Designing Multi-Agent Systems from Ontology Models

Artur Freitas(✉), Rafael H. Bordini, and Renata Vieira

PUCRS, Porto Alegre, Brazil
artur.freitas@acad.pucrs.br, {rafael.bordini,renata.vieira}@pucrs.br
http://smart-pucrs.github.io/

Abstract. This chapter presents our proposal for the development of multi-agent systems designed as ontology models supporting code generation and reasoning. The foundation of such work takes into consideration ontologies for agent-oriented software engineering aligned with the JaCaMo framework. These techniques are implemented in a tool that supports multi-agent systems core code generation for JaCaMo. The underlying ontology also allows for reasoning about the multi-agent systems models under development. Such comprehensive approach, therefore, spans through the modelling, programming, and verification of agent-oriented software.

Keywords: Ontologies for agents · Reasoning in agent-based systems · Development techniques, methodologies, tools and platforms

1 Introduction

The design of complex systems, such as Multi-Agent Systems (MAS), should consider models that are clear to communicate, provide support during programming, and allow reuse and reasoning over the specification [6]. The use of modelling methodologies help us to understand complex problems and their potential solutions through abstractions. Thus, in this context, research investigating ontologies to support the modelling of MAS has been carried out [6,11,16]. Well-known MAS development frameworks, such as JaCaMo [1], integrate different technologies and languages for the design of MAS. In this chapter, we propose an ontology-based MAS development approach where a common basic language is used to present and specify a MAS, resulting in the integration of their different aspects and also serving for core code generation in JaCaMo [1].

It should be noted from the start that, although the general approach can be applied to any agent-oriented platform, the fact that there is not overall agreement on concepts and terms used in Agent-Oriented Software Engineering (AOSE), we need specific ontologies for each platform. While we here concentrate on the well-known JaCaMo framework [1], work on alignment with upper ontologies might in the future facilitate also the integration of different approaches to agent-oriented development.

D. Weyns et al. (Eds.): EMAS 2018 Workshops, LNAI 11375, pp. 76–95, 2019.
https://doi.org/10.1007/978-3-030-25693-7_5

An important contribution of agent-oriented programming as a new paradigm was to provide ways to help programmers in developing autonomous systems. For example, agent programming languages typically have high-level programming constructs which facilitate (compared to traditional programming languages) the development of systems that are continuously running and reacting to changes in the dynamic environments where such autonomous systems usually operate [1]. Agent-oriented paradigms are normally used to develop very complex systems, where not only are many autonomous entities present in a shared environment but also they need to interact in complex ways and need to have social structures and norms to regulate the overall social behaviour that is expected of them.

This chapter is organised as follows. Section 2 focuses on alternative modelling approaches for engineering MAS. Section 3 introduces the topic of programming such systems using JaCaMo. Section 4 presents our model-based techniques to support code generation for JaCaMo. Section 5 explores the issue of reasoning with ontology models. In Sect. 6 we discuss the results of an experiment that was conducted to evaluate the proposed framework. Section 7 concludes this chapter and highlights some research directions for future work.

2 Multi-Agent Systems Modelling Approaches

Current AOSE methodologies (such as Prometheus [12]) are usually deficient in at least one area of MAS development [15], such as agent internal design, interaction design, or organisation modelling. Also, currently we have separate approaches to address the modelling and programming of MAS, resulting in gaps and conceptual divergences in AOSE [6,7]. While JaCaMo [1] is a programming platform that uses three *different* formalisms for coding, Prometheus [12] is an agent modelling approach that does not apply or explore any formal (logic-based) representation as part of its technique. This work addresses issues stemming from those facts investigating an ontology-based model-driven engineering approach as an integrated global model of MAS characteristics, where ontology models support MAS verification and programming. Although the advantages of ontologies for agents are clear, few MAS platforms currently integrate ontology techniques [6,15]. Limited ontological support is provided by a number of existing AOSE methodologies since they do not incorporate ontologies throughout the entire development lifecycle nor consider ways in which ontologies can be used to account for interoperability and verification during design [15].

Several models and methodologies can be found in literature to formalise and define the processes of MAS design and implementation. For example, Prometheus [12] is one of the best-known MAS modelling methodology for developing intelligent agent systems. It defines a development process with associated deliverables for assisting developers to design, document, and build agent systems based on concepts such as goals, beliefs, plans, and events. The Prometheus [12] methodology encompasses three phases: *system specification*, *architectural design*, and *detailed design*. Among future work for Prometheus [12] there is the introduction of social concepts to improve its current models, however these improvements are not available yet in the latest official version of

the Prometheus Design Tool (PDT). Therefore, some aspects of MAS are not covered by the models of Prometheus, which also does not explore the use of formal or explicit ontologies as part of its approach. Ontologies for MAS are being proposed and investigated to support programming and reasoning over specifications, and they can also offer code generation features and help in organising the many concepts involved in the modelling, development, and verification of MAS. In this direction, ontologies have been considered in several different approaches in AOSE [6,11,16]. Ontologies are defined as knowledge representation structures composed of concepts, properties, individuals, relationships, and axioms.

It is possible to find in literature ontologies for the environments of MAS [11]. Environments play an essential role in MAS, and their semantic representation improves the way agents reason about the objects with which they interact and the overall environment where they are situated. This is important because most agent-oriented programming languages are weak in allowing the developer to model the environment within which the agents will execute [2]. The use of an environment ontology adds three important features to existing multi-agent approaches [11]: *(i)* ontologies provide a common vocabulary to enable environment specification by agent developers (since it explicitly represents the environment and agent essential properties, defining environments in ontologies facilitates and improves the development of multi-agent simulations); *(ii)* an environment ontology is useful for agents acting in the environment because it provides a common vocabulary for communication within and about the environment (it allows interoperability of heterogeneous systems); and *(iii)* environment ontologies can be defined in ontology editors with graphical user interfaces, making easier for those unfamiliar with programming to understand and design such ontologies.

Research on ontologies for MAS environments [11] had already foreseen the relationship between the environment and other MAS dimensions, since they mention the intention of looking at higher-level aspects of environments, i.e., social environment aspects of agents, such as the specification of social norms and organisations in agent societies. In fact, on the MAS organisation dimension, there is a semantic description of MAS organisations [16] to specify an ontology for organisational characteristics of the Moise meta-model. This approach helps agents in becoming aware, querying, and reasoning about their social and organisational context in a uniform way, making possible to convert between ontology and Moise specifications, thus providing more flexibility for modelling and developing in this domain. This semantic description of Moise [16] provides other benefits such as increased modularisation, knowledge enriching with metadata, reuse of specifications, and easier integration. With the semantic web effort aiming to represent the information in semantic formats, the MAS community can take advantage of these new technologies in AOSE development tasks such as to integrate organisational models, to monitor organisations, and to analyse agent societies [16].

Next section introduces the JaCaMo as a unified programming framework for these MAS characteristics recently discussed.

3 Programming in JaCaMo

MAS programming in JaCaMo [1] requires the development of code in Jason [3], CArtAgO [14], and Moise [9]. Jason [3] is an AgentSpeak language implementation that focuses on agent actions and mental concepts and provide to programmers features such as speech-act based agent communication, plan annotation, architecture customisation, distributed execution, extensibility through internal actions, among other functionalities. On the environment side of agent systems, CArtAgO [14] is a platform to support the artifact notion in MAS. Artifacts are function-oriented computational devices which provide services that agents can exploit to support their individual and social activities [14]. Lastly, the specification of agents at the organisation level can be achieved using an organisation modelling language, such as Moise [9]. Moise explicitly decomposes the specification of an organisation into its structural, functional, and normative dimensions.

JaCaMo resulted from one of the earliest approaches aimed at explicitly investigating the integration of all the dimensions of MAS from a design and programming point of view. Most previously existing approaches had considered either only the agent-organisation dimensions, or the agent-environment dimensions [1]. The combination of these dimensions of MAS into a single programming paradigm with a concrete working platform has a major impact on the ability to program complex distributed systems. The authors of JaCaMo pointed out, as future work, the desire for an Integrated Development Environment (IDE) to facilitate the process of design, development, and execution of JaCaMo applications, potentially reusing and integrating existing Jason, CArtAgO, and Moise tools and technologies [1]. Thus, recognising the importance achieved by JaCaMo, this research direction is one of the motivations in this chapter.

JaCaMo is one of the few fully operational platforms combining all three dimensions of MAS, to the best of our knowledge, and arguably one of the best-known (e.g., given it is highly cited). Thus, our proposed techniques for modelling and code generation address the design of MAS with an eye on implementations using JaCaMo as the target programming platform specifically. However, as noted earlier, the overall approach could also be recreated for other agent development platforms as well. Other frameworks for MAS development provide some support for environments, or some organisational notions such as roles, but without including a fully-fledged organisational model and first-class environment abstractions that are provided by JaCaMo.

4 Code Generation Techniques for Multi-Agent Systems Designed as Ontology Models

In our work we present two different techniques for code generation based on models specified using an ontology of MAS obtained from the literature [5,6]. One technique is the iterative drag-and-drop of elements from ontology to transform them into the different parts of code that compose a JaCaMo project: Jason, CArtAgO, Moise, or the `jcm` file. The other technique is the automatic generation

of the initial files and code of JaCaMo projects that match the ontology-specified content. Both techniques are implemented in our tool called Onto2JaCaMo. Our work employs the ontology of MAS obtained from [5,6] as the basis for the code generation techniques, and we refer to it as OntoMAS. For details about the ontology, we refer to its references, so that we can focus here on its applications. When using an ontology for modelling MAS, the underlying idea is that the MAS project conception should start by its modelling as an ontology. This is done by extending the ontology top-level concepts, and adding new classes, instances and relationships in order to specify the corresponding desired project to be implemented in terms of agent-oriented concepts [6].

In OntoMAS, a particular MAS begins to be modelled by *extending* the ontology, which is done by creating new subclasses to its top-level concepts. Then, individuals are created in the process of *instantiating* the extended ontology. From an instantiated model, it is possible to perform *reasoning* and obtain an inferred specification, which can be explored for *verification* purposes such as, for example, in model checking approaches. Then, a model specified using OntoMAS can be used in our techniques for supporting MAS *programming*, which are incorporated into the Onto2JaCaMo tool. Such an approach also allows designers to gradually refine from high-level abstract views to elements directly available in concrete features of MAS programming platforms. The designers may apply the desired level of completeness in their models, which will later result in a code with a corresponding detailing. Figure 1 illustrates how OntoMAS and Onto2JaCaMo fit in the phases of AOSE in the proposed methodology. Currently, an ontology editor tool, such as Protégé [10], should be used to interact with OntoMAS during the MAS modelling. OntoMAS is currently formalised in OWL (Web Ontology Language), which is a computational logic-based Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. Future research, besides, could consider new languages to be used for OntoMAS if they offer some sort of advantages in terms of knowledge representation and reasoning.

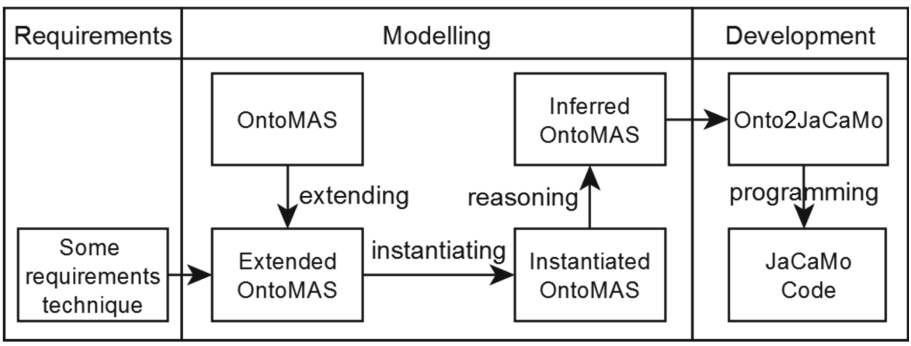

Fig. 1. Methodology using OntoMAS and Onto2JaCaMo.

4.1 Mapping Elements from the MAS Ontology to JaCaMo Code

Initially, lets make a mapping of where elements from OntoMAS [5,6] are usually found in a JaCaMo project. There are concepts to deal with the *Agent Dimension* with a clear relation to Jason (such as *Agent*, *Plan*, and *Belief*), concepts to deal with the *Environment Dimension* to establish a relation with CArtAgO (such as *Artifact*, *Space*, and *Operation*), and concepts to deal with the *Organisation Dimension* to address Moise specifications (such as *Group*, *Role*, and *Norm*).

From the agent dimension, we are not interested in defining any possible and generic characteristics of any kind of agent, such as physical agents. Instead, we are interested in specifying only the concepts of virtual agents that make sense in the context of programming for this dimension. Thus, the OntoMAS ontology contains the following 6 top-level concepts to represent the agent dimension: *Agent*, *Plan*, *Action*, *AgentGoal*, *Belief*, and *Message*. Figure 2 summarises the main concepts, subclasses and properties in the agent dimension of OntoMAS.

As already mentioned, the use of OntoMAS ontology sometimes requires to create subclasses that specialise the given top-level domain concepts. A subclass of *Agent* represents a type of agent, such as for example, *Player*. When defining a given concept as a subclass of *Agent*, this concept represents all individual agents of that kind. Subclasses of *Agent* are usually found in JaCaMo as the `.asl` files. An instance of a subclass of *Agent* represents an individual agent of that corresponding type, such as for example *playerJohn*. Instances, such as *playerJohn*, are usually found in JaCaMo as individual agents defined by an `agentID` in the `.jcm` file.

A *Plan* is a procedure composed of actions and it is triggered inside agents. The definition of each plan should be represented as an instance of the *Plan* concept. Thus, instances of plans represent the specification of a plan, such as for example *chooseMovement*. The specification of a plan is found in JaCaMo inside the `.asl` code of the type of agent that contains such plan. From this modelling perspective adopted in OntoMAS, the designer does not need to create subclasses of *Plan*, but this possibility is allowed. There are classes in this dimension that can be applied just by creating instances, which we argue that is the most simple way. However, the modeller is allowed to create subclasses to achieve an additional layer of expressiveness.

There are two kinds of *Actions* represented in OntoMAS: *ExternalAction* and *InternalAction*. An *ExternalAction* is what the agent does that affects the environment, such as the act of opening a door. An *InternalAction* is how an agent act to manipulate its mental state, for example, forgetting some belief. While internal actions may be defined by local actions in the agent's state, external actions may refer to performing operations of artifacts that are situated in an environment. The definition of an action is represented by creating an instance of *Action*, such as for example *openDoor*. Actions are usually found in JaCaMo in the body of agents' plans. Similarly with plans, the designer does not need to create further subclasses of *Action*, but this possibility is allowed. For example, the subclass *openDoor* could have two different instances according to different door handles, *openDoor-barhandle* and *openDoor-knobshandle*.

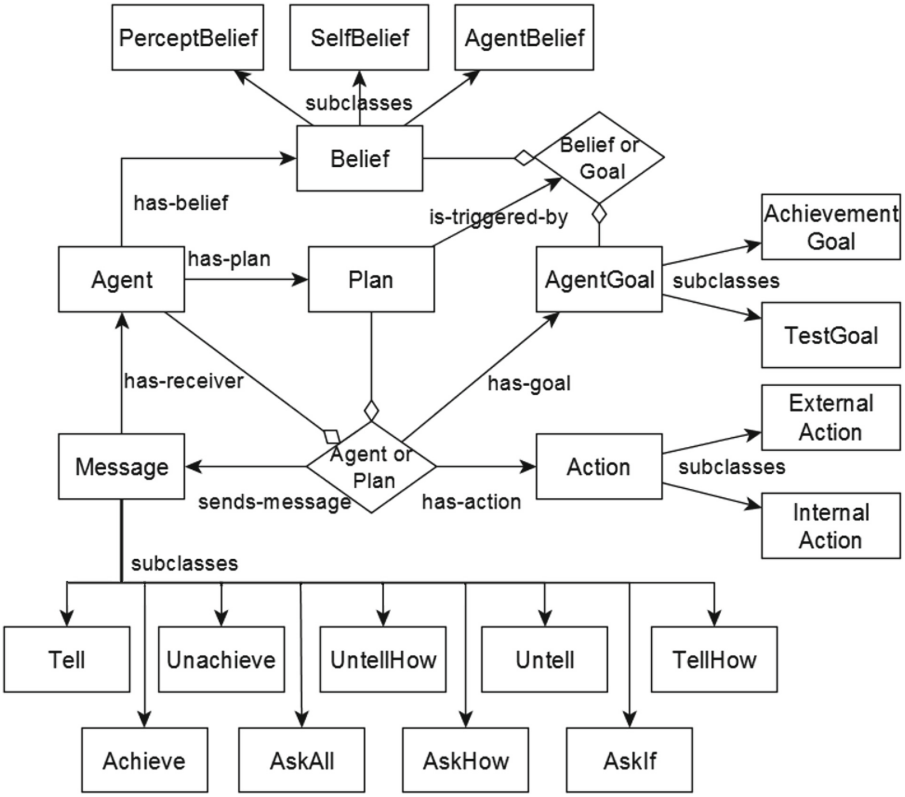

Fig. 2. Concepts, subclasses and properties in the agent dimension of OntoMAS.

An *AgentGoal* represents some agent individual desire to be achieved. Goals can be in one of the two following types. An *AchievementGoal* represents a state of the world (objective) that an agent can have intention to attain, such as having the door opened. A *TestGoal* is a check on the agent's beliefs in order to verify if a given belief holds, for example, querying the belief about the door being closed. Both achievement and test goals may fail, but for any plan that is using them in order to continue its execution and finish with success, its goals must be completed. The definition of a type of goal that agents may pursue is represented by creating an instance of *AgentGoal*, such as for example to achieve *doorOpened*. Goals are usually found in JaCaMo inside agent code (`.asl` files).

The *Belief* encodes the knowledge of agents, which can be one of the three types, as follows. *PerceptBeliefs* are obtained from environment perception, for example, the belief *stoveLit* to represent the state perceived from a device. *AgentBeliefs* are beliefs obtained from some other agent, for example, when an agent is told by other about something. *SelfBeliefs* are obtained by internal agent reasoning, for example, when an agent believes in something but not because it was perceived from the environment nor it was told by other agent. The definition

of a type of belief is specified by creating an instance of *Belief*, such as for example $preferredMove$, which can be a *SelfBelief*. Beliefs are usually found in JaCaMo inside the code of agents (`.asl` files).

A *Message* is a communication that goes from one agent to another. The definition of a type of message is represented by creating an instance of *Message*, such as for example $informLocation$. Sending a *Message* may be a part of a plan in agents. The message types correspond to which performative is part of the *sender* agent's intention, for example, if it is delegating a goal (*AchieveMessage*), informing a belief (*TellMessage*), requesting a plan (*AskHowMessage*), etc. There are 9 different types that a message can assume, each representing its illocutionary force, all of them depicted in Fig. 2 as subclasses of *Message*.

After explaining the concepts illustrated in Fig. 2, we can now discuss other topic depicted there, namely the properties that take place in relationships among concepts of the agent dimension. Plans may contain actions, which means that when a given plan is being executed, its corresponding actions may be performed. This is represented by connecting instances of these concepts using the *has-action* property, for example, $chooseMovement$ *has-action* $openDoor$. The same is true for plans that may start the pursue of goals, defined through the property *has-goal*, as exemplified by $chooseMovement$ *has-goal* *doorOpened*. Also about plans, they may be triggered by an event involving a belief or a goal, which is given by the property named *is-triggered-by*. To indicate that a given plan sends a specific message, the *sends-message* property may be used. There is no need to specify for agents the *has-action* and *sends-message* properties if they were all specified for plans, a general rule can make inferences to check if an agent contains plans that have actions and send messages, in such case the agent will also present these properties too. We refer to Sect. 5 for more details about rules and reasoning over OntoMAS models.

Some properties work with the concept of *Agent* as its domain or range. We have explained that the *Agent* concept may have both subclasses (e.g., *Player*) and instances (e.g., $playerJohn$). When it is desired to use a property to connect between instances, the semantic is the same as explained in the previous paragraph. For example, agents may have beliefs, as expressed by the $has\text{-}belief$ property. If $playerJohn$ has some belief, lets call $preferredMove$, then these instances have to be connected using the mentioned $has\text{-}belief$ property. However, if all agents of that type (*Player*) have such belief, then a "subclass of" restriction should be used in that concept. This is represented as: *Player* is a subclass of $has\text{-}belief$ *value* $preferredMove$. The same principles are applied to: the *has-goal* property, which indicates the goals of agents; the *has-plan* for indicating the plans of agents; and the *sends-message* property, which indicates which messages the agent sends. To connect an instance of message with an instance of agent that should receive it, the property *has-receiver* can be applied (e.g., $informLocation$ *has-receiver* $playerJohn$).

We point out to reference [5] for further details on the ontology meta-model that are not tackled here in this chapter. However, we next briefly explain the dimensions of environment and organisation, but not with the same level of detail employed above to show the concepts and properties of the agent dimension.

Each subclass of *Artifact* is found in CArtAgO as a Java class, and instances of *Artifact* subclasses represents an object/tool/resource of that type, which may be found in JaCaMo in the `.jcm` file that describes the initial artifacts of a system; however, other artifacts instances may be created after the initialisation of the MAS. *Spaces* are initialised in the JaCaMo project file, but agents may make reference to spaces in their code too. *Operations* are found in CArtAgO as methods of the artifact that implements such procedures. Instances of *Percept* (*ObservableProperty* or *ObservableEvent*) are found in the Java code of artifacts through methods provided by the CArtAgO API to manipulate them (such as *defineObsProperty*, *getObsProperty*, *updateObsProperty*, and *signal*).

Subclasses of *Group* can be found in the XML that specifies an organisation in Moise, and their instances take place in the JaCaMo project file, as well as in the code of agents in Jason that can make references to groups (e.g., `join_group`). Instances of *Role* are found in the Moise XML file, and the code of Jason agents can make reference to such roles too (e.g., `adopt_role`). Instances of *OrganisationGoal* are also found in the Moise file, and the code of agents in Jason can make references to those goals (for example, agents may have plans to act when a goal is assigned to them by the organisation). Lastly, instances of *Missions* and *Norms* are defined in the Moise XML file of a JaCaMo system.

The classes and properties in OntoMAS are modelled in three sub-ontologies, one for each dimension: agent, environment, and social organisation. The integration and connections among concepts in the dimensions of OntoMAS are encoded by means of concepts, object properties, and rules which determine how elements are allowed to relate among each other. To illustrate, in order to specify the location of agents' instances in spaces from the environment dimension, the property *is-in* may be used, such as, for example $playerJohn$ *is-in* $classRoom$ (considering $classRoom$ an instance of *Space*). The property $is\text{-}focused$ connects an agent with an instance of artifact in which that agent is focused, such as $playerJohn$ $is\text{-}focused$ $homeComputer$ (considering $homeComputer$ an instance of $Artifact$). Then, some properties may be obtained by inference over elements from different dimensions. If an agent ($?a$) is in a space ($?s$), and this space provides some percept ($?p$), then this agent can have such percept ($?a$ *can-perceive* $?p$). This is specified through the following rule:

$$\textit{is-in}(?a, ?s), \textit{provides-percept}(?s, ?p) \text{ –> } \textit{can-perceive}(?a, ?p).$$

As another example to illustrate important things to represent in the ontology, when relating concepts from the dimensions of agent and organisation, we may desire to specify that a given agent is adopting a role. This may be done with the property *adopts-role*. If a characteristic affects only some individuals of a group, then it should be defined as an object property in those affected instances. In this case, for example, supposing $redSoccerTeam$ as an instance of the *Group* concept, if the $redSoccerTeam$ contains the $playerJohn$ agent, then these instances should be related using the object property *contains-agent*.

4.2 Drag-and-Drop Transformation Technique from the Multi-Agent Systems Ontology to JaCaMo

The idea of using an ontology for providing drag-and-drop operations from models to code in JaCaMo has been already mentioned in literature [5,6]. In this chapter we explain how the elements of an ontology model can be dragged to generate code for the different parts of JaCaMo, such as Jason, CArtAgO, Moise, or the JaCaMo project file that defines the specification that initialises the corresponding system. Each element from an ontology model can be transformed in MAS code in several different ways.

To exemplify the drag-and-drop conversions, let us take a look at how instances of the *ObservableProperty* concept may be employed in the code of each of the different parts of JaCaMo. Suppose there is an instance of *ObservableProperty* called *temperature*, defined at the *Environment Dimension*. If a programmer makes a drag-and-drop of *temperature* in this dimension, a code automatically created as suggestion may be to update the value of such observable property. Thus, the following code can be created:

```
getObsProperty(temperature).updateValue(newValue);
```

In Jason, making a drag-and-drop using this same instance of *ObservableProperty* may give origin to a plan triggered by the observation of such property:

```
+temperature : true <- planBody.
```

However, if dropped in the middle of a plan, then just the corresponding belief identified by *temperature* is generated. When a JaCaMo system is running, the observable properties provided by environmental artifacts become beliefs to agents that are focusing on those artifacts, and when they become beliefs, some plans may be triggered by the belief addition event. Instances of observable properties are not applicable for drag-and-drop code transformations in the case of Moise or JaCaMo project file. We have summarised the information about the drag-and-drop operations provided by Onto2JaCaMo for transforming from the ontology to JaCaMo code in Table 1. This table shows the generation when the desired outcome is the *Agent Dimension* of JaCaMo (i.e., Jason). Similarly, there are strategies to convert the ontology to the *Environment Dimension* of JaCaMo (CArtAgO), to the *Organisation Dimension* (Moise), and also to the initialisation setup of JaCaMo (the `.jcm` file). However, the tables illustrating these other mappings were not included in this chapter for the sake of space (see reference [5] for further details).

4.3 Core Code Generation Technique from OntoMAS to JaCaMo

The technique proposed in this subsection is related to the idea of using an ontology for the automatic generation of skeleton code for each of the JaCaMo languages. Elements from an ontology of MAS should have their resulting code counterparts in Jason, CArtAgO, and Moise. Therefore, it would be possible

to directly transform an ontology-based MAS specification into initial code for JaCaMo. While when using drag-and-drop programmers are iteratively transforming elements from their ontology model into code, this code generation technique uses another perspective, which is to generate an initial structure of a corresponding project in JaCaMo to what is specified in the ontology model.

The generation of initial agent files and code for Jason considers mainly the subclasses and instances of the *Agent Dimension* of the employed ontology. For example, each subclass of *Agent* becomes an `.asl` file with its corresponding plans, actions, goals, beliefs, and messages. However, characteristics defined at other dimensions, such as the environment, although not directly applicable to generate the initial code at the agent level, may be considered to suggest implementation alternatives for programmers (at least for them to be aware of). For example, for an agent that is expected to receive a given percept, a plan triggered by the addition event of that percept may be suggested as a situation that programmers are likely to have to handle.

Similarly, the initial files of the CArtAgO part of a JaCaMo project derive mainly from the *Environment Dimension* of the ontology in use, and the Moise initial code is generated based on the *Organisation Dimension*. Subclasses of *Artifact* become the Java files with their corresponding operations as methods, and observable properties are initialised. All the organisation elements (subclasses, instances, and relationships) are considered in the generation of the initial XML file of a Moise organisation. Lastly, the JaCaMo project file considers characteristics from all the three dimensions, and relationships from their integration.

To exemplify the initial project generation, consider the *ObservableProperty* instance used in previous examples, *temperature*. If it is said that an artifact type (e.g., *computer*) has this property, then such observable property definition must compose the *init*() method of the *computer* artifact class in the format:

```
defineObsProperty("temperature", initialValue);
```

Considering Jason, Moise, or `.jcm` files, instances of observable properties are not directly applicable for the automatic code generation in this case. However, a plan triggered by the addition event of the related observable property could be suggested to the agents' programmers as a situation worth to be handled.

How each element from OntoMAS models can be transformed into the initial structure of files and code for Jason is shown in Table 2. This same principle is applied to CArtAgO, Moise, and the project file, albeit, it is not possible to illustrate all those tables in this chapter. However, we have complete definitions for OntoMAS models as starting point to generate skeleton code for each part of JaCaMo programming (Jason, CArtAgO, Moise, and the `.jcm` project file).

The so-called core code generation technique presented in this subsection creates the first skeleton code for a JaCaMo project that was modelled using the OntoMAS ontology. The drag-and-drop technique is a way to complement and iteratively evolve the programming of such systems. Compared to a fully hand-written code, developers would lack tools that could provide means to integrate the modelling and programming of their MAS.

Table 1. Drag-and-drop code generation for Jason (*Agent Dimension*) from ontology elements.

Instance of the Ontology Element	Drag-and-Drop Code for Jason	Explanation
Agent Dimension		
ExternalAction	`actionName();`	an external action invocation inside a plan's body representing an agent acting in the environment.
InternalAction	`.actionName();`	an internal action invocation inside a plan's body representing an action that an agent performs mentally.
Agent	`agentName`	the identification of the individual agent in order to send messages, or perform some other tasks.
Belief	`+beliefName[source(value)];`	a belief addition event with source's value defined by the belief's subtype: self, percept, or other agent.
AchievementGoal	`!achievementGoalName;`	an initial goal for that agent; or a goal that has to be achieved during the execution a plan.
TestGoal	`?testGoalName;`	a goal that has to be tested during the execution of a plan.
Message	`send(receiver,illocutionaryForce, propositionalContent);`	the act of sending the corresponding instance of message
Plan	`is_triggered_by : true <- actions; goals.`	a plan with its triggering condition, context (that by default is true), and a body composed (mainly) of actions and goals.
Environment Dimension		
Space	`joinWorkspace("workspaceName");`	an action for that agent to join the corresponding workspace.
Artifact	`focus(artifactName);`	the action of focusing on that instance of artifact.
Operation	`operationName();`	the invocation of the corresponding operation in the body of a plan representing the execution of that operation by an agent.
ObservableProperty	`+propertyName : true <- planBody.`	a plan triggered by the observation of the corresponding property.
ObservableEvent	`+eventName : true <- planBody.`	a plan triggered by the observation of the corresponding event.
Organisation Dimension		
Group	`join_group(groupName);`	an action of joining in the given group.
Role	`adopt_role(roleName,groupName);`	the action to adopt the given role in the specified group.
Mission	`commit_mission(missionName, schemeId);`	an action to commit with the instance of mission in the given scheme.
Norm	`+normName: true <- planBody.`	a plan triggered by the perception of the given norm.
OrganisationGoal	`+!goalName : true <- planBody.`	a plan triggered by the addition event of the specified goal.

Table 2. Template core code generation for Jason (*Agent Dimension*) from ontology elements.

Instance of the Ontology Element	Base Code for Jason	Explanation
Agent Dimension		
Agent (subclass)	`agentSubclass.asl` file	a type of agent that contains all related elements such as plans, goals, and beliefs.
Belief	`beliefName[source(value)]`	an initial belief in the corresponding .asl file with the source's value defined by the belief's type (self, percept, or other agent).
AchievementGoal	`!achievementGoal.`	an initial achievement goal in the corresponding .asl file.
Message	`!sendMsgName <- .send(receiver,illocForce, propositionalContent);`	a plan to send the corresponding message. A plan for the receiver agent may be created using as triggering condition the given propositionalConcent and illocutionaryForce.
Plan	`+is_triggered_by : true <- actions, goals.`	a plan in which it has a triggering condition, a context, and a body composed (mainly) of actions and goals. The plan is inserted in the `.asl` file of the agent type that has it.
ExternalAction, InternalAction, TestGoal, Agent	not directly applicable for automatic code generation for Jason.	
Environment Dimension		
ObservableProperty	`+propName : true <- planBody.`	it is not directly applicable for automatic code generation. However, a plan triggered by the addition event of the related observable property may be suggested as a situation that is likely desired to be handled.
ObservableEvent	`+eventName : true <- planBody.`	it may be suggested (however, it is not mandatory) a plan triggered by the observation of the corresponding event.
Space, Artifact (instance and subclass), Operation	not directly applicable for automatic code generation for Jason.	
Organisation Dimension		
Group (instance and subclass), Role, Mission, Norm, OrganisationGoal	not directly applicable for automatic code generation for Jason.	

4.4 The Onto2JaCaMo Tool for Multi-Agent Systems Development

For effective and efficient software development, preferably all tasks and activities during the development process should be adequately supported by tools [13]. The quality of any software tool support can be assessed by considering the degree of support for the different phases and tasks [13], e.g., design tools, which besides the creation and editing of design models also often support consistency checking and/or code generation.

We have implemented the techniques previously explained in Subsects. 4.2 and 4.3 in a software tool to support MAS development, which we refer to as Onto2JaCaMo. It consists of a plug-in for Eclipse that loads instantiated models based on the ontology of MAS obtained from [5,6] to provide code generation for JaCaMo. Eclipse [4] is an open source software development project that provides an IDE in which a basic unit of function, or a component, is called a plug-in. Eclipse is already the standard IDE for JaCaMo development, and it was indeed an interesting choice since Eclipse is recognised as a mature IDE, and one of the most widely used by programmers [13].

Onto2JaCaMo is easily installed by adding its `.jar` file in the Eclipse plug-ins folder. It can be activated to appear visually in the graphical interface of Eclipse by following this sequence: $Window \rightarrow Show\ View \rightarrow Other... \rightarrow JaCaMo\ Ontology \rightarrow Ok$. When it is enabled, Onto2JaCaMo requests to be informed about the OWL file corresponding to an instantiated ontology so that it can be loaded. The plug-in was designed to be used in the "JaCaMo Perspective" of Eclipse (or related perspectives, such as Jason). The tool loads OWL ontologies and provides three model-based programming features to generate MAS code: drag-and-drop, conversion from ontology to code, and auto-complete from instantiated ontologies.

In the drag-and-drop functionality, the developer can visualise and navigate through the ontology concepts, instances, and properties. These elements from the model can be dragged to the code in files being edited in Eclipse. For example, the programmer may perform a dragging and dropping operation using the action `pass_ball` to be inserted in a plan of agents of type "player". Similarly, it is possible to provide developers the auto-complete feature from ontology to agent code, which is activated when the developer is typing MAS code (or press the auto-complete shortcut: "ctrl + space"). Then, the available options based on the ontology are presented to programmers as suggestions. One example is when coding the plan's context, which may be composed of ontology-based queries (e.g., verifying if an individual belongs to a concept).

The Onto2JaCaMo tool is able to generate code fragments based on design information, which is known as forward engineering [13]. This is in the opposite direction of extracting design information out of existing application code, the so called reverse engineering. A drawback of forward or reverse engineering techniques is that after a generated artifact has been changed manually, forward or reverse engineering cannot be reapplied without losing the changes, as stated in the called "post editing problem". The combined support of forward and reverse engineering, such that changes in one artifact can always be merged

into the other without compromising consistency or losing changes, is referred to as round-trip engineering [13]. The Onto2JaCaMo tool presented in this work does not address such advanced and complex concepts of synchronisation as yet; these are, however, interesting topics for future work.

5 Ontology-Based Reasoning Support for Agent Systems

Model verification refers to the processes and techniques that the model developer uses to ensure that their model is correct and matches any agreed-upon specifications and assumptions. OntoMAS can be explored with its available reasoning mechanisms to implement model verification in the context of MAS. The literature reports that most practical approaches for verification of MAS are done on code, and most of the work done on model checking within the MAS research area is quite theoretical [2]. However, there are approaches that use model checkers typically to verify properties of particular aspects of a given MAS. While this has the advantage of proving properties of systems that will be deployed, it is also often useful to check properties during systems' design.

Considering this context, semantic reasoners may provide, for example, consistency checking and inferences about the MAS specified as an ontology. Ontologies empower the execution of semantic reasoners that provide functionalities such as *consistency checking*, *concept satisfiability*, *classification*, and *realisation*. In other words, reasoners are able to automatically infer logical consequences from a set of axioms. The possibility to reason about the model can provide support for various consistency checks during the MAS project design and implementation. For example, when considering only MAS organisations, it is possible to check for conflicts considering the existing norms, roles, and missions. When an instantiation of MAS organisation is combined with instantiated agents, it is possible to check for other kinds of inconsistencies integrating information from more than one dimension, such as whether the agents contain the required capabilities to achieve the existing organisation goals. Organisation goals are assigned to agents playing the organisation roles, and an agent playing a specific role may not have the required plans to achieve the goals that the organisation will assign to it. Reasoning can be applied also to verify consistency among the norms in the organisation. The combination of some norms can result in contradictions, for example, when a prohibition occurs together with an obligation or permission. These contradictions can appear when considering the missions of just one isolated role, or when combining the missions of two or more roles.

When analysing the knowledge about the environment, it can be checked whether agent actions are valid in a given environment configuration. If there is an agent action that does not exist in the environment, the invocation of such an action in run-time will result in failure. Thus, the verification of characteristics over an instantiated model at design time may prevent future errors to happen during the execution time of the corresponding JaCaMo specified project.

The use of ontology enables the creation of *rules*, which can be coded in the Semantic Web Rule Language (SWRL). Such rules can be inherited from the base

OntoMAS ontology, and new ones can be added specifically for an extension and instantiation of OntoMAS, when defining a desired MAS scenario. All elements in the ontology are taken into consideration when semantic reasoners are executed for making inferences. For example, one general rule is that if an agent `a` is in a space `s`, and this space `s` can provide an observable property `p`, then it can be inferred that the agent `a` is able to perceive `p` if it chooses to do so. This rule is coded as follows:

```
is-in(?a,?s), provides-percept(?s,?p) -> can-perceive(?a,?p).
```

In such reasoning mechanism it is possible to relate elements from any dimension (e.g., agent) with elements from other dimensions (e.g., environment). Lets suppose now a more complex example for inferences about a modelled MAS. We already commented that agents join organisations by playing organisation-defined roles, and it is expected that such agents have in their codes the required plans to handle the goals that the organisation may assign to them. Organisation goals are assigned to agents, for example, if there is an obligation norm on that role, and an agent that adopts such role should have a plan for achieving that goal. Lets represent this with a new property to specify that `Agents should-have-plan-for Goals`. This can be inferred, for example, if there is an obligation norm `n` that targets a role `r`, and there is an agent `a` that adopts the role `r`, then, the conclusion is that the agent `a` should have a plan for the goal `g`, where `g` is a goal from mission `m`, which is the mission for the obligation norm `n`. The following rule exemplifies how to make this inference:

```
ObligationNorm(?n), targets-role(?n,?r), adopts-role(?a,?r),
targets-mission(?n,?m), has-goal(?m,?g) ->
                                should-have-plan-for(?a,?g).
```

As we have exemplified using some rules in this subsection, more complex information can be incrementally inferred from the basic conceptualisation proposed by OntoMAS. Also, it allows extensions to be made on top of it, by including for example new concepts, properties, and so on.

As another example, it can be inferred which operations and percepts can be obtained from each space based on which artifacts are situated in it (the concept of *Space* from the ontology refers to the called *Workspaces* of CArtAgO). A rule may be used as follows:

```
contains-artifact(?s,?a), provides-percept(?a,?p) ->
                                provides-percept(?s,?p).
```

This rule can be read as: if the space `s` contains an artifact `a`, and `a` provides a percept `p`, then `s` provides `p`. The same reasoning principle applies to operations from artifacts that are located in some space. Moreover, another general rule about environments is that the percepts and operations of sub-spaces are also provided by the spaces that contain them.

6 Evaluating Onto2JaCaMo

Our initial evaluation of Onto2JaCaMo indicates that it facilitates coding in JaCaMo, mainly for beginners or for those who are not fully aware about how to implement some agent concepts. Users have reported that it improves the understanding about the operation of JaCaMo and how to program particular behaviours. Also, Onto2JaCaMo helps avoid syntactical errors as it provides code templates, which is important because the auto-complete shortcut from Eclipse ("ctrl + space") does not work in all JaCaMo extensions. Thus, more agility can be obtained in JaCaMo code generation. Lastly, during development, it is interesting to visualise the system's ontology, so that the idea defined in models may be followed easier when programming. Most importantly, it avoids some of the most common types of bugs made by programmers such as mistyping names since now the ontology provides the vocabulary to be used in the code.

Before starting our experiments regarding the evaluation of the programming techniques implemented in Onto2JaCaMo, the participants received the required prior instructions on these topics in order to perform the tasks with the minimum required knowledge, such as, for example, how to load and how to use ontology models in Onto2JaCaMo. The participants received the Onto2JaCaMo plug-in, where they had to load their previously instantiated ontology models and use the tool to support the model-based development of their agent code. Each participant had previously defined their own application scenario to work with. After finishing the programming of their MAS using the drag-and-drop provided by Onto2JaCaMo, the participants were surveyed by means of questionnaires to extract their perceptions and opinions about the techniques and tool according with statements that followed a 5-point Likert scale. Some criteria have received only positive evaluations from all participants, such as that Onto2JaCaMo is easy to understand, provides coding support, offers advantages for programming, and enables a better understanding of JaCaMo.

Thus, we observe that the proposed plug-in helps in code consistency (e.g., it facilitates coding using the same terms), and it prevents developers from using terms outside the ontology-based model. In summary, such approach provides an overview about agent systems to be visualised within the programming context, combined with features of dragging content from models to MAS code. As lessons learned from our practical experiences, we have observed that more MAS code could be generated from the proposed modelling approach, and that the ontology could be used in a technique to constrain the MAS coding (i.e., to indicate errors or mismatches between model and code). Also as future work, we have noticed that Onto2JaCaMo could provide model editing features (for example, to include new instances), which would discard the need of an ontology editing tool to update the ontology model. Another point for improvement that was highlighted by our practices, although a very complex one, is the automatic update of the ontology when the MAS code changes [6], in the direction of synchronising model and code. This might be solved by implementing features to highlight mismatches between MAS code and its corresponding model in order to keep both aligned.

In a last part of our experiments, the participants created theirs models and later programmed manually their code, which means without using the core code generation mechanism. That allowed us to compare the code that our technique creates automatically from the ontology models with the code actually programmed by the participants. Through these comparisons, we have observed the correspondences and similarities between elements in the code that was automatically generated from the specification in contrast with the code that was manually programmed. These similarities between these two sources of code are indicative of the correctness of the proposed model-based code generation technique. We have analysed that some key elements in the ontology models created by the participants, the corresponding code that was automatically generated from these elements by using the proposed techniques, and the code actually programmed by them. We were able to confirm that the model-based technique for generating code is indeed offering a program reasonably similar to the code structured created by the programmer[1], given the analysed aspects. We argue that if the starting codes were created based on converting their corresponding models, then it would be easier for programmers to align their initial code with the design and continue their programming based on that. The similarities between what can be automatically generated with what was manually created indicate that the code generation is in the correct direction and it provides more agility for developers that have their systems modelled before they start coding. We point out to our reference [5] for further details on our evaluations that would not be possible to tackle here in this chapter.

7 Final Remarks

In this chapter, we have proposed development techniques focusing on the JaCaMo platform, on the basis of ontologies that support the modelling and programming of MAS. Our proposal considers MAS designed as ontology models as the foundation for a MAS engineering process that allows core code generation. We have explored the research direction of reasoning with these ontology models, which allows the implementation of inference mechanisms in agent-based systems such as, for example, to reason about action, plans, knowledge, beliefs, goals, and norms in MAS. Producing software code for complex and highly detailed systems directly in programming environments by first using a specification, modelling, or design mechanism may avoid many problems. Without a proper modelling, it may be difficult to find potential bugs when they eventually appear in the implementation. Features derived from our approach are techniques for: *(i)* integrating design and code; *(ii)* supporting MAS programming with automatic code generation through model-based development; and *(iii)* performing verification with focus on the use of semantic reasoning and model checking.

[1] The model is at a higher abstraction level than the code, so sometimes only a structure or code skeleton can be created and programmers have to complete it in order to obtain a fully executable and running system.

Ontologies that serve as the basis of agent models could also inform agents in reasoning about their own system or even other systems or projects. These would allow agents to be able to share their implementation with others, or to execute inferences about its own implementation. In this context, an approach that provides for MAS the ability to interact with ontologies may be applied [8]. As future experiments, it would be interesting to consider more complex and distributed scenarios of software development, for example where teams of software engineers need to work together to develop a single MAS. These teams would be composed of persons playing different roles such as requirement engineer, designer, programmer, etc. In this context, it should be investigated how much a modelling and programming approach that is based on an ontology would help the team to communicate, synchronise, and coordinate the development of the desired MAS. Moreover, a viewpoint that should also be considered in future work is the comparison between using and not using the approach proposed in this chapter (similar to what is done with experiments conducted on the basis of a control group). Moreover, as we have highlighted in this chapter, new features may be added to Onto2JaCaMo, such as refactoring mechanisms for model and code synchronisation. Another related point would be to automatically identify mismatches between current MAS code and its corresponding model. That would contribute towards implementing round-trip engineering features in the context of MAS development (combined use of forward and reverse engineering).

For the sake of better explaining our approaches through examples, Sect. 4 discussed mostly the possibility of extending the OntoMAS ontology by creating new subclasses. We claim that the approach is more extensible than shown here, for example, one may decide to add new classes, properties or even rules, but in these cases, the consistency of the obtained ontology may be a major problem to deal with, especially in terms of future research directions. However, it is not an easy task to extend an ontology if the users do not have any solid prior knowledge about it. For example, Sect. 5 depicted general rules of agent systems which can be refined and extended for specific domains, which means that OntoMAS is extensible not only by adding new subclasses but in every part of its components. Literature often claims that it is worth considering what someone else has done and checking if existing sources can be refined and extended for the required particular domain and task. Reusing existing ontologies may be a requirement if a system needs to interact with other applications that have already committed to particular ontologies or controlled vocabularies. Lastly, the OntoMAS ontology[2] and the Onto2JaCaMo plug-in[3] can be found in the addresses given as footnotes.

Acknowledgements. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior – Brasil (CAPES) – Finance Code 001.

[2] OntoMAS ontology: http://www.inf.pucrs.br/linatural/wordpress/index.php/recursos-e-ferramentas/ontomas/.

[3] Onto2JaCaMo plug-in: http://www.inf.pucrs.br/linatural/wordpress/index.php/recursos-e-ferramentas/onto2jacamo/.

References

1. Boissier, O., Bordini, R.H., Hübner, J., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Sci. Comput. Program. **78**(6), 747–761 (2013)
2. Bordini, R.H., Dastani, M., Winikoff, M.: Current issues in multi-agent systems development. In: O'Hare, G.M.P., Ricci, A., O'Grady, M.J., Dikenelli, O. (eds.) ESAW 2006. LNCS (LNAI), vol. 4457, pp. 38–61. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75524-1_3
3. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. Wiley, Chichester (2007)
4. Budinsky, F.: Eclipse Modeling Framework: A Developers Guide. The Eclipse Series. Addison-Wesley, Boston (2004)
5. Freitas, A.: Model-driven engineering of multi-agent systems based on ontology. Ph.D. thesis, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, RS, Brazil (2017). http://tede2.pucrs.br/tede2/handle/tede/7930
6. Freitas, A., Bordini, R.H., Vieira, R.: Model-driven engineering of multi-agent systems based on ontologies. Appl. Ontol. J. **12**, 157–188 (2017)
7. Freitas, A., Cardoso, R.C., Vieira, R., Bordini, R.H.: Limitations and divergences in approaches for agent-oriented modelling and programming. In: Baldoni, M., Müller, J.P., Nunes, I., Zalila-Wenkstern, R. (eds.) International Workshop on Engineering Multi-Agent Systems, pp. 88–103 (2016)
8. Freitas, A., Panisson, A.R., Hilgert, L., Meneguzzi, F., Vieira, R., Bordini, R.H.: Applying ontologies to the development and execution of multi-agent systems. Web Intell. J. **15**(4), 291–302 (2017)
9. Hübner, J.F., Boissier, O., Kitio, R., Ricci, A.: Instrumenting multi-agent organisations with organisational artifacts and agents. Auton. Agents Multi-Agent Syst. **20**(3), 369–400 (2010)
10. Musen, M.A.: The Protégé project: a look back and a look forward. AI Matters **1**(4), 4–12 (2015)
11. Okuyama, F.Y., Vieira, R., Bordini, R.H., da Rocha Costa, A.C.: An ontology for defining environments within multi-agent simulations. In: Workshop on Ontologies and Metamodeling in Software and Data Engineering (2006)
12. Padgham, L., Winikoff, M.: Prometheus: a methodology for developing intelligent agents. In: Giunchiglia, F., Odell, J., Weiß, G. (eds.) AOSE 2002. LNCS, vol. 2585, pp. 174–185. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36540-0_14
13. Pokahr, A., Braubach, L.: A survey of agent-oriented development tools. In: Fallah-Seghrouchni, A.E., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming, Languages, Tools and Applications, pp. 289–329. Springer, Boston (2009). https://doi.org/10.1007/978-0-387-89299-3_9
14. Ricci, A., Viroli, M., Omicini, A.: CArtAgO: an infrastructure for engineering computational environments in MAS. In: Weyns, D., Parunak, H.V.D., Michel, F. (eds.) International Workshop Environments for Multi-Agent Systems, pp. 102–119 (2006)
15. Tran, Q.N.N., Low, G.: MOBMAS: a methodology for ontology-based multi-agent systems development. Inf. Softw. Technol. J. **50**(7–8), 697–722 (2008)
16. Zarafin, A.M.: Semantic description of multi-agent organizations. Master's thesis, Automatic Control and Computers Faculty, Computer Science and Engineering Department - Politehnica University of Bucharest (2012)