How to Program Organizations and Roles in the JADE Framework

Matteo Baldoni[1], Guido Boella[1], Valerio Genovese[1],
Roberto Grenna[1] and Leendert van der Torre[2]

[1]Dipartimento di Informatica. Università di Torino - IT.
E-mail: {baldoni,guido,grenna}@di.unito.it; valerio.click@gmail.com
[2]Computer Science and Communications, University of Luxembourg, Luxembourg
E-mail: leon.vandertorre@uni.lu

Abstract. The organization metaphor is often used in the design and implementation of multiagent systems. However, few agent programming languages provide facilities to define them. Several frameworks are proposed to coordinate MAS with organizations, but they are not programmable with general purpose languages. In this paper we extend the JADE framework with primitives to program in Java organizations structured in roles, and to enable agents to play roles in organizations. Roles facilitate the coordination of agents inside an organization and offer new abilities (*powers*) in the context of organizations to the agents which satisfy the *requirements* necessary to play the roles. To program organizations and roles, we provide primitives which enable an agent to enact a new role in an organization to invoke powers.

1 Introduction

Organizations are the subject of many recent papers in the MAS field, and also among the topics of workshops like COIN, AOSE, CoOrg and NorMAS. They are used for coordinating open multiagent systems, providing control of access rights, enabling the accommodation of heterogeneous agents, and providing suitable abstractions to model real world institutions [10].

Many models have been proposed [13], applications modeling organizations or institutions [17], software engineering methods using organizational concepts like roles [21]. However, despite the development of several agent programming languages among which 3APL [20], few of them have been endowed with primitives for modeling organizations and roles as first class entities. Exceptions are MetateM [12], J-MOISE+ [15], and the Normative Multi-Agent Programming Language in [19]. MetateM is BDI oriented and not a general purpose language and is based on the notion of group. J-MOISE+ is more oriented to programming how agents play roles in organizations, while [19], besides not being general purpose, is more oriented to model the institutional structure composed by obligations than the organizational structure composed by roles. On the other hand, frameworks for modelling organizations like SMoise+ [16] and MadKit [14] offer limited possibilities to program organizations. The heterogeneity of solutions shows a lack of a common agreement upon a clear conceptual model of what an organization is; the ontological status of organizations has been studied only

recently and thus it is difficult to translate the organizational model into primitives for programming languages. Moreover, it is not clear how an agent can interact with an organization, whether the organization has to be considered like an object, a coordination artifact, or as an agent [11]. The same holds for the interaction with a role. Thus, in this paper, we address the following research questions: *How to program organizations? How to introduce roles? How agents interact with organizations by means of the roles they play?* And as subquestions: *How to specify the interaction between an agent and its roles? How to specify the interaction among roles?* Moreover, *How can an agent start playing a role?* and, finally, *What is the behaviour of an agent playing a role?*

We start from the ontological model of organizations developed in [6], since it provides a precise definition of organizations and specifies their properties, beyond the applicative needs of a specific language or framework, and allows a comparison with other models. Moreover, this model has been successfully used to give a logical specification of multiagent systems and to introduce roles in object oriented languages [2]. This approach, due to the agent metaphor, allows to model organizations and roles using the same primitives to model agents, like in MetateM [11]. Despite the obvious differences with agents, like the lack of autonomy and of an independent character in case of roles, the agent metaphor allows to understand the new concepts using an already known framework. In particular, interaction between agents and organizations and the roles they play can be based on *communication protocols*, which is particularly useful when agents and organizations are placed on different platforms.

For the description of the interaction among players, organizations and roles we adopt the model of [5]. For what concerns the modeling of how an agent plays a role we take inspiration from [9]. We implement this conceptual model with JADE (Java Agent DEvelopment framework) [3], providing a set of classes which extends it and offer the primitives for constructing organizations when programming multiagent systems. We extend JADE not only due to its large use, general purpose character and open-source philosophy, but also because, being developed in Java, it allows to partly adopt the methodology used to endow Java with roles in the language powerJava [2]. Given the primitives for modeling agents and their communication abilities, transferring the model of [6] in JADE is even more straightforward than transferring it in Java. The extension of JADE consists in a new library of classes and protocols. The classes provide the primitives used to program organizations, roles and player agents, by extending the respective class.

The paper is organized as follows. In Section 2, we summarize the model of organizations and roles we take inspiration from. In Section 3, we describe how the model is realized by extending JADE, and in Section 4 we describe the communication protocols which allow the interaction of the different entities. In Section 5 we underline related and future work.

2 A model for organizations and roles

In [6] a definition of the structure of organizations given their ontological status is given, roles do not exist as independent entities but they are linked to organizations (in other words, roles are not simple objects). Organizations and roles are *not autonomous*, but

act via *role players*, although they are description of complex behaviours: in the real world, organizations are considered legal entities, so they can even act like agents, albeit via their representative playing roles. So, they share some properties with agents, and, in some aspects, can be modelled using similar primitives. Thus, in our model roles are entities, which contain both state and behaviour: we distinguish the *role instance associated with a player* and the *specification of a role* (a role type). As recognized by [8] this feature is quite different from other approaches which use roles only in the design phase of the system, as, e.g., in [21].

Goals and beliefs, attributed to a role (as in [9]) describe the behaviour expected from the player of the role, since an agent pursues his goals based on his beliefs. The player should be aware of the goals attributed to the roles, since it is expected to follow them (if they do not conflict with other goals). Most importantly, roles work as "interfaces" between organizations and agents: they give "powers" to agents, extending the abilities of agents, allowing them to operate inside the organization and inside the state of other roles. If on the one hand roles offer powers to agents, they request from agents to satisfy a set of *requirements*, abilities that the agents must have [7].

The model presented in [6] focuses on the dynamics of roles in function of the communication process: role instances evolve according to the speech acts of the interactants. Where speech acts are powers, that can change not only the state of its role, but also the state of other roles (see [4]). For example, the commitments made by a speaker of a promise or by commands made by other agents playing roles which are empowered to give orders. In this model, sets of beliefs and goals (as [9] does) are attributed to the roles. They are the description of the expected behaviour of the agent. The powers of roles specify how the state of the roles changes according to the moves played in the interactions by the agents enacting other roles.

Roles are a way to *structure the organization*, to *distribute responsibilities* and a *coordination means*. Roles allow to encapsulate all the interactions between an agent and an organization and between agents in their roles. The powers added to the players can be different for each role and thus represent different affordances offered by the organization to other agents to interact with it [1].

However, this model leaves unspecified how, given a role, its player will behave. In [9], the problem of formally defining the dynamics of roles is tackled identifying the actions that can be done in an *open system* such that agents can enter and leave. In [9] four operations to deal with role dynamics are defined: *enact* and *deact*, which mean that an agent starts and finishes to occupy (play) a role in a system, and *activate* and *deactivate*, which mean that an agent starts executing actions (operations) belonging to the role and suspends the execution of the actions. Although is possible to have an agent with multiple roles enacted simultaneously, only one role can be *active* at the same time: when an agent performs a power, it is playing only one role in that moment.

3 Organizations, roles, and players in JADE

We introduce organizations and roles as first class entities in JADE, with behaviours, albeit not autonomously executed, and communication abilities. Thus, organizations and roles can be implemented using the same primitives of agents by extending the JADE

`Agent` class with the classes `Organization` and `Role`. Analogously, to implement autonomous agents who are able to play roles, the `Player` class is defined as an extension of the `Agent` class. The `Role` class and its extensions represent the *role types*. Their instances represent the role instances associated with an instance of the `Agent`.[1] Organizations and roles, however, differ in two ontological aspects: first roles are associated to players; second, roles are not independent from the organization offering them. Thus, the `Role` class is subject to an invariant, stating that it can be instantiated only when an instance of the organization offering the role is present. Conversely, when an organization is destroyed all its roles must be destroyed too.

A further difference of role classes is that to define "powers", they must access the state of the organization they belong too. To avoid making the state of the organization public, the standard solution offered by Java is to use the so-called "inner classes". Inner classes are classes defined inside other classes ("outer classes"). An inner class shares the namespace of the outer class and of the other inner classes, thus being able to access private variables and methods. The class `Role` is defined as an inner class of the `Organization` class. Class extending the `Role` class must be inner classes of the class extending the `Organization` class. In this way the role can access the private state of the organization and of the other roles. Since roles are implemented as inner classes, a role instance must be on the same platform as the organization instance it belongs to. Moreover, the role agent can be seen as an object from the point of view of the organization and of the other roles which can have a reference to it, besides sending messages to it. In contrast, outside an organization the role agent is accessed by its player (which can be on a different platform) only as an agent via messages, and no reference to it is possible. So not even its public methods can be invoked.

The inner class solution for roles is inspired to the use of inner classes to model roles in object oriented programming languages like in powerJava [2]. The use of inner classes is coherent with the organization of JADE, where behaviours are often defined as inner classes with the aim to better integrate them with the agent containing them.

3.1 Organizations

To implement an organization it's necessary to extend `Organization`, subclass of `Agent`, which offers protocols necessary to communicate with agents who want to play a role, and the behaviours to manage the information about roles and their players. Moreover, the `Organization` class includes the definition of the `Role` inner class that can be extended to implement new role classes in specific organizations. To support the creation and management of roles the `Organization` class is endowed with the (private) data structures and (private) methods to create new role instances and to keep the list of the AIDs (Agent IDs) of role instances which have been created, associated with the AIDs of their players. Since roles are Java inner classes of an organization, the organization code can be written in Java mostly disregarding what is a JADE application. Moreover, the inner class mechanism allows the programmer to access the role state and viceversa, while maintaining the modularity character of classes.

[1] Nothing prevents to have organizations which play roles in other organizations, like in [8], for this, and other combinations, it is possible to predefine classes and extend them.

The `Enact` protocol allows starting the interaction between player and organization. A player sends a message to an organization requesting to play a role of a certain type; if the organization considers the agent authorized to play that role type, it sends to the caller a list of powers (what the role can do) and requirements (what the role can ask to player what to do). At this point, the player can compare his requirements list with the one sent from the organization and communicate back if he can play the role or if he can't. The operation of leaving a role, *Deact*, is asked by the player to the role itself, so the class organization does not offer any methods or protocols for that.

For helping players to find quickly one or more organizations offering a specific role, Yellow Pages are used. They allow to register a pair *(Organization, RoleType)* for each role in each organization; the interested player will only have to query the Yellow Pages to obtain a list of these couples and choose the best for itself.

3.2 Roles

A role is implemented by extending the `Role` class which offers the protocols to communicate with the player agent. Notice that, since the *Role* class is an inner class of the *Organization* class, this class should be an inner class of the class that extends `Organization` as well. The protocols to communicate with the player agent allows: (*i*) To receive the request of invoking powers; (*ii*) To receive the request to deact the role; (*iii*) To send to the player the request to execute a requirement; (*iv*) To receive from the player the result of the execution of a requirement; (*v*) To notify the player about the failure of executing the invoked power or the failure to receive all the results of the requested requirements. The role programmer, thus, has to define the powers which can be invoked by the player, and to specify them in a suitable data structure used by the `Role` class to select the requests of powers which can be executed.

Allowing a player to invoke a power, which results in the execution of a method by the role, could seem a violation of the principle of autonomy of agents. However, the powers are the only way the players have to act on an organization and, the execution of an invoked power may request, in turn, the execution by the player of requirements needed to carry on the power.

Moreover, since the player may refuse to execute a requested requirement (see Section 3.3), and the requirements determine the outcome of the power, this outcome, thus, varies from request to request, and from player to player. Since role instances are not autonomous, the invocation of a power is not subordinated to the decision of the role to perform it or not. In contrast with powers, a requirement cannot be invoked. Rather it is requested by the role, and the player autonomously decides to execute it or not. In the latter case the player is not complying anymore with its role and it is deacted. To remark this difference we will use the term *invoking a power* versus *requesting a requirement*.

Requests for the execution of requirements are not necessarily associated with the execution of a power. They can be requested to represent the fact that a new goal has been added to the role. For example, this can be the result of a task assignment when the overall organization is following a plan articulated in subtasks to be distributed among the players at the right moments [16]. In case the new goal is a requirement of the player the method *requestRequirement* is executed, otherwise, if it is a power of the role, a *requestResponsibility* is executed. The method *requestResponsibility* asks to the

player to invoke a power of the role, while the method *requestRequirement* invokes a requirement of the player. It returns the result sent by the player if he complies with the requirement. The failure of executing of a requirement results in the deactment of the role. Analogously to requirements when the role notifies its player about the responsibility, it cannot be taken for granted that it will invoke the execution of the power. Note that both methods can be invoked also by other roles or by the organization itself, due to the role's limited autonomy.

Other methods are available only when agents are endowed with beliefs, which could be represented, for instance, in Jess. The method *sendInform* is used to inform the player that the beliefs of the roles are changed. This does not imply that the player adopts the conveyed beliefs as well. Methods a*ddBelief* and *addGoal* are invoked by the role's behaviours or by other roles' to update the state of the role.

Besides the connection with its player, the role is an agent like any other, and it can be endowed with further behaviours and further protocols to communicate with other roles of the organization or even with other agents. At the same time it is a Java object as any other and can be programmed, accessing both other roles and the organization internal state to have a better coordination.

3.3 Players

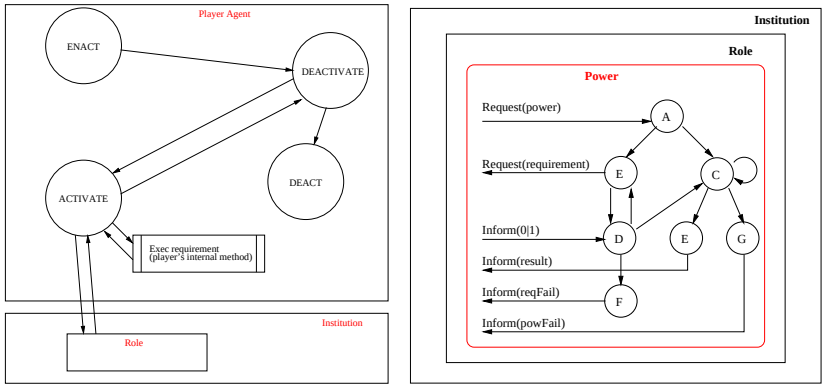
Players of roles in organizations are JADE agents, which can reside on different platforms with respect to the organization. To play a role, a special behaviour is needed; for this reason the `Player` class is offered. An agent which can become a player of roles extends the *Player* class, which, in turn, extends the *Agent* class. This class defines the states of the role playing (enact, active, deactivated, deacted), the transitions from one state to the others, and offers the protocols for communicating with the organization and with the role. A player agent can play more than one role. The list of roles played by the agent, and the state of each role, is kept in an hashtable.

The enactment procedure takes the AID of an organization and of a role type and, if successful, it returns the AID of the role instance associated to this player in the organization. From that moment the agent can activate the role and play it. The activate state allows the player to receive from the role requests of requirement execution and responsibilities (power invocation). Analogously, the *Player* class allows an agent to deact and deactivate a role.

The behaviour of playing a role is modelled in the player agent class by means of a *finite state machine* (FSM) behaviour. The behaviour is instantiated for each instance of the role the agent wants to play, by invoking the method *enact* and specifying the organization AID and the role type. The states are inspired to the model of [9].

Enact: The communication protocol (which contains another FSM itself) for enacting roles is entered. If it ends successfully with the reception of the new role instance AID the deactivated state is entered. The hashtable containing the list of played roles is updated. Otherwise, the deacted state is reached.

Activate: This state is modelled as a FSM behaviour which listens for events coming from outside or inside the agent. If another behaviour of the agent decides to invoke a power of the role by means of the *invokePower* method (see below), the behaviour of the activated state checks if the power exists in the role specification and sends an



(a) (b)

Fig. 1. (a) The states of role playing. - (b) The behaviour of roles - A: ManagePwRequest (manages the request from player); B: ManageReqRequest (if a requirement is needed); C: Execute (executes the called power); D: MatchReq (checks if all requirements are ok); E: InformResult (sends results to player); F: InformFail (sends fail caused by requirement missing); G: InformPowerFail (sends fail caused by power failure)

appropriate message to the role agent. Otherwise an exception is raised. If another behaviour of the agent decides to deactivate the role the deactivated state is entered. If a message requesting requirements or invoke powers arrives from the role agent it plays, the agent will decide whether or not to comply with the new request sent by the role. First of all it checks that the required behaviour exists or there has been a mismatch at the moment of enacting a role. If the role communicates to its player that the execution of a power is concluded and sends the result of the power, this information is stored waiting to be passed back to the behaviour which invoked the power upon its request (see *receivePowerResult*). The cyclic behaviour associated with this state blocks itself if no event is present and waits for an event.

Deactivated: The behaviour stops checking for the invocation of requirements or powers from respectively the role and the player itself, and blocks until another behaviour activates the role again. The messages from the role and the power invocations from other behaviours pile up in the queue waiting to be complied with, until an activation method is called and the active state is entered.

Deact: The associated behaviour informs the role that the agent is leaving the role and cleans up all the data concerning the played role in the agent.

One instance of this FSM, that can be seen in Figure 1, is created for each role played by the agent. This means that for a role only one power at time is processed, while the others wait in the message queue. Note that the information whether a role is activated or not is local to the player: from the role's point of view there is no difference. However, the player processes the communication of the role only as long as it is acti-

vated, otherwise the messages remains in the buffer. More sophisticated solutions can be implemented as needed, but they must be aware of the synchronization problems.

The methods in the `Player` can be used to program the behaviours of an agent when it is necessary to change the state of role playing or to invoke powers. We assume that invocations of powers to be *asynchronous*. The call returns a call id which is used to receive the correct return value in the same behaviour. It is left to the programmer to stop of a behaviour till an answer is returned by JADE primitive *block*. This solution is coherent with the standard message exchange of JADE and allows to avoid using more sophisticated behaviours based on threads.

- *enact(organizationAID, roleClassName)*: to request to enact a role an agent has to specify the AID of the organization and the name of the class of the role. It returns the AID of the role instance or an exception is raised.
- *receivePowerResult(int)*: to receive the result of the invocation of a power.
- *deact(roleAID)*, *activate(roleAID)*, *deactivate(roleAID)* respectively deacts the role, activates a role agent that is in the *deactivate* state, and temporarily deactivates the role agent.
- *invokePower(roleAID, power)*: to invoke a power it is sufficient to specify the role AID and the name of the behaviour of the role which must be executed. It returns an integer which represent the id of the invocation.
- *addRequirement(String)*: when extending the *Player* class it is necessary to specify which of the behaviours defined in it are requirements. This information is used in the *canPlay* private method which is invoked by the *enact* method to check if the agent can play a role. This list may contain non truthful information, but the failure to comply with the request of a commitment may result in the deactment to the role as soon as the agent is not able to satisfy the request to execute a certain requirement.

Moreover, an agent, in order to be a player, has to implement an abstract method to decide whether to execute the requirements upon request from the roles. The method *adoptGoal* is used to preserve the player autonomy with reference to requirement requests from the role he's playing. The result is *true* if the player decides to execute the requirement.

4 Interaction

In this section we describe the different protocols used in the interaction between agents who want to play roles and organizations, and between players and their roles. All protocols use standard FIPA messages, to enable also non JADE agents to interact with organizations without further changes. Figure 2 describes the sequence diagram of the interaction.

4.1 Agents and the organization

Behind the *enacting* state of the player described in the previous section, there is an *enactment protocol* inherited, respectively, as concern the initiator and the receiver, from

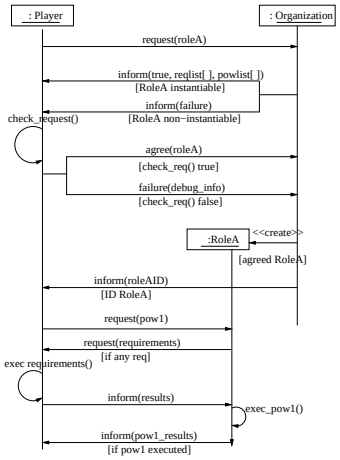


Fig. 2. The interaction protocol

the classes `Player` and `Organization`. It forwards from the player to the organization the request of enacting a specified role, and manages the exchange of information: sending the specification of requirements and powers of the roles and checking whether the player complies with the requirements.

The organization listens from messages from any agent (even if some restrictions can be posed at the moment of accepting to create the role), while the subsequent communication between player and role is private. After a request from an agent the behaviour representing the protocol forks creating another instance of itself to be ready to receive requests of other agents in parallel.

The first message is sent by the player as initiator and is a `request` to enact a role. The organization, if it considers the agent authorized to play the role, returns to the candidate player a list of specifications about the powers and requirements of the requested role which are contained in its knowledge base, sending an `inform` message containing the list; otherwise, it denies to the player to play the role, answering with an `inform` message, indicating the failure of the procedure. In case of positive answer, the player, invoking the method *canPlay* using the information contained in the player about the requirements, decides whether to respond to the organization that it can play the role (`agree`) or not (`failure`).

The first answer results in the creation of a new role instance of the requested type and in the update of the knowledge base of the organization with the information that the player is playing the role. To the role instance the organization passes the AID of the role player, i.e., the initiator of the enactment, so that it can eventually filter out the messages not coming from its player. An `inform` is sent back to the player agent, telling him the played role instance's AID, The player, in this way, can address messages to the role and it can identify the messages it receives from the role it plays. Then the agent updates its knowledge base with this information, labeling the role as still deactivated. The protocol terminates in both the player and the organization. This

completes the interaction with the organization: the rest of the interaction, including deacting the role, passes through the role instance only.

4.2 Players and their roles

The interaction between a player and its role is regulated by three protocols: the request by the role of executing a requirement, the invocation of a power by the player, and the request of the role to invoke a power. In all cases, the interaction protocol works only between a player and the role instances it plays. Messages following the protocol but which do not respect this constraint are discharged on both sides.

We start from the first case since it is used also in the second protocol during the execution of a power. According to Dastani et al. [9] if a role is activated, the player should (consider whether to) adopt its goals and beliefs. Since our model is distributed, the role is separated from its player: the goals (i.e., the requirements) and beliefs of the role have to be communicated from the role to its player by means of a suitable communication protocol. Each time the state of the role changes, since some new goal is added to it, the agent is informed by the role about it: either a requirement must be executed or a power must be invoked. In this protocol, the initiator is the role, which starts the behaviour when its method *requestRequirement* is invoked.

First of all, the agent checks if the requested requirement is in the list of the player's requirements, but this does not mean that it will be executed. Since the player agent is autonomous, before executing the requirement, it takes a decision by invoking the method *adoptGoal* which is implemented by the programmer of the player. The protocol ends by informing the role about the outcome of the execution of the requirement or the refusal of executing it, using an "inform" (see bottom of Figure 1 b).

This protocol is used inside the protocol initiated by the player for invoking a power of the role. After a request from the player, the role can reply with the request of executing some requirements which are necessary for the performance of the power. In fact, in the behaviour corresponding to the power, some invocation of the method *requestRequirement* can be present. The protocol ends with the role informing the agent about the outcome of the execution of the power.

A third protocol is used by the role to remind the agent about its responsibilities, i.e., the role asks its player to invoke a power executing the method *requestResponsibility*. In this case the object of the request is not a requirement executable by the player, but a power, i.e., a behaviour of the role. So the player has to decide whether and when to invoke the power.

In principle, the programmer could have invoked a power directly from the role, instead of requesting it by means of *requestResponsibility*. However, with this mechanism we want to model the case where the player is obliged to invoke the power but the decision of invoking the power is left to the player agent who can have more information about when and how invoke the power. It is left to the programmer of the organization to handle the violation of such obligations.

The final kind of interaction between a player and its role is the request of a player to deact the role. While deactivation is an internal state of the player, which is not necessarily communicated to the role, deacting requires that the role agent is destroyed and that the organization clears up the information concerning the role and its player.

5 Conclusions

In this paper we use the ontological model of organizations proposed in [6] to program organizations. We use as agent framework JADE since it provides the primitives to program MAS in Java. We define a set of Java classes which extends the agent classes of the JADE to have further primitives for building organizations structured into roles. To define the organizational primitives JADE offered advantages but also posed some difficulties. First of all, being based on Java, it allowed to reapply the methodology used to implement roles in powerJava [2] to implement roles as inner classes. Moreover, it provides a general purpose language to create new organizations and roles. Finally, being based on FIPA speech acts, it allows agents programmed in other languages to play roles in organizations, and viceversa, JADE agents to play roles in organizations not implemented in JADE. However, the decision of using JADE has some drawbacks. For example, the messages used in the newly defined protocols can be intercepted by other behaviours of the agents. This shows that a more careful implementation should use a more complex communication infrastructure to avoid this problem. Moreover, since JADE behaviours differently from methods do not have a proper return value, they make it difficult to define requirements and powers. Finally, due to the possible parallelism of behaviours inside an agent, possible synchronization problems can occur.

Few agent languages are endowed with primitives for modeling organization. MetateM [11] is one of these, and introduces the notion of group by enlarging the notion of agent with a context and a content. The context is composed by the agents (also groups are considered as agents like in our model organizations are agents) which the agent is part of, and the content is a set of agents which are included. The authors propose to use these primitives to model organizations, defining roles as agents included in other agents and players as agents included in roles. This view risks to leave apart the difference between the play relation and the role-of relation which have different properties (see, e.g., [18]). Moreover it does not distinguish between powers. Finally MetateM is a language for modeling BDI agents, while JADE has a wider applicability and is built upon on the Java general purpose language.

About S-Moise+ features [16], we will improve our system with agent sets and subset as particular inner classes in the Organization class. Very interesting is the matter of cardinality, constraint that we will implement considering both minimum than maximum cardinality allowed for each group.

The principles of permission will be implemented through a specific new protocol, called `Permissions`, which will allow to a role a call to another role's power, if and only if the first role's player can show (at the time of execution) his credentials (additional requirements); if no additional requirement is given, the other role's power invocation cannot be done.

Another future work is related to Obligations [16]; we are going to implement them by particular requirements that have to produce some result in a fixed time. If no result is produced, then a violation occours and this behaviour is sanctioned in some way. Planning goals too will be realized by requirements, that can be tested one after another to play single missions.

References

1. M. Baldoni, G. Boella, and L. van der Torre. Modelling the interaction between objects: Roles as affordances. In *Proc. of KSEM 2006*, *LNCS* 4092, pp. 42–54. Springer, 2006.
2. M. Baldoni, G. Boella, and L. van der Torre. Interaction between Objects in powerJava. *Journal of Object Technology*, 6(2):7–12, 2007.
3. F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.
4. G. Boella, R. Damiano, J. Hulstijn, and L. van der Torre. ACL semantics between social commitments and mental attitudes. In *Proc. of AC 2005 and AC 2006*, *LNAI* 3859, pp. 30–44. Springer, 2006.
5. G. Boella, V. Genovese, R. Grenna, and L. der Torre. Roles in coordination and in agent deliberation: A merger of concepts. In *Proc. of Multi-Agent Logics, PRIMA 2007*.
6. G. Boella and L. van der Torre. Organizations as socially constructed agents in the agent oriented paradigm. In *Proc. of ESAW'04*, *LNAI* 3451, pp. 1–13, 2005. Springer.
7. G. Cabri, L. Ferrari, and L. Leonardi. Agent roles in the brain framework: Rethinking agent roles. In *The 2004 IEEE Systems, Man and Cybernetics Conference, session on "Role-based Collaboration"*, 2004.
8. A. Colman and J. Han. Roles, players and adaptable organizations. *Applied Ontology*, 2007.
9. M. Dastani, B. van Riemsdijk, J. Hulstijn, F. Dignum, and J.-J. Meyer. Enacting and deacting roles in agent programming. In *Procs. of AOSE'04*, pages 189–204, New York, 2004.
10. J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: an organizational view of multiagent systems. In *Proc. of AOSE'03*, *LNCS* 2935, pp. 214–230, 2003. Springer.
11. M. Fisher. A survey of concurrent metatem - the language and its applications. In *ICTL*, pages 480–505, 1994.
12. M. Fisher, C. Ghidini, and B. Hirsch. Organising computation through dynamic grouping. In *Objects, Agents, and Features*, pages 117–136, 2003.
13. D. Grossi, F. Dignum, M. Dastani, and L. Royakkers. Foundations of organizational structures in multiagent systems. In *Procs. of AAMAS'05*, pages 690–697, 2005.
14. O. Gutknecht and J. Ferber. The madkit agent platform architecture. In *Agents Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, 2000.
15. J. F. Huebner. J-Moise$^+$ programming organizational agents with Moise$^+$ and Jason. In *http://moise.sourceforge.net/doc/tfg-eumas07-slides.pdf*, 2007.
16. J. F. Huebner, J. S. Sichman, and O. Boissier. S-moise+: A middleware for developing organised multi-agent systems. In *Proc. of AAMAS Workshops*, *LNCS* 3913, pp. 64–78. Springer, 2005.
17. A. Omicini, A. Ricci, and M. Viroli. An algebraic approach for modelling organisation, roles and contexts in MAS. *Applicable Algebra in Engineering, Communication and Computing*, 16(2-3):151–178, 2005.
18. F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering*, 35:83–848, 2000.
19. N. Tinnemeier, M. Dastani, and J.-J. C. Meyer. Orwell's nightmare for agents? programming multi-agent organisations. In *Proc. of PROMAS'08*, 2008.
20. W. van der Hoek, K. Hindriks, F. de Boer, and J.-J. C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
21. F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *IEEE Transactions of Software Engineering and Methodology*, 12(3):317–370, 2003.