ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal
Regular Issue, Vol. 7 N. 3 (2018), 63-98
eISSN: 2255-2863
DOI: http://dx.doi.org/10.14201/ADCAIJ2018736398

JAMDER: JADE to MULTI-Agent Systems Development Resource

Yrleyjander S. Lopes[a], Mariela I. Cortés[a],
Enyo José Tavares Gonçalves[b] and Robson Oliveira[a]

[a] Universidade Estadual do Ceará, Fortaleza, CE – Brazil
[b] Universidade Federal do Ceará, Quixadá, CE – Brazil
yrleyjander@gmail.com, mariela@larces.uece.br, enyo@ufc.br, rob.oliveira89@gmail.com

KEYWORD

Multi-Agent Systems; Framework; JADE; Model-Driven Architecture

ABSTRACT

The semantic gap is distinguished by the difference between two descriptions generated using different representations. This difference has a negative impact on the developer productivity and probably, the quality of the written code. In software development context, the coding phase aims at coding the system consistent with the detailed project developed with a group of designed models. This paper presents an endeavor to consolidate different agent type definitions and implementation concepts for Multi-Agent Systems (MAS) involving the adaptation of the JADE framework regarding the theoretical concepts in MAS. Additionally, it contains a standardization of code generation. The main benefit of the proposed extension is to include the agent internal architectures, entities and relationships in an implementation framework and increase the productivity by code generation, ensuring the consistency between design and code. The applicability of the extension is illustrated by developing a multi-agent system for Moodle.

1. Introduction

A Multi-Agent System (MAS) involves a rich variety of entities such as organizations, environments, agent and object roles, that each one has relationships and associated behaviors (Freire *et al.*, 2013). In particular, a single MAS can be composed of agents with different architectures, where each architecture involves some specific behaviors and attributes associated to the agent. In this context, the high complexity to develop MAS requires a set of methods and tools to assist in the construction of this system considering the particularity of each entity. Russell and Norvig (2002), defined initially four architectures for agents: simple reflex agents, model-based reflex agents, goal-based agents and utility-based agents. BDI (belief-desire-intention) is another known architecture which involves the concept of one agent needs to store the steps (beliefs), according to its goals (desires) and make plans (intentions).

On another hand, the agent-oriented development paradigm requires adequate techniques to explore its benefits and features to support the construction and maintenance of this type of software. As it is the case with any new software engineering paradigm, the successful and widespread deployment of MASs requires modelling languages that explore the use of agent-related abstractions and promote the traceability from the

design models to code. To reduce the risk when adopting new technology, it is convenient to present it as an incremental extension of known and trusted methods and to provide explicit engineering tools that support industry-accepted methods of technology deployment (Castro *et al.*, 2006). The transition between the modelling and implementation stages is usually accomplished in a non-systematic way, and frequently, aspects considered in the modelling have no counterpart in the implementation. In this context, the existence of a conceptual metamodel is crucial since represents the ontology of the entities in a MAS. The framework TAO+ (Taming Agents and Objects) provides an ontology that covers the fundamentals of software engineering based on agents and objects and supports the development of MAS in large-scale (Freire, 2013). TAO+ presents the definition of each abstraction, as a concept of its ontology, and establishes the relationships between them.

On another hand, the use of methods and tools to support the development activities allow an increase in productivity and, in general, ensures the correctness of the generated artefacts (produced code). More specifically, frameworks and platforms provide an environment for implementation. Barely, a framework accomplishes all characteristics for different types of agents and other entities to increase the complexity of agents in an environment.

The Java Agent Development[1] framework (JADE), widely used in the development of MAS, has the following characteristics: (i) it has a platform in the Java language, (ii) supports distributed systems and (iii) is free. However, the agent implementation in JADE is limited mainly to agents, behaviors and messages. JADE also has a platform, which contains the necessary environment for the agent lifecycle and containers where agents reside.

This paper presents an extension of the JADE framework to provide the adequate infrastructure focused on implementation of agents according to the settings covered in (Freire, 2013). Additionally, an approach based on model-driven architecture (MDA[2]) (Mellor, 2004) to support the code generation is presented in order to promote a fast and consistent development for different types of agent architecture. The paper is organized as follows: in Section 2, related works are presented. Section 3 presents the theoretical referential about the entities in the TAO+ metamodel, JADE framework and the MDA approach. Section 4 presents the extension of JADE and the code generation mechanism. Section 5 a case study is illustrated. Finally, Section 6 presents the conclusions and suggestions for future works.

2. Related Works

A set of frameworks and modelling languages have been developed in order to support the implementation of MAS. In general, these tools are associated with an object-oriented programming language, in order to compose entities and provide an environment for their execution.

2.1. MAS Modeling Languages and Tools

Jadex (JADE XML) (Braubach, 2003) (Pokahr *et al.*, 2005) is an agent-oriented reasoning mechanism that the agents are written in XML and Java programming language. One of the main aspects of Jadex is not to present a new programming language. It uses the already existent object-oriented environment. In adding, the utilization of XML language in the definition of the agent features increases the development complexity. On another hand, a platform update entails that code created using the previous version can be incompatible with the new one.

JaCaMo (Boissier *et al.*, 2011) is a framework for Multi-Agent Programming builds upon three existing platforms, Jason for programming autonomous agents, *Moise* for programming organizations, and CArtAgO for programming shared environments. A software system programmed in JaCaMo is defined by the organization of autonomous BDI agents based on concepts as roles, groups, mission and schemes; autonomous agents are implemented in Jason; working in shared distributed artefact-based environments. The JaCaMo meta-model

1 Java Agent Development. http://jade.tilab.com/
2 Object Management Group. http://www.omg.org.

defines dependencies, connections and conceptual mappings and synergies between all the different abstractions available in the meta-models associated to each level of abstraction.

Opal (Purvis *et al.*, 2002) is a platform for the development of MAS-based abstract architecture developed by FIPA[3]. It provides a framework formed by specialist officers who provide essential services for the development MAS, with, for example, registration and search of instantiated agents in the system. An agent is made of a combination of several micro-agents. These micro-agents are lower levels in a MAS.

JADE (Castro *et al.*, 2006) is widely used in various market sectors such as telecommunications, JADE provides a robust and mature infrastructure, and provides many features that are needed for implementing multi-agent systems, which includes yellow pages, message exchange and support for ontologies. JADE implements a task-oriented model, in which agents have a set of behaviors. No cognitive abilities, such as a reasoning cycle, are provided for agents.

Santos (2006) provides a metamodel that represents the concepts that define an agent system, as well as the relationships between them. The code generation uses a prototype tool developed in Velocity[4] and the modeling information is represented by XML (Extensible Markup Language) file that contains the agent structures. The author uses a prototype of a modeling tool, called MAS Modeler (Santos, 2006), where the textual information is filled through step-by-step screens. After that, this structure is stored in an XML file and can be used by the Velocity template for code generation in the Semanticore framework (Blois and Lucena, 2004), but only for agents.

TAOM4E (Morandini et at., 2011) is an agent-oriented modeling environment and supports a model-driven, an agent-oriented software development. It has been designed taking into account MDA recommendations. The TAOM4E architecture allows for a flexible integration of different tools. The tool is a plug-in for ECLIPSE Platform, but it allows the code generation only to BDI agents.

The Prometheus Design Tool (PDT) (Padgham and Winikoff, 2004) is a graphical tool that is used to design a MAS following the Prometheus Methodology. PDT is integrated into the Eclipse platform, enabling the users to accomplish the full development life-cycle of an agent-oriented application in one IDE. Similarly to TAOM4E, the code generation is targeted only to BDI agents.

Among all the frameworks highlighted in this section, only JADEX and JADE have an IDE (Integrated Development Environment) with free support for code debugging. Considering that any of them would need the extension to best suit the modelling language, then JADE showed the fittest since is not necessary the utilization of another language for redefining the agent structure as JADEX (Nunes, 2008).

2.2. MDD Approaches for MAS Development

Several methodological approach and frameworks to the model-driven design of multiagent systems have already been proposed (Gómez-Sanz *et al.*, 2010)(Ficher *et al.*, 2012). In this context, model transformation techniques are one of the key aspects of the model-driven development approach. In (Gascuena *et al.*, 2014) model-to-model and model-to-text transformations are presented to automate the development process to generate ICARO code from the INGENIAS model.

Studies on Domain-Specific Languages (DSLs) and Domain-Specific Modeling Languages (DSMLs) for agents have emerged in the context of MAS development (Challenger, 2014). A DSML for MAS is presented in (Gascuena *et al.*, 2012), where the abstract and the concrete syntax was presented using the Meta-object Facility (MOF) and GMF, respectively. Finally, the code generation for the JACK[5] agent platform. However, the developed modeling language is based on the metamodel (Challenger *et al.*, 2014) of one of the specific MAS methodologies called Prometheus (Padgham and Winikoff, 2004). A similar study was done in Fuentes-Fernandez *et al.* (2010) which is based on INGENIAS methodology (Pavon *et al.*, 2005) for MAS development.

More specifically, DSM and DSML may provide the required abstraction and support a more fruitful methodology for the development of the development of Semantic Web enabled MASs. In this domain, autonomous

3 http://www.fipa.org/.
4 Velocity. Apache Velocity Project. http://velocity.apache.org).
5 http://www.agent-software.com.au/products/jack/.

agents can evaluate semantic data and collaborate with semantically-defined entities of the Semantic Web, like Semantic Web Services. Challenger *et al.* (2016) presents a new DSL-base methodology for the development of MAS with semantic web. This study presented a DSL called Semantic web-Enabled Agent Language (SEA_L) (Demirkol, 2013) (Getir *et al.*, 2014) and includes code generation and constraints to check the programs regarding the implementation of SEA_L agents.

3. Theoretical Referential

This section describes the concepts related to the TAO+ metamodel including the entities needed for MAS and Jade definitions.

3.1. MAS Entities in Taming Agents and Objects (TAO+)

The framework TAO+ provides an ontology that covers the fundamentals of Software Engineering based on agents and objects and supports the development of MAS in large-scale (Freire, 2013). TAO+ presents the definition of each abstraction, as a concept of its ontology, and establishes the relationships between them (Figure 1). Gonçalves *et al.* (Gonçalves *et al.*, 2010) (Gonçalves *et al.*, 2015) describe the entities needed for a MAS as such as the internal architectures of agents as follow:

- Object: It is a passive element that has state and behavior and can be related to other elements.
- Agent: It is an autonomous, adaptive and interactive element. Its basic behavioral feature is action and Structural/Mental and behavioral aspects depends of its internal architecture:
- The Simple reflex Agent has not structural/mental features and it has Perception and Action (oriented by Condition-Action Rules) as behavioral features;
    - The Model-based reflex Agent has the structural/mental feature Belief and it has Perception, Next-function and Action (oriented by Condition-Action Rules) as behavioral features;
    - The MAS-ML agent has goal and belief as structural/mental features and it has Plan and Action (oriented by the Plan chosen according to the goal);
    - Goal-based Agent has goal and belief as structural/mental features and it has Perception, Next Function, Goal-formulation Function, Problem-formulation function, Planning and Action as behavioral features;
    - The Utility-based Agent has goal and beliefs as structural/mental features and it has Perception, Next Function, Goal-formulation Function, Problem-formulation function, Utility Function Planning and Action as behavioral features.

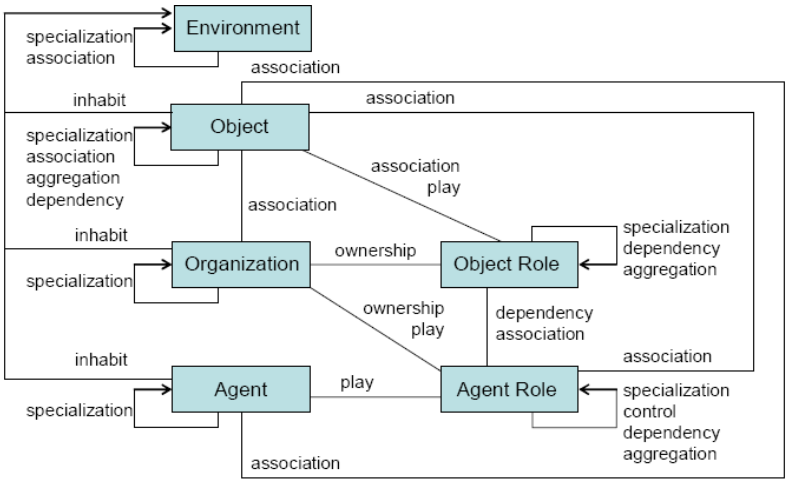

Figure 1: Entities and Relationships on TAO (Gonçalves et al., 2010)

- Organization: It is an element that groups agents, which play roles and have common goals. It may restrict the behavior of their agents and their sub-organizations through the concept of axiom, which define the actions that must be performed.
- Object Role: It is an element that guides and restricts the behavior of an object in the organization. An object role can add information, behavior and relationships to the object that plays the role.
- Agent Role: It is an element that guides and restricts the behavior of an agent in the organization. An agent role defines (i) duties that define an action that must be performed by an agent, (ii) rights that define an action that can be performed by an agent and (iii) protocol that defines an interaction with the other elements.
- Environment: It is an element that represents the habitat for agents, objects and organizations. An environment can be heterogeneous, dynamic, open and distributed.

Silva *et al.* (2007) define the following relationships in TAO+: Inhabit, Ownership, Play, Specialization/ Inheritance, Control, Dependency, Association and Aggregation /Composition. The relationships will not be described here since the framework and code generation proposed in this work approaches implicitly from the diagram projected and all validations of these relationships are also now covered by the modelling tool.

3.2. JADE Agent Definitions

JADE framework includes classes and services to facilitate the MAS coding phase. Among the available features, some are listed below: publishing services through yellow pages, location service through white pages, support to ontologies and communication of protocols compatible with FIPA[6] (*Foundations of Intelligent Physical Agents*) standard. Indeed, JADE is an object-oriented framework written in Java.

The architecture in JADE contains containers where the agents reside and the system can be distributed on different platforms. Each agent is registered in the service AMS (*Agent Management System*) provided by JADE that guarantees the oneness of the agents. To find other agents, another service is provided, DF (*Directory Facilitator*) and it works like a *Yellow Pages* service.

JADE agent classes inherit (in) directly from jade.core.Agent (Bellifemine *et al.*, 2007), that represents a common basic class for the agent definition by the user. Therefore, from the programmer standpoint, a JADE agent is simply an instance of Java class that inherits from Agent class (Castro *et al.* 2006). This implies, in case of feature inheritance, in order to support the basic interactions with the agent platform (registration,

6 http://www.fipa.org.

remote configuration and management). Initially, a basic set of methods must be coded to run the basic agent's behaviors, such as methods to send or receive messages, related to the utilization of standard communication protocols, registration of several domains, among others.

The JADE life cycle for an agent is defined according to FIPA. The first step is the execution of the agent's constructor, followed by the assignment of an identifier for the agent to be entered into the system. The *setup()* method runs from the time that the agent starts its activities. A agent behavior is designed by the overridden of the *setup()*.

The kinds of behaviors in JADE represent the actions of agents (Castro *et al.* 2006). The main class related to behavior is the *jade.core.Behavior*, whose subclasses implement specific purposes, such as behavior composition or task duration. This class defines two basic methods: *action()* and *done()*. The *action()* method contains the code of the behavior to be executed by the agent. After its execution, the *done()* method is automatically executed to check if the behavior has been finalized or not. The class must maintain the state of the execution, so the *done()* method returns a logical value false (equivalent to the false literal in Java) as long as necessary to perform the *action()* method. Otherwise, it must return a logical value true (equivalent to the true literal in Java).

3.3. Model-Driven Architecture

The model-driven architecture (MDA) is presented as an appropriate approach to assist in the code generation because the code can be generated several times without compromising the model. The OMG[7] defines some standardization in this process that is not necessarily associated with a specific platform. Thus, the concepts may be applied to different modeling and implementation languages.

The transformation process takes place through the steps proposed by OMG. The generated artifacts in each of these steps are independent of a specific tool to be used, for example, when the same application can be implemented in different languages. The concepts of each of these steps are described as follows (Beydeda *et al.*, 2005):

- CIM (Computation Independent Model): rules for functional requirements;
- PIM (Platform-Independent Model): relations between properties and their entity relationships.
- PSM (Platform-Specific Model): definition of how the system will work on a specific platform.
- PDM (Platform Description Model): definition of how the PIM to PSM will work.

Model transformation is the process of converting one model to another model of the same system and represents the central point in the model-driven development. High-level models are transformed into low-level models. In this sense, the idea of generating one model from another in an automatic/semi-automatic manner intends to provide a simple and fast way of software development.

4. Jade Extension and Mapping for MAS

As explained in the previous section, besides agent concept, the conceptual framework TAO+ offers other aspects that can be involved in a MAS like object, organization, object role and agent role. Each entity specifies different structural properties and behaviors.

Considering the resources and features offered at the conceptual level, adjustments in JADE are required in order to turn the properties and behaviors to corresponding implementation components. The resultant JADE extension is called JAMDER[8] (JADE to MAS Development Resource).

7 Object Management Group. http://www.omg.org.
8 https://bitbucket.org/yrley/jamder/src.

4.1. Agents

Agents have their architecture definition and interact with an environment. Beside the BDI definition, there are the reactive and proactive agents defined by Russell and Norvig. Taking this knowledge, five different architectures or types of agents can be identified: (i) simple reflex agents, (ii) model-based reflex agents, (iii) goal-based agents with searching (planning), (iv) utility-based agents, and (v) goal-oriented agents with plans. On this assumption, the JADE Agent class must be extended to incorporate these five agent types (Figure 2).

Except for goal-based agents with plans, the other architectures have similar characteristics according to perception and functions used, but agents based on goals with planning and utility-based agents do not share the condition-action rules. Therefore, three hierarchies of agents at modeling level were established, (i) goal-based agent with plan, (ii) reflex agents (simple reflex agent and model-based reflex agent) and (iii) cognitive agents (goal-based agent with planning and utility-based agent). This classification is based on the structural and behavioral characteristics of the agents.

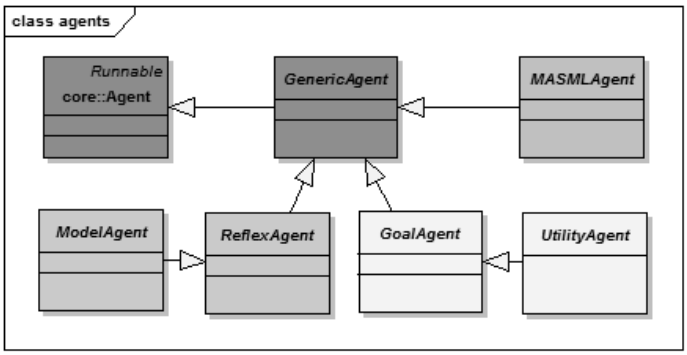

Figure 2: JAMDER Agent Hierarchy

The class *jamder.agents.GenericAgent*, which extends *jade.core.Agent*, was defined to represent these properties and other common attributes between the three branches. The common attributes are: (i) the list of agent roles, (ii) the list of organizations which the agent can be participating, (iii) the environment where it is, and (iv) the list of actions that it can perform. These lists are instances of *java.util.Hashtable* class. Except for the list of agent roles, the other lists are protected to ensure access only by its sub-classes. All lists with their five access methods are defined as following headers:

- getXXX(name) - returns the instance of the element name of the list;
- addXXX(name, XXX) - adds the element XXX in the list;
- removeXXX(name) - removes the element name from the list and returns the list;
- removeAllXXX() - removes all elements of the list;
- getAllXXX() - returns the list of elements XXX;

The agents need sensors to capture the perceptions; this concept is represented by the new class *jamder.behavioral.Sensor* which inherits from *jade.core.behaviors. TickerBehavior*. It is responsible for capturing perceptions from the environment, time by time. The period should be defined by the developer in *Sensor*. The perception is handled by the abstract method called *percept(perception)* of *GenericAgent*.

In JADE, the agent plays at least one agent role which contains the beliefs, goals and actions. By acquiring the role, the agent acquires the beliefs and goals of the role. The agent can only perform their actions if the same actions are defined by the agent role.

4.1.1. *Agent's Structural Characteristics*

JADE classes that can be used to represent the properties of agents are essential. So, it is needed to verify the concepts of each one, which is: beliefs (*Belief*), goals (*Goal*), plans (*Plan*), messages (*ACLMessage*) (FIPA), actions (*Action*) and condition (*Condition*).

In JADE were not found correlatives for beliefs and goals. Due to the similar structure between belief and goal (each one is composed of fields: name, type and value) and like these fields are the properties of each agent, except for simple reflex agent and model-based reflex agent, the *jamder.structural.Property* class was created and contains these fields, which are inherited by *jamder.structural.Belief* and *jamder.structural.Goal* representing the beliefs and goals, respectively.

Goals can be simple or composed, then, the respective classes' *jamder.structural.LeafGoal* and *jamder.structural.CompositeGoal* represent these features and inherit from *Goal*. Also, the *Goal* class contains a list of plans that are associated with a goal and the boolean attribute, *achieved*, indicates if the goal was achieved or not.

4.1.2. *Agent's Behavioral Characteristics*

Actions are the behavioral characteristics of agents (Weiss, 1999), and therefore should be implemented in association with the JADE class, *jade.core.behaviors.Behavior*. A class called *jamder.behavioral.Action* is created for this purpose as a subclass of *Behavior*. A list of actions indicates the actions which the agent can take. An action in the list is actually performed by calling the *addBehavior(action)* method.

The execution of actions only occurs if all its preconditions are attended. In contrast, post-conditions define the conditions that will be true when the execution of the action is completed successfully. In this sense, the *Action* class defines two attributes: the pre-conditions and post-conditions lists, both represented by a *jamder.behavioral.Condition*, defined as a subclass of *Property*. *Action* has a method called *execute()*, responsible for implementing the concrete actions for the *Action*. The pre-conditions and post-conditions also have five access methods, in a manner analogous to the list.

Each plan defines a set of actions and the execution order. The corresponding class in JADE is *SequentialBehavior*. Thus, the *jamder.behavioral.Plan* was created and inherits from *SequentialBehavior*. This class defines the associated goal and an action list, whose basic sorting in the plan is determined by the abstract method *execute()*. The agent's actions in the plan have to match with the actions in the agent role, i.e., actions must be defined in both, the agent and the agent role.

The agents communicate with each other through messages, which are stored in a queue. Asynchronously, the agent decides what to do when reading each message. According to Weiss (1999), a message is defined by a label that specifies the type of message, content, sender (agent responsible for sending the message) and the recipient. In JADE there is a corresponding class named *jade.lang.acl.ACLMessage* (JADE).

The basic class *jade.core.Agent* defines a list of received messages of *ACLMessage* type. Considering that the proposed agent types inherit from this class, this feature also is inherited by the agents in JAMDER. The received messages from the queue are stored until it is read and can be retrieved through the *receive()* method of *Agent*.

MAS-ML defines that the agent has a queue of incoming messages and a queue of the sent messages, but in JADE the track of sent messages is not stored. To attend this requirement, the *GenericAgent* class is defined to store the sent messages by an agent through of the method *sendMessage(ACLMessage)*.

JADE provides an identifier for each agent, consisting of an internal body of *jade.core.AID*. This attribute is created automatically and contains only the elements needed to be located, their addresses access and local name. It can be retrieved through the method *getAID()* of *Agent*. This feature is important to address and handle the sending of messages. Agent Types in JAMDER In the following subsections, each internal architecture is described how it is structured in JAMDER according to its type defined by TAO+.

Simple Reflex Agent. In JAMDER, this agent is defined by the *jamder.agents.ReflexAgent* class, which inherits from *GenericAgent*. This class contains a list of condition-action rules with the type *Hashtable<String, String>*, where the first name represents the perception and the second one corresponds to the action's identifier.

Beliefs and goals are not components of the agent role structure, in consistence with the corresponding agent structure.

Model-based Reflex Agent is characterized by storing the history of actions, which can be used in the decision-making, for example, it does not perform the same step (Russell and Norvig, 2002). This agent is represented by the class *jamder.agents.ModelAgent* that extends the *ReflexAgent* class and includes an attribute to represent the historical knowledge or beliefs as a list of *jamder.structural.Belief*. Additionally, the *ReflexAgent* class incorporates the abstract method *nextFunction(belief, perception*), responsible for mapping the perceptions and the current internal state (representation of model or environment) to a new updated internal state, showing the next action to be selected (Russell and Norvig, 2002).

Other agents can not access beliefs, but the concrete class that inherits from this agent can change their beliefs. So, their methods are protected. This kind of agent by acquiring the role will incorporate all the beliefs come from role, i.e., if there are similar beliefs, the agent's beliefs will be replaced by the role ones. The method *addAgentRole(name, role*) is overloaded to incorporate this principle. As the goals are not a part of this agent type, its respective *agentRole* (*ModelAgentRole*) also does not have this characteristic.

Goal-based Agent with Planning can generate an action plan at runtime. This kind of agent also has a set of goals to be achieved by the agent. Plans are composed by actions, sequentially. The class for this agent, which inherits from GenericAgent, is *jamder.agents.GoalAgent*. This class incorporates three additional abstract methods, the formulate goal function that takes the current state (belief or model) and returns a goal formulated, the formulate problem function that identifies the necessary actions for attempt the state and the goal and the planning method that returns a result of the actions. Also, this agent has perceptions and the next function which are present in this hierarchy trough the perceives list and *nextFunction(belief, perception)* method.

By purchasing the role, the goal-based agent will also incorporate the goals of the role. If there are the same goals, the agent's goals will be replaced. The *addAgentRole (name, role)* method in this class is overloaded to incorporate this principle.

The abstract method to implement the formulate goal function, represented by *formulateGoalFunction (belief)*, receives a belief and returns the formulated goal. The developer must provide the algorithm that implements this method. Similarly, the abstract method responsible for formulating problem function represented by *formulateProblemFunction (belief, goal)*, returns a problem to be used in planning. The problem is an action subset of agent actions (Russell and Norvig, 2002). Finally, the planning method, *planning(actions)*, receives the problem (action list) and returns a sequential list of these actions to achieve the goal. According to Gonçalves *et al.* (2011), the planning is based on the available actions in the action list, to create a sequence of actions (plan). Such as the reactive agents, this hierarchy also needs a sensor to take care of the environment perceptions. There is the *percept(perception)* method in this agent which uses the *Sensor* class to achieve this purpose.

Utility-based Agent has the same structure of the goal-based agent with planning. Also, the utility function is incorporated to guide the agent behavior. When the design can be linked to more than one goal to be reached, these goals can be conflicting, and so the utility function is inserted to determine the degree of value to the goals associated. Therefore, the class that represents this kind of agent is *jamder.agents.UtilityAgent*. This class inherits from *GoalAgent*. The utility provides a way that the probability of success can be considered about the goal importance (Russell and Norvig, 2002).

The *utilityFunction(action)* abstract method is used to identify the priority of the actions, based on the received action, and checks the priority of the goals about the agent. This method is called multiple times when at the formulation of the problem, it identifies the actions that will be executed. The return of this method consists of a value indicating the action with the highest priority.

Goal-based Agent with Plan. The goal-based agent with plan is frequently named BDI agent by in the context of agent-oriented software engineering too (Nunes *et al.*, 2011). It inherits from *GenericAgent* and contains beliefs and goals attributes and the plans and actions. The formulate goal function, formulate problem function, next function, utility function and perception are not considering in this case. An important difference is that its plans are defined in modeling time, which makes them statics. The class that represents this agent is the *jamder.agents.MASMLAgent*.

4.2. The other entities in MAS

In addition to the agent concept, JAMDER considers the representation of the other entities that frequently appear in MAS. Thus, specific representations of the implementation of agent role, organization, environment, object and object role are included in the framework.

4.2.1. *Agent Roles*

A role is an element that guides and restricts the social behavior of an agent or sub-organization in the organization. Each instance of the agent role is a member of an organization and determines what the agent can and should do within an organization (Silva *et al.*, 2003). To represent the agent role concept in MAS, JAMDER follows the agent hierarchy defined in Figure 4, which assigns a specific agent role related to the specific agent type. This definition of agent roles was necessary because an agent role adds new beliefs and goals to the beliefs and goals in the agent, but depending on the agent, some of them do not contain these characteristics. Based on the definition, it was created the *jamder.role.AgentRole* class and other associated classes to adequate the properties for it. This class is related to *ReflexAgent* and any structural attribute is required in this case because the selection process is based on condition-action rules. The basic properties for agent roles are:

- owner – Organization instance where the role is defined. This property can be recovered through getOwner() method. As owner does not change, its definition is done inside the AgentRole constructor;
- name – agent role identifier (String), defined and recovered by the access methods respectively, getName() and setName(String name);
- player – indicates who is exercising the role. It is represented by the GenericAgent class. It can be an agent or organization instance, this last one in case of a sub-organization. As player also does not change, its definition is done inside the AgentRole constructor;

The life-cycle of an agent role starts when it is associated to an entity (agent or sub-organization) (Silva *et al.*, 2003) (Silva *et al.* 2007) (Gonçalves *et al.*, 2010). This means that agent role instance is created after the creation of the associated entity. The link between the entity and the agent role can be canceled, in this case, it is needed that the entity has other agent roles in the same organization. Otherwise, the entity instance is also canceled, because the entity should have a link with at least one organization.

The agent roles work as actions that the agents need to execute and when the agent has some role, it incorporates the beliefs or goals from the role, if it has. In JAMDER, the status defined for an agent role include: *ACTIVATE*, *DEACTIVATE* and *CHANGE*. This information is obtained using *getAgentRoleStatus()* method in *AgentRole* class. The status CHANGE informs that the agent is migrating from one environment to another or from one organization to another. The *activeRole* restarts the actions of the agent playing the role and in the other hand, the *changeDeactivateRole(AgentRoleStatus)* method returns the status and deactivates the role.

The duties and rights of the roles define the actions assigned to the agent playing the role related to responsibilities and permissions, respectively (Weiss, 1999). These attributes in *AgentRole* are defined as an instance of *Hashtable<String, XXX> class,* where XXX represents an instance of *jamder.behavioral.Duty* or *jamder.behavioral.Right*. The classes *Duty* and *Right* were created in JAMDER to identify an attribute that defines the associated action. The Action class used by the agents also represents an action associated to duties and rights, which each duty or right has only one action. The creation of an agent role instance involves the analysis of right and duties calling the *initialize* method in the class constructor.

A protocol defines a group of messages that an agent can send to other agents (Bellifemine, 2007). In JADE there are several protocols FIPA compliant to standardize the communication between the agents. A communication protocol is defined by the type and the behavior of the communication attendees. Thus, the protocols in the *AgentRole* class are defined using an instance of *Hashtable<String, Behavior>* where *Behavior* represents an extension of the starter or participant class. Roles played by model-based agents (*ModelAgent*) require the definition of beliefs that are incorporated by the agent when committing to the role. The belief of the agent role also is defined by Belief class. In case of the agent has beliefs with the same name defined in its agent role, the belief in the role replaces the belief in agent. The class to represent this type of agent role in JAMDER is

jamder.role.ModelAgentRole that extends the *AgentRole* class and incorporates the belief attribute. This agent role is related only to model agents, which means, only ModelAgent instances can exercise this agent role type.

Finally, the *jamder.role.ProactiveAgentRole* class inherits from *ModelAgentRole* class and incorporates goals. The goal characteristic also is represented by Goal class. Proactive agents can play this role, i.e., *GoalAgent*, *UtilityAgent* and *MASMLAgent*. As sub-organization behaves like an agent, it also can exercise this agent role type.

4.2.2. *Object and Object Role*

In the other hand, the object and the object role are represented by the classes *Object* from Java and *jamder.roles.ObjectRole* in JAMDER, respectively. The object only answers requests that were asked. The *ObjectRole* class defines three attributes: role name, the object that will play the role and the organization to which it belongs. This role assists the execution of an object similar to what happen with agent role regarding and agent. It also is defined in an organization and its attributes are explained as follow: object role identifier (*name*); an object that will exercise this role (*object*); and the organization which this role is a member (*owner*).

4.2.3. *Organization*

The organization is a place where the agents act, it is in one environment and can stay in only one, it means, it cannot change or move to another environment. It is defined in JAMDER as *Organization* class is defined as an extension of *MASMLAgent* (Figure 3) and describes a JADE container where agent roles and object roles are defined. An organization can contain sub-organizations, recursively. The attribute *Hashtable<String, Organization>* are used to represent sub-organizations including the name and the sub-organization instance, respectively. Analogously, a sub-organization instance is an organization and can contain a new sub-organization, thus this relationship is established with the *superOrganization* attribute.

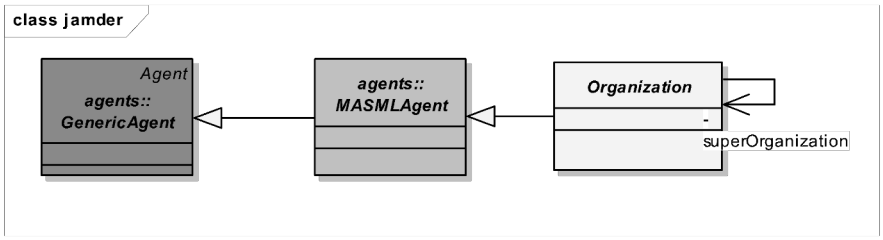

Figure 3: Organization hierarchy in JAMDER

JADE has a class named *ContainerID* that represents the identifier for the organization. Each *Organization* instance contains the attribute *containerID* with the respective container (JADE) associated to the organization on JAMDER. As *Organization* inherits from *MAS_MLAgent*, the inheritance of *ContainerID* is not possible.

Due the *Organization* class extends the *MASMLAgent* class. Thus, the structural features and access methods of this entity, such as *Beliefs*, *Goals*, *Plans* and *Actions* are incorporated. Additionally, an organization includes axioms used to govern the agent actions. The axiom concept is represented by *jamder.Structural.Axiom*, it is a subclass of *Property* since shares the same structure and restricts the actions of agents and sub-organizations that dwell on the organization. Axiom is an organization particularity and is handled in *Organization* class.

The agents in the organization are obtained through the roles that the organization has once the agent role knows the organization is involved and the agent who performs. As *Organization* inherits the list of roles from *MAS_MLAgent*, this list can be performed in two ways:

- In the case of an organization, the list consists of all agents roles accepted by the organization, or the roles to be stored or played by an agent or by a sub-organization;

- If the instance has the function of sub-organization, the list of agent roles consists in the agent roles exerted by sub-organization. The superOrganization attribute must contain the instance of the Organization which belongs regarding to execute as sub-organization.

This class also defines the agent roles and object roles. The attribute is specified as an instance of *Hashtable<String, XXX>*, where *XXX* can be performed by *AgentRole* or *ObjectRole*. This attribute stores all the agent roles or object roles that the agent or object can perform. Its access methods follow the same way of the other attributes. In the Organization, the properties of sending and receiving messages are treated as the same way as agents.

4.2.4. *Environment*

Finally, the environment represents the JADE platform, where reside the containers (represented by the organization entity) and agents, and defines methods for managing (adding and removing) other entities. This entity is represented by the *Environment* class in JAMDER. When an environment instance is created, the *Environment(name, host, port)* constructor receives the platform name, the machine name (host) and the machine port that will be stored at platform. By creating an environment, the platform creates the main container that will hold the platform services like Directory Facilitator (yellow pages) and Agent Management System, where this last one manages the next organizations (containers) and agents that will be created in this platform.

When the JADE platform is started, it can provide services needed for the other entities' lifecycle, as for an example, the agent searching service. In this case, as *Environment* class in JAMDER does not have super class from JADE, it acts as an adapter of one environment using the JADE platform. Each platform acquires naturally one identifier that is an instance of *jade.core.PlatformID*. Similarly, *Enviroment* class has the *ID* attribute, where it is an instance of *PlatformID*. Additionally, other attributes compound the environment as: *name*, *address* and so on. In addition to these properties, the *Environment* class also has other methods that will manage the environment, for example, by adding an organization in the environment, *addOrganization(String, Organization)* method increases one Organization instance in the list of organizations, it also creates a container in the JADE platform.

The reverse process, i.e., the organization and agent removal, needs to be analyzed with caution because several dependencies need to be observed. Firstly, the relationships with agents, sub-organizations and objects that inhabit the organization must be disposed. If an organization has agents and objects, but if some agent also participates in another organization, only this will cancel the agent roles linked to the organization being deleted. Otherwise, the agent itself is also deleted. The *removeOrganization(String key*) method implements this behavior if the organization contains sub-organizations, will run recursively.

Assuming that the environment knows all agents the supports, the creation procedure for agents is required in this class through *addAgent(String key, GenericAgent agent*), where the key is the agent's name and agent instance. Remove an agent is simpler than remove of an organization because the agent has fewer dependencies. The agent removal process initially verifies the roles which organizations the agent is packed. When removing the contract with the agent role, the instance of the agent turns out not to exist. The *removeAgent(String name)* performs this behavior. The changes that the agent perceives happen in own environment. These changes come from the actions that the agents themselves perform, that is, when an agent performs some action, it can change any state of the environment.

Even if the agent can move between environments, it is not possible be in both environments at the same time. Consequently, the agent must cancel all roles and relationships that it exercises before getting to another environment, as well as in organizations in which it exercises a role. Upon entering the new environment, the agent is instantiated in an organization and starts to exercise an agent role in this organization. If the migration happens between organizations, it changes the status of agent roles to CHANGE and its actions are completed or ended.

In JADE, every agent has three methods that help when that it is moving around. These methods are overloaded in *GenericAgent* to adapt the migration of staff to update organizations and the environment. The *beforeMove()* method contains the mechanism responsible for performing an action before the agent gets around. In

its code, the status of agent roles is modified to CHANGE. The *afterMove()* method is executed after the agent migration and, in its content, the developer can provide information about what can be done after the agent gets around. If the mobility is between organizations, the agent roles (of the organization) which are with the status CHANGE switch to ACTIVATE status and actions assigned to this role are again exercised. If mobility is between environments, it will exercise the actions of the agent role that is obtained in the organization of the target environment. Finally, the *doMove(Location)* method is operable to verify the proper performance of the migration. The location parameter is a superclass of ContainerID and PlatformID, and is responsible for informing the organization or the environment ID, respectively.

5. Code Generation

In the context of the MDA approach, automated support to the code generation for MAS is proposed. In the proposed process, the following components for generating code are applied: MAS-ML Tool (PIM); Java/JAMDER framework (PSM); and Acceleo templates (PDM). MAS-ML Tool (Silva *et al.*, 2007) (Gonçalves *et al.*, 2015) is an Eclipse[9] plugin which follows an approach directed by models to support the modeling of MAS-ML 2.0 language (Gonçalves *et al.*, 2011) by the concepts and abstractions defined in MAS.

The MAS-ML Tool supports the modeling of the following diagrams: organization diagram, role diagram and class diagram, bases on the MAS-ML metamodel. This tool generates the masml file (XMI - XML Metadata Interchange) that stores the structure of data entities and structural and behavioral aspects defined in MAS-ML 2.0. These files are the input to the transformation to code generation process using the plugin Acceleo in order to support the concept of MDA since allows the code generation in different code languages and incremental development.

To formalize the code generation in Acceleo, it is needed to establish a template for each entity through a language defined by the OMG, the MTL (Model Transformation Language). When the template is executed on Eclipse, it will need the MAS-ML model representation (.masml files) and the output folder to store the JAMDER classes).

5.1. Environment

The environment is an instance of a class that inherits from Environment class in JAMDER and its representation occurs through *EnvironmentClass* in MAS-ML Tool. As this instance inherits the methods defined in Environment, it is necessary only, on code generation, the constructor of this class calls the superclass and create organizations, agents and agent roles in that order. The need to create agent role instances is only to link the organization and agent instances, where the agent role construct does this link. The mapping of creating and organizing bodies agents occurs through inhabits the relationship between the environment and these entities. The objects and object roles are also created in the environment where the roles are obtained by the organization through the ownership relationship. In turn, the objects are achieved by the object through the play relationship.

If needed additional methods or attributes, the MASML Tool offers the Operation and Property components to this need, respectively. Figure 4 shows the template to generate the environment.

9 www.eclipse.com

```
public class (c.name /) extends Environment {
  public (c.name /) (String name, String host, String port) {
        super(name, host, port);
   (for(i : Inhabit | c.inhabit))
       (let organ : OrganizationClass = i.org.name)
        (if (i.org -> size() > 0) )
           Organization (organ.name/) = new Organization("(organ.name/)", this, null);
           addOrganization("(organ.name/)", (organ.name/));
           (if (i.org.play -> size() > 0) )
           (let ar : AgentRoleClass = i.org.play.agentRole)
             AgentRole (ar.name/) = new AgentRole("(ar.name/)",
(ar.ownership.owner.name/), (organ.name/));
           (/let)  (/if)

           (for(ow : Ownership | i.org.ownership))
                    (for(ob : ObjectRoleClass | ow.objectRole))
                      Object (ob.play.name/) = new Object();
                      addObject("(ob.name/)", (ob.name/));
                      ObjectRole (ob.name/) = new ObjectRole("(ob.name/)",
(organ.name/), (ob.play.name/));
                    (/for)
           (/for)
            (/if)
       (/let)
     (/for)

     (for(i : Inhabit | c.inhabit))
        (if (i.agentClass -> size() > 0) )
           (let a : AgentClass = i.agentClass)
          GenericAgent (a.name /) = new (a.name /)("(a.name /)", this, null);
           AgentRole (a.play.agentRole.name/) = new
AgentRole("(a.play.agentRole.name/)", (a.play.agentRole.ownership.owner.name/),
(a.name/));
           addAgent("(a.name /)", (a.name /));
           (/let)
        (/if)
     (/for)
  }
  // Additional attributes
  (for(p : Property | c.ownedProperty))
     (p.visibility/) (p.type.toString()/) (p.name/);
  (/for)

  // Additional methods
  (for(o : Operation | c.ownedOperation))
     (o.visibility /) (o.returnValue /) (o.name /)
        ((for(p:Parameter | o.parameter) separator(',')) (p.type/) (p.name/) (/for)) {}
  (/for)
}
(/file)
(/template)
```

Figure 4: Environment template

An important feature to note is that the agent role needs to know that the agent or sub-organization is exercising this role, while the agent or sub-organization needs to know what their role of the initial agent. To solve this problem the agent role parameter starts to null. In creating the agent role, its constructor takes as a parameter the instance of the agent or sub-organization where this builder sets the agent or sub-organization, the role will exert initially through *addAgentRole* method (*AgentRole*).

5.2. Object and Object Role

The object role in JAMDER is represented by MAS-ML Tool for *ObjectRoleClass* entity. The structure of this class does not have many attributes, in turn, generate new object of this class inherits only roles. While the object entity is any Java object and that the Acceleo already creates through its pre-defined entity Class. Figure 5 shows the details of this template.

```
(template public generateJava(c : ObjectRoleClass))
(comment @main /)
(file (c.name + '.java', false, 'UTF-8'))
import jamder.roles.ObjectRole;
import jamder.Organization;
public class (c.name /) extends ObjectRole {
   //Constructor
   public (c.name /) (String name, Organization owner, Object object) {
          super(name, owner, object);
   }
}
(/file)
(/template)
```

Figure 5: ObjectRole template

5.3. AgentRole

The *AgentRoleClass* entity MAS-ML Tool supports three types of agent role defined by MAS-ML 2.0. If the role has not beliefs and objectives, the role of the agent will be *AgentRole*. If the role is not just beliefs, the role of the agent will be *ModelAgentRole*. And if the role contains all the available structure, it will be the kind of *ProactiveAgentRole*. The template in .mtl dealing the agent role creates the corresponding inheritance to those found features (Figure 6).

```
(template public generateJava(c : AgentRoleClass))
...
import jamder.behavioral.*;

(if (c.superClass->size() > 0) )
public class (c.name /) extends (c.superClass.name /) {
(else)
        (if ((c.ownedBelief->size() <= 0) and (c.ownedGoal->size() <= 0)) )
public class (c.name /) extends AgentRole {
        (elseif ((c.ownedBelief->size() > 0) and (c.ownedGoal->size() <= 0)) )
public class (c.name /) extends ModelAgentRole {
        (elseif ((c.ownedBelief->size() > 0) and (c.ownedGoal->size() > 0)))
public class (c.name /) extends ProactiveAgentRole {
        (/if)
(/if)
   //Constructor
   public (c.name /) (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);

   for(b : Belief | c.ownedBelief))
   addBelief("(b.name/)", new Belief("(b.name/)", "(b.type/)", "(b.default/)"));
   (/for)
   (for(g : Goal | c.ownedGoal))
   addGoal("(g.name/)", new LeafGoal("(g.name/)", "(g.type/)", "(g.default/)"));
   (/for)
```

```
    (for(r : Right | c.ownedRight))
    addRight("(r.name/)", new Right());
    (/for)
    (for(d : Duty | c.ownedDuty))
    addDuty("(d.name/)", new Duty());
    (/for)
    (for(p : ProtocolClass | c.protocol))
    addProtocol("(p.name/)", null);
    (/for)

     initialize();
    }
}
```

Figure 6: AgentRole template

At the end of this template is held to call the *initialize()* method in *AgentRole* class, JAMDER framework, to check the rights and duties shall be exercised by role. Regarding the protocol, as in JADE there are different types and they somehow inherit from *Behavior*, and the developer needs to inform what type of existing protocol on JADE will be used.

5.4. Agent

The template to the agent generation is analogous to the agent role template since each agent inherits from the class corresponding to the JAMDER agent, and the agent type depends on the components that the agent contains. Figure 7 shows the template for the creation of all agent types defined in MAS-ML2.0.

```
(query public possuiUtility(actions : OrderedSet(ActionClass)) : Boolean =
actions.actionSemantics.toString().equalsIgnoreCase('<<Utility-Function>>') /)
(template public generateJava(c : AgentClass) )

(comment @main /)
(file (c.name + '.java', false, 'UTF-8'))
import jamder.behavioral.*;
...
import jamder.agents.*;

    (if (c.superClass->size() > 0) )
public class (c.name /) extends (c.superClass.name /) {
       (elseif ((c.ownedBelief->size() <= 0) and (c.ownedGoal->size() <= 0)) )
public class (c.name /) extends ReflexAgent {
        (elseif ((c.ownedBelief->size() > 0) and (c.ownedGoal->size() <= 0)) )
public class (c.name /) extends ModelAgent {
        (elseif (c.ownedPlan->size() > 0) )
public class (c.name /) extends MASMLAgent {
        (elseif (c.ownedPlanning->size() > 0) )

        (if (c.possuiUtility(c.owendAction).toString().contains('true')))
public class (c.name/) extends UtilityAgent {
        (else)
public class (c.name /) extends GoalAgent {
        (/if)
(/if)
```

```
    //Constructor
    public (c.name.toUpperFirst() /) (String name, Environment env, AgentRole agRole) {
    super(name, env, agRole);

    (for(b : Belief | c.ownedBelief))
       addBelief("(b.name.concat('B')/)", new Belief("(b.name.concat('B')/)", "(b.type/)",
         "(b.default/)"));
    (/for)

    (for(g : Goal | c.ownedGoal))
       Goal (g.name.concat('G')/) = new LeafGoal("(g.name.concat('G')/)", "(g.type/)",
         "(g.default/)");
       addGoal("(g.name.concat('G')/)", (g.name.concat('G')/));
    (/for)
...
```

Figure 7: Agents template

Note that the agent modeling needs to be consistent with your brand, that is, its structure must comply with the MAS-ML 2.0 specification, so in his generation, agent properties also match your type of agent presently. To facilitate the definition of the agent type or inheritance from its structure, checking the diagram of the components in the following order is made:

- If the agent inherits from another agent, the inheritance relationship is held;
- If the agent does not define beliefs and goals, this agent inherits from ReflexAgent;
- If the agent defines beliefs, but do not have goals, this agent inherits from ModelAgent;
- If the agent defines a pre-defined plan, this agent inherits from MASMLAgent;
- If the agent defines a plan to set (planning) and does not have utilityFunction(), this agent inherits from GoalAgent;
- If the agent defines a plan to set (planning) and has an utilityFunction(), this agent inherits from UtilityAgent;

The *Action* of *AgentClass* component defined in MAS-ML Tool works in two ways in the representation of agents. Each attribute of this component has a field named *ActionSemantics* that may or may not be completed.

If the *ActionSemantics* field is not filled in MAS-ML Tool, that means this attribute is one of the actions that the agent can run, i.e., is an Action of JAMDER instance. An important point about the actions is that in MAS-ML, they can have preconditions and post-conditions, but in the current version of MAS-ML Tool, this feature is not contemplated yet. On another hand, if the *ActionSemantics* field is filled, it works as one of the methods which perform the agent, depending on the agent structure. The allocation or definition of methods will depend on the type of agent that the designer wants to set. Completion of *ActionSemantics* field can contain and run one of the following situations (Figure 8):

- <<Next-Function>>: indicates that the agent has the function of nextFunction() type and is useful for reactive agents with knowledge, ModelAgent;
- <<Formulate-Problem-Function>>: indicates that the agent has the function of formulateProblemFunction() type and is used for planning with agents, GoalAgent;
- <<Formulate-Goal-Function>>: indicates that the agent has the function of formulateGoalFunction() type and is used for planning with agents, GoalAgent;
- <<Utility-Function>>: indicates that the agent has the function of utilityFunction() type and is used for planning with agents, UtilityAgent.

Although these methods contain some stereotype that identifies its function, the method also contains a name, but in JAMDER, the method names are abstract and already defined. Regarding resolve this issue, the generated code comprises two methods for each property *ActionSemantics* filled in case the JAMDER method and the method with the name used in modeling, where the first refers to the second method. For example, if the agent contains the property <<*Next-Function*>> as *nextGroups*, the generator generates the *nextFunction*() method which calls the *nextGroups*() method.

If the agent contains any attribute in the Planning compartment, it indicates this agent (*GoalAgent*) contain the function *planning*(), used to assemble the plan at runtime. The plan creation happens in two ways, Plan or planning. The first form consists in a plan defined in modeling time, which comprises in its structure the *ownedAction* attribute that holds its actions and is part of the agent. The second form is a plan without action, but which may be built at run time by the agent. Both types of MAS-ML Tool plan are instances of Plan class in JAMDER.

```
  ...
   (for(ac : ActionClass | c.owendAction))
     (if (ac.actionSemantics.toString().trim().size() <= 0))
       Action (ac.name.concat('Ac')/) = new Action("(ac.name.concat('Ac')/)", null, null);
       addAction("(ac.name.concat('Ac')/)", (ac.name.concat('Ac')/));
     (/if)
   (/for)
...

(for(a : ActionClass | c.owendAction))
 (if (a.actionSemantics.toString().equalsIgnoreCase('<<Next-Function>>')))
protected Belief nextFunction(Belief belief, String perception) {
        return (a.name/)( belief, perception);
}
private Belief (a.name/)(Belief belief, String perception) {
        return null;
}
 (elseif (a.actionSemantics.toString().equalsIgnoreCase('<<Formulate-Problem-Function>>')))
protected List<Action> formulateProblemFunction(Belief belief, Goal goal) {
        return (a.name/)( belief, goal);
}
private List<Action> (a.name/)(Belief belief, Goal goal) {
        return null;
}
 (elseif (a.actionSemantics.toString().equalsIgnoreCase('<<Formulate-Goal-Function>>')))
protected Goal formulateGoalFunction(Belief belief) {
        return (a.name/)(belief);
}
private Goal (a.name/)(Belief belief) {
        return null;
}
 (elseif (a.actionSemantics.toString().equalsIgnoreCase('<<Utility-Function>>')))
protected Integer utilityFunction(Action action) {
        return (a.name/)( action);
}
private Integer (a.name/)(Action action) {
        return 0;
}
 (/if)
(/for)
}
(/file)
(/template)
```

Figure 8: Agent template (Continuation)

The lists of messages send and received from the agent are not defined when the agent's creation. Methods of access to these lists are inherited from *GenericAgent* class of JAMDER.

5.5. Organization

The *OrganizationClass* node in MAS-ML Tool and the *Organization* class in JAMDER represent the organization concept. Its structure is similar to the agents in terms of beliefs, goals, actions and plans. Beyond these components, the organization structure comprises the axioms.

The process of creating the sub-organizations is the same as an organization, however, when the constructor's parameter, *org,* is not null, this means that the current organization has a super-organization. This rule is detailed in JAMDER. Figure 9 shows the template for *Organization*.

```
(template public generateJava(c : OrganizationClass))
(comment @main /)
(file (c.name + '.java', false, 'UTF-8'))
import jamder.Organization;
...
import jamder.behavioral.*;

(if (c.superClass -> size() > 0) )
public class (c.name /) extends (c.superClass.name /) {
(else)
public class (c.name /) extends Organization {
(/if)
   //Constructor
   public (c.name /) (String name, Environment env, AgentRole agRole, Organization org) {
     super(name, env, agRole);

     (for(b : Belief | c.ownedBelief))
       Belief (b.name.concat('B')/) = new Belief("(b.name.concat('B')/)", "(b.type/)",
        "(b.default/)");
           addBelief("(b.name.concat('B')/)", (b.name.concat('B')/));
        (/for)

     (for(g : Goal | c.ownedGoal))
       Goal (g.name.concat('G')/) = new LeafGoal("(g.name.concat('G')/)", "(g.type/)",
        "(g.default/)");
       addGoal("(g.name.concat('G')/)", (g.name.concat('G')/));
     (/for)

    (for(a : ActionClass | c.ownedAction))
      (if (a.actionSemantics.toString().trim().size() <= 0))
        Action (a.name/)Ac = new Action("(a.name/)", null, null);
        addAction("(a.name/)", (a.name/)Ac);
      (/if)
    (/for)

    (for(p : PlanClass | c.ownedPlan))
      Plan (p.name.concat('Plan')/) = new (p.name/)("(p.name.concat('Plan')/)",
        (p.owendGoal.name/)G);
      addPlan("(p.name.concat('Plan')/)", (p.name.concat('Plan')/));
      (for(ac : ActionClass | p.ownedAction))
```

```
        (p.name.concat('Plan')/).addAction("(ac.name.concat('Ac')/)",
        (ac.name.concat('Ac')/));
      (/for)
    (/for)
  }
}
(/file)
(/template)
```

Figure 9: Organization template

Each .mtl file created contains the generation template of each entity in Acceleo, which enables the code generation to JAMDER from models designed in MAS-ML Tool. A similar process can be used to generate code to other frameworks or programming language. The process the process guarantees the code correctness for construction.

6. CASE STUDY

This chapter illustrates the generation of code to develop a MAS for the collaborative learning environment Moodle[10]. Moodle is an Open System Course Management Source - Course Management System (CMS), also known as Learning Management System (LMS) or a Virtual Learning Environment (VLE), which is a representation of the interaction between students and teachers in a virtual environment. The case study's purpose is to illustrate the application of the proposed approach by creating a multi-agent system for Moodle in MAS-ML Tool. After that, the model is used to input to the code generation process. Initially, it must create an Acceleo project in the Eclipse tool containing the classes and Acceleo templates (.mtl file) to generate code in JAMDER.

To perform code generation in this MAS prototype design, it was used MAS-ML Tool organization diagram (Figure 12) including: agents, organization, agent roles and environment and the relationships play, ownership and inhabit, between these entities. Each template will be executed individually using the diagram that contains the entity needed to the template to generate them. However, all properties of the entities as beliefs, goals, among others, have been omitted from this diagram; due to the limit of display space in the text of the job, however, the structure of the entities have the properties which will be detailed in the following sections.

The representation of the environment for this project has only one instance of Environment in JAMDER where this instance contains other entities, here represented as *MoodleEnv*. Having the knowledge that the goal of the Moodle system is to bring students and teachers virtually, the representation of the organization is only one instance of Organization in JAMDER, in this case, represented by *MoodleOrg*. Because of these characteristics, all actors and roles this MAS are in the same organization and therefore in the same environment.

10 http://moodle.org.

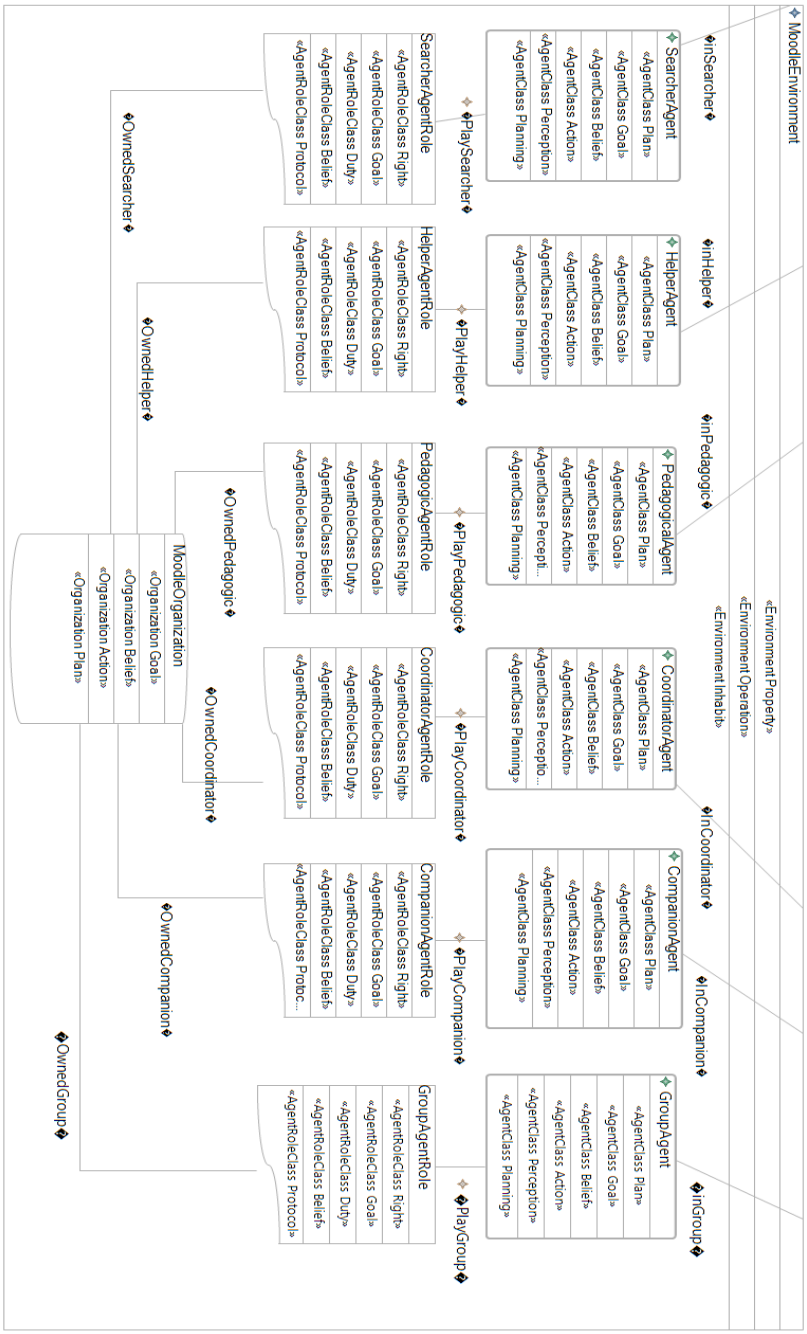

Figure 10: Moodle prototype in MAS-ML Tool

An important point to note in this MAS prototype is that the beliefs of agents and agent roles were represented in modeling as .pl files where these files contain information of beliefs to be read by the generated class. The function of this file just holds the beliefs and their information will be obtained from their lecture, adapting the code after being generated by the tool. Next, details of agent and agent roles entities involved in this MAS prototype are presented.

6.1. Agents

Based on the idea of Moodle system the following agents have been identified for this environment in MAS-ML 2.0.

6.1.1. *Companion Learning Agent*

This type of agent should be able to choose independently among a predetermined range of affective interaction strategies such as messages of support. It presents encouraging messages (positive reinforcement) when the user, through the manifested interactions, gives evidence that is straightforward to follow discussions and/or the proposed tasks and/or content, and even when the student poses a lot of rhythms higher than the average in your class or workgroup. Because of the need to keep class notes for comparison and send messages quickly, this agent is characterized as a reactive agent based on knowledge. Its structure is given below and illustrated in Figure 11.

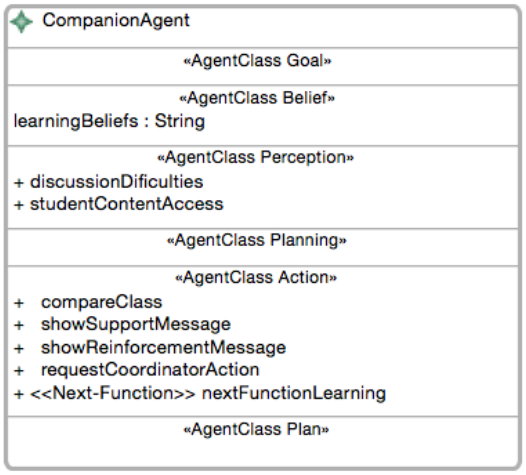

Figure 11: CompanionAgent structure

This agent must know the affective interaction strategies and messages of support (emotional imprint messages) that will be given to the user by their beliefs. When the user through the perceptions of interactions, shows signs of difficulty to follow the discussion and/or the proposed tasks and/or content, the perceptual attribute is identified by *discussionDifficulties* and can be obtained through the emoticon faces. The perception *studentContentAccess* is designed to determine if the student is accessing the content.

The *compareClass* action compares with the class and verifies that the student is much higher or much lower, however, before this comparison, it is necessary that the precondition *studentAverage* is different from zero. The *showSupportMessage* action displays a message of support, but the *noDifficulty* precondition must be satisfied and the student must be getting content. The action *showReinforcementMessage* has an opposite effect, where *studentWithDifficulty* precondition must be true. The *requestCoordinatorAction* action requests an action of another agent to contact the coordinator to meet the preconditions of *classTipMsgs* and *difficultyFeatures*, among others. Figure 14 shows the code generated for *CompanionAgent* in JAMDER.

According to Figure 12, the generated code for this agent has in its constructor a call to its super class, in this case, *ModelAgent*. Soon after, there is the creation of instances of actions related to this agent, the *addAction*(*String*, *Action*) method, which advises that these instances are being included in the agent's actions list and finally, perceptions are defined. The methods designed for this agent were *nextFunction*(*Belief*, *Perception*) and *learningNextFuction*(*Belief*, *Perception*).

```
import jamder.behavioral.*;
...
import jamder.agents.*;

public class CompanionAgent extends ModelAgent {
   //Constructor
   public CompanionAgent (String name, Environment env, AgentRole agRole) {
     super(name, env, agRole);
     addBelief("learningsBeliefsB", new Belief("learningBeliefsB", "String",""));

     Action compareClasspAc = new Action("compareClassAc", null, null);
     addAction("compareClassAc", compareClasspAc);
     Action showSupportMessageAc = new Action("showSupportMessageAc", null,null);
     addAction("showSupportMessageAc", showSupportMessageAc);
     Action showReinforcementMessageAc = new Action("showReinforcementMessageAc", null, null);
     addAction("showReinforcementMessageAc", showReinforcementMessageAc);
     Action requestCoordinatorActionAc = new Action("requestCoordinatorActionAc", null, null);
     addAction("requestCoordinatorActionAc", requestCoordinatorActionAc);

     addPerceive("discussionDifficulties", null);
     addPerceive("studentContentAccess", null);
   }

   protected Belief nextFunction(Belief belief, String perception) {
        return learningNextFunction(belief, perception);
   }
   private Belief learningNextFunction(Belief belief, String perception) {
        return null;
   }
}
```

Figure 12: CompanionAgent class in JAMDER

6.1.2. *Learning Assistant (Pedagogical)*

This agent should be able to accompany the student in the different disciplines that participate to contribute to the user through tips, suggestions and messages about the topic ongoing and not only affective nature of messages (support). It is a goal-based agent with planning because he needs to create a study plan, suggesting disciplines for the student based on the disciplines that the student is doing (Figure 13).

The pedagogic agent perceives the existing disciplines(s) that the student is enrolled. From perceptions, the actions belonging to the agent include or relate the disciplines in which the student is enrolled and suggest other disciplines that the student attends.

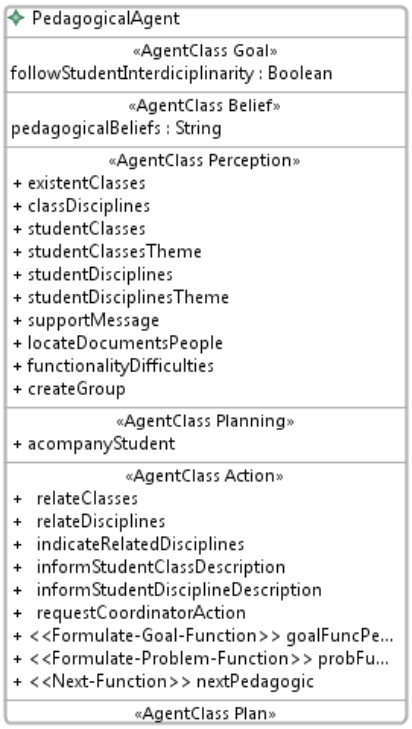

Figure 13: PedagogicalAgent structure

As a goal-based agent with planning, methods belonging to this type of agent need to be implemented by the developer, for example, to set up his run-time plan to reach his goal, where the plan is to *acompanyStudent*. Figure 14 shows the code generated for *PedagogicalAgent* in JAMDER.

```
import jamder.behavioral.*;
...
import jamder.agents.*;
public class PedagogicalAgent extends GoalAgent {
   //Constructor
   public PedagogicalAgent (String name, Environment env, AgentRole agRole) {
   super(name, env, agRole);

     addBelief("pedagogicalBeliefsB", new Belief("pedagogicalBeliefsB", "String", ""));
     Goal followStudentdisciplinarityG = new LeafGoal("followStudentdisciplinarityG",
        "Boolean", "");
     addGoal("followStudentdisciplinarityG", followStudentdisciplinarityG);

     Action relateClasssesAc = new Action("relateClasssesAc", null, null);
     ...
     Action("requestCoordinatorActionAc", null, null);
     addAction("relateClasssesAc", relateClasssesAc);
     ...
     addAction("requestCoordinatorActionAc", requestCoordinatorActionAc);

     addPerceive("existentClasses", null);
     ...
```

```
import jamder.behavioral.*;
...
```

```
      addPerceive("createGroup", null);

      Plan acompanyStudentPlan = new Plan("acompanyStudentPlan", null);
      addPlan("acompanyStudentPlan", acompanyStudentPlan);
   }
      protected Plan planning(List<Action> actions){
      return null;
   }

   protected Goal formulateGoalFunction(Belief belief) {
      return goalFuncPedagogical(belief);
   }
   private Goal goalFuncPedagogical(Belief belief) {
      return null;
   }
   protected List<Action> formulateProblemFunction(Belief belief, Goal goal) {
      return probFuncPedagogical(belief, goal);
   }
   private List<Action> probFuncPedagogical(Belief belief, Goal goal) {
      return null;
   }
   protected Belief nextFunction(Belief belief, String perception) {
      return nextPedagogic(belief, perception);
   }
   private Belief nextPedagogic(Belief belief, String perception) {
      return null;
   }
   public void percept(String perception) { }
}
```

Figure 14: PedagogicalAgent class in JAMDER

6.1.3. *Information Search Engine Agent*

This type of agent finds people within Moodle environment which are involved in disciplines related to a particular topic of interest. The purpose of this agent is to do ongoing research and autonomously, show personal documents and contacts wherever locate potential common interests. Figure 15 shows the structure of the agent.

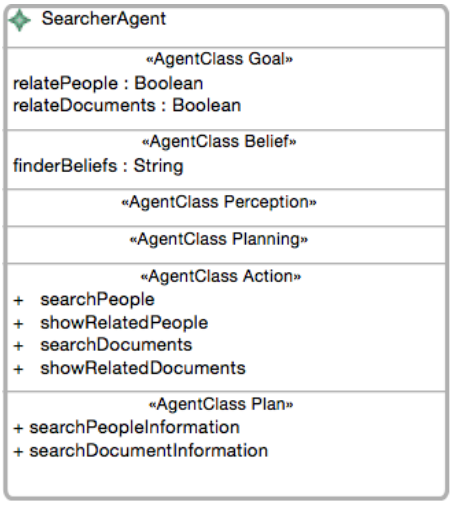

Figure 15: SearcherAgent structure

The agent beliefs store the people, groups and disciplines. The goal *relatePeople* checks people in Moodle environment that are involved with projects or subjects in common. The goal *relateDocuments* search pages as documents, projects or other digital files you have in common, for example, the keyword.

The actions belonging to this agent can be in four ways. The *searchPeople* action, as the name implies, located the people who are related to the same user topic. The *showRelatedPeople* action is used to display the result, where the precondition of this action is the result of the location of other people is different from zero. In the same way, the *searchDocuments* and *showRelatedDocuments* actions are used for documents.

Due to this agent already knows what to do and how to do, it is classified as a plan with goal-based agent. In this case, it has two predefined plans. The *searchPeopleInformationPlan* plan has two actions, *searchPeople* and *showRelatedPeople*, in that order, to achieve the goal *relatePeople*. Also, in the same way, the *searchDocumentInformationPlan* plan uses the *searchDocuments* and *showRelatedDocuments* actions, in this order, to achieve the goal *relateDocuments*. The Figure 16 shows the code for *SearcherAgent* generated in JAMDER.

The *searchPeople* and *showRelatedPeople* actions for the *searchPeopleInformation* plan and the *searchDocuments* and *showRelatedDocuments* actions for the *searchDocumentInformation* plan have been included manually in their respective plans after generation of this class for the MAS-ML Tool; the current version has not yet the association between plan and semantically actions.

```
import jamder.behavioral.*;
...
import jamder.agents.*;

public class SearcherAgent extends MASMLAgent {
   //Constructor
   public SearcherAgent (String name, Environment env, AgentRole agRole) {
   super(name, env, agRole);
     addBelief("finderBeliefsB", new Belief("finderBeliefsB", "String", ""));

     Goal relatePeopleG = new LeafGoal("relatePeopleG", "Boolean", "false");
     addGoal("relatePeopleG", relatePeopleG);
     Goal relateDocumentsG = new LeafGoal("relateDocumentsG", "Boolean","false");
     addGoal("relateDocumentsG", relateDocumentsG);

     Action searchPeopleAc = new Action("searchPeopleAc", null, null);
     addAction("searchPeopleAc", searchPeopleAc);
     Action showRelatedPeopleAc = new Action("showRelatedPeopleAc", null, null);
     addAction("showRelatedPeopleAc", showRelatedPeopleAc);
     Action searchDocumentsAc = new Action("searchDocumentsAc", null, null);
     addAction("searchDocumentsAc", searchDocumentsAc);
     Action showRelatedDocumenstAc = new Action("showRelatedDocumentsAc", null, null);
     addAction("showRelatedDocumentsAc", showRelatedDocumenstAc);

     Plan searchPeopleInformationPlan = new Plan("searchPeopleInformationPlan",
        relateDocumentsG);
     addPlan("searchPeopleInformationPlan", searchPeopleInformationPlan);
     Plan searchDocumentInformationPlan = new Plan("searchDocumentInformationPlan",
        relatePeopleG);
     addPlan("searchDocumentInformationPlan", searchDocumentInformationPlan);
   }
   public void percept(String perception) { }
}
```

Figure 16: SearcherAgent class in JAMDER

6.1.4. *Agent provides help on Moodle*

This agent is the simple reactive type and has a list of several insights into the difficulties the user has, and before this, it chooses the appropriate action. This agent realizes at what point the user is at the same time independently offers tips on how to make the best use of a particular functionality. Figure 17 shows that this agent is made only of perceptions and actions, consequently it is a simple reactive agent. To match the perception, the agent performs one of the actions to address the perception. They have no pre or posconditions. Figure 18 shows the code generated for *HelperAgent* in JAMDER.

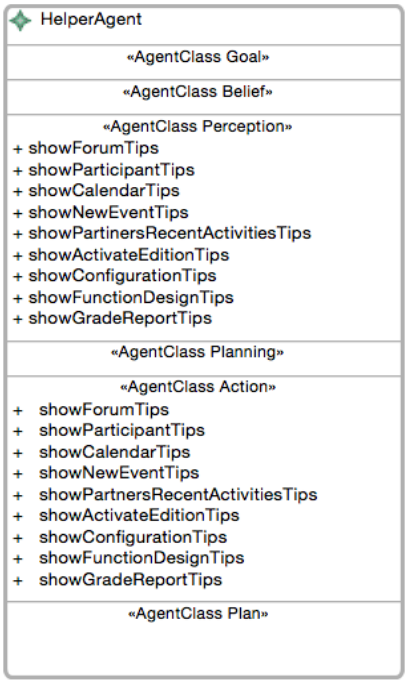

Figure 17: HelperAgent structure

```
import jamder.behavioral.*;
...
import jamder.agents.*;

public class HelperAgent extends ReflexAgent {
   //Constructor
   public HelperAgent (String name, Environment env, AgentRole agRole) {
   super(name, env, agRole);

   Action showForumTipsAc = new Action("showForumTipsAc", null, null);
   addAction("showForumTipsAc", showForumTipsAc);
   ...
   Action showGradeReportTipsAc = new Action("showGradeReportTipsAc", null, null);
   addAction("showGradeReportTipsAc", showGradeReportTipsAc);

   addPerceive("showForumTips", null);
   ...
   addPerceive("showGradeReportTips", null);
   }
}
```

Figure 18: HelperAgent class in JAMDER

6.1.5. *Coordinator Agent*

This type of agent should be able to centralize the requests of the agents and the information they send to each other, making this agent an intermediary between other agents. The objective of coordinator agent is to order the actions of agents (*requestAgentsActions*).The *requestActionPlan* plan uses two actions to achieve this goal: *checkAgentAction* and *requestAction* where the first one checks if the agent can perform the task and the second one asks the agent to perform the action after verifying that the agent can perform (*checkAgentAction*) (Figure 19).

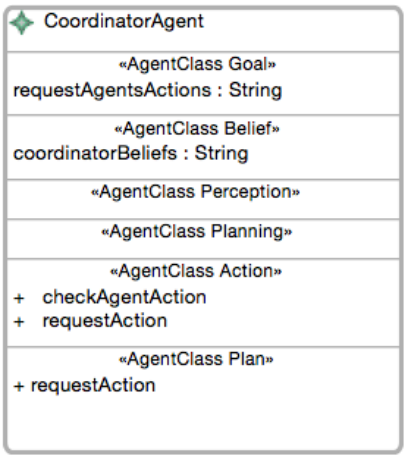

Figure 19: CoordinatorAgent structure

Similarly, to the SearcherAgent, the CoordinatorAgent is classified as a plan with a goal-based agent. Thus, the code generation is omitted in this case.

6.1.6. *GroupAgent*

This agent should be able to assist autonomously users, students and educators, the composition of working groups taking into account affinity themes or learning profiles. For this, it must consider certain criteria established by a trainer of one or more classes, or by the user interested in integrating the working groups. Figure 20 shows the structure of *GroupAgent*.

The beliefs of this agent store courses by subject and the teacher or learning user profiles. The *createGroupHelp* planning is mounted on agent runtime and has two goals, *createGroupByLearningProfile* and *createGroupByAffinity*. They form groups, but the difference is the assumption used to plan the best way and hence suggest ways to create or join groups, following a theme or profile of users who fall into the group to set. This choice shows two goals chosen to be reached during the execution time.

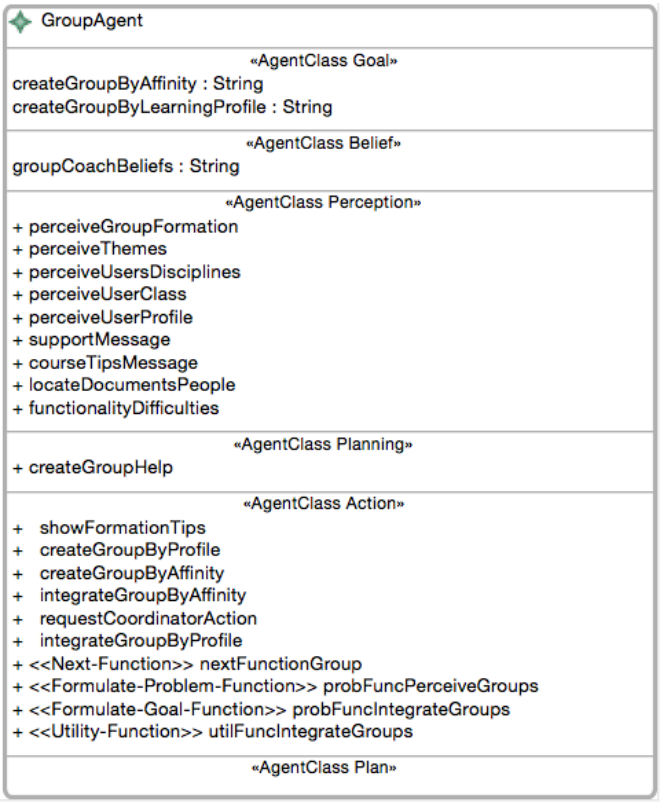

Figure 20: GroupAgent structure

Figure 21 shows the code generated for *GroupAgent* in JAMDER. As a utility-based agent, this has the methods defined in MAS-ML 2.0 and are determined by methods:

- <<next-function>> nextFunctionGroup: gives creation suggestions or integration groups;
- <<formulate-problem-function>> probFuncPerceiveGroups: recognizes the potential group formation according to a theme or suggested profile;
- <<formulate-goal-function>> probFuncIntegrateGroups: assists the user in forming the subject of thematic groups, thus giving greater value to the relationship of collaboration between the members of the group to provide better learning;
- <<utility-function>> utilFuncIntegrateGroups: creates a balance between training issues and profiles;

In this example of generated code, the actions were entered manually because the current version of MAS-ML tool still does not provide the inclusion of actions information on their structure. Namely: *checkAgentAction* and *requestAction*.

```
import jamder.behavioral.*;
...
import jamder.agents.*;

public class GroupAgent extends UtilityAgent {
   //Constructor
  public GroupAgent(String name, Environment env, AgentRole agRole) {
    super(name, env, agRole);
    addBelief("groupCoachBeliefsB", new Belief("groupCoachBeliefsB", "String", ""));
```

```
    Goal formGroupByAffinityG = new LeafGoal("formGroupByAffinityG ", "Boolean", "false");
    addGoal("formGroupByAffinityG", formGroupByAffinityG);
    Goal formGroupByLearningProfileG = new LeafGoal("formGroupByLearningProfileG ", Boolean,
        "");
    addGoal("formGroupByLearningProfileG ", formGroupByLearningProfileG);

    Action showFormationTipsAc = new Action("showFormationTipsAc", null, null);
    addAction("showFormationTipsAc", showFormationTipsAc);
    ...
    Action("integrateGroupByProfileAc", null, null);
    addAction("integrateGroupByProfileAc", integrateGroupByProfileAc);

    addPerceive("perceiveGroupFormation", null);
    addPerceive("perceiveThemes", null);
    ...
    addPerceive("funciontalityuDifficulties", null);

    Plan createGroupHelpPlan = new Plan("createGroupHelpPlan", null);
    addPlan("createGroupHelpPlan", createGroupHelpPlan);
  }

  protected Plan planning(List<Action> actions){
        return null;
  }

  protected Belief nextFunction(Belief belief, String perception) {
        return nextFunctionGroup (belief, perception);
  }
  private Belief nextFunctionGroup (Belief belief, String perception) {
        return null;
  }
  protected List<Action> formulateProblemFunction(Belief belief, Goal goal) {
        return probFuncPerceiveGroups(belief, goal);
  }
  private List<Action> probFuncPerceiveGroups(Belief belief, Goal goal) {
        return null;
  }
  protected Goal formulateGoalFunction(Belief belief) {
        return probFuncIntegrateGroups(belief);
  }
  private Goal probFuncIntegrateGroups(Belief belief) {
        return null;
  }
  protected Integer utilityFunction(Action action) {
        return utilFuncIntegrateGroups(action);
  }
  private Integer utilFuncIntegrateGroups(Action action) {
        return 0;
  }
  public void percept(String perception) { }
}
```

Figure 21: GroupAgent class in JAMDER

6.2. Agent Roles

In a MAS system, agents can exercise more than one agent role in the same organization, but this MAS prototype, each agent has a specific role. Because of this granularity, the definitions of agent roles are the same as agents regarding beliefs, goals and actions, but the roles have other components: rights and duties. Figure 22 shows more details of the proposed roles for the agent. The structure of each agent role proposed for each agent based on the rights and duties is defined as follows:

Learning agent role (*CompanionAgentRole*) is the type of knowledge-based role as it needs to store students' grades through beliefs. It contains the right *displaySupportMsg* , it may or may not display it and the duty to *compareClasses* as to perform some action needs to compare the class at first.

Pedagogical agent role (*PedagogicAgentRole*) is a kind of role for a proactive agent, because depending on how the student evolution is, the actions that the agent will use this role will make the agent's plan.

Searcher Information Agent Role (*SearcherAgentRole*) is a kind of proactive agent role because his actions will be used in a pre-defined agent plan. Their rights are *displayRelatedPeople* and *displayRelatedDocuments* to display people or documents relating to the current user interests. His *searchPeople* and *searchDocuments* duties identify the obligation to locate people or documents relating to the current user.

Moodle Helper Agent Role (*HelperAgentRole*) is a kind of simple reactive agent role, therefore, does not have beliefs or goals. It provides help in the operation of Moodle.

Forming Groups Agent Role (*GroupAgentRole*) is a kind of proactive agent role due to own beliefs and goals needed to assist users in the composition of working groups. This role has no duty but has several rights.

Figure 22: Part of Organization diagram with only proposed agent roles

Coordinator Agent Role (*CoordinatorAgentRole*) is a kind of proactive agent role. It centralizes the requests of the agents and the information they send to each other. Thus, their behavior is an intermediary between other agents. This role does not have duties, however, contains *checkAgentAction* right, that can check which action the agent is taking, and *requestAction* right, that may request another user to perform an action. Figure 23 shows all agent roles code generated in JAMDER proposed in this case study.

```
import jamder.structural.*;
...
import jamder.behavioral.*;

public class CompanionAgentRole extends ModelAgentRole {
   //Constructor
   public CompanionAgentRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);
   addBelief("companionBeliefs.pl", new Belief("companionBeliefs.pl", "String", ""));
   addRight("displaySupportMsg", new Right());
   addDuty("compareClasses", new Duty());
   initialize();
   }
}

import jamder.behavioral.*;

public class PedagogicAgentRole extends ProactiveAgentRole {
   //Constructor
   public PedagogicAgentRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);
   addBelief("pedagogicBeliefs.pl", new Belief("pedagogicBeliefs.pl", "String", ""));
   addGoal("assistStudentDiscipline", new LeafGoal("assistStudentDiscipline", "String", ""));
   addRight("suggestRelatedDiscipline", new Right());
   ...
   addRight("informDisciplineDescToStudent", new Right());
   addDuty("relateDisciplines", new Duty());
   addDuty("relateCourses", new Duty());
   initialize();
   }
}

public class HelperAgentRole extends AgentRole {
   //Constructor
   public HelperAgentRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);

   addRight("displayForumTips", new Right());
   addRight("displayMateTips", new Right());
   addRight("displayCalendarTips", new Right());
   initialize();
   }
}
```

Figure 23 AgentRoles proposed classes generated in JAMDER

6.3. Environment and Organization

Finally, the Organization and Environment classes for this MAS are shown in Figure 24. The structure of these classes is simpler because only contains entities that inhabit them and the instances that inhabit them.

The transformations of the code on models with charts in Acceleo proposed just generate the skeleton of classes and methods of the entities and their characteristics. Some entities required including, as a standard nomenclature, a suffix due to the possible existence of numerous properties in the entity and thus, organizing better its structure, namely: *Plan* (Plan), *Action* (Ac), *Goal* (G), *Belief* (B). This standardized names deletes only the *Duty* classes, *Right*, and protocol, as they are generated, their instances are created and are not directly referenced in the same method without the need for a variable. This nomenclature adopted provides a better understanding of the characteristics involved, giving the possibility of being used in the constructor of the entity more easily. Another advantage is that it helps to identify quickly what kind of class or variable was generated and prevents possible coding errors, improving the code quality.

```
import jamder.Organization;
...
import jamder.behavioral.*;

public class MoodleOrganization extends Organization {
   //Constructor
   public MoodleOrganization (String name, Environment env, AgentRole agRole, Organization
org) {
     super(name, env, agRole, org);
   }
}

import jamder.environment;
...
import jamder.agents.GenericAgent;

public class MoodleEnvironment extends Environment {
  public MoodleEnvironment (String name, String host, String port) {
    super(name, host, port);
    Organization MoodleOrganization = new Organization("MoodleOrganization", this, null);
        addOrganization("MoodleOrganization", MoodleOrganization);
        GenericAgent HelperAgent = new HelperAg("HelperAgent", this, null);
        AgentRole HelperAgentRole = new AgentRole("HelperAgRole", MoodleOrganization,
HelperAgent);
    addAgent("HelperAgent", HelperAgent);
    GenericAgent SearcherAgent = new SearcherAgent ("SearcherAgent", this, null);
    AgentRole SearcherAgentRole = new AgentRole("SearcherAgentRole", MoodleOrganization,
SearcherAgent);
    addAgent("SearcherAgent", SearcherAgent);

    GenericAgent CompanionAgent = new CompanionAgent ("CompanionAgent", this, null);
    AgentRole CompanionAgentRole = new AgentRole("CompanionAgentRole", MoodleOrganization,
CompanionAgent);
    addAgent("CompanionAgent", CompanionAgent);

    GenericAgent CoordinatorAgent = new CoordinatorAgent("CoordinatorAgent", this, null);
    AgentRole CoordinatorAgentRole = new AgentRole("CoordinatorAgentRole",
MoodleOrganization, CoordinatorAgent);
    addAgent("CoordinatorAgent", CoordinatorAgent);

    GenericAgent GroupAgent = new GroupAgent ("GroupAgent", this, null);
    AgentRole GroupAgentRole = new AgentRole("GroupAgentRole", MoodleOrganization,
GroupAgent);
    addAgent("GroupAgent", GroupAgent);

    GenericAgent PedagogicalAgent = new PedagogicAgent("PedagogicalAgent", this, null);
    AgentRole PedagogicAgentRole = new AgentRole("PedagogicAgentRole", MoodleOrganization,
PedagogicAgent );
    addAgent("PedagogicalAgent", PedagogicalAgent);
  }

  // Additional attributes and methods
}
```

Figure 24: MoodleOrg and MoodleEnv classes in JAMDER

In this study case, JAMDER was applied to the prototype of a real problem, illustrating the suitability of JADE extension for implementing entities in MAS-ML 2.0 context. The JAMDER source code and the full study case, as well as the templates used, can be obtained via the website https://bitbucket.org/yrley/jamder/src/.

7. Conclusion

The reduction of the semantic gap between the representations in design and implementation phases contributes to the increase of the developer's productivity and improves the consistency between design and code representations of the entities concerned. Additionally, the existence of a mapping between entities ensures consistency and traceability to code.

In this paper, we present a mapping between entities and the support mechanisms provided for JADE. The focus of the work is the different internal architectures of agents, namely: simple reflex agent, model-based reflex agent, a goal-based agent with planning, utility-based agent and goal-based agent with a plan. As a result, a JADE extension is presented through a set of adaptations to support the implementation of the scheduled architectures. In this way, some JADE classes have been extended because they share a common structure and behavior, such as: *Agent* and *Behavior*. In another case, new classes were created because there was no corresponding concept in JADE, namely: *Belief*, *Goal* (*CompositeGoal* and *LeafGoal*), *Plan*, *Action*, *AgentRole, ObjectRole*, *Axiom*, *Duty*, *Right*, *Environment, Organization* and *Condition*.

The applicability of the proposed extension is illustrated in a case study where the mapping on entity-level between modeling and its implementation through JAMDER code could be observed. This case study offered an explanation of a known system in the MASs field, the Moodle which facilitates the understanding of different types of agents and their implementation. The structural properties for the proposed agents were shown; however, their behavioral properties must be developed by the developer through the methods for each agent. Each template developed in this work is performed individually, one for each type of entity. As future work, we consider establishing a chain of calls for these templates sequentially that enables a reduction of time to generate the classes as well.

This work constitutes the step towards for promoting the development of MASs using at the same time, many agent types and JADE. Based on the presented mapping, the strategy to be followed in the process of coding can be traced. Furthermore, in this codification process, other frameworks for implementation could also be adapted to the variety of agents.

8. References

Bellifemine, F. L., Caire, G., and D. Greenwood, Developing Multi-Agent Systems with JADE, Wiley, 2007b. (Wiley Series in Agent Technology).

Beydeda, S., Book M., Gruhn, V. Model-driven Software Development. Birkhäuser, 2005.

Blois, M., Lucena, C. Multi-agents Systems and the Semantic Web - The Semantic Core Agent-Based Abstraction Layer. In: ICEIS, Porto. 2004.

Braubach L., Lamersdorf W., Pokahr A. Jadex: Implementing a BDI-Infrastructure for Jade agents. Exp, vol. 3, num 3. 2003.

Castro, J.; Alencar, F.; Silva, C. 2006. Agent-Oriented Software Engineering. In: Karin Breitman; Ricardo Anido. (Org.). Updates in Computers. Rio de Janeiro: PUC-Rio, p. 245-282.

Challenger M., Mernik M., Kardas G., Kosar T. Declarative specifications for the development of multi- agent systems, Computer Standards & Interfaces 43: 91-115. 2016.

Challenger M., Demirkol S., Getir S., Mernik M., Kardas G., Kosar T. On the use of a domain-specific modeling language in the development of multiagent systems, Engineering Applications of Artificial Intelligence 28: 111-141 (2014).

Demirkol S., Challenger M., Getir S., Kosar T., Kardas G., Mernik M. A DSL for the development of software agents working within a semantic web environment. Computer Science and Information Systems, 10(4): 1525-1556 (2013).
Fischer Klaus, Warwas Stefan. A Methodological Approach to Model Driven Design of Multiagent Systems. 13th International Workshop on Agent-Oriented Software Engineering XIII - Volume 7852.
Freire, Emmanuel Sávio; Gonçalves, Enyo José Tavares; Cortés, M. I.; Brandão, M. TAO+: Extending the Conceptual Framework TAO to Support Internal Agent Architectures in Normative Multi-Agent Systems. Electronic Notes in Theoretical Computer Science, v. 292, p. 57-69, 2013.
Fuentes-Fernandez R., Garcia-Magarino I, Gomez-Rodriguez A.M., Gonzalez-Moreno J. C. A technique for defining agent-oriented engineering processes with tool support Eng. Appl. Artif. Intell. 23 (3) (2010).
Gascuena J.M., Navarro E., Fernandez-Caballero A. Model-driven engineering techniques for the development of multi-agent systems Eng. Appl. Artif. Intell., 25 (1) (2012).
Gascuena J. M., Navarro E., Fernandez-Caballero A., Martínez-Tomas R.. Model-to-model and model-to-text: looking for the automation of VigilAgent. Expert Systems, 31(3): 199-212 (2014)
Getir S., Challenger M., Kardas G. The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems. International Journal of Cooperative Information Systems 23(3): 1-53 (2014).
Gómez-Sanz J. J., Fernández C. R., Arroyo J. Model driven development and simulations with the INGENIAS agent framework. Simulation Modeling Practice and Theory, 18(10): 1468-1482 (2010)
Gonçalves, E. J. T.; Cortés, M. I.; Campos G. A.; Gomes G. F.; Da Silva V. T. Towards the modeling reactive and proactive agents by using MAS-ML. In: the 2010 ACM Symposium, 2010, Sierre. Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10. v. 2. p. 936-937
Gonçalves, E. J. T., Farias, K., Cortés, M.I. and Silva, V.T., MAS-ML Tool: A Modeling Environment for Multi-Agent Systems. In: 13th International Conference on Enterprise Information Systems (ICEIS), 2011, Beijing, China. Proceedings of the 13th International Conference on Enterprise Information Systems, 2011.
Gonçalves E. J. T., Cortés M. I., Campos G. A. L., Lopes Y. S., Freire E. S. S., Silva, V. T., Oliveira K. S. F., Oliveira M. A. MAS-ML2.0: Supporting the modeling of multi-agent systems with different agent architectures. JSS (The Journal of Systems and Softwares) 108 (77-109), 2015.
Jennings, N. R. and Wooldridge, M., On Agent-based Software Engineering. Artificial Intelligence, v. 117, p. 277-296, 2000.
Nunes I., Lucena C., Luck M. BDI4JADE: a BDI layer on top of JADE. In: Ninth International Workshop on Programming Multi-Agent Systems (ProMAS 2011), 2011, Taipei. Ninth International Workshop on Programming Multi-Agent Systems (ProMAS 2011), 2011. p. 88-103.
Mellor S. J., Scott K., Uhl A., Weise D. MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley. 2004.
Morandini M., Nguyen C. D., Penserini L., Perini A., and Susi A.. Tropos modeling, code generation and testing with the Taom4E tool. CEUR Proceedings of the 5th International i* Workshop (iStar 2011)
Padgham, Lin and Winikoff, Michael. Developing Intelligent Agent Systems: A Practical Guide. ISBN 0-470-86120-7, John Wiley and Sons. (2004).
Pokahr A., Braubach L., Lamersdorf W. Jadex: A BDI Reasoning Engine Multi-agent Programming Languages, Platforms and Applications, Springer (2005), pp. 149-174.
Purvis M., Nowostawski M., Cranefield S. Opal: a Multi-level infrastructure for agent-oriented software development. In: Department of Information Science, Dunedin, New Zealand, University of Otago, p. 1-25, 2002.
Russell S. and Norvig P., Artificial Intelligence: A Modern Approach (International Edition). 2. ed. New Jersey: Pearson US Imports & PHIPEs, 2002.
Silva, V.T. da, Choren, R. and Lucena, C.J.P., MAS-ML: A Multi-Agent System Modeling Language. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA); In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications; Anaheim, CA, USA, ISBN:1-58113-751-6, ACM Press, pp. 304-305. 2007.

Santos, D. R.; Ribeiro, M. B.; Bastos, R. M. A Comparative Study of Multi-Agent Systems Development Methodologies. In: XX Simpósio Brasileiro de Engenharia de Software (SBES 2006). Florianópolis. p. 18-35 (2006)

Silva V., Garcia A., Brandão A., Chavez C., Lucena C. and Alencar P. Taming Agents and Objects in Software Engineering. In: Garcia, A.; Lucena, C.; Zamboneli, F.; Omicini, A; Castro, J. (Eds.), Software Engineering for Large-Scale Multi-Agent Systems, Springer-Verlag, LNCS 2603, pp. 1-26, 2003, ISBN 978-3-540-08772-4. 2003.

Weiss, G. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press, Massachusetts.1999.

Zambonelli F., Jennings N., Wooldridge M. Organizational abstractions for the analysis and design of multi-agent systems. In: Ciancarini, P.; Wooldridge, M. (Eds.) AgentOriented Software Engineering, LNCS 1957, Berlin: Springer, p. 127 (2001)