



A combined specification language and development framework for agent-based application engineering

José Alberto R.P. Sardinha ^{a,*}, Ricardo Choren ^b, Viviane Torres da Silva ^a,
Ruy Milidiú ^a, Carlos J.P. de Lucena ^a

^a *Pontifical Catholic University of Rio de Janeiro, Computer Science Department, Rua M. de S. Vicente 225, Gávea, 22451-900 Rio de Janeiro, RJ, Brazil*

^b *Military Institute of Engineering, Department of Systems Engineering, Pça Gen. Tibúrcio 80, Praia Vermelha, 22290-270 Rio de Janeiro, RJ, Brazil*

Received 8 March 2005; received in revised form 1 February 2006; accepted 11 February 2006

Abstract

Software agents are used to solve several complex problems. Software engineering processes and tools are vital to support all the development phases of agent-based systems, and guarantee that these systems are built properly. 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.

© 2006 Published by Elsevier Inc.

Keywords: Software agents; Agent oriented software engineering; Framework

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 quality affects 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 are needed to guide the analysis and design phases, agent architectures are required for the design of individual components, and integrated supporting infrastructures are essential to obtain productivity (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) and Zambonelli et al. (2003), and there are also some platforms or

* Corresponding author. Tel.: +55 21 2540 6915x133; fax: +55 21 2511 5645.

E-mail addresses: sardinha@inf.puc-rio.br (José Alberto R.P. Sardinha), choren@de9.ime.eb.br (R. Choren), viviane@inf.puc-rio.br (V.T. da Silva), milidiu@inf.puc-rio.br (R. Milidiú), lucena@inf.puc-rio.br (C.J.P. de Lucena).

architectures for agent-based system implementation, such as Howden et al. (2001) and Bellifemine et al. (2005). However, there very few methods that attempt to properly combine agent design and implementation initiatives. Such methods are important not only to allow the fast and easy development of agent systems but also to foster the interest of commercial organizations in agent systems. Consequently, this can provide further dissemination of agent technologies (Luck et al., 2004).

In this paper, we propose a development method for building multi-agent systems that goes from the requirements elicitation to the actual implementation of the agent system. This method combines some software engineering techniques in a straightforward manner, and consequently presents some progress to the agent-oriented field. Three phases are proposed in this method: (i) an agent design phase, (ii) a migration or mapping path from the design results to an agent framework, and (iii) the actual implementation of the agent system.

The design phase uses an agent-oriented modeling language called ANote (Choren and Lucena, 2004, 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.

The generated artifacts in the design phase are then mapped to a framework called Agent's SYnergistic Cooperation (ASYNC) framework (Sardinha et al., 2003b). The ASYNC object-oriented framework is used to implement the agents and the system environment. The framework also provides a communication infrastructure and uses some hot spots in order to implement the agent's goals, actions and interaction protocols. Finally, 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 a 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.

In order to demonstrate the applicability of the proposed method, the LearnAgents (Sardinha et al., 2005) case study is 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 Trading Agent Competition (TAC) travel scenario (Wellman et al., 2003). We present 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 ANote with a brief description of its concepts and artifacts. Section 3 presents a summarized explanation of the ASYNC framework. Section 4 presents our proposed method for building agent-based systems, which is the main contribution of this work. This section also uses the LearnAgents system to illustrate all the phases of our approach. 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, 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 (Fig. 1) that identifies the set of concepts and relationships supported by ANote. The conceptual meta-model is used to define scope and to identify the elements that can be present in ANote diagrams. This meta-model 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, environmental 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.

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 refinement 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 relationships. In this view, no details about agent behavior are provided since its purpose is to specify the systems structure. Agent classes specify the roles that will perform the goals elicited in the goal view and that compose the system organizations. When two agents need to inter-

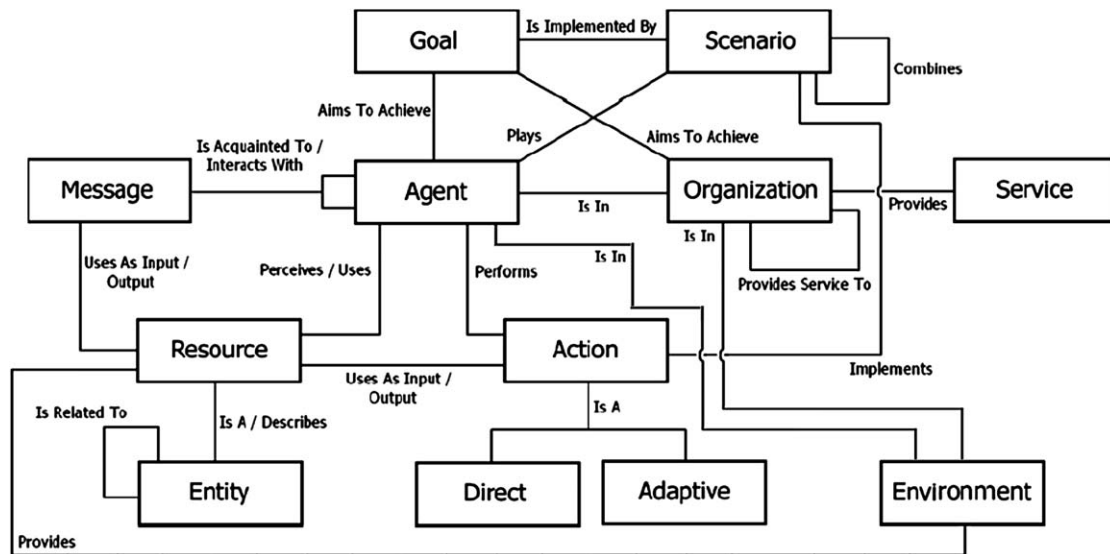


Fig. 1. The ANote meta-model structure.

act in the system, there must be a structural relationship between their agent classes, which is represented in the agent view as an association relationship.

The scenario view captures agent behavior in scenarios, which are textual representations of how goals are achieved by agents. A scenario serves 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 modeling agency characteristics. This is done by generating courses of action for both normal 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 action. It shows the agent's internal actions and their sequence, using state charts representing action states and transitions. Moreover, 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 environmental view is responsible for specifying the agents' knowledge base in terms of resources provided by the environment resources. 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 resources can be seen in [Cranfield 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 from agents in another organization.

3. The ASYNC framework

Agent infrastructures are used to support the development and operation of agent-based systems ([Luck et al., 2004](#)). In fact, they are fundamental engines that underlie autonomous components for an 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, such as Jack ([Howden et al., 2001](#)) and Jade ([Bellifemine et al., 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 complex agent systems. These are some examples of systems that use the ASYNC framework: (i) evolutionary agents for creating offerings in a retail market ([Milidiu et al., 2001](#)), (ii) agents that negotiate automatically for goods ([Sardinha et al., 2003a,b](#)), (iii) agents

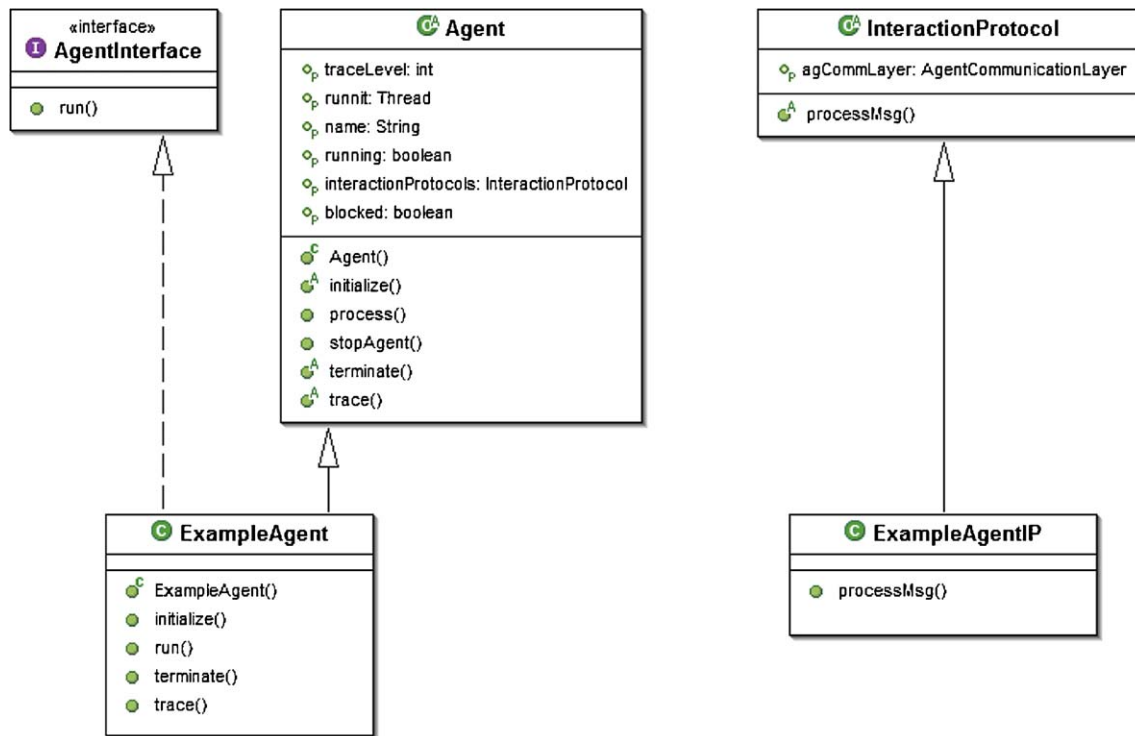


Fig. 2. An example of an instantiated agent.

that learn to play automatically in two-person games (Sardinha et al., 2003a,b), (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 object-oriented classes and interfaces for the development of agents and their environment. A detailed description of all these classes and interfaces can be found in Sardinha et al. (2003a,b). ASYNC also provides a design pattern (Sardinha et al., 2004) to implement machine-learning techniques in 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.

Fig. 2 presents an example of an instantiated agent from the ASYNC framework. The implementation of each agent requires two classes, such as the ExampleAgent class and the ExampleAgentIP class. The ExampleAgent class normally implements some basic services, such as: agent initialization, agent suspension, sensory of the environment and action execution. This class must extend the abstract class Agent and implement the AgentInterface, which are both part of the ASYNC framework. The second class implements the agent's interaction protocols, and consequently defines the way an agent can interact with other agents in the system. This class must extend the abstract class InteractionProtocol from ASYNC.

The following section presents the three phases of our method: (i) the specification of a multi-agent system using ANote, (ii) the mapping relations from the ANote artifacts to the ASYNC framework structure, and (iii) some resulting implementation code.

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 min. 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 designed as a multi-agent system because we believe that the game's complex requirements are easily implemented 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's 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 auctions and is vital for the agents to select their actions. Therefore, to build the knowledge base, the system shall: (i) constantly check (sense) the auctions' current prices and states (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 ask prices of 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 several auctions. To negotiate the system shall: (i) select a scenario with a high profit and low risk, in order to maximize the utility function (sub-goal classify best allocations); (ii) define the number and the types of travel goods, which the system must send bids to the auctions (sub-goal create bidding orders); (iii) and, finally, define a price for each bid and send it to the proper auction (sub-goal send bids to auctions). The goal view diagram for the LearnAgents system is shown in Fig. 3.

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 sub-goals in the lowest level of the goal view diagram to agents. Table 1 shows the relation between the sub-goals and the agent classes, and Fig. 4 shows the system's 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 these 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.

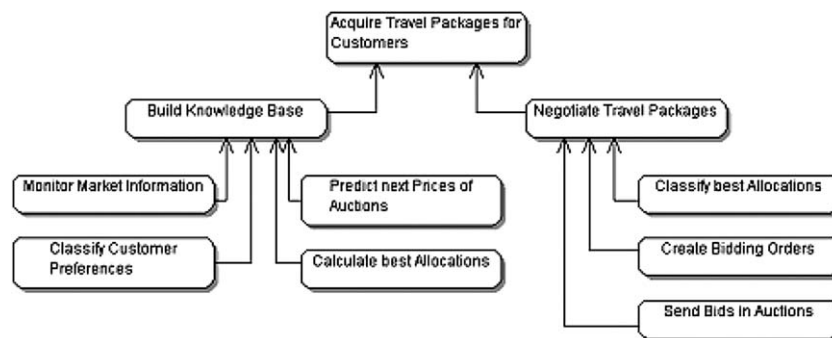


Fig. 3. The LearnAgents goal view diagram.

Table 1
The relation between system goals and agent classes

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

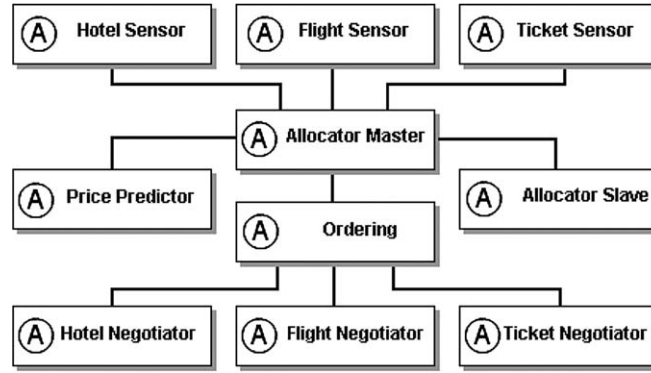


Fig. 4. The LearnAgents agent view diagram.

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 (Fig. 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 its main action plan and interacts with the Hotel, Flight and Ticket Negotiator agents.

The planning and interaction view diagrams are presented in Figs. 6 and 7, respectively, and they show the action plan and the interaction protocol for the “classify best allocations” scenario shown above.

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 environmental view diagram (Fig. 8).

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

Fig. 5. The classify best allocations scenario view diagram.

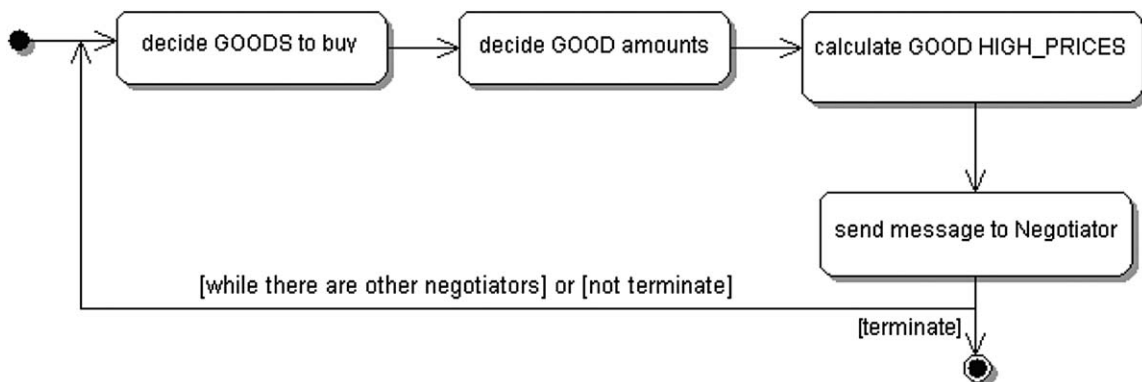


Fig. 6. The classify best allocations planning view diagram.

<Classify Best Allocations>

[<Allocator Master>] [<Ordering>] [<Hotel Negotiator>] [<Flight Negotiator>] [<Ticket Negotiator>]

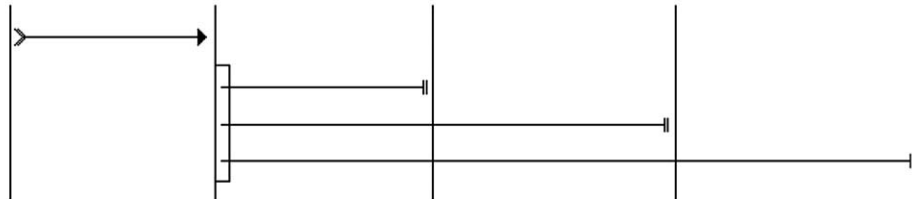


Fig. 7. The classify best allocations interaction view diagram.

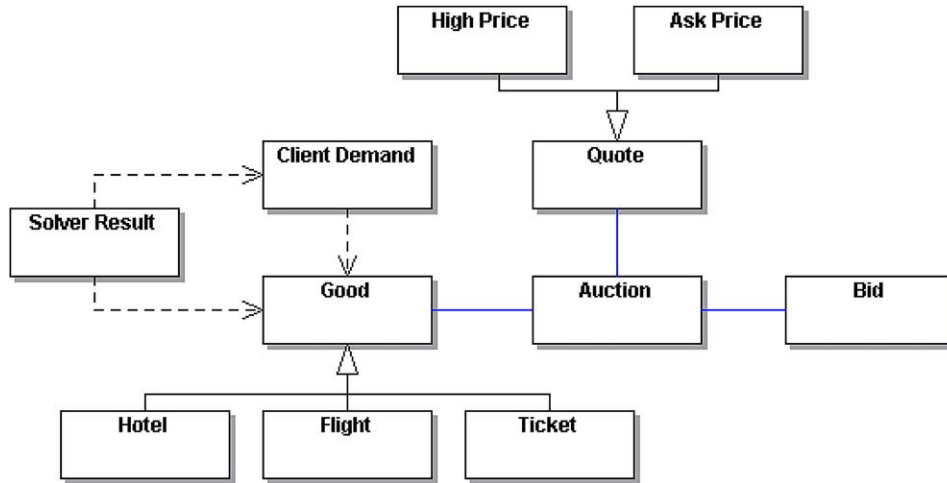


Fig. 8. The LearnAgents environmental view diagram.

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 an 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.

To guide the mapping process, a set of Prolog (Bratko, 2000) rules was 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. Implementing the system environment does this. The environment is a class that will have the references to all the agents and resources that will comprise the system. This is the system main class and it will be responsible for creating the agents,

Table 2
Mapping elements in the specification 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 environmental diagram	Simple class

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 a particular diagram did not directly specify this environment – it is the result of the entire model set. The Prolog code in lines 1–3 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. Lines 3 and 5 state 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" = AsyncIPEExtension,
    append (AnoteClass, AsyncIPEExtension, AsyncIPClass).
5. async (interactionProtocols, IPClassName, IPextends, IPimplements, IPMethods):-
    IPextends = 'InteractionProtocols',
    IPimplements = "".
```

The third step consists of mapping the actions 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 executes – 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 planning diagrams, shall be placed as methods in a concrete class that implements InteractionProtocols. 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 environmental diagram, are implemented as Java Classes. The Prolog code in line 2 presents the mapping rule of the resources in the environmental diagram to simple Java classes in ASYNC.

1. `anote (resources, ResourceNames).`
2. `anote2asyncResources (ResourceClasses):-`
`anote (resources, ResourceNames),`
`ResourceClasses = ResourceNames.`
3. `async (resources, ResourceClasses).`

Fig. 9 shows the implementation of the Ordering agent in the LearnAgents system using the rules presented above.

The Prolog code in lines 1–8 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 lines 3 and 4, (iii) produces the methods of the ASYNC classes in lines 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),`



Fig. 9. The Ordering agent.

4.3. The method phase 3: code

The Ordering agent internal actions are selected from the planning view diagram (Fig. 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 must also provide an implementation for the abstract methods of `Agent` (`initialize`, `terminate`, `trace`) and methods of `AgentInterface` (`run`). The Java code is presented in lines 1–36.

```
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 planning view diagram (Fig. 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 must also implement the abstract method of `InteractionProtocols` (`processMsg`). The Java code is presented in lines 1–24.

```

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. }
```

Fig. 10 presents the output of the Prolog program that guides the mapping process for the Ordering agent. The `ANote` class is requested and the rules produce the ASYNC concrete class names, class extensions, interface implementations and methods included in the ASYNC concrete classes.

```

SWI-Prolog -- c:/Documents and Settings/Beto Sardinha/Desktop/Trabalho/Doutorado/Agentes e E...
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (Multi-threaded, Version 5.4.6)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- go.
Anote Class = "Ordering".
Java Class = OrderingAgent
Extends = Agent
Implements = AgentInterface
Methods to implement = [Constructor, Initialize, Terminate, Trace, Run, Decide Goods
to Buy, Decide Good Amounts, Calculate Good High Price]
Java Class = OrderingAgentIP
Extends = InteractionProtocols
Implements =
Methods to implement = [Process Message, Send Message to Negoatiators]

Yes
2 ?-
```

Fig. 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, 2002a,b) 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 easier 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 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 ANote for system specification and the ASYNC framework for system implementation.

The method begins with the specification of the multi-agent system using 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. Finally, we show how this mapping guides the code generation of an application.

This method intends to contribute to the progress of research and deployment of agent technology. There are many available methodologies and platforms for agent-based development, but they are completely independent. We do not claim that they should be intrinsically connected, but industry will only adopt the agent paradigm if there is a 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 an XML configuration file, which is similar to the PROLOG rules we described here.

Acknowledgements

This work has been partially supported by the ESSMA Project under grant 552068/2002-0 (CNPq, Brazil).

References

- Bellifemine, F., Caire, G., Trucco, T., Rimassa, G., 2005. JADE Programmer's Guide. Version 3.3, TILab S.p.A.
- Bratko, I., 2000. Prolog Programming for Artificial Intelligence, third ed. Addison Wesley.
- Castro, J., Kolp, M., Mylopoulos, J., 2002. Towards requirements-driven information systems engineering: the Tropos Project. *Information Systems* 27 (6), 365–389.
- Choren, R., 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.
- Choren, R., Lucena, C., 2005. Modeling multi-agent systems with ANote. *Journal on Software and Systems Modeling* 4 (2), 199–208.
- Cranefield, S., Purvis, M., 1999. UML as an ontology modeling language. In: *Proceedings of the IJCAI'99 Workshop on Intelligent Information Integration*, 46–53.
- Czarnecki, K., 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*.
- d'Inverno, M., Luck, M., 2004. *Understanding Agent Systems*, second ed. Springer.
- DeLoach, S., 1999. Multiagent systems engineering: a methodology and language for designing agent systems. In: *Proceedings of Agent Oriented Information Systems (AOIS99)*.
- Fayad, M., Schmidt, D., 1999. *Building Application Frameworks: Object-Oriented Foundations of Design*, first ed. John Wiley & Sons.
- Howden, N., Ronnquist, R., Hodgson, A., Lucas, A., 2001. JACK intelligent agents: Summary of an agent infrastructure. In: *Proceedings of the Fifth International Conference on Autonomous Agents, Workshop on Infrastructure for Agents, MAS and Scalable MAS*, 251–257.
- Huget, M., 2002. Generating code for agent UML sequence diagrams. In: *Proceedings of Agent Technology and Software Engineering (AgeS)*.
- Luck, M., McBurney, P., Preist, C., 2004. A manifesto for agent technology: towards next generation computing. *Autonomous Agents and Multi-Agent Systems* 9, 203–252.
- Milidiu, R., Lucena, C., Sardinha, J., 2001. An object-oriented framework for creating offerings. *Proceedings of the International Conference on Internet Computing (IC'2001)*, vol. 1. CSREA Press, pp. 119–123.
- Mitchell, T., 1997. *Machine Learning*. McGraw-Hill.
- Padgham, L., Winikoff, M., 2002a. Prometheus: a pragmatic methodology for engineering intelligent agents. In: *Proceedings of the Workshop on Agent-oriented Methodologies at OOPSLA*.
- Padgham, L., Winikoff, M., 2002b. Prometheus: a methodology for developing intelligent agents. In: *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*.
- Sardinha, J., Milidiu, R., Lucena, C., Paranhos, P., 2003a. An OO Framework for building intelligence and learning properties in software agents. In: *Proceedings of the Second International Workshop on Software Engineering for Large-Scale Multi-Agent Systems at ICSE 2003*, 30–36.
- Sardinha, J., Ribeiro, P., Lucena, C., Milidiu, R., 2003b. An object-oriented framework for building software agents. *Journal of Object Technology* 2 (1), 85–97.
- Sardinha, J., Garcia, A., Milidiu, R., Lucena, C., 2004. The agent learning pattern. In: *Proceedings of the Fourth Latin American Conference on Pattern Languages of Programming, SugarLoafPLOP'04*.
- Sardinha, J., Milidiu, R., Paranhos, P., Cunha, P., 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.
- Wellman, M., Cheng, S., Reeves, D., Lochner, K., 2003. Trading agents competing: Performance, progress, and market effectiveness. *IEEE Intelligent Systems* 18, 48–53.
- Wooldridge, M., Ciancarini, P., 2001. Agent-oriented software engineering: The state of the art. In: Ciancarini, P., Wooldridge, M. (Eds.), *Agent-Oriented Software Engineering*, LNCS, vol. 1957. Springer, pp. 1–28.
- Zambonelli, F., Jennings, N., Wooldridge, M., 2003. Developing multiagent systems: the Gaia methodology. *ACM Transactions on Software Engineering and Methodology* 12 (3), 317–370.