An Object-Oriented Framework for Implementing Agent Societies

Viviane Torres da Silva Mariela Inés Cortés Carlos José Pereira de Lucena
{viviane,mariela, lucena}@inf.puc-rio.br

PUC-Rio Inf.MCC 32 /04 September, 2004

Abstract. The goal of multi-agent system architectures and frameworks is to provide reusable agent-oriented classes that can be extended and customized in order to implement domain specific systems. This paper proposes an object-oriented framework for implementing agent societies. Agent societies are multi-agent systems in which it is required or important to represent agents playing roles in organizations. By using the framework, it is possible to implement several agents; each playing different roles in different organizations. The framework contemplates agents, roles, organizations and environments as first-order entities. For each entity, a detailed lifecycle is prescribed by describing the execution states and the events that cause the (allowed) transition of the entity from one state to another. Based on graphic representations of the lifecycle models, the computational models of each entity could be explored. The computational models define the behavior of the entities associated with each state described in the lifecycle models. As a consequence, both the structural and the dynamic aspects of the ASF (Agents Society Framework) framework are presented in the paper.

Keywords: Framework, agent societies, multi-agent systems, object-oriented systems

Resumo. O objetivo de arquiteturas e frameworks para sistemas multi-agentes é prover conjuntos de classes orientadas a agentes que possam ser reutilizadas e que possam ser estendidas e customizadas para implementar sistemas de domínios específicos. Este artigo propõe um framework orientado a objetos para implementar sociedades de agentes. Sociedade de agentes são sistemas multi-agentes nos quais é necessário ou importante representar agentes desempenhando papéis em organizações. Usando o framework proposto, é possível implementar vários agentes, cada um desempenhando inúmeros papéis em diferentes organizações. O framework lida com agentes, papéis, organizações e ambientes como entidades de primeira ordem. Para cada entidade, um detalhado ciclo de vida é definidos descrevendo os estados de execução e eventos que causam as transições (permitidas) de uma entidade a partir de um estado para outro. Baseada em representações gráficas dos modelos de ciclo de vida, os modelos computacionais de cada entidade podem ser explorados. Os modelos computacionais definem os comportamentos das entidades associados a cada estado descrito nos modelos de ciclo de vida. Como conseqüência, tanto a estrutura quanto a dinâmica do framework ASF (Framework para Sociedades de Agentes) são exploradas no artigo.

Palavras-chave. Framework, sociedade de agentes, sistemas multi-agentes, sistemas orientados a objetos

1 Introduction

Advances in agent technologies depend on improving architectures and frameworks for building multi-agent systems (MAS). Frameworks and architectures offer a general-purpose support that is extended and customized in order to implement domain specific systems. MAS frameworks provide programmers with reusable object-oriented classes that can be used to implement agent-oriented systems. In this paper, we propose an object-oriented framework [8] focused on the implementation of agent societies. Agent societies are systems in which it is required or important to represent agents, organizations and agents playing roles in organizations.

The concept of agent societies (or organizations) has become an important research area in the field of agent-related systems [42][23]. Models [16][9], methodologies [39][4] and modeling languages [37][30][36] propose the use of organizations and roles for modeling multi-agent systems. In [42], the authors suggest that an organization's perspective can make the design of a system less complex, can increase the system's efficiency and can make it easier to manage. Besides, in [10], the authors point out some drawbacks of "agent centered systems" and propose "organization centered systems" to solve the weaknesses of such an approach.

An agent-oriented framework must support the essential concepts and notions of agent-based computing [31]. Since our goal is to provide an object-oriented framework for implementing agent societies, there is a need for eliciting the properties of such systems. Our proposed framework must provide capabilities to deal with social characteristics. Numerous authors identify several properties that must be considered when implementing agent societies. In [22], the author identifies agents, organizations, interactions and environments as central MAS concepts. In [11], the authors define agents, roles and groups (or organizations) as the main concepts in agent societies. Other authors [6][7] also agree that multi-agent societies are composed of agents playing roles in organizations and inhabiting environments. In the context of organizations, Zambonelli and others [41] suggest that the autonomous behavior of agents should be designed by mimicking the behavior and structure of human organizations. For each agent, a specific role is assigned characterizing the position of the agent in the organization. Therefore, a multi-agent system framework created to implement agent societies should define agents as the main abstractions. Since agent societies are composed not only of agents, other MAS entities such as organizations, roles and environments must also be encapsulated as first class abstractions. For instance, it is possible to implement agents playing different roles and dynamically changing their roles by representing agent roles and organizations as first order elements. In addition, it is also possible to define the properties of each role and organization in the system.

Many authors acknowledge the Belief-Desire-Intention (BDI) model as one of the most useful MAS architectures [32][33]. In this model, the agent's mental attributes are used to determine the state of the agent and possible attitudes related to events and messages. The advantage of using mental attributes in the design and realization of agents and multi-agent systems is a natural (human-like) modeling and high-level of abstraction [31]. To allow a smooth transition between the modeling and implementation phase, the BDI paradigm has to be supported at the implementation level, as well. Rao and Georgeff [15][32] have adopted the BDI model and transformed it into an execution model for software agents, based on the

notion of beliefs, goals and plans. In order to use the BDI model at the implementation level, a MAS framework should provide support for defining beliefs, goals and plans as properties of agents and organizations.

In the BDI model, agents are defined as being goal-oriented entities [6][7]. Agents execute plans in order to achieve their goals. Since the organization concept extends the agent concept [35], organizations should also be defined as goal-oriented entities. The behavior of agents is characterized by the goals that agents have and the plans that they execute. Different agents can define different plans. Furthermore, different agents also have different strategies for selecting the next goal to be achieved and several strategies to select the plan that will be executed to achieve the goal. Therefore, the MAS framework should support the definition of numerous plans and goals and different policies to select the goals and plans.

Another important characteristic associated with agents is their interaction capability. Agents interact by sending and receiving messages instead of calling methods of other entities. Many agent applications use ACL, an agent communication language proposed by FIPA [12], in order to provide a communication pattern between the interactive agents. For that reason, agent frameworks should support the definition of ACL compliant messages.

In conclusion, the following are some important characteristics that should be provided by a framework designed to implement agent societies: (i) to support the implementation of agents, roles, organizations and environments as first-order abstractions, (ii) to define agents and organizations as goal-oriented entities, (ii) to model agents and organizations according to the BDI model, and (iv) to use ACL to provide the communication between agents and organizations. Our goal was to develop an object-oriented framework to implement agent societies based on the aforementioned characteristics. The framework defines classes to represent agents, roles, organizations, environments and their properties. The framework also states the relationships between the classes and the implementation of some methods.

Although several MAS platforms have been published in the literature [1][2][3][5][19], none of them support the definition of environments, organizations and agent roles. Upon using these platforms, it is not possible to explicitly implement agents playing different roles in organizations and moving from one organization to another. In addition, although some platforms support mobile agents, they do not propose any mechanism to represent environments. By explicitly representing the environment, it is possible to express the interaction between environments and mobile agents and to clearly group all its properties and characteristics.

From the set of analyzed platforms, architectures and frameworks, the platform that is most closely related to our approach is Jadex [19][31]. Both ASF and Jadex allow for the development of goal-oriented agents by following the BDI model and implementing a similar execution model. The main difference between Jadex and ASF is that Jadex does not support the definition of agent role. Agents are modeled as single threads and do not change their roles during execution. In ASF, an agent can execute concurrently in several execution threads playing different roles in different organizations. In systems where it is not necessary to represent roles and organizations, platforms such as Jadex and Zeus [5] can be used.

The ASF framework was developed based on a conceptual framework called TAO [25] that defines a core set of abstractions that characterize agent societies. The abstractions, their properties and

relationships are briefly presented in Section 2. The structural aspects of the proposed framework, i.e., the classes that it defines and the relationships between the classes, are presented in Section 3. The framework is composed of several object-oriented modules and each module represents an agent-related abstraction. The dynamic aspects of the framework, i.e., the implementation of application-independent methods that are identified in the classes, are described based on the analysis of the entity lifecycle modules described in Section 4. The lifecycle models define the states and the (allowed) transitions from one state to another. According to states defined in the lifecycle models, the computational models of each entity are presented by detailing methods of the classes identified in the framework structure. The computational models, presented in Section 5, identify the common behavior of the entities during the execution of their states. By common behavior we mean behavior that is not dependent on the application being implemented and that, thus, can be described in the framework. In Section 6, an example of the framework instantiation is presented. Section 7 describes some related work and compares the existing different approaches. Finally, Section 8 presents our conclusions and plans for future work.

2 Agent Societies

An agent society is composed not only of agents [18][29][34][38] but also of organizations [4][28][34][42], roles [28][40][42], environments [18][16][28][40] and objects. Objects are passive elements that have control over their states and can modify their states during their lifetime. However, an object has no control over its behavior, meaning that it does whatever any other element asks it to do and only when it is asked to do so Agents, on the other hand, are autonomous, interactive and adaptive entities [29] that play roles in organizations [40][42]. Agents are goal-oriented entities that execute plans and actions in order to archive their goals. Agents are defined based on their goals, beliefs, plans and actions. During the execution of the agent, its set of goals, beliefs, plans and actions can be modified, as well as the set of roles that it is playing.

Environments represent the habitat of objects, agents and organizations. An environment can be a passive entity (object), or can be an active entity (by having agency characteristics such as autonomy, adaptation and interaction). The most important difference between an active environment and an agent is that an environment does not play roles.

MAS agents and sub-organizations are grouped together by organizations [4][28][34]. The term organization is used to represent partitions and groups of entities such as departments, communities and societies [10]. Like agents, organizations are autonomous, interactive and adaptive entities that have goals, beliefs, plans and actions.

An organization can define a set of axioms that agents and sub-organizations must obey. The term axiom is used to group three related terms: rule [42][28], law and norm [24]. The axioms characterize the global constraints of the organization that agents and sub-organizations must obey. An organization also defines roles that must be played by agents and sub-organizations within it and the roles that are played by objects. The organization that does not play roles in any other organization is called the main-organization.

The two most important properties of a role are (i) its definition in the context of an organization [10] and (ii) the fact that its instance must be played by an agent, by an object or by a sub-organization. A role guides and also restricts the behavior of the instances that play the role. There are two kinds of roles: object roles and agent roles. An object role guides and restricts the behavior of an object through the description of a set of features that are viewed by other elements. An object role may restrict access to the attributes and methods of an object instance, but may also add other attributes and even methods to the object instance that plays the role. An object role manipulates the object related to it. An object is not aware of the role that it is playing. It is the object role that knows the object that it is associated with. The entity that wants to access the object interacts through the object roles that it plays.

1. An agent role guides and restricts the behavior of an agent through the description of a set of goals, beliefs and actions. An agent role defines the duties, rights and protocols that restrict the behavior of the agent that is playing the role. Duties define actions that must be executed by an agent; rights are actions that an agent can execute; and protocols define interactions between agent roles. Each agent role instance is a member of one organization and is played by one agent or sub-organization [10]. The agent (or organization) playing the role instance must execute according to the goals, duties, rights and protocols specified at the agent role instance that it is playing [40]. Every agent and sub-organization plays at least one role in an organization, i.e., each agent and sub-organization is assigned to at least one specific role in the system [28][42]. However, an object is not compelled to play roles.

3 The Structure of the ASF Framework

The object-oriented ASF framework supports the implementation of agents, roles, organizations and environments as first-order abstractions. However, entities such as environments, organizations, agents and roles that are commonly used in MASs are not abstractions available in object-oriented systems (OOS). Consequently, it is not possible to directly map any one of these entities into an OO class because these entities and an OO class have different features.

In order to implement MAS entities using an object-oriented programming language such as Java, it is necessary to create sets of classes that represent the new entities. The framework is composed of sets of object-oriented modules and their relationships. Each module represents an MAS entity by mapping the entity into a set of classes and relationships. Each set of classes is composed of an abstract class that represents the entity and other concrete or abstract classes that represent the properties of the entity. The framework is described based on the definitions of MAS entities presented in Section 2. Details about the execution of the entities and implementation of the methods are presented in Section 5 based on the entities lifecycle model described in Section 4.

3.1 Agent Module

The agent module describes a set of classes and relationships that are used to represent agents and their properties. The MAS entity *agent* is represented by the abstract class called *Agent* that extends the Java class *Thread*. The use of threads makes the agent an active entity. Several threads can be associated with

an agent to represent the several roles that the agent is playing. The relationship between the agent thread and the role that it is playing is detailed in Sections 5.1 and 5.5.

Since every agent has a set of goals, beliefs, plans and actions, the abstract class that represents an agent must define attributes to store the values associated with these properties. Goals and beliefs are defined as attributes in Section 2 and are represented by the classes *Belief* and *Goal* that define the following attributes: name, value and value type. Furthermore, the class *Goal* also defines attributes to store the goal priorities and the goal type. Jadex predefines some goals types. ASF does not predefine the types of the goals. The class *Goal* also defines two flags to indicate if the agent has tried to achieve the goal and if the goal has been achieved.

A plan is composed of a set of actions and is associated with a goal that it can achieve. Each plan defines the sequence (or the order) of the actions that will be executed when the plan is called. Since each plan defines its particular execution, it is not possible to create a concrete class called *Plan* to represent any plan. An abstract class called *Plan* is created to define the abstract structure of plans. The abstract class *Plan* defines two attributes and one method. One attribute stores the associated goal and the other a list of actions. The method called `execute` implemented in the abstract class *Plan* defines a basic execution of every plan by calling the actions in the list by following their predefined sequence. The basic execution can be specialized (re-implemented) by the concrete plans – plans defined in the application – that extend the abstract class *Plan*. Section 6 illustrates the specialization of the method according to an application plan.

An action defines the tasks of an agent. While executing an action, an agent can, for instance, send and receive messages, call methods and change its state. Since each action defines different executions, an abstract class called *Action* is created to represent the structure of every action. This class defines two attributes (pre-conditions and post-conditions) and a method called `execute` that must be implemented by the concrete actions defined in the application. The pre-conditions and post-conditions are instances of the class *Condition* that has the attributes condition type, condition name and condition value.

In addition, since agents are interactive entities that communicate by sending and receiving messages, a class called *Message* is created and associated with the class *Agent*. The class *Agent* defines two attributes to store the incoming and outgoing messages. In summary, the agent model is represented by eleven classes and their relationships, as illustrated in Figure 1.

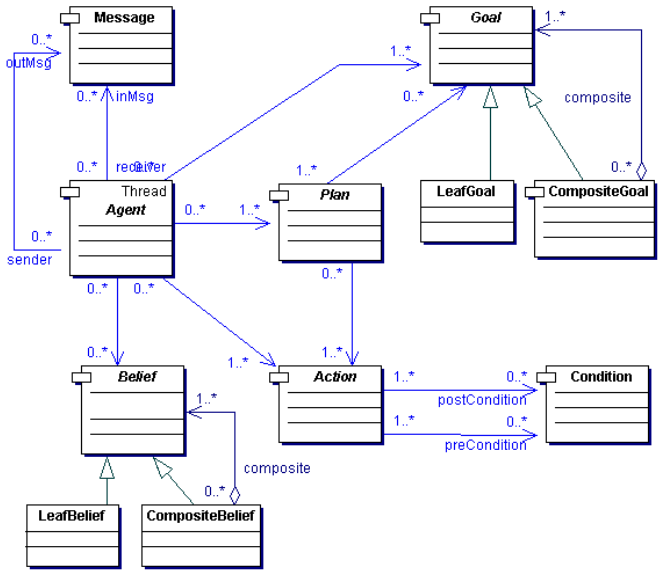

Figure 1. Classes representing an agent and its properties.

3.2 Organization Module

The organization module defines the classes related to the representation of organizations and their properties. A generic organization is represented by the abstract class *MainOrganization* or by the abstract class *Organization* that extends the class *MainOrganization*. The class *MainOrganziation* extends the java class *Thread* making it possible to define active organizations. Since organizations also have goals, beliefs, plans and action, the class *MainOrganization* defines attributes to store the list of goals, beliefs, plans and actions of an organization. Therefore, the classes *Goal*, *Belief*, *Plan* and *Action* are related to the class *MainOrganization*. Besides defining these properties, organizations also define axioms. Axioms are represented by the class *Axiom* related to the class *MainOrganization*. In addition, since organizations interact by sending and receiving messages, the class *MainOrganization* is also related to the class *Message*.

Two different classes were created to represent organizations: classes *MainOrganization* and *Organization*. Organizations that do not play roles should extend the class *MainOrganization* and the ones that play roles, i.e., sub-organizations, should extend the class *Organization*. The relationships between these two classes and between the class *Organization* and the class that represent the roles (see Section 3.6) explore the differences between the classes. The class *MainOrganization* is related to the class *Organization* in order to indicate sub-organizations and to indicate the organizations where sub-organizations play roles. In summary, the organization model is represented by 12 classes and their relationships as illustrated in Figure 2.

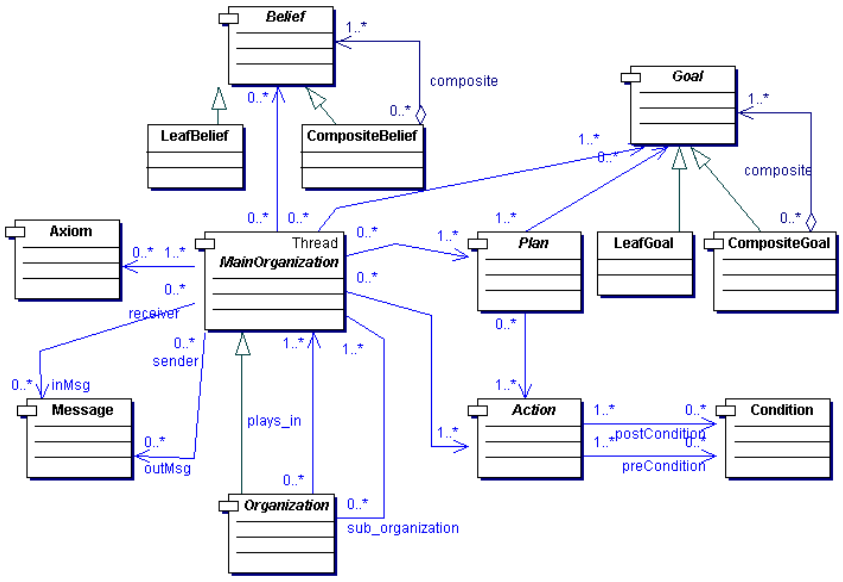

Figure 2. Classes that represent an organization and its properties.

3.3 Agent Role Module

Agent roles are represented based on the agent role module. This module describes the classes and relationships that define agent roles. The entity *agent role* is represented by the abstract class *AgentRole*. The classes that correspond to the goals, beliefs, duties, rights and protocols – properties of agent roles – are associated with the class *AgentRole*.

The classes *Right* and *Duty* identify an action and therefore are related to the class *Action*. Protocols are represented by the abstract class called *Protocol* that is related to the class *Message*. A protocol defines a sequence of incoming and outgoing messages. Since each protocol defines a sequence of messages, concrete classes representing application protocols must extend the abstract class *Protocol*. Figure 3 illustrates the agent role model composed of 12 classes and their relationships.

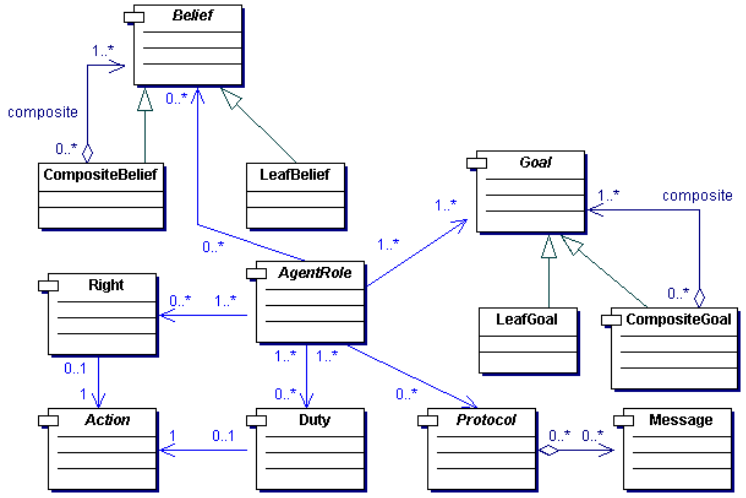

Figure 3. The class that represents an agent role and its related properties.

3.4 Object Role Module

The object role module is very simple. It is composed of one abstract class that represents the object role itself. Since the properties of object roles are attributes and methods, there is no need for defining a class to represent their properties. Figure 4 illustrates the object role model.

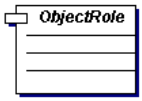

Figure 4. The class that represent object roles.

3.5 Environment Module

Like the object role module, the environment module is also very simple. The module is composed of one abstract class that represents the environment itself. Since the properties of an environment depend on the application description, its properties must be defined when creating the concrete class that will represent the application environment. The environment model is depicted in Figure 5.

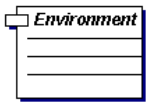

Figure 5. The abstract class Environment and its related classes.

3.6 Grouping the Modules

Once the entity modules are defined, it is important to understand how these modules are related in order to compose the object-oriented framework. The relationships between the modules are represented by the relationships between the abstract classes associated with agent, organization, agent role, object role and environment.

Environments are the habitat of agents, organizations and objects. Therefore, the abstract class *Environment* is associated with the abstract classes *Agent* and *MainOrganization* and with the Java class *Object*. The class *Environment* defines attributes to stores agents, organizations and objects.

Agents play agent roles in organizations. The abstract class *Agent* is related to the abstract class *AgentRole* to represent the roles that agents play and to the abstract class *MainOrganization* to represent the organizations where agents play roles. The class *AgentRole* is also associated with the class *Organization* to indicate organizations that can play roles and with the class *MainOrganization* to indicate owners of the agent role. The class *MainOrganization* is also related to the class *ObjectRole* in order to define the owners of object roles. Finally, the class *ObjectRole* is related to the class *Object* to identify the objects that play roles. Figure 6 shows the relationship between the abstract classes and Figure 7 illustrates the OO framework by composing the modules.

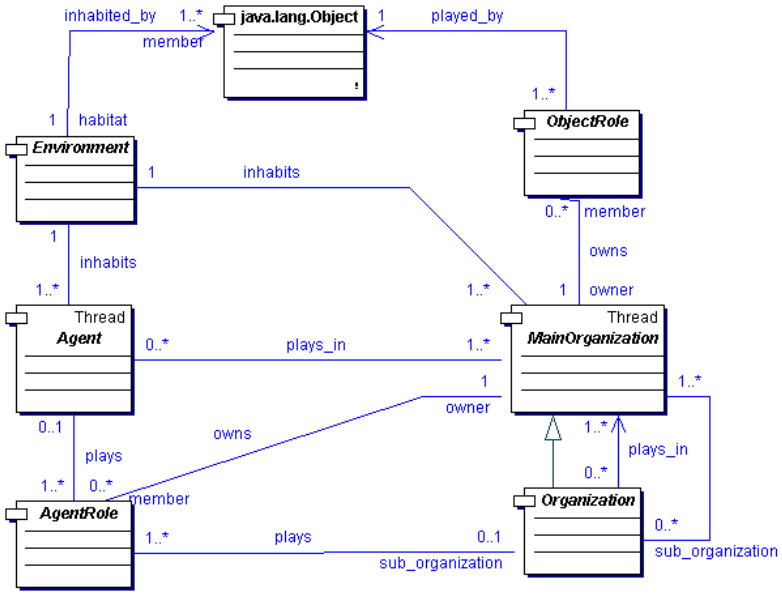

Figure 6. The relationships between the modules

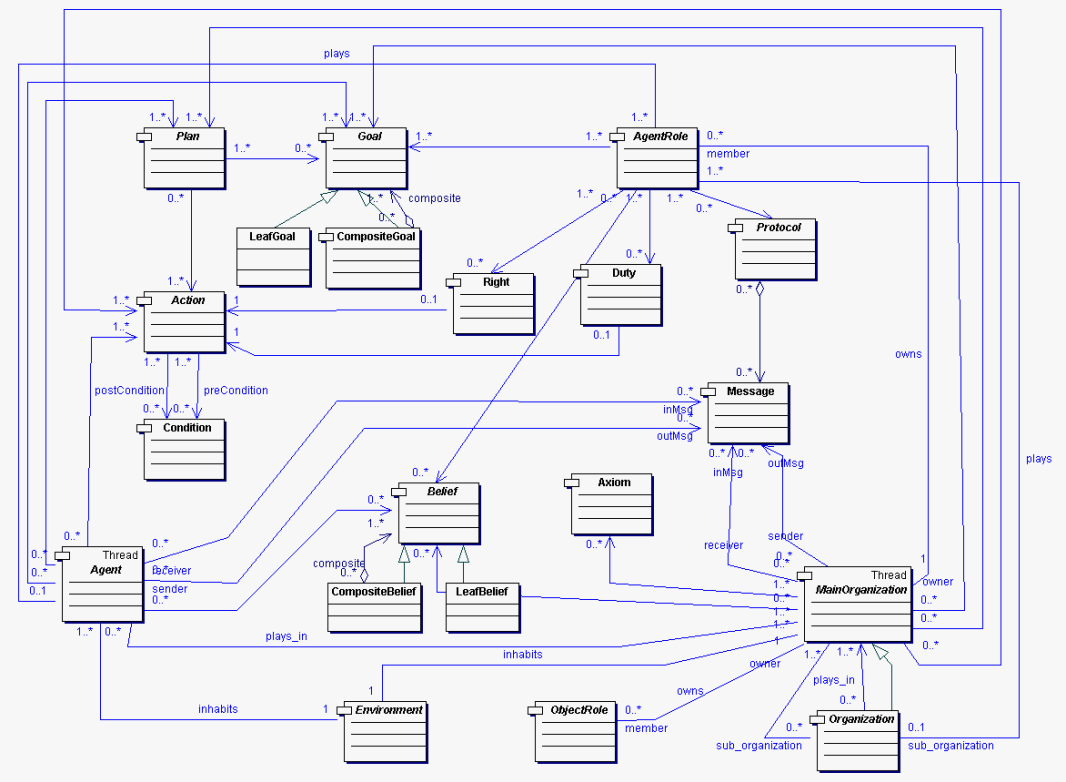

Figure 7. The object-oriented framework for implementing agent societies

4 The Entities Lifecycle Models

The lifecycle model of an entity prescribes the different execution states of the entity and the events that cause the (allowed) transition from one state to another. A lifecycle model is closely related to the computational model that describes how the execution occurs when an entity is in the states defined in the lifecycle model.

The lifecycle models presented in this Section are basic models, which provide a minimal lifecycle for objects, object roles, agents, agent roles, organizations and environments. The models may be extended by defining additional states and transitions, if so required. Other lifecycle models for agents have already been proposed in [20][27]. However, those lifecycle models do not relate agents to the roles that they play, which is an important characteristic of societies of agents.

4.1 Object and Object Role Lifecycle Models

The lifecycle models for objects and object roles are similar. Although such models are composed of the same states, as depicted in Figure 8, the definition of each state is different. The lifecycle model for objects is a well-know lifecycle model composed of three states. In the *start* state objects are created and associated with the environment where they inhabit. After being created, an object changes its state. In the *running* state, an object executes whenever one of its methods is called by an entity. The destruction of the object occurs when an entity kills the object. The destruction is represented by the *death* state. During the *death* state the register of the object in the environment is deleted.

Figure 8. Lifecycle model for Object and Object Roles

The *start* state of the object roles lifecycle model indicates the creation of an object role, the association of the object role with an object that will play the role, and the association of the object role with the organization where the role will be played. After being created, the object role can execute. In the *running* state the object role also executes whenever an entity calls one of its methods. The destruction of object roles is represented by the *death* state. In such state, the object role is destroyed. Before being destroyed, the object role must inform the organization where the role is being played. Note that there is no need for informing the object that is playing the role since objects are not aware of the object roles being played.

4.2 The Agent Roles Lifecycle Model

The agent roles lifecycle model is composed of four states, as illustrated in Figure 9. A role is created when an agent or a sub-organization commits to play the role in an organization. It is destroyed when the commitment is canceled. The creation of the role is represented by the *start* state and its destruction is represented by the *death* state.

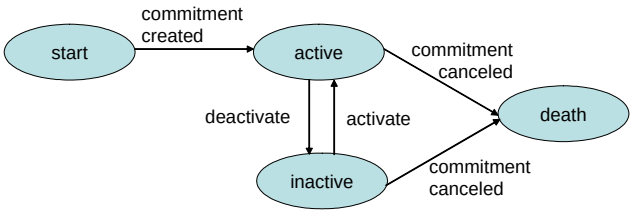

Figure 9. Agent roles lifecycle model

When a role is created, it automatically changes its state and becomes an *active* role. A role in the *active* state is being played by the entity associated with it, i.e., there is an agent role instance associated with an entity instance that is playing the role. An active role can be destroyed or can become inactive. When the entity playing the role cancels the commitment with the organization, the role is canceled, i.e., the role instance is destroyed. An active role becomes inactive when the entity stops playing the role but does not cancel the commitment with the organization to play the role. The *inactive* state represents the deactivation of the role. For instance, when an entity leaves the organization where it is playing roles without canceling the commitments with the organization, its roles become inactive. A role instance in the *inactive* state exists but is not being played, i.e., there is an agent role instance associated with an entity instance but the role is not being played. An inactive role can become active when the entity associated with the role starts playing the role. For instance, if the entity returns to the organization it can reactivate its roles. An active or an inactive role can be destroyed if the entity associated with the role cancels the commitment.

Note that agent roles are created, destroyed, activated and deactivated in the context of the entities (agents or sub-organizations) that are playing the roles. Agent roles are entities that depend on other entities in order to exist. Section 4.3 presents the lifecycle models for agents and sub-organizations by describing the influence caused by the lifecycle model for agent roles.

4.3 The Agents and Sub-organizations Lifecycle Models

The lifecycle models for agents and sub-organizations are equivalent. Like agents, sub-organizations also play roles and can move from one organization to another and from one environment to another. Their lifecycle model, presented in Figure 10, consists of five states, and a number of transitions between these states. The lifecycle model of agents and sub-organizations is directly influenced by the roles that they play. Each state and each transition defined in the model is somehow related to the roles that the entity plays. For instance, when an agent (or sub-organization) is created, a role also is created to be played by the entity and when an agent (or sub-organization) is destroyed all the roles that it was playing also are destroyed.

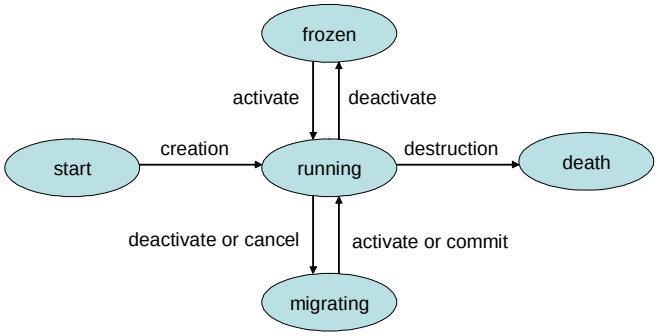

Figure 10. Lifecycle model for Agents and Sub-organizations

The first state of the agents and sub-organizations lifecycle models is the *start* state. In the *start* state, the agent (or sub-organization) is created (or instantiated) to play a role in an organization and to inhabit an environment. In such a state a role instance must be created to be played by the agent (or sub-organization) that must be registered in the environment. After its creation, the agent (or sub-organization) begins its execution in the *running* state. While executing, an agent (or sub-organization) can change its roles by committing to new roles and by canceling, activating or deactivating its roles.

In the *frozen* state, all of the roles of an agent (or sub-organization) are deactivated. This means that the roles associated with the agent (or sub-organization) are not being played; the entity is not executing its roles. In order to return to the *running* state, an agent (or sub-organization) must activate one of its deactivated roles. In the running state, at least one role associated with the entity is being played, i.e., one role is being executed.

As stated before, agents (and sub-organizations) can move from one environment into another. In order to move to another environment, an agent must deactivate or cancel all the roles that it was playing in the previous environment. An agent (or sub-organization) cannot play roles in different environments. When it arrives in another environment, it needs to commit to a new role or activate one of its roles. The *migrating* state characterizes the transition from one environment to another. The agent enters the *migrating* state by canceling or deactivating its roles. Next, its context is delivered to the destination node where the process is resumed and the agent re-enters the *running* state. The agent enters the *running* state by committing to a new role in an organization that inhabits the new environment or by activating one of its roles defined in one of the new environment's organizations.

The migration of sub-organizations is more complex than the migration of agents. The complexity of sub-organizations migration is related to the internal characteristics of sub-organizations. Since roles are being played in a sub-organization, before it moves to another environment, agents and other sub-organizations playing roles in it must stop playing their roles. The role instances being played in the sub-organization may be destroyed or may become inactive before the sub-organization moves to the other environment. Therefore, in order to migrate, a sub-organization must cancel or deactivate its own roles and the roles being played by other entities.

When the sub-organization arrives in another environment, it needs to commit to a new role or activate one of its roles by re-entering in the *running* state. Moreover, the roles being played in the sub-

organization before it moved also can be activated by the agents and sub-organizations that were playing those roles and also moved to the new environment.

The destruction of an agent depends on the destruction of the roles the agent is playing. On the other hand, the destruction of a sub-organization depends on the destruction of the roles it is playing and also on the destruction of the roles being played in it by agents and other sub-organizations. The destruction of agents and sub-organizations also depends on the cancellation of their record in the environment. Such destruction is represented by the *death* state.

4.4 Main-Organizations and Environments Lifecycle Models

As stated before, main-organizations and environments do not play roles and are not mobile entities. Therefore, their lifecycle models (Figure 11) are different from the models for agents and sub-organizations since they are not influenced by roles.

Figure 11. Lifecycle model of Main-organizations and environments

The first state of the lifecycle model of main-organizations or environments is the *start* state. The start state represents the creation of main-organizations and environments and the registration of the new main-organization in the environment. After being created, the main-organization and environment automatically enter into the *running* state. While executing, main-organizations, for instance, can permit or deny the entrance of an agent to play one of the roles that it defines and environments can permit or deny the entrance of foreign agents. When a main-organization is destroyed, its record in the environment must also be destroyed. The destruction of main-organizations and environments is represented by the *death* state.

5 The Entities Computational Models

The computational models are defined based on the lifecycle models of entities and on the description of agents, organizations, environments and agent roles presented in Section 4. The computational models are responsible for identifying the common behavior of the entities realized during the execution of their states. The dynamic aspects of the MAS framework are characterized through the identification of the entities' common behavior.

5.1 The Environment Computational Model

The environment lifecycle model is composed of three states. The following sub-sections will present the computational models for each one of them.

5.1.1 The Start State

Every MAS is composed of one environment. The creation of an environment does not depend on any other entity. Since every agent, organization and object inhabits an environment, the environment must be created before the creation of other entities. In order to create an environment, it is only necessary to

instantiate an application class that extends the abstract class *Environment* defined in the OO framework. Figure 12 illustrates the creation of a *Virtual_Marketplace* environment instance.

```
Environment env = new Virtual_Marketplace ();
```

Figure 12. The creation of an environment

5.1.2 The Running State

As stated before, an environment can be implemented as a passive entity such as an object or an active entity such as an agent. When the environment is an active entity it has its own control thread. In both cases, the environment stores the record of agents, organizations and objects that inhabit the environment. While executing, environments can, for instance, send messages to and receive messages from other entities and also control the entrance of foreign agents and organizations. Since the behavior of environments can vary and cannot be previously defined, it is not possible to describe the computational model of the running state of environments.

5.1.3 The Death State

An environment can only be destroyed after all the agents, organizations and objects have been destroyed. Therefore, it is important to check if the entities have already been destroyed. If there still is any entity inhabiting the environment, it must be destroyed before destroying the environment. Since agents and organizations play roles, the destruction of these entities are more complex than the destruction of objects. Objects are destroyed by canceling their record in the environment. Figure 13 illustrates the destruction process for environments. The method *destroy()* defined in the abstract class *Environment* represents the computational model of every environment being destroyed. Independently of the application, every environment must destroy the agents, organizations and objects before being destroyed.

```
public abstract class Environment extends Thread
{
   ...
   public void destroy()
   {
      //Destroying the agents
      Vector vAgents = getAgents();
      if (vAgents != null)
      {
         Enumeration enumvAgents = vAgents.elements();
         while (enumvAgents.hasMoreElements()) {
            Agent agentAux = (Agent)enumvAgents.nextElement();
            agentAux.destroy();
         }
      }
      //Destroying the organizations
      Vector vOrganizations = getOrganizations();
      if (vOrganizations != null)
      {
         Enumeration enumvOrg = vOrganizations.elements();
         while (enumvOrg.hasMoreElements()) {
            MainOrganization orgAux = (MainOrganization)enumvOrg.nextElement();
```

```
            orgAux.destroy();
         }
      }
      //Destroying the objects
      Vector vObjects = getObjects();
      if (vObjects != null)
      {
         Enumeration enumvObjects = vObjects.elements();
         while (enumvObjects.hasMoreElements()) {
            Object objAux = enumvObjects.nextElement();
            cancelObjectRegister(objAux);
         }
      }
   }
   ...
}
```

Figure 13. The destruction process of environments

5.2 Objects Computational Models

The computational models for objects are the simplest computational models since the creation and destruction of objects are very simple and it is not possible to predict any application independent behavior during the execution of objects.

5.2.1 The Start State

The object creation depends on the creation of an environment the object will inhabit. After its creation, the object must be recorded in the environment. The computational model related to the creation of an object is illustrated in Figure 14. This figure illustrates the object *book* being created and being registered in the environment.

```
Book book = new Book ();
env.registerObjects(book); //Registering the object in the environment
```

Figure 14. The creation of an object

5.2.2 The Running State

During the running state, objects execute their methods. While executing the methods, they can, for instance, call methods and change their current state. Since it is not possible to describe the default behavior for every object, it is not possible to describe the computational models of their running state.

5.2.3 The Death State

In order to destroy an object, it is necessary to cancel the record of the object in the environment. After the destruction of the record, the object can be destroyed. Figure 12 shows the object *book* being destroyed in the environment.

```
env.cancelObjectRegister(book);
```

Figure 12. The destruction of an object

5.3 Object Roles Computational Models

The object roles computational models are more complex than the objects computational models because the creation and destruction of object roles are more complex than the creation and destruction of objects.

5.3.1 The Start State

The creation of an object role depends on the creation of an organization where the object role will be played. The constructor method of an object role receives the organization where the role will be played as an input parameter. After its creation, the object role must be associated with the object that will play the role. Note that the object does not know about the role it is playing. It is the object role that refers to the object. Figure 13 depicts the computational model associated with the start state of object roles lifecycle models. The figure exemplifies the creation of an object role by instantiating the class extension of the abstract class *ObjectRole* defined in the framework and the association between the role and the object that will play the role.

```
Offer bookOffer = new Offer(mainOrg);
bookOffer.setObject(book);
```

Figure 13. The creation of object roles

5.3.2 The Running State

Object roles execute when their methods are called. During the execution of the methods, an object role can, for instance, call its methods, call methods of the object associated with it and change its running state. As in the running state of objects, it is not possible to describe the computational models of the running state of object roles. The behavior of object roles depends on the application domain.

5.3.3 The Death State

When destroying object roles, it is necessary to remove the role from the list of roles being played by objects in an organization. Note that it is not required to inform the object playing the role. As stated before, objects are not aware of the object roles they are playing. The destruction of object roles is represented by the method *destroy()* shown in Figure 14. Since the process of destroying an object role is common to any object role, such method can be defined in the abstract class *ObjectRole*.

```
public abstract class ObjectRole
{
   ...
   public void destroy ()
   {
      //object playing role
      setObject(null);

      //deleting role of the organization where it was being played
      MainOrganization organization = getOwner();
      Vector vRoles = organization.getObjectRoles();
      Enumeration enumvRoles = vRoles.elements();
      while (enumvRoles.hasMoreElements()) {
         ObjectRole roleAux = (ObjectRole)enumvRoles.nextElement();
```

```
            if (roleAux == this)
            vRoles.remove(roleAux);
        }
    }
    ...
}
```

Figure 14. The destruction of object roles

5.4 Agent Roles Computational Model

The agent roles lifecycle model is composed of four states. The following sub-sections will present their computational models.

5.4.1 The Start State

The creation of a role instance depends on the creation of the organization instance where the role instance will be played. A role instance is created to be played in an organization. Thus, the constructor method of an agent role must receive the organization where the role will be played as an input parameter. In addition, a role name that is unique in the scope of the environment inhabited by the organization should be indicated in order to identify the role being created. After its creation the agent role should be associated with an agent or a sub-organization that will play the role. A role created but not associated with an entity is not being played. In Figure 15, an instance of the agent role *Buyer* is created to be played in the main organization. The agent role class *Buyer* extends the abstract class *AgentRole* defined in the framework.

```
AgentRole agRole = new Buyer (mainOrg);
agRole.setRoleName ("Buyer01");
```

Figure 15. The creation of an agent role

5.4.2 The Active State

The active state represents the execution of an agent role. An agent role is executed in the context of an agent or a sub-organization. Each agent and sub-organization thread is associated with a role and the execution of an entity thread represents the execution of a role. A role can define duties and rights that are associated with the entity that is playing the role but cannot specify how the entity will execute in order to follow the duties and the rights. Therefore, there is no computational model associated with the *active* state of agent roles. The computational model associated with the execution of agent roles are described by the computational models of the running states of agents and sub-organizations. The execution of sub-organizations and agents are presented in Sections 5.6.2 and 5.5.2, respectively.

5.4.3 The Inactive State

The inactive state of a role characterizes a role that exists but is not being played. In such a state, the entity thread associated with the roles is not running, i.e., it is suspended. In the proposed framework, a thread can be suspended and resumed by executing the methods represented in Figure 16. The flag *threadSuspended* is used to indicate if a thread is suspended. Since every agent role is associated with an

agent (or sub-organization) thread, these methods can be used by the agent to activate and deactivate a role.

The method *checkIfSuspended* in Figure 17 is used to verify if the entity thread was suspended and, thus, should wait until it is resumed. During the execution of the entity (agent or sub-organization) it must check if the thread was suspended. Figure 18 represents the method *run()* associated with the execution of agents.

```
public abstract class AgentRole
{
   ...
   public synchronized void supendThread()
   {
      threadSuspended = true;
   }
   public synchronized void resumeThread()
   {
      threadSuspended = false;
   }
   protected boolean threadSuspended()
   {
      return threadSuspended;
   }
   ...
}
```

Figure 16. Suspending and resuming a thread

```
public abstract class Agent extends Thread
{
   ...
   private void checkIfSuspended(AgentRole role)
   {
      if (role.threadSuspended())
      {
         synchronized (this)      {
            while (role.threadSuspended())
               System.out.println("Suspended");
            System.out.println("Resumed");
         }
      }
   }
   ...
}
```

Figure 17. Checking if the thread was suspended

```
public abstract class Agent extends Thread
{
   ...
   public void run()
   {
      ...
      AgentRole currentRole = getCurrentRole();
      if (currentRole != null)
      {
         //Cheking if thread was stopped
```

```
         while (continueExecution && !checkIfStopped(currentRole)) {
            //Checking if thread was suspended
            checkIfSuspended(currentRole);
         }
      }
      ...
   }
   ...
}
```

Figure 18. The method run() of an agent checking if the thread was suspended

5.4.4 The Death State

When a role is destroyed, the entity that was playing the role stops playing it. If it is the case, the entity thread associated with the role is stopped. In the proposed framework, a thread is stopped by executing the method *stopThread()*defined in the *AgentRole* class and represented in Figure 19. The flag *threadStopped* defined in roles is used to indicate if the thread related to the role was stopped.

During the execution of the entity, it checks if it must stop playing the role, as illustrated in Figure 21 by the method *run()*. When the thread realizes that it must stop, it automatically destroys the role associated with the thread. In order to destroy a role, the role must be removed from the list of roles being played by the agent and from the list of roles being played in an organization. The computational model associated with the destruction of a role is illustrated by the *destroy()* method in Figure 19. If the role being destroyed is the unique role of the entity, the entity must also be destroyed.

```
public abstract class AgentRole
{
   ...
   protected boolean threadStopped()
   {
      return threadStopped;
   }
   public void stopThread()
   {
      threadStopped = true;
   }
   protected void destroy ()
   {
      //agent playing role
      setAgent(null);
      //organization playing role
      setOrganization(null);

      //The current role must be removed from the list of the roles being played
      //in an organziation
      MainOrganization organization = getOwner();
      Vector vRoles = organization.getAgentRoles();
      Enumeration enumvRoles = vRoles.elements();
      while (enumvRoles.hasMoreElements()) {
         AgentRole roleAux = (AgentRole)enumvRoles.nextElement();
         if (roleAux == this)
            vRoles.remove(roleAux);
      }
```

```
   }
   ...
}
```

Figure 19. Stopping a thread and destroying a role

```
public abstract class Agent extends Thread
{
   ...
   protected boolean checkIfStopped(AgentRole role)
   {
      retunr role.threadStopped();
   }
   ...
}
```

Figure 20. Checking if the thread was stopped

```
public abstract class Agent extends Thread
{
   ...
   public void run()
   {
      Vector vPlansExecuted = new Vector();
      boolean continueExecution = true;
      AgentRole currentRole = getCurrentRole();
      if (currentRole != null)
      {
            //Cheking if thread was stopped
            while (continueExecution && !checkIfStopped(currentRole))
            {
               ...
            }
            //The thread was stoped

            //The current role must be removed from the list of the roles being played
            Vector vRoles = getRolesBeingPlayed();
            vRoles.remove(currentRole);

            //The current role must be destroyed
            currentRole.destroy();

            //Verify if the agent is playing other roles
            vRoles = getRolesBeingPlayed();
            if (vRoles == null)
               //If the agent is not playing other roles, the agent must be destroyed
               destroy();
        }
      }
   }
   ...
}
```

Figure 21. The method run() checking if the thread was stopped

5.5 Agent Computational Model

The lifecycle model of agents is composed of the start, running, frozen, migrating and death states. The computational model for the migrating state will not be presented here since mobile agents are out of the scope of the paper. The computational model for agents will be presented based on the other states.

5.5.1 The Start State

The creation of an agent instance must be immersed in an environment instance and cannot inhabit more than one environment at the same time. Thus, the creation of an agent instance depends on the creation of an environment instance. In order to immediately relate the agent and the environment, the constructor method of agents must receive the environment that the agent will inhabit as an input parameter.

Moreover, an agent is created to play a role in an organization. The creation of an agent instance depends on the creation of a role instance that will be played by the agent and in an organization instance that defines the role. Therefore, when an agent is created it is also associated with an organization and a role to be played. The constructor method of agents also receives an organization and a role as input parameters.

Figure 22 presents an example of the creation of an agent instance based on the *User_Agent* class. When creating an agent, its name should be indicated since agents interact by sending and receiving messages addressed to their names. Each name should be unique in the whole environment.

```
Agent agent = new User_Agent (env, mainOrg, agRole);
agent.setAgentName ("UserAgent::Viviane");

Thread agentThread = new Thread(agent, agRole.getRoleName());
agentThread.start();
agRole.setAgent (agent);
```

Figure 22. The creation of an agent

Threads are used in order to implement autonomous agents. Each thread is associated with an agent playing one role. The number of threads associated with an agent is equivalent to the number of roles being played by the agent. For each new role played by an agent, a new thread is created and associated with the agent and with the new role. Figure 22 depicts a thread being created and associated with the *User_Agent* instance and with the agent role *agRole* instance.

5.5.2 The Running State

After its creation, the agent starts to execute in order to achieve its goals and the goals of the role that it is playing. The agent *running* state describes the execution of agents while trying to achieve the goals.

In this paper we are talking about goal-oriented agents. Agents that are goal-oriented and execute plans in order to achieve goals have the same basic behavior. Every goal-oriented agent begins its execution by selecting a goal to be achieved. Based on the selected goal, a plan that makes the achievement of the goal possible is selected. After selecting a plan, the plan and its associated actions are executed.

The selection of goals and plans and the execution of plans are domain-independent behavior that every goal-oriented agent executes. Figure 23 illustrates a domain-independent state machine that

characterizes the *running* state of such agents. The basic executions of the agents are divided into three phases: goal selection, plan selection and plan execution.

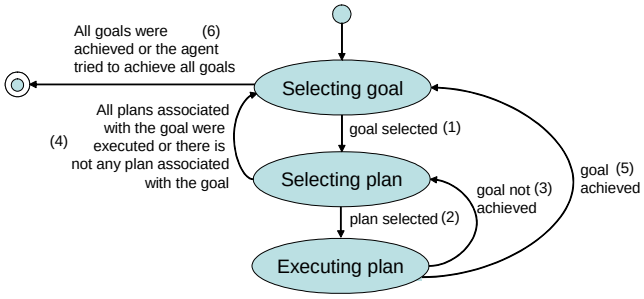

Figure 23. Phases of the agent *running* state

The first task that the agent performs is to select a goal. If there is any goal to be achieved (1), it selects a plan (2) based on the goal and on the plans that it has previously executed. After executing the plan, if the plan did not achieve the goal (3), another plan is selected to achieve the same goal. If there is no other plan to be executed associated with the goal (4) or if the goal was achieved (5), the agent selects another goal. When the agent achieves all its goals or tried to achieve them (6), the agent can decide to keep trying to achieve its goals or to give up.

By assuming that every MAS agent is a goal-oriented and a plan-based entity, it is possible to implement a common algorithm that is executed by every agent while playing roles. The algorithm is implemented in the method *run()* illustrated in Figure 24 according to the state machine depicted in Figure 23. Every agent begins its execution by choosing a goal and a plan and, then, executing a plan. The algorithms used to select a goal and a plan and the algorithm implemented by the plan are domain-dependent algorithms, i.e., different applications can define different algorithms. Therefore, in the proposed framework, those algorithms are abstract classes and should be extended according to the application.

```
public abstract class Agent extends Thread
{
   ...
   public void run()
   {
      Vector vPlansExecuted = new Vector();
      boolean continueExecution = true;
      AgentRole currentRole = getCurrentRole();
      if (currentRole != null)
      {
            //Cheking if thread was stopped
            while (continueExecution && !checkIfStopped(currentRole))
            {
               //Checking if thread was suspended
               checkIfSuspended(currentRole);

               //Selecting a goal to be achieved
               Goal goal = selectingGoalToAchieve();

               //Cheking if thread was stopped
               while (goal != null)
```

```
            {
               //Selecting a plan to be executed
               Plan plan = selectingPlan(vPlansExecuted, goal);

               //Cheking if thread was stopped
               while (plan != null)
               {
                  //Checking if thread was suspended
                  checkIfSuspended(currentRole);

                  //Executing a plan
                  executingPlan(plan);

                  checkIfSuspended(currentRole);
                  if (checkIfStopped(currentRole))
                     break;
                  vPlansExecuted.add(plan);
                  if (!goal.getAchieved())
                     //Secting another plan
                     plan = selectingPlan(vPlansExecuted, goal);
                  else {
                     //Goal achieved
                     //If goal type eguals maintaim, the agent must always try to achieve the goal
                     //but now it has low priority in order to let other goals to be achieved
                     if (goal.getGoalType().equals("maintain")) {
                        goal.setAchieved(false);
                        goal.setPriority(1);
                     }
                     plan = null;
                  }
               }
               //The goal was achieved or
               //all plans associated with the goal were executed or
               //there is not any plan associated with the goal
               //Selecting another goal
               if (checkIfStopped(currentRole))
                  break;
               //Selecting another goal
               goal = selectingGoalToAchieve();
               vPlansExecuted.clear();
            }
            //There is not any other goal to be achieved:
            //all goals where achived or
            //or the agent tried to achieve all goals.
            if (checkIfStopped(currentRole))
               break;
            continueExecution = checkIfWillContinue();
         }
         //The thread was stoped

         //The current role must be removed from the list of the roles being played
         Vector vRoles = getRolesBeingPlayed();
         vRoles.remove(currentRole);

         //The current role must be destroyed
         currentRole.destroy();

         //Verify if the agent is playing other roles
```

```
            vRoles = getRolesBeingPlayed();
            if (vRoles == null)
               //If the agent is not playing other roles, the agent must be destroyed
               destroy();
         }
      }
   }
   ...
}
```

Figure 24. The method run() of agents

Goal Selection

An agent chooses the goal that it will try to achieve according to the strategies that it defines. An agent can use several strategies to select a goal. Different strategies can be defined by different applications.

A goal is selected from the set of goals of the agent itself and from the set of goals defined by the roles that the agent is executing. In [6], the authors describe three types of role enactment by the agent: selfish enactment (agent gives priority to its own goals), social enactment (agent gives priority to the goals of the role) and maximally social enactment (agent only uses the goals of the role). These three types of role enactment use three different strategies to select the goal to be achieved.

A goal strategy selects goals based on priorities associated with the agent and role goals. For instance, the priorities can be calculated based on the agent mental state and on the messages sent and received by the agent. A goal is a high priory goal if, for example, the agent receives a message that influences the execution of the goal.

Since the beliefs store the memories of an agent, the agent beliefs can store the goals that the agent has tried to achieve. Such information can also influence the selection of the next goal. For example, a goal that has been previously selected but has not been achieved receives low priority. The agent can retry to achieve such goal later.

Since agents are goal-oriented entities, an agent must stop its execution when it achieves all its goals and all the goals of the roles that it is playing. The reasons for an agent to not achieve a goal and to stop its execution will not be described in this paper.

Plan Selection

After selecting a goal, an agent needs to execute a plan in order to achieve the goal. From its set of plans an agent must select one according to the goal that it wants to achieve. The agent can use different strategies in order to select a plan. The plan strategy adopted by each agent is determined by the application.

An agent can have no plan associated with the goal that it wants to achieve, can have only one plan or can have several plans. If there is no plan associated with the goal, the agent will need to choose another goal. Although such a goal has high priority, it cannot be achieved since the agent has no plan associated with the goal. The information about the inexistence of such a plan may influence the selection of the next goal.

In the case there are several plans associated with the goal two different strategies can be used. One strategy selects one plan and the other selects several plans to be executed in parallel. In case the plan

strategy has chosen only one plan and the agent could not achieve the goal by executing the plan, another plan associated with the same goal can be selected. A plan strategy can access the beliefs of the agent in order to verify which plans have been executed. If all plans associated with the goal have been executed, the agent should select another goal. In case numerous plans are being executed in parallel, when a plan achieves the goal the agent can stop executing the other plans.

Plan Execution

The execution of a plan involves the execution of its actions. Each plan defines the sequence of the actions to be executed and the conditions that must be satisfied in order to trigger the actions. Such conditions can be defined based on the pre-conditions and post-conditions of actions. The pre-conditions defined by the action must be satisfied before the execution of an action. After the execution of an action, its post-condition must be guaranteed.

The set of plans and actions of an agent depends on the application description. Moreover, what an agent will do while executing its plans and actions will also depend on the application. During the execution of its plans and actions, an agent can, for instance, change its role, send and receive messages, call object methods, migrate to another environment and change its mental states, i.e., change the set of goals, beliefs, plans and actions.

5.5.3 The Frozen State

An agent is frozen when all its threads have been suspended, i.e., when all its roles have been deactivated. The agent in the *frozen* state exists but is not executing. Since an agent has one thread associated with each one of its roles, the *frozen* state of an agent is a consequence of the deactivation of all its roles. All the roles of the agent are in the *inactive* state when the agent is in the *frozen* state. Since the computational model for the *inactive* state of roles has already been described in Section 5.4.3, there is no need for describing the computational model of the *frozen* state of agents.

5.5.4 The Death State

When an agent is destroyed, all the roles that it is playing also are destroyed. The threads associated with the roles should be stopped. In addition, the environment that is the habitat of the agent must be informed about the destruction of the agent. When an agent is destroyed, its record in the environment must also be destroyed. The method *destroy()* of an agent, illustrated in Figure 25, stops all its threads and cancels the record of the agent in the environment.

```
public abstract class Agent extends Thread
{
   ...
   public void destroy()
   {
      //Stoping all threads
      Vector vRoles = getRolesBeingPlayed();
      Enumeration enumvRoles = vRoles.elements();
      while (enumvRoles.hasMoreElements()) {
         AgentRole roleAux = (AgentRole)enumvRoles.nextElement();
         roleAux.stopThread();
      }
```

```
      //Canceling the register
      Environment env = getEnvironment();
      env.cancelObjectRegister(this);
   }
   ...
}
```

Figure 25. The destruction of an agent

When it is destroyed, an agent should liberate the resources it was using and, depending on the application, it may need to inform another agent about its destruction. The method *destroy()* implemented in the *Agent* class can be specialized by the application agents.

5.6 The Organization Computational Model

This section presents the computational models of the start, running and destroy states of main-organizations and sub-organizations. The differences between the lifecycle models of sub-organizations and main-organization occurs in the frozen and the migrating states. Both states are presented in the sub-organizations lifecycle model but are not presented in the main-organizations lifecycle model. Since the scope of the paper is not to present mobile entities, the computational model for the migrating state will not be presented.

5.6.1 The Start State

An organization instance must be immersed in an environment instance and cannot inhabit more than one environment at the same time. In order to immediately relate the organization and the environment, the constructor method of an organization must receive the environment that the new organization will inhabit as an input parameter. When creating an organization, the name of the organization should also be indicated since organizations interact by sending and receiving messages addressed to their names. Each name must be unique to the entire environment.

Almost all organizations play roles in other organizations. As previously stated, the only organization that does not play roles is the main-organization. When creating sub-organizations, besides defining the environment that it will inhabit, it is also necessary to indicate the role that it will play and the organization instance where the role will be played. It is not necessary to specify a role to create a main-organization since the main-organization does not play roles. The main-organization must be the first organization to be created since it does not depend on any other organization.

The creation of any organization is composed of two phases: the instantiation of an organization instance based on an organization class and the creation of a thread associated with the organization instance. In the case of sub-organizations, the thread is associated with the initial role that the organization will play. In Figure 26 there are examples of the creation of a main-organization instance based on the *General_Store* organization class and of a sub-organization instance based on the *Imported_Bookstore* organization class. Figure 26 omits the creation of the initial role of the sub-organization. The creation of roles is detailed in Section 5.4.1.

```
//Main-organization
```

```
MainOrganization mainOrg = new General_Store (env);
mainOrg.setOrganizationName("General_Store");

Thread mainOrgThread = new Thread(mainOrg, "General_Store");
mainOrgThread.start();

//Sub-organization
Organization subOrg = new Imported_Bookstore (env, mainOrg, orgRole);
subOrg.setOrganizationName("ImportedBookStore::Amazon");

Thread subOrgThread = new Thread(mainOrg,orgRole.getRoleName());
subOrgThread.start();
orgRole.setOrganization(subOrg);
```

Figure 26. The creation of a main-organization

5.6.2 The Running State

Such as agents, organizations are goal-oriented entities. Organizations select goals to be achieved, select plans according to the goals and execute these plans. The selection of goals and plans and the execution of plans characterize domain-dependent behavior as described in Section 5.2.2. In addition, both sub-organizations and agents play roles. Therefore, the method *run()* of every sub-organization is equivalent to the method *run()* of agents illustrated in Figure 24. This method is implemented in the abstract class *Organization*.

However, the method *run()* of main-organizations (see Figure 27) is not equivalent to the method *run()* of sub-organizations. Although main-organizations are goal-oriented entities, they do not play roles. Since the main-organization does not play roles and there is only one thread associated with it, it is automatically destroyed when this thread stops. In addition, the main-organization thread cannot be suspended. The feature related to suspension of a thread is linked to roles being played by entities. The method *run()* of main-organization is implemented in the abstract class *MainOrganization*.

```
public abstract class MainOrganization
{
   ...
   public void run() {
      Vector vPlansExecuted = new Vector();
      boolean continueExecution = true;
      //Cheking if thread was stopped
      while (continueExecution && !checkIfStopped())
      {
         //Selecting goal to be achieved
         Goal goal = selectingGoalToAchieve();

         //Cheking if thread was stopped
         while (goal != null)
         {
            //Selecting plan to be executed
            Plan plan = selectingPlan(vPlansExecuted, goal);

            //Cheking if thread was stopped
            while (plan != null)
            {
```

```
                //Executing plan
                executingPlan(plan);

                if (checkIfStopped())
                   break;

                vPlansExecuted.add(plan);
                if (!goal.getAchieved())
                   //Secting another plan
                   plan = selectingPlan(vPlansExecuted, goal);
                else {
                   //Goal achieved
                   //If goal type eguals maintaim, the agent must always try to achieve the goal
                   //but now it has low priority in order to let other goals to be achieved
                   if (goal.getGoalType().equals("maintain")) {
                      goal.setAchieved(false);
                      goal.setPriority(1);
                   }
                   plan = null;
                }
             }
             //The goal was achieved or
             //all plans associated with the goal were executed or
             //there is not any plan associated with the goal
             //Selecting another goal
             if (checkIfStopped())
                break;

             goal = selectingGoalToAchieve();
             vPlansExecuted.clear();
          }
          //There is not any other goal to be achieved:
          //all goals where achived or
          //or the agent tried to achieve all goals.
          if (checkIfStopped())
             break;
          continueExecution = checkIfWillContinue();
       }
       /* The thread was stoped */
       destroy();
      }
    ...
}
```

Figure 27. The method run() of main-organization

5.6.3 The Frozen State

Like an agent, an organization is frozen when all its threads have been suspended. The organization exists but is not executing. The *frozen* state of an organization is a consequence of the inactivate states of its roles. Since the computational model of the *inactive* state of roles has already been described in Section 5.4.3, there is no need for describing the computational model for the *frozen* state of organizations.

5.6.4 The Death State

In order to destroy a sub-organization, it is necessary to destroy the roles played by the sub-organization and the roles being played in the sub-organization. Note that the destruction of organizations and sub-organizations are different since there is no need for destroying the roles being played by main-organizations. As stated before, they do not play roles.

In the case of object roles, the roles can be destroyed without informing the object that is playing the roles. Objects are not concerned about the roles that they are playing. However, before destroying an agent role, the entity (agent or sub-organization) that is playing the role must be informed. The entity must stop playing the roles before it is destroyed. After destroying the roles, the record of the organization in the environment is canceled. Figure 28 depicts the destruction process of sub-organizations.

```
public abstract class Organization extends MainOrganization
{
   ...
   public void destroy()
   {
      //Destroying the roles being played
      Vector vRoles = getRolesBeingPlayed();
      Enumeration enumvRoles = vRoles.elements();
      while (enumvRoles.hasMoreElements()) {
         AgentRole roleAux = (AgentRole)enumvRoles.nextElement();
         vRoles.remove(roleAux);
         roleAux.destroy();
      }

      //Destroying the agent roles played in the organization
      //The thread of the entity playing the role must be stoped
      vRoles = getAgentRoles();
      enumvRoles = vRoles.elements();
      Agent agent;
      while (enumvRoles.hasMoreElements()) {
         AgentRole roleAux = (AgentRole)enumvRoles.nextElement();
         roleAux.stopBeingPlayed();
         vRoles.remove(roleAux);
      }

      //Destroying the object roles played in the organization
      //It is not necessary to destroy the object playing the role
      vRoles = getObjectRoles();
      enumvRoles = vRoles.elements();
      while (enumvRoles.hasMoreElements()) {
         ObjectRole objRoleAux = (ObjectRole)enumvRoles.nextElement();
         vRoles.remove(objRoleAux);
      }

      //Canceling the register
      Environment env = getEnvironment();
      env.cancelOrganizationRegister(this);
   }
   ...
}
```

Figure 28. The destruction of an organization

6 Using the Proposed Framework

The proposed framework needs to be extended in order to generate an application. This section describes the extensions applied to the framework according to a simple virtual marketplace example. An electronic commerce example was chosen since it is referred to in the literature [17][21][26] as an MAS benchmark.

6.1 The Virtual Marketplace Example

Virtual marketplaces are markets located in the Web where users buy and eventually sell items. We suppose there are several *virtual marketplaces* sharing the same characteristics. The *virtual marketplaces* being modeled in this Section have the following characteristics. Each *virtual marketplace* is composed of a *main-market* where users are able to negotiate *books*. The environment where the application is described is called *VirtualMarketplace*.

In the *main-market* (called *GeneralStore*), users buy the *books* available in the market. Agents called *UserAgent* represent the users in the market. Such agents play the role *Buyer* whenever they want to buy an item. The buyers look for a seller, created by the market to negotiate with the buyer, and send to the seller a description of the desired item. Agents called *StoreAgent* playing the role *Seller* represent the vendors of the market.

The seller is responsible for verifying if there is an item with the same characteristics in the *environment*. The *environment* stores all the items to be sold in these markets. If the item is found, the seller informs the buyer of its offer. The buyer can accept or reject the seller's offer. If the buyer does not accept the offer, the negotiation is ended. On the other hand, if the buyer accepts the offer, the seller sends the bill to the buyer. Next, the buyer sends the payment to the seller. The object roles *Desire* and *Offer* represent the roles of *Book* when buyers desire a book and sellers make offers to buyers.

6.2 Implementing Agents

An application agent is implemented by extending the abstract class *Agent* and by implementing the abstract methods *selectingPlan()*, *executingPlan()*, *selectingGoalToAchieve()* and *checkIfWillContinue()*. It is unnecessary to implement the running method of agents since it is implemented in the abstract class. On the other hand, concrete plan and action classes must be created extending the abstract classes *Plan* and *Action* and implementing their methods called *execute()*.

The method *selectingPlan()* describes the algorithm used by the application agent to select a plan while the method *selectingGoalToAchieve()* is used by the application agent to select a goal. The method *executingPlan()* defines the algorithm used to execute the selected plan and the method *chekcIfWillContinue()* is used to check if the agent will continue to execute. This method is called after the agent has tried to achieve all its goals and has not yet achieved at least one of them. If the method returns *true* the agent will try again to achieve the goal it has not yet achieved.

Two different agents were implemented in the virtual marketplace: the user agent and the store agent, both extending the abstract class *Agent*. We will focus on implementing the *UserAgent* class in order to exemplify the implementation of the methods. The methods *selectingPlan()*, *executingPlan()* and

checkIfWillContinue() were implemented in a very simple way. On the other hand, the strategy used in the *selectingGoalToAchieve()* is a complex one.

The plan that is selected using the *selectingPlan()* method is the first plan of the agent list that achieves a given goal and that has not yet been executed. The *executingPlan()* method finds out the role associated with the current thread and calls the *execute()* method of the plan sending the role as a parameter. Each plan is executing in the context of the role. The *execute()* method of each plan calls the *execute()* method of its actions.

The *checkIfWillContinue()* method returns always true. It indicates that the agent does not stop trying to achieve its goals. Note that other implementations could be used. After the agent has tried several times to achieve the goals, it could give up and stop its execution.

The strategy used in the method *selectingGoalToAchieve()* considers the goals of the role[1] being played by the agent and the goals of the agent. The goals are selected according to their priorities. In this example the goals of the roles have high priorities. If the agent has tried to achieve all the goals of the role, it will, then, try to achieve its own goals. Since the goals of the roles that the agent is playing are associated with the goals of the agent, the agent can have no goal to be achieved. If there is still an agent goal to be achieved, the agent may try to achieve this goal. If the agent goal is related to another agent role, the agent selects another goal since the thread related to the role will try to achieve it. However, if the goal is not associated with any of its roles, the agent may try to play another role in order to achieve such goal[2].

6.3 Implementing Agent Roles

The implementation of an application agent role depends solely of the implementation of concrete protocols. Concrete protocols are implemented extending the abstract class *Protocol* and implementing the method *execute()*.The method *execute()* of the protocols must inform the message that can be sent and the ones that can be received.

In the virtual marketplace example, two agents negotiate while executing the *SimpleNegotiation* protocol. As stated before in Section 6.1, the seller sends the buyer an offer. The buyer can accept or reject the offer. If the buyer accepts the offer, the seller sends the bill and the buyer pays for the item. When the buyer receives the price of the item, the method *execute()* of the class *SimpleNegotiation* returns a message informing that the buyer can accept the price or reject it. When the buyer receives the bill, the method *execute()* returns a message indicating that the buyer may send the payment.

6.4 Implementing Organizations

Since organizations extend agents, an application organization is implemented by extending the abstract classes *MainOrganization* or *Organization* and also by implementing the abstract methods *selectingPlan(), executingPlan(), selectingGoalToAchieve()* and *checkIfWillContinue()*. In addition, concrete plans and actions classes must also be created extending the abstract classes *Plan* and *Action* and implementing the methods *execute()*.

---

[1] The role is related to the current thread executing the method *selectingGoalToAchieve()*.

[2] The agent goals are achieved by executing agent roles.

Since main-organizations do not play roles, the implementation of the methods *selectingGoalToAchieve()* and *executingPlan()* of the main-organization *GeneralStore* are different from the implementation of the respective methods of the *UserAgent*. The main-organization method *selectingGoalToAchieve()* selects the high priority organization goal. The method *executingPlan()* calls the plan execution without relating it to a role.

6.5 Implementing Environments

An application environment is implemented by extending the abstract class *Environment*. The application environment class must be implemented according to the application property. In the event the application environment is a passive element, it should be implemented as an object. The abstract class *Environment* should be extended by adding new methods to represent the behavior of the application environment and new attributes to store its state.

In the event the application environment is an active element, it should be implemented as an agent. Besides extending the abstract class *Environment*, new classes must be created to represent the goals, beliefs, plans and actions of the environment. The implementation of an active environment can be described based on the agent module defined in Section 3.1.

A passive environment was created when implementing the virtual marketplace example. The class *VirtualMarketplace* environment was created by extending the abstract class *Environment*. The methods *increaseStock()* and *decreaseStock()*also were created to implement an inventory being increased and decreased, respectively.

6.6 Implementing Objects and Object Roles

The implementation of an object really is quite simple. It is not necessary to extend any class. The new class that represents the object can be implemented only according to the application specification. On the other hand, to implement object roles it is necessary to extend the abstract class *ObjectRole* and implement methods and attributes according to the characteristics of new object role class. Figure 29 illustrates some application classes that represent the entities of the application *Virtual Marketplace* extending the abstract classes of our proposed framework.

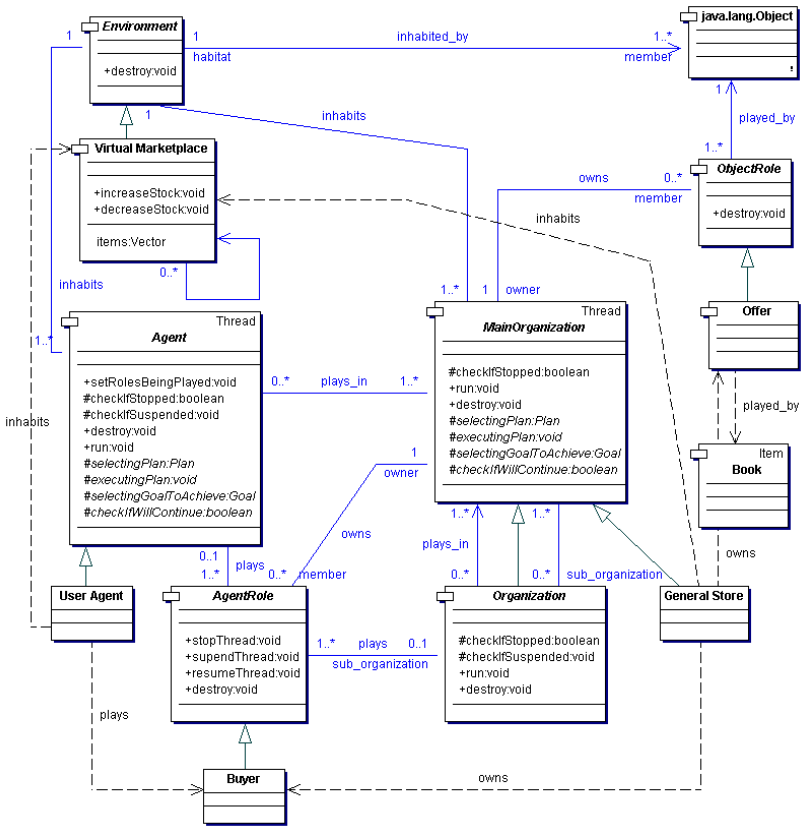

Figure 29. Some application classes extending the framework

7 Related Work

There are several frameworks (or platforms) for building agents and multi-agents systems published in the literature [1][2][3][5][14][19]. We have analyzed several platforms (Jadex, FIPAOS [14], ZEUS [5], Kaos [2] and Desire [3]) in order to find out if any of them support the implementation of agent societies. The evaluation focused on investigating if these platforms support the implementation of agent roles and organizations. In fact, only KAoS supports the implementation of agent roles. However, in KAoS, roles are restricted to conversations and are not associated with organizations.

While analyzing the different platforms, we felt the need for an approach that models not only agents but also other MAS entities such as organizations, roles and environments. In this context, we proposed the ASF framework to implement agent societies. Due to the characteristics of ASF, Jadex is the framework that is more related to our approach. Jadex is a Java based FIPA compliant agent environment that allows the development of goal-oriented agents by following the BDI model. Jadex provides a framework and a set of development tools to simplify the creation and testing of agents. In Jadex, the BDI model, based on the mental attitudes, was adopted and transformed into an execution model for software agents, based on the notion of belief, goals, and plans. Jadex incorporates this model into the JADE [1]

agents, by introducing beliefs, goals and plans as first class objects that can be created and manipulated inside the agent.

The definition of goals and beliefs are virtually the same in both models. Goals and beliefs are represented as objects with several attributes. Besides, in Jadex and ASF, the agent developer can define the strategy used when selecting a goal. Jadex has no built-in generic deliberation mechanism to select goals. On the other hand, ASF provides a default mechanism for selecting goals.

The structure of plans defined in Jadex and in ASF is similar. Plans are represented by procedures that define the execution of the plans and are related to goals that may be achieved after their execution. The differences between the plans defined in Jadex and in ASF are related to the selection of the plans and their execution. In Jadex, plans are selected according to their filters. Agents do not select the plans that are automatically selected by the system. When a filter is triggered, the plan is instantiated and executed. In ASF, agents define algorithms to select their plans. A plan only executes when it is selected by an agent (or an organization).

The goal of Jadex is to simplify the development of agents. Jadex is especially useful for developing simple systems with only interacting agents. The instantiating process is simplified by using an ADF (Agent Definition File) to define the agents' attributes and to supply the plans' implementations. Differently, ASF can be used to model complex systems where agents can play different roles in organizations. The complex systems characterize agent societies. The instantiation process of the ASF framework is more complex than the Jadex platform since it includes the instantiation of several entities and the implementation of plans, actions and algorithms to select the plans and the goals.

8 Conclusion and Future Work

In this paper we propose an object-oriented framework for implementing agent societies. The framework supports the implementation of agents, roles, organizations and environments as first-order abstractions. This approaches allows (i) the definition of roles, organizations and environments, (ii) the implementation of agents and sub-organizations playing roles, changing and canceling their roles, (iii) the association among roles and organization that are the owners of the roles, and (iv) the movement of agents and sub-organizations from an organization to another in order to play different roles.

The framework is composed of several object-oriented modules where each module represents an MAS entity in terms of Java classes and relationships. The set of object-oriented modules define the structural aspects of the framework. In order to define the dynamic aspects of the framework, i.e., the behavior of the MAS entities, their lifecycle models were defined. For each MAS entity, a detailed lifecycle model is presented by describing the execution states and the events that cause the (allowed) transition of the entity from one state to another. Based on the graphic representation of the lifecycle models, the computational models of each entity were explored. The computational models define the behavior of the entities associated with each state described in the lifecycle models. As a consequence, both the structural and the dynamic aspects of the ASF framework are covered in the paper.

The main difference between our proposed frameworks and other frameworks, architectures and platforms presented in the literature is the ability to implement agent societies. None of the analyzed

approaches explicitly models agent roles and organizations. Therefore, it is not possible to represent agents entering in different organizations to play different roles when using these approaches. On the other hand, since our proposed framework deals with diverse entities, it is not a simple task to extend the framework in order to create an application. It is necessary to understand the relationships between the different entities in order to understand the framework. Moreover, several classes must be extended to implement not only agents but also roles, organizations and environments. Other frameworks that only represent agents are simpler to understand and extend. Thus, other approaches should be used when there is no need for implementing agent societies.

In order to improve the ASF framework, it is important to turn the framework into a FIPA [13] compliant one. The scope of FIPA architecture includes: a model of services and discovery of services available to agents and other services, message transport interoperability, supporting various form of ACL representations, supporting various forms of content language, and supporting multiple directory services representations. In addition, there is also a need for extending the framework to support mobile and distributed agents. These features may allow the migration of agents (and sub-organizations) from one environment to another and the communication between agents (or sub-organizations) in different environments.

Affiliation of the authors

* PUC-Rio, Computer Science Department, Rua Marques de São Vicente, 225 – Ed. Pe Leonel Franca, 13o andar, 22453-900, Rio de Janeiro, RJ, Brazil, {viviane,mariela,lucena}@inf.puc-rio.br

Acknowledgments

This work is partially supported by CNPq/Brazil under the project “ESSMA”, number 5520681/2002-0.

References

1. F. Bellifemine, A. Poggi, G. Rimassa. “Developing multi-agent systems with a FIPA-compliant agent framework.” In: Software - Practice and Experience, 2001 no. 31, pp. 103-128.

2. J. Bradshaw, S. Dutfield, P. Benoit and J. Woolley. “KAoS: Toward an Industrial-Strength Open Agent Architecture.” In: Software Agents, J.M. Bradshaw (Ed.), Menlo Park, Calif., AAAI Press, 1997, pages 375-418.

3. F. Brazier, B. Dunin Keplicz, N. Jennings, J. Treur. “DESIRE: Modeling Multi-Agent Systems in a Compositional Formal Framework.” In: International Journal of Cooperative Information Systems Vol: 6, 1997.

4. G. Caire, F. Chainho, R. Evans. “Agent-oriented analysis using Message/UML.” In: Agent-Oriented Software Engineering, M. Wooldridge, G. Weiss, P. Ciancarini (Eds). Second International Workshop, AOSE 2001, LNCS 2222 Stringer, Canada, p. 119-135. 2002.

5. J. Collis and D. Ndumu. “Zeus Technical Manual.” Intelligent Systems Research Group, BT Labs. British Telecommunications. 1999.

6. M. Dastani, V. Dignum, F. Dignum. “Role-Assignment in Open Agent Societies.” In Proceedings of the Second International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2003): pp 489-496. Melbourne, Australia, ACM Press, 2003.

7. R. Depke, R. Heckel, J. Kuster. “Roles in Agent-Oriented Modeling.” International Journal of Software Engineering and Knowledge Engineering, v.11, n.3, p. 281-302. 2001.

8. M. Fayad, D. Schmidt. “Building Application Frameworks: Object-Oriented Foundations of Framework Design”, Wiley Computer Publishing, 1999.

9. J. Ferber, O. Gutknecht. “A meta-model for the analysis and design of organizations in multi-gents systems” In: Proceedings of the International Conference on Multi-Agent Systems, Demazeau, Y., Ed., IEEE Press. , pp. 128-135, 1998.

10. J. Ferber, O. Gutknecht, F. Michael. “From Agents to Organizations: an Organizational View of Multi-Agent Systems.” In: Proceeding of the Fourth International Workshop on Agent-Oriented Software Engineering (AOSE), Australia, 2003.

11. J. Ferber, O. Gutknecht, C. Jonker, J. Mueller, J. Treur. “Organization Models and Behavioral Requirements Specification for Multi-Agent Systems.” In: Proceedings of the ECAI 2000 Workshop on Modeling Artificial Societies and Hybrid Organizations, 2000.

12. FIPA; Foundation of Intelligent Physical Agent. Available at URL http://www.fipa.org/, 2004.

13. FIPA Abstract Architecture Specifications, version L-2002, FIPA. Available at: <http://www.fipa.org/repository/architecturespecs.html>. Accessed in: February 14th 2004.

14. FIPAOS. FIPA Agent Platform Open-source, The Foundation for Intelligent Physical Agents. 2004. Available at: <http://fipaos.sourceforge.net>. Accessed in: August 12th 2004.

15. M. Georgeff, B. Pell, M. Pollack, M Tambe, M. Wooldridge. “The Belief-Desire-Intention model of agency.” Proceedings of Agents, Theories, Architectures and Languages (ATAL). 1999.

16. M. Hannoun, O. Boissier, J. Sichman, C. Sayettat. “MOISE: An organizational model for multi-agent systems” In: Proc. International Joint Conference 7th. Ibero-American Conference on Artificial Intelligence (IBERAMIA'00) and 15th. Brazilian Symposium on Artificial Intelligence (SBIA'00), Atibaia, Brasil.

17. M. He, N. Jennings, H. Leung. “On agent-mediated electronic commerce.” In: IEEE Transaction on Knowledge and Data Engineering, v.15, n.4, p.985-1003. 2003.

18. I. Ishida, L. Gasser, M. Yokoo. “Organization self design of production systems.” In: IEEE Transaction on Knowledge and Data Engineering, v.4, n.2, p.123-134. 1992.

19. Jadex – BDI Agent System, University of Hamburg, Germany. Availabel at: <http://vsis-www.informatik.uni-hamburg.de/>. Accessed in: August 12th 2004.

20. M. Jang, G. Agha. “On Efficient Communication and Service Agent Discovery in Multi-agent Systems,” Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS '04), pp. 27-33, May 24-25, Edinburgh, Scotland, 2004.

21. N. Jennings, M. Wooldridge. “Applications of Intelligent Agents.” In: N. Jennings, M. Wooldridge (Eds.), Agent Technology: Foundations, Applications, and Markets, pp. 3-28, 1998.

22. N. Jennings. “On Agent-Based Software Engineering.” Artificial intelligence, vol. 117, no 2, p.277-96. Elsevier, March 2000.

23. C. Lemaître. “Multi-Agent Organization Approach.” In: P. Ciancarini, M. Wooldridge (Eds.), Agent-Oriented Software Engineering. Springer-Verlag, 2001.

24. E. Letier, A. Lamsweerde. “Agent-based Tactics for Goal-Oriented Requirements Elaboration.” In: Proceedings of International Conference on Software Engineering (ICSE02), Florida, 2002.

25. J. Lind. “MASSIVE: Software Engineering for Multi-agent Systems.” PhD Dissertation, Universität des Saarlandes, Saarbrücken, Germany, 2000.

26. A. Lomuscio, M. Wooldridge, N. Jennings. “A classification scheme for negotiation in electronic commerce.” In: International Journal of Group Decision and Negotiation, v.12, n.1, p.31-56. 2003.

27. D. Mobach, B. Overeinder, N. Wijngaards, F. Brazier. “Managing agent life cycles in open distributed systems.” In: Proceedings of the 2003 ACM symposium on Applied computing, pp: 61 - 65, 2003.

28. J. Odell, H. Parunak, M. Fleisher. “The Role of Roles in Designing Effective Agent Organizations.” In: A. Garcia, C. Lucena, F. Zamboneli, A. Omicini, J. Castro (Eds.) Software Engineering for Large-Scale Multi-Agent Systems. LNCS 2603, Berlin: Springer, 2003.

29. OMG: Object Management Group. Available at: <http://www.omg.org>. Accessed in: February 14th 2004.

30. Parunak, H. and Odell, J. (2002), "Representing social structures in UML," In Agent-Oriented Software Engineering II, Wooldridge, M., Weiss, G. and Ciancarini, P., Eds., LNCS 2222, Springer-Verlag, Berlin, pp. 1-16.

31. A. Pokahr, L. Braubach, W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for Jade Agents. In Exp in Search of Innovation, vol. 3, n. 3., September 2003. Available at: <http://exp.telecomitalialab.com>. Accessed in: August 12th 2004.

32. A. Rao, P. Georgeff. "Modeling Rational Agents within a BDI-Architecture". In Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, pp. 473-484, 1991.

33. A. Rao, P. Georgeff. "An Abstract Architecture for Rational Agents." In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, pp. 439-449, 1992.

34. Y. Shoham. Agent-Oriented Programming. Artificial Intelligence, v.60, 1993.

35. V. Silva, A. Garcia, A. Brandao, C. Chavez, C. Lucena, P. Alencar. "Taming Agents and Objects in Software Engineering." In: A. Garcia, C. Lucena, F. Zambonelli, A. Omicini, J. Castro (Eds.) Software Engineering for Large-Scale Multi-Agent Systems. LNCS 2603, Berlin: Springer, 2003.

36. V. Silva, C. Lucena. "From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language," In: K. Sycara, M. Wooldridge (Edts.), Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers, 2004.

37. G. Wagner. "Agent-Oriented Analysis and Design of Organizational Information Systems" In: Proceedings of Fourth IEEE International Baltic Workshop on Databases and Information Systems, Vilnius, Lithuania, 2000.

38. M. Wooldridge, P. Ciancarini. "Agent-Oriented Software Engineering: the State of the Art." In: P. Ciancarini, M. Wooldridge. (Eds.) Agent-Oriented Software Engineering, LNCS 1957, Berlin: Springer, p. 1-28. 2001.

39. M. Wooldridge, N. Jennings, D. Kinny. "The Gaia methodology for agent-oriented analysis and design." Journal of Autonomous Agents and Multi-Agent Systems, 3, pp. 285-312, 2000.

40. L. Yu, B. Schmid. "A Conceptual Framework for Agent-Oriented and Role-Based Work on Modeling." In: G. Wagner, E. Yu (Eds.). Proceedings of the 1st International Workshop on Agent-Oriented Information Systems, 1999.

41. F. Zambonelli, N. Jennings, M. Wooldridge. "Organizational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems." In: International Journal of Software Engineering and Knowledge Engineering, 2001.

42. F. Zambonelli, N. Jennings, M. Wooldridge. "Organizational abstractions for the analysis and design of multi-agent systems." In: P. Ciancariani, M. Wooldridge (Eds.) Agent-Oriented Software Engineering, LNCS 1957, Berlin: Springer, p. 127-141. 2001.