In this view, complex goals can be functionally decomposed into their constituent goals and flows, providing a description as a hierarchical tree of goals.

The agent view specifies the agent classes in the application solution and their re-lationships. In this view, no details about agent behavior are provided since its pur-pose is to specify the systems structure. Agent classes specify the roles that will per-form the goals elicited in the goal view and that compose the system organizations. When two agents need to interact in the system, there must be a structural relation-ship between their agent classes, which is represented in the agent view as an asso-ciation relationship.

The scenario view captures agent behavior in scenarios, which are textual repre-sentations of how goals are achieved by agents. A scenario serves for two purposes: to illustrate how goals can be achieved or fail; and to show the circumstances in which an agent may adapt (or learn) or have an autonomous behavior, thus model-ing agency characteristics. This is done by generating courses of action for both nor-mal and emergent (adaptive or exceptional context) behavior. A scenario specifies the following parts: main agent, prerequisites, usual action plan, interaction and variant action plan(s).

The planning view describes the action plans depicted in a scenario's courses of actions. It shows the agent's internal actions and their sequence, using state charts representing action states and transitions. Besides, it introduces notation to represent agent adaptation with adaptive transitions to variant actions (emergent behavior). Adaptive transitions, along with their tags, allow system designers to show when and under what circumstances an agent should change its behavior.

The interaction view is used to represent the set of messages agents exchange while executing an action plan. In this view, interactions are represented as conversation diagrams that describe the discourse between agents, i.e., the message protocols and the states of interactions.

A multi-agent system is not only composed of agents: it has other non-agent components that build the environment, i.e. the agent world. The ontology view is responsible for specifying the environment resources and the agents' knowledge base. In ANote non-agent components are modeled as objects. Thus the UML class diagram, possibly detailed with OCL, is used to represent the system environment. Some advantages of using a UML subset to model ontology can be seen in (Cranefield and Purvis, 1999).

The organization view models the agent societies. In this view, an organization is seen as an implementation unit that offers services (set of goals), accessed by an interface (set of message protocols). Organizations can participate in a dependency relationship that shows how organizations are arranged in a client-server model, i.e. how the agents in an organization require services form agents in another organization.

## 3 The ASYNC Framework

Agent infrastructures are concerned with developmental and operational support for agent systems (Luck et al., 2004). In fact, they are the fundamental engines underlying the autonomous components that support effective behavior in real world, dynamic and open environments (d'Inverno and Luck, 2004).

There are now a large number of agent development environments and toolkits, for instance (Howden et al., 2001; Jade, 2005). However, not all of the available tools are sufficiently mature for mission usage. In this sense, the ASYNC framework is another initiative to provide an infrastructure to enable the development of agent systems. The framework has been used to implement complex agent-based systems, such as: (i) evolutionary agents for creating offerings in a retail market (Milidiu et al., 2001), (ii) agents that negotiate automatically for goods (Sardinha et al., 2003b), (iii) agents that learn to play automatically in two-person games (Sardinha et al., 2003a), (iv) a multi-agent system for a complex procurement scenario (Sardinha et al., 2005), and (v) a multi-agent system for a supply chain management scenario.

The framework provides a structure (figure 2) for agent development and offers a communication infrastructure in a distributed environment. Moreover, the framework has hot spots that implement the agent's goals, actions and interaction protocols. ASYNC also provides a design pattern (Sardinha et al., 2004) to implement machine learning techniques in the agents. Machine learning (Mitchell, 1997) algorithms are crucial to provide well-known strategies to support the construction of adaptable agents, especially in unpredictable, heterogeneous environments, such as the Internet.
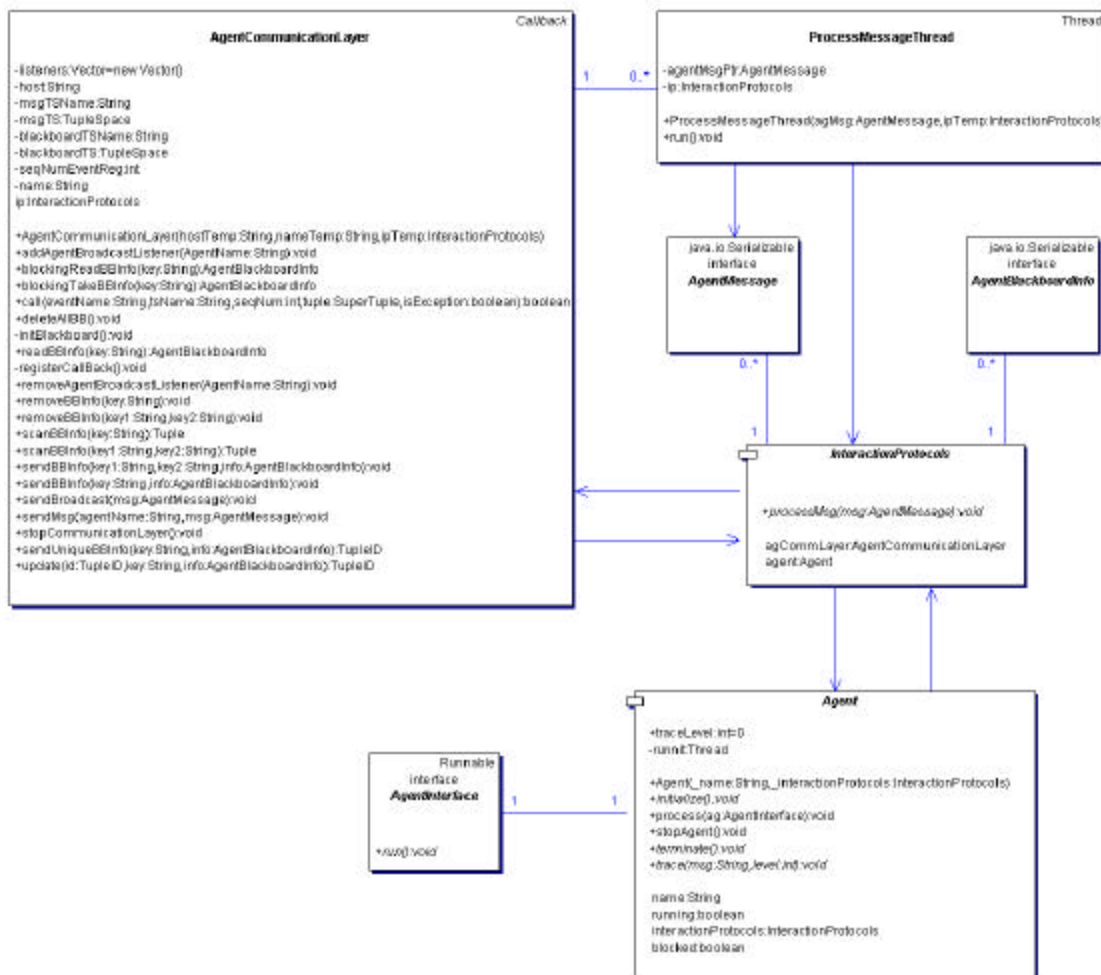
**Figure 2.** The ASYNC framework structure

The ASYNC framework is composed of the Agent and InteractionProtocols abstract classes, the ProcessMessageThread and AgentCommunicationLayer final classes, and the AgentMessage, AgentBlack-BoardInfo and AgentInterface interfaces. The Agent abstract class offers the agent basic functionalities: initialize, stop and process. These methods are responsible for the agent start up (relating it to its resources, actions and interaction protocols), agent suspension and action execution. Besides, it has two other methods, terminate and trace, which are responsible for the ending code and message display respectively. The *name* attribute specifies the agent's name in the system, which has to be unique.

AgentInterface is responsible for turning the subclass that inherits Agent into a thread. This subclass will have to implement a method called run, and it will be in charge of starting the agent's private activities.

InteractionProtocols is an abstract class that defines the way an agent can interact with other agents in the system. All the code related to interaction is placed in this class. A subclass of InteractionProtocols also requires the implementation of a method called processMsg, which is called every time a new message is received from another agent. The ProcessMessageThread class is in charge of processing mes-

sages received by agents. In fact, it creates a new thread for every incoming message, which will automatically call the abstract method processMsg.

AgentMessage specifie the system's message format and AgentBlackBoardInfo specifies the blackboard message format. Implementations of these interfaces depend on the problem domain since they define the structure of the messages exchanged in the system, either directly or though the blackboard.

AgentCommunicationLayer is a class that implements the entire communication infrastructure needed for agents to interact in a distributed environment. This infrastructure is a layer over IBM TSpaces (TSpaces, 2005).

## 4  The LearnAgents Application

A trading agent is a computer program that acts on online markets on behalf of one or more clients, trying to satisfy their preferences. In the trading agent competition (TAC), agents have the goal of assembling travel packages composed of flight tickets, hotel rooms, and event tickets, for a period of five days. In a game, eight agents, each representing eight clients, compete for a limited amount of goods on a total of 28 different auctions, for 9 minutes. The agents receive a score based on how well client preferences are satisfied (sum of client utilities) and the average score over a number of games determines the winner.

The LearnAgents (Sardinha et al., 2005) is an application that participated in the TAC Classic 2004. Although a competitor can be developed as a single agent, LearnAgents was realized as a multi-agent system because we believe that the game complex requirements were better developed by creating modular distributed entities.

### 4.1  The Method Phase 1: Specification

ANote uses a goal-oriented requirements elicitation technique as the first step in the modeling process. The ultimate goal of the LearnAgents systems is to maximize the total satisfaction of its clients with the minimum expenditure in the travel auctions, i.e. to acquire travel packages with as much profit as possible. According to the game rules, this profit is defined as the sum of the utilities of the eight clients in the TAC game, minus the costs of purchasing the travel goods in the auctions.

This high-level goal can be decomposed into two intermediate sub-goals: to build a knowledge base for decision making and to negotiate travel packages based on this knowledge base. The system knowledge base has information about the current prices of the running auctions and this information will be decisive to the proper agent execution in the game. Therefore, to build the knowledge base, the system shall: (i) constantly check (sense) the auctions' current prices (sub-goal monitor market information); (ii) calculate the client preferences, in this case, expressed as an utility table for each desired travel package (sub-goal classify customer preferences); (iii) estimate the prices that will be asked in the auctions, according to the competitors' activity in the TAC system (sub-goal predict next prices of auctions); and (iv)

enumerate a list of different negotiation scenarios, based on the predicted prices and the client preferences (sub-goal calculate best allocations). These two last sub-goals require some learning and adaptation features in the multi-agent system, which is why the system is called LearnAgents.

Using the information in the knowledge base, the system can now negotiate in the several open auctions. To negotiate the system shall: (i) select a scenario with a high profit and low risk, in order to try to maximize the utility function (sub-goal classify best allocations); (ii) define the number and the types of travel goods, which should be bought to satisfy the selected scenario, to create bids to the auctions (sub-goal create bidding orders); (iii) and, finally, define a price to each generated bid and send it to the proper auctions (sub-goal send bids to auctions). The goal view diagram for the LearnAgents system is shown in figure 3.
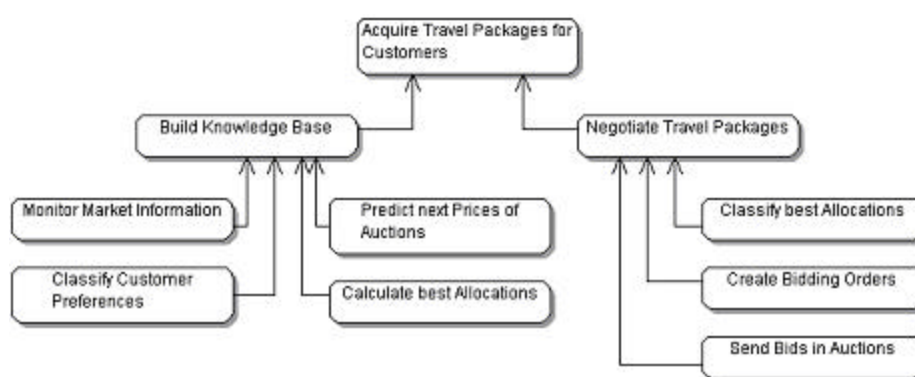


**Figure 3.** The LearnAgents goal view diagram

The agent structure must be defined from the goal hierarchy, with the identification of the agent classes that will be responsible for actually achieving the system functional goals (leaf nodes in the goal view diagram). So, the designer should relate the lowest level of sub-goals in the goal view diagram to agents. Table 1 shows the relation between the sub-goals and the agent classes, and figure 4 shows the system's agent view diagram.

| Goal | Agent |
| --- | --- |
| Monitor Market Information | Hotel Sensor, Flight Sensor, Ticket Sensor |
| Classify Customer Information | Hotel Sensor |
| Predict Next Prices of Auctions | Price Predictor |
| Calculate Best Allocations | Allocation Master, Allocation Slave |
| Classify Best Allocations | Ordering |
| Create Bidding Orders | Ordering |
| Send Bids in Auctions | Hotel Negotiator, Flight Negotiator, Ticket Negotiator |

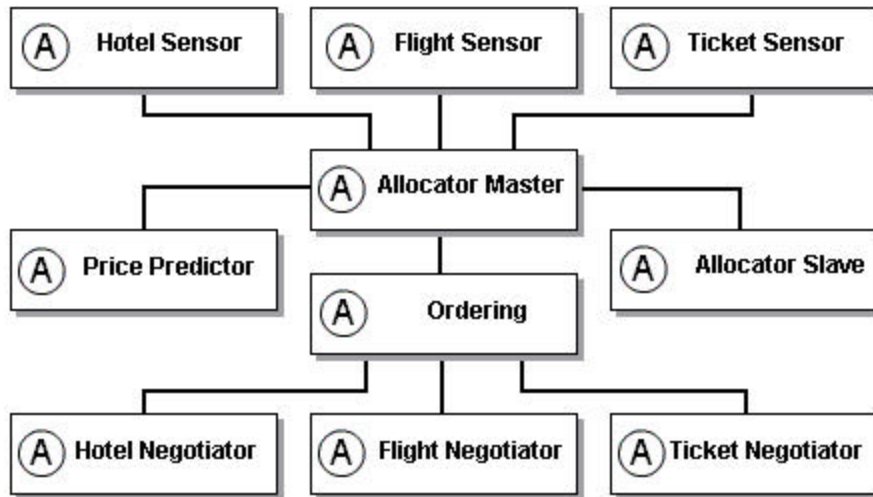**Table 1.** The relation between system goals and agent classes

**Figure 4.** The LearnAgents agent view diagram

The Hotel, Flight and Ticket Sensor agents update the knowledge base with quote prices, auction states (open or closed), number of goods purchased, and number of goods in negotiation in the TAC environment. The Price Predictor agent predicts hotel and flight prices after quotes have been updated. The Allocator Master and Slave agents are responsible for calculating and combining all the allocation scenarios. The Ordering agent decides the amount of travel goods to buy based on the scenarios. And, finally, The Hotel, Flight and Ticket Negotiator agents negotiate the travel goods in the auctions based on the decision made by the Ordering agent.

Each agent goal is further specified with one or possibly more scenarios that will include information about the contexts in which these goals will be achieved. For brevity reasons, only the scenario for the "classify best allocations" goal is shown here (figure 5). The Ordering agent can start to perform its actions to achieve the goal just after it receives a message from the Allocator Master agent with the several negotiation scenarios. Then, it executes it main action plan and interacts with the Hotel, Flight and Ticket Negotiator agents.

| CLASSIFY BEST ALLOCATIONS | |
|---|---|
| Main Agent | Ordering Agent |
| Preconditions | Message from Allocator Master received |
| Main Action Plan | `while true`<br>`    for each Negotiator Agent do`<br>`        decide GOODS to buy`<br>`        decide GOOD amounts`<br>`        calculate GOOD HIGH_PRICES`<br>`        send message to Negotiator Agent`<br>`    end do`<br>`end while` |
| Interaction | Hotel Negotiator, Flight Negotiator, Ticket Negotiator |
| Variant Action Plan | |

**Figure 5.** The Classify Best Allocations scenario view diagram

The planning and interaction view diagrams are derived from the scenario diagram. Figures 6 and 7 present the action plan and the interaction protocol for the "classify best allocations" scenario shown above.
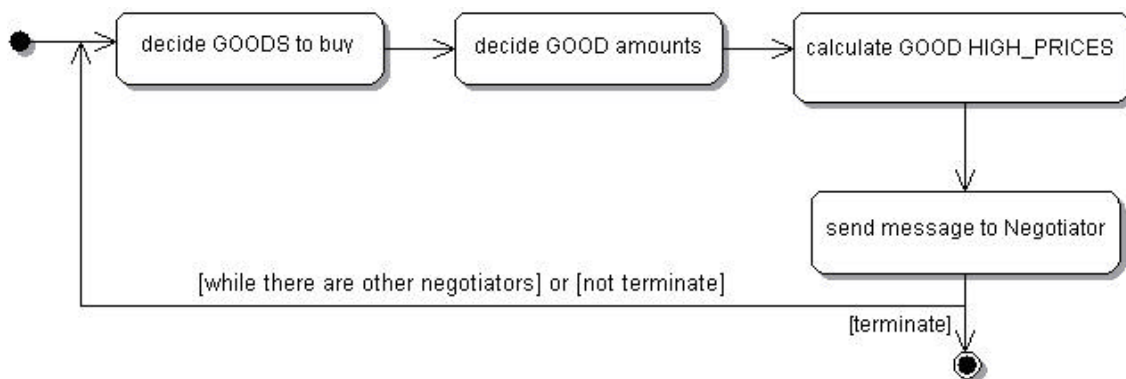


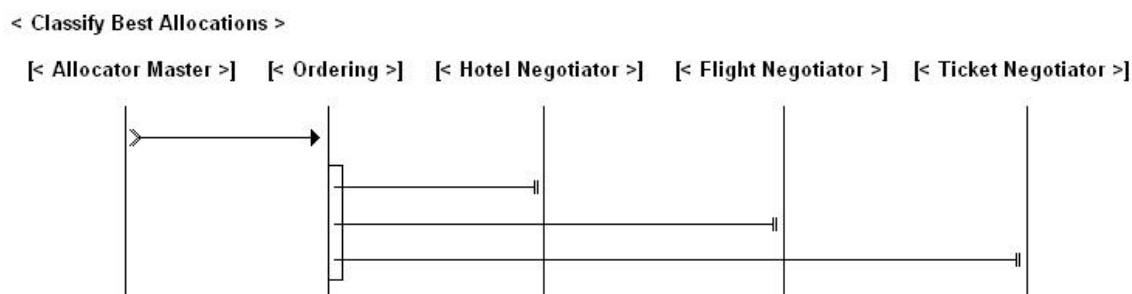**Figure 6.** The Classify Best Allocations planning view diagram



**Figure 7.** The Classify Best Allocations interaction view diagram

The non-agent components that build the system environment and agent knowledge are related to the information about client preferences, goods, auctions and bids. In fact, all the non-agent system resources are modeled in the ontology view diagram (figure 8).
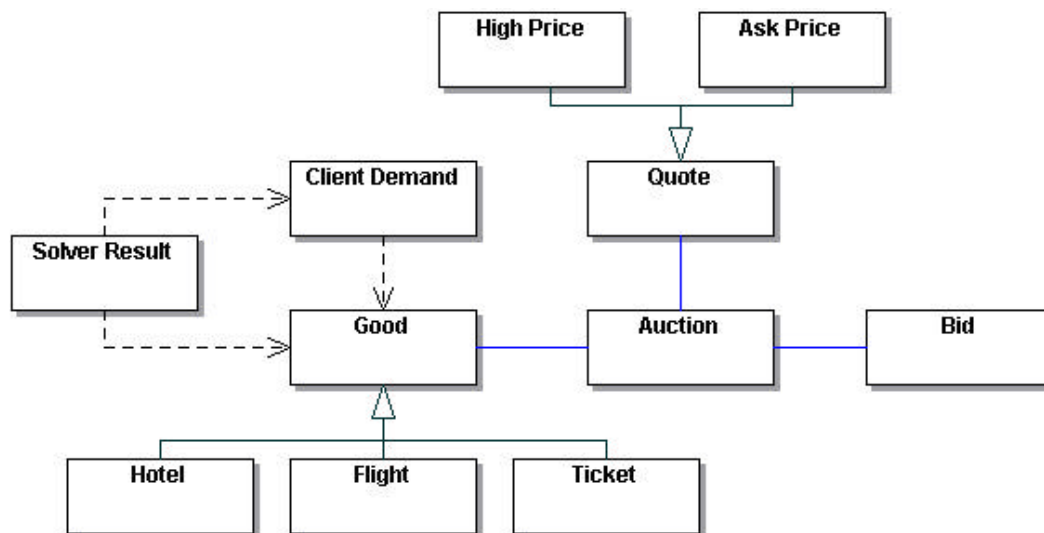
Since each competitor builds an agent society, the LearnAgents is just one organization. The LearnAgents organization interacts with the SICS TAC server, which is not another agent organization – it is just a "middleware" for hosting the competition. Thus the organization view diagram is very simple, composed of only one organization, which has all the agents.

## 4.2 The Method Phase 2: Mapping the Specification to the Framework

In order to map the ANote models into code by extending the ASYNC framework, we used a visitor-based approach (Czarnecki and Helsen, 2003). In such approach, the mapping provides a visitor mechanism to traverse the ANote models and to write code extending the ASYNC framework. Each ANote model is analyzed to find out the modeling elements that should be transformed. The transformation rules are applied to those elements, generating Java code that extends the framework. Table 2 summarizes the mapping process from the ANote models to the ASYNC framework.

| ANote concept | ASYNC framework implementation |
| --- | --- |
| Environment and Organization | Main class for the Environment, Vector Attribute in Main class for each Organization, Attribute in Main class for each Agent and Resource |
| Agent class | Concrete class of Agent, Concrete class of InteractionProtocols |
| Action state (Internal) in the planning diagram | Method of the Concrete class of Agent |
| Action state (Interaction) in the planning diagram | Method of the Concrete class of InteractionProtocols |
| Resource objects in the ontology diagram | Simple class |

**Table 2.** Mapping elements in the specification to the ASYNC framework

To guide the mapping process, a set of Prolog (Bratko, 2000) rules were specified. The transformation process steps are presented in this section along with the related rules. The first step of the process is responsible for creating the system per se. This is done by implementing the system environment. The environment is a class that will have the references to all the agents and resources that will compose the system. This is the system main class and it will be responsible to create the agents, the resource objects, and to handle the communication infrastructure. This main class must have attributes for each agent and resource in the system. Moreover, the organizations in ANote specify a list of agents that participate in a society, and each organization is also mapped as a Vector attribute in the main class with an agent list. It is important

to notice that this environment was not directly specified by a particular diagram – it is the result of the entire model set. The Prolog code in lines 1, 2 and 3 defines the lists of resources, organizations and agent classes, respectively. The mapping to the ASYNC main class is produced in line 4. Line 5 presents the ASYNC environment class with methods and attributes.

```
1. anote(resources,ResourceNames).

2. anote(organizations,OrganizationsNames).

3. anote(classes, AnoteClasses).

4. anote2asyncEnvironment(EnvClass,EnvAttributes,EnvMethods):-
       anote(organizations,OrganizationsNames),
       anote(resources,ResourceNames),
       anote(classes,AnoteClasses),
       EnvClass = 'MainClass',
       append(AnoteClasses,OrganizationsNames,X),
       append(X,ResourceNames,EnvAttributes),
       EnvMethods = ['Main'].

5. async(environment,EnvClass,EnvAttributes,EnvMethods).
```

In the next step, each agent class represented in the ANote agent view is transformed into two concrete classes. One of them extends the ASYNC class *Agent* and implements the *AgentInterface* class. The second one extends the ASYNC class *InteractionProtocol* to provide the interaction between the agents. The Prolog code in line 1 presents the definition of an ANote class. Line 3 and 5 states the ASYNC classes that are used in the transformation. Lines 2 and 4 depict the rules used in this step. Rule 2 generates the mapped class by appending "Agent" to the ANote class and rule 4 generates the mapped class by appending "AgentIP" to the ANote class.

```
1. anote(class,AnoteClass).

2. anoteClass2asyncIAClass(AnoteClass,IAClassName) :-
       "Agent" = AsyncExtension,
       append(AnoteClass,AsyncExtension,IAClassName).

3. async(internalAction,IAClassName,IAextends,IAimplements,IAMethods):-
       IAextends = 'Agent',
       IAimplements = 'AgentInterface'.

4. anoteClass2asyncIPClass(AnoteClass,AsyncIPClass) :-
       "AgentIP" = AsyncIPExtension,
       append(AnoteClass,AsyncIPExtension,AsyncIPClass).

5. async(interactionProtocols,IPClassName,IPextends,IPimplements,IPMethods)
   :-
               IPextends = 'InteractionProtocols',
               IPimplements = ''.
```

The third step consists of mapping the actions associated with the agents and defined in the ANote planning diagrams. Such actions can be subdivided in two groups: internal actions and interactions. The agent internal actions are implemented as methods of the concrete class that extends Agent and implements AgentInterface.

The autonomy and reasoning features of an agent shall also be implemented as methods in this class since these features directly affect the way an agent execute, in other words, it affects its actions. Thus the basic structure of an agent, i.e. its identity, basic features and action plans are encapsulated in this particular concrete class. The internal actions of an agent are presented in line 1. The Prolog code in line 2 presents the mapping rule of the internal actions of the agent to methods of the ASYNC class.

```
1. anote(actions,internal,AnoteClass,IActions).

2. anoteAction2asyncIAMethod(AnoteClass,IAMethods) :-
          anote(actions,internal,AnoteClass,IActions),
          IAMethods =

     ['Constructor','Initialize','Terminate','Trace','Run'|IActions].
```

All the interactions, specified in the interaction diagrams, will be placed in a concrete class that implements InteractionProtocols. 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. The actions that represent interaction with other agents are presented in line 1. The Prolog code in line 2 presents the mapping rule of the actions related to interaction to methods of the ASYNC class.

```
1. anote(actions,interactionProtocol,AnoteClass,IPActions).

2. anoteAction2asyncIPMethod(AnoteClass,IPMethods) :-
     anote(actions,interactionProtocol,AnoteClass,IPActions),
     IPMethods = ['Process Message'|IPActions].
```

All the resources, specified in the ontology diagram, are implemented as Java Classes. The Prolog code in line 2 presents the mapping rule of the resources in the ontology diagram to simple Java classes in ASYNC.

```
1. anote(resources,ResourceNames).

2. anote2asyncResources(ResourceClasses):-
     anote(resources,ResourceNames),
     ResourceClasses = ResourceNames.

3. async(resources,ResourceClasses).
```

Figure 9 shows the implementation of the Ordering agent from the LearnAgents system using the rules presented above.
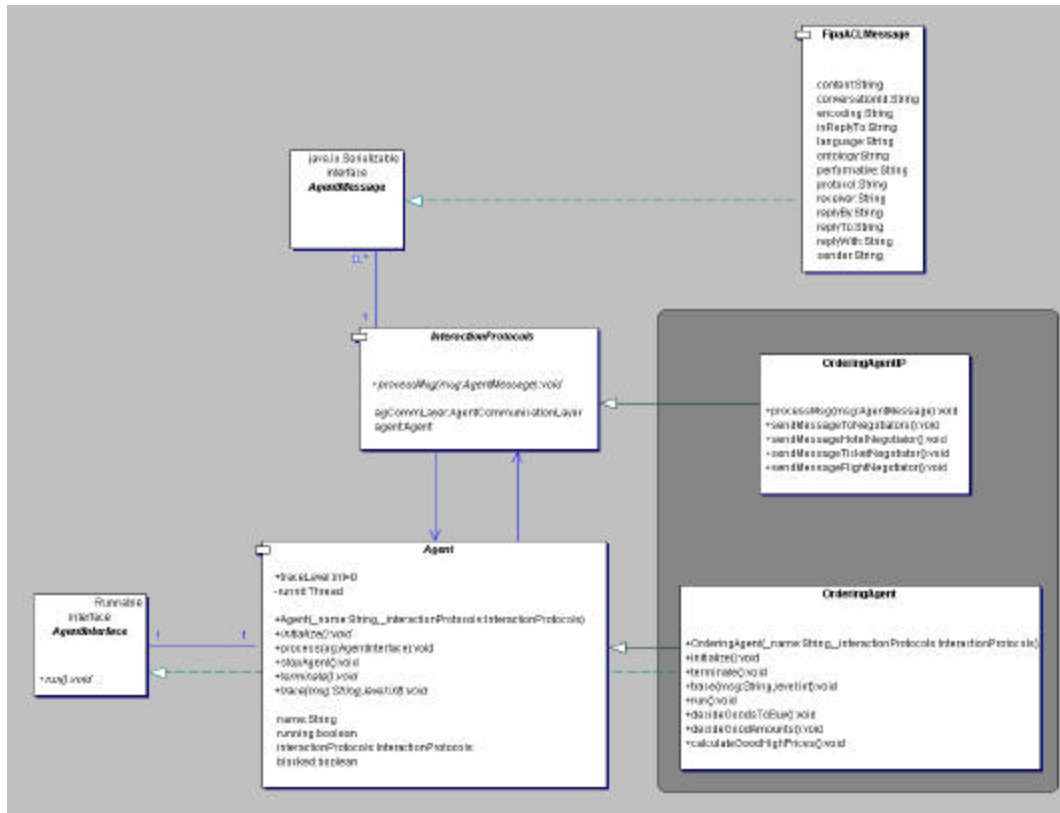
**Figure 9.** The Ordering agent

The Prolog code in lines 1 to 8 resumes the mapping of an Agent. The program runs the following steps: (i) tests in line 1 if the ANote class is true, (ii) produces the mapping of the ANote class to the ASYNC classes in line 3 and 4, (iii) produces the methods of the ASYNC classes in line 5 and 6, and (iv) specifies in lines 7 and 8 the class extension and interface implementation of the ASYNC classes.

```
1. mapping(AnoteClass) :-
2.      anote(class,AnoteClass),
3.      anoteClass2asyncIAClass(AnoteClass,AsyncInternalActionClass),
4.      anoteClass2asyncIPClass(AnoteClass,AsyncIPClass),
5.      anoteAction2asyncIAMethod(AnoteClass,IAMethods),
6.      anoteAction2asyncIPMethod(AnoteClass,IPMethods),
7.      async(internalAction,AsyncInternalActionClass,
                IAextends,IAimplements,IAMethods),
8.      async(interactionProtocols,AsyncIPClass,
                IPExtends,IPImplements,IPMethods),
```

In the ASYNC framework, the developer can choose to implement the agent in-teraction using messages or using TSpaces tuples. If the software agent uses message passing in order to communicate, a class that specifies the message format shall im-plement the interface AgentMessage. For instance, if the system uses FIPA ACL as the agent communication language, a concrete class that implements AgentMessage shall be done to build the messages' structure according to ACL. This helps to modularize all the system messages since they are all built from the same template. If the agent also uses a blackboard for communication, another class that specifies the

message format shall implement the interface AgentBlackBoardInfo. The entities specified in the ontology diagram are directly mapped to classes in the multi-agent system environment.

## 4.3 The Method Phase 3: Code

The Ordering agent internal actions are selected from the planning view diagram (figure 6). These internal actions can be classified as follows: (i) Decide goods to buy, (ii) Decide good amounts, and (iii) Calculate good high prices. The Prolog code in line 2 presents these actions.

```
1. anote(class,"Ordering").

2. anote(actions,internal,"Ordering",
      ['Decide Goods to Buy','Decide Good Amounts',
      'Calculate Good High Price']).
```

Consequently, the methods decideGoodsToBuy, decideGoodAmounts, and calculateGoodHighPrices are included in the Java class called OrderingAgent. This class must extend the Agent abstract class and implement the AgentInterface interface. The OrderingAgent class also has code for the initialize, run, terminate, trace, and constructor methods. The Java code is presented in lines 1 to 36.

```
1.  public class OrderingAgent extends Agent implements AgentInterface
2.  {
3.      public OrderingAgent(String name, InteractionProtocols iP){
4.          super(name, iP);
5.      }
6.      public void initialize() {
7.      ...
8.      }
9.      public void terminate() {
10.     ...
11.     }
12.     public void trace(String msg, int level) {
13.     ...
14.     }
15.     public void run() {
16.         int NumNegotiators=3;
17.         while(true) {
18.             for(int i=0;i<NumNegotiators;i++){
19.                 decideGoodsToBuy();
20.                 decideGoodAmounts();
21.                 calculateGoodHighPrices();
22.                 ((OrderingAgentIP)getInteractionProtocols()).
23.                 sendMessageToNegotiators();
24.             }
25.         }
26.     }
27.     public void decideGoodsToBuy() {
28.     ...
29.     }
30.     public void decideGoodAmounts() {
31.     ...
```

```
32.    }
33.    public void calculateGoodHighPrices() {
34.    ...
35.    }
36. }
```

The Ordering agent actions related to interactions are also selected from the plan-ning view diagram (figure 6). The only action that can be classified as an interaction action is the "Send message to Negotiators". The Prolog code in line 2 presents this action.

```
1. anote(class,"Ordering").

2. anote(actions,interactionProtocol,"Ordering",
        ['Send Message to Negoatiators']).
```
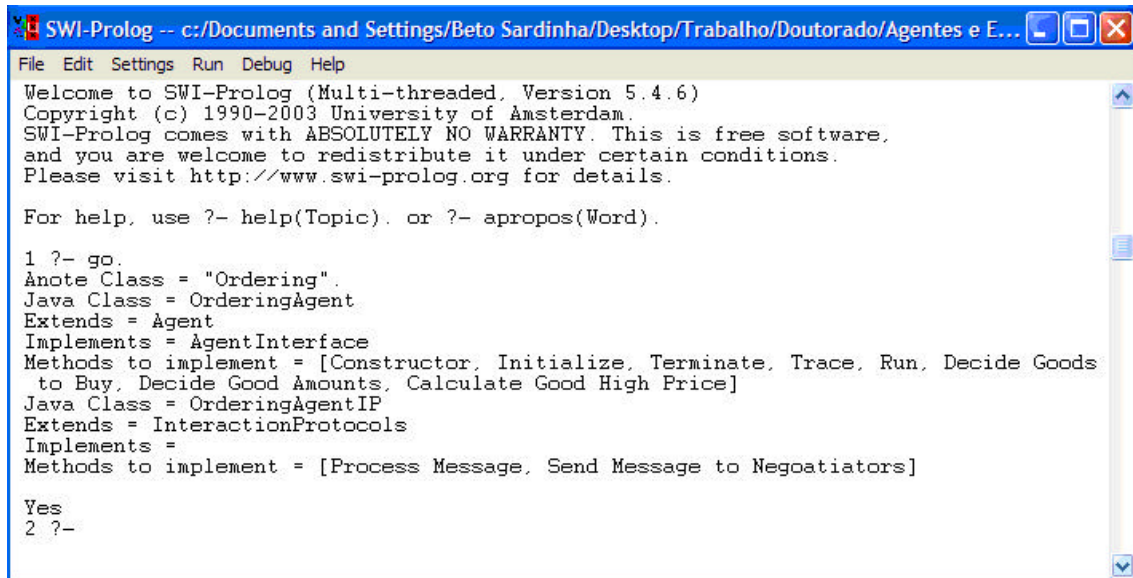
Consequently, the method sendMessageToNegotiators is included in the Java class called OrderingAgentIP. This class must extend the InteractionProtocol abstract class. The OrderingAgentIP class also has code for the processMsg method. The Java code is presented in lines 1 to 24.

```
1. public class OrderingAgentIP extends InteractionProtocols
2. {
3.    public void processMsg(AgentMessage msg) {
4.         FipaACLMessage msgReceived = (FipaACLMessage)msg;
5.         ...
6.    }
7.    public void sendMessageToNegotiators(){
8.         sendMessageFlightNegotiator();
9.         sendMessageHotelNegotiator();
10.        sendMessageTicketNegotiator();
11.   }
12.   public void sendMessageHotelNegotiator() {
13.        FipaACLMessage msg = new FipaACLMessage();
14.        getAgCommLayer().sendMsg("HotelNegotiator",msg);
15.   }
16.   public void sendMessageTicketNegotiator() {
17.        FipaACLMessage msg = new FipaACLMessage();
18.        getAgCommLayer().sendMsg("TicketNegotiator",msg);
19.   }
20.   public void sendMessageFlightNegotiator() {
21.        FipaACLMessage msg = new FipaACLMessage();
22.        getAgCommLayer().sendMsg("FlightNegotiator",msg);
23.   }
24. }
```

Figure 10 presents the output of the Prolog program that guides the mapping pro-cess for the Ordering agent. The ANote class is requested and the rules produce the ASYNC concrete class names, class extensions, interface implementations and meth-ods included in the ASYNC concrete classes.

**Figure 10.** The Prolog mapping program output

# 5 Related Work

A lot of methodologies and modeling languages have been proposed for modeling multi-agent systems and several platforms and frameworks have also been proposed for implementing them. However, little work has been done on proposing the mapping of agent-oriented design models into code. Methodologies, such as Gaia (Zambonelli et al., 2003) and MaSE (DeLoach, 1999), do not provide any guideline to the implementation. In (Zambonelli et al., 2003), the authors affirm that Gaia does not directly deal with implementation issues. Although DeLoach (1999) affirms that the primary focus of MaSE is to help in the requirements, analysis, design, and implementation phases, the methodology does not describe how the design models are implemented in any existing platform.

In (Castro et al., 2002), the authors propose a mapping from the i* concepts used by the Tropos methodology to a BDI agent-oriented development environment called Jack (Howden et al., 2001). Each i* concept is mapped to a BDI concept that is then mapped to a Jack abstraction. Although the work describes the mappings, it does not exemplify it. It does not demonstrate the mapping of some i* concepts used to model the agent-oriented application to Jack abstractions. Moreover, it does not detail the implementation of agents and plans that respectively extend the *Plan* and *Agent* abstractions proposed in Jack. In this paper, we present and demonstrate the mapping from design to implementation abstraction by using the LeanAgent application.

The Prometheus methodology (Padgham and Winikoff, 2002b) also provides a "start-to-end" support from specification to detailed design and implementation. The authors also propose the use of Jack to implement the Prometheus models. In (Padgham and Winikoff, 2002a), the Jack Development Environment is presented as a supporting tool for modeling Prometheus detailed designs and for implementing agent-oriented systems by using Jack. However, neither the mapping of the detailed

design artifacts into the implementation abstractions provided by Jack is described nor an example of a system modeled using Prometheus and implemented using Jack is presented.

Huget (2002) demonstrates the generation of code from AUML sequence diagrams. The authors focus on exemplifying the mapping from an agent interaction protocol to Java code. However, the mapping is not based on any implementing agent-oriented architecture or framework. Such technologies provide programmers with reusable abstractions that can be used to implement agent-oriented systems. By extending and customizing frameworks, the process of implementing an application becomes easer and quicker since part of the application code is already written and compiled in the framework (Fayad and Schmidt, 1999).

# 6  Conclusion and Future Work

In this paper we have sought to provide a method to transform agent-oriented analysis models into code. This method is based on the work undertaken in the development of a set of case studies, using the ANote modeling language for system specification and the ASYNC framework for system implementation.

The method presented here begins with the specification of the multi-agent system solution. This is done with the use of ANote diagrams. Each ANote diagram focuses on a specific concept, thus defining a modeling view. After the specification, there is a mapping process to the ASYNC framework. This mapping shows, in detail, how the outcomes of the modeling phase (the ANote diagrams) are translated to an agent implementation platform. Then, we have shown how this mapping guides the application code generation.

This method intends to contribute for the progress of research and deployment of agent technology. Many are the proposals for methodologies and platforms for agent-based development, but these are completed independent. We do not claim that they should be intrinsically connected, but industry will only adopt the agent paradigm if there is consistent support for the development and implementation of agent-based applications. Besides, the work presented here has focused primarily on our research and development aspects. Even though we have taken a rather research-influenced approach, the work presented here is not invalidated. It suggests instead that there is further work to be done on all aspects of agent technologies to completely support the development process.

The research reported here is still in progress. We are finishing a software development environment, called Albatroz, to support the method presented here. The environment provides a tool for visual modeling using ANote, a tool for model transformation and a tool for partial code generation. In fact, the tool aims at a higher purpose: to allow the transformation from ANote models to any agent-oriented implementation platform. To do so, the transformation tool requires a XML configuration file, which is similar to the PROLOG rules we described here.

# References

Bratko, I., 2000. Prolog Programming for Artificial Intelligence, Addison Wesley, 3rd edition.

Castro, J., Kolp, M. and Mylopoulos, J, 2002. Towards Requirements-Driven Information Systems Engineering: the Tropos Project, Information Systems, No 27(6), 365-389.

Choren, R. and Lucena, C, 2005. Modeling Multi-agent Systems with ANote, Journal on Software and Systems Modeling (SoSyM). DOI: 10.1007/s10270-004-0065-y, ISSN: 1619-1374.

Choren, R. and Lucena, C, 2004. Agent-Oriented Modeling Using ANote. In: Proceedings of the Third International Workshop on Software Engineering for Large- Scale Multi-Agent Systems at ICSE 2004, 74-80.

Cranefield, S. and Purvis, M., 1999. UML as an Ontology Modeling Language. In: Proceedings of the IJCAI'99 Workshop on Intelligent Information Integration, 46-53.

Czarnecki, K. and Helsen, S, 2003. Classification of Model Transformation Approaches. In: Proceedings of the Workshop on Generative Techniques in the context of Model Driven Architecture at OOPSLA.

DeLoach, S, 1999. Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems. In: Proceedings of Agent Oriented Information Systems (AOIS99).

Fayad, M. and Schmidt, D, 1999. Building Application Frameworks: Object-Oriented Foundations of Design, John Wiley & Sons, 1$^{st}$ edition.

Huget, M, 2002. Generating Code for Agent UML Sequence Diagrams. In: Proceedings of Agent Technology and Software Engineering (AgeS).

Howden, N., Ronnquist, R., Hodgson, A. and Lucas, A., 2001. JACK Intelligent Agents: Summary of an agent infrastructure. In: Proceedings of the 5th International Conference on Autonomous Agents, Workshop on Infrastructure for Agents, MAS and Scalable MAS, 251–257.

d'Inverno, M. and Luck, M., 2004. Understanding Agent Systems, Springer, 2nd edition.

JADE Programmer's Guide, Accessed in: 02/2005, http://sharon.cselt.it/projects/jade/doc/programmersguide.pdf.

Luck, M., McBurney, P. and Preist, C., 2004. A Manifesto for Agent Technology: Towards Next Generation Computing, Autonomous Agents and Multi-Agent Sytems 9, 203–252.

Milidiu, R., Lucena, C., and Sardinha, J., 2001. An object-oriented framework for creating offerings. In: Proceedings of the International Conference on Internet Computing (IC'2001), CSREA Press, v.1, 119-123.

Mitchell, T., 1997. Machine Learning. McGraw-Hill.

Padgham, L. and Winikoff, M., 2002. Prometheus: a pragmatic methodology for engineering intelligent agents. In: Proceedings of the Workshop on Agent-oriented Methodologies at OOPSLA.

Padgham, L. and Winikoff, M., 2002. Prometheus: a methodology for developing intelligent agents. In: Proceedings of the first International Joint Conference on Autonomous Agents & Multiagent Systems.

Sardinha, J., Milidiú, R., Paranhos, P., Cunha, P. and Lucena, C., 2005. An Agent Based Architecture for Highly Competitive Electronic Markets. In: Proceedings of the 18th International FLAIRS Conference (The Florida Artificial Intelligence Research Society) (in press).

Sardinha, J., Garcia, A., Milidiú, R. and Lucena, C., 2004. The Agent Learning Pattern. In: Proceedings of the fourth Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'04.

Sardinha, J., Milidiú, R., Lucena, C., and Paranhos, P., 2003. An OO Framework for building Intelligence and Learning properties in Software Agents. In: Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems at ICSE 2003, 30-36.

Sardinha, J., Ribeiro, P., Lucena, C. and Milidiú, R., 2003. An Object-Oriented Framework for Building Software Agents. Journal of Object Technology 2(1), 85-97.

TAC web site, http://www.sics.se/tac. Accessed in: 02/2005.

TSpaces, http://www.almaden.ibm.com/cs/TSpaces/. Accessed in: 02/2005.

Wooldridge, M. and Ciancarini, P, 2001. Agent-Oriented software engineering: The state of the art. In P. Ciancarini and M. Wooldridge, (eds.), Agent-Oriented Software Engineering, vol. 1957 of LNCS, Springer, 1-28.

Zambonelli, F., Jennings, N., and Wooldridge, M., 2003. Developing Multiagent Systems: The Gaia Methodology. ACM Transactions on Software Engineering and Methodology 12(3), 317-370.