powerJADE: Organizations and Roles as Primitives in the JADE Framework

Matteo Baldoni, Guido Boella, Mauro Dorni, Andrea Mugnaini, and Roberto Grenna

Abstract. **This document shortly describes powerJADE, an improved JADE framework [4] which provides the primitives to manage organizations and roles.**

I. THE MODEL OF ORGANIZATIONS AND ROLES

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 [12].

Many models have been proposed [16], applications modeling organizations or institutions [22], software engineering methods using organizational concepts like roles [26]. However, up to now, on the one hand, despite the development of several agent programming languages among which 3APL [25], few of them have been endowed with primitives for modeling organizations and roles as first class entities (exceptions are MetateM [15], J-Moise+ [19], and the Normative Multi-Agent Programming Language in [24]). On the other hand, frameworks for modelling organizations like S-Moise+ [20] and MadKit [17] offer limited possibilities to program organizations.

The heterogeneity of solutions show a lack of a common agreement upon a clear conceptual model of what is an organization; the ontological status of organizations has been studied only recently and thus it is difficult to translate the organizational model in primitives for programming languages.

Our proposal, the introduction of primitives for organizations and roles in JADE, is mainly based on the ontological model of organizations and roles of [7]. However, since this model disregards the problem of how agents play roles in organizations, in [6] we integrated our model with the model of role playing of [11]. The model of [7] is focused on the definition of the structure of organizations, given their ontological status, which is only partly different from the one of agents or objects. On the one hand, roles do not exist as independent entities, since they are linked to organizations. Thus they are not components like objects. Moreover, organizations and roles are not autonomous and act via role players. On the other hand, organizations and roles 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.

Thus, in our model, roles are entities, which contain both state and behaviour: we must distinguish with the term "role", the role instance associated with a player, while the general specification of a role is defined as a "role type". For each agent, when it asks to an organization to play a role of a given type, an instance is created, representing the possibility for the player of interacting with the organization, and with the other roles, and the state of the interaction between the agent and the organization. As recognized by [10] this feature is quite different from other approaches which use roles only in the design phase of the system, as, e.g., in [26].

The goals, together with the beliefs, attributed to a role (as also in [11]) 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 of the agent).

Most importantly, roles work as "interfaces" between organizations and agents: they give so called "powers" to agents, extending their abilities, allowing them to operate inside the organization and inside the state of other roles. An example of such powers, called "institutional powers" in [21], is the signature of a director which counts as the commitment of the entire institution. If, on the one hand, roles offer powers to agents, they request from agents who want to play roles a set of requirements, abilities that the agents must have, like also in [9]. Thus, the set of roles an agent can play is not determined a priori, but it depends on which abilities are required by the role of an institution.

Powers are invoked by players on their roles, but they are executed by the roles, since they have both state and behaviour. Boella and van der Torre's [7] model 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 an example of powers which change not only the state of the role making the speech act, but also of other roles (see [5]). E.g. 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 [11] 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 interaction 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 [2].

Using the distinction of Omicini [22], we use our model [7] as an objective coordination mechanism, like for example artifacts: organizations are first class entities of the MAS rather than a mental constructions which agents use to coordinate themselves.

However, this model leaves unspecified, how, given a role, its player will behave. In Dastani [11], 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 [11] 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 means 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, he is playing only one role in that moment.

II. ORGANIZATIONS, ROLES, AND PLAYERS IN JADE

In this section we describe how we use our model in [7] to extend the JADE framework with primitives for programming organizations and roles.

JADE, to program a MAS, offer as basic elements the class *Agent*, different classes of behaviours (like finite state machines), and protocols with the relative speech act definitions, compliant with FIPA [13]. We have to introduce organizations and roles as first class entities, located in another platform with respect to their member agents, with behaviours - albeit not autonomously executed - and communication abilities. Thus, organizations and roles can be implemented using the same primitives as agents. In JADE this amounts to having special extensions of the *Agent* class, *Organization* and *Role* respectively, which can be further extended to program organizations and roles. Analogously, to implement agents who are able to play roles, the *Player* class is defined, which extends the JADE *Agent* class. Differently from *Organization* and *Role*, this class is used to implement autonomous agents. The behaviours and the communication protocols - described in the next section - which allow organizations to work, are inherited from these three classes. E.g. all organizations must offer a suitable protocol to allow an agent to ask playing a role, to verify if the agent fulfills the requirements of the role, and to enact the role. The extensions of the *Role* class, like, e.g., *Buyer*, represent the role types. Their instances, the role instances associated with an instance of the *Agent*. 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" as described in the previous section, they must access the state of the organization they belong too: only in this way it is possible that the signature of a role has the effect of modifying the private state of the organization adding a commitment. To avoid making the state of the organization public, the standard solution offered by Java is to use the so-called "inner classes", which are classes defined inside other classes (called "outer classes").

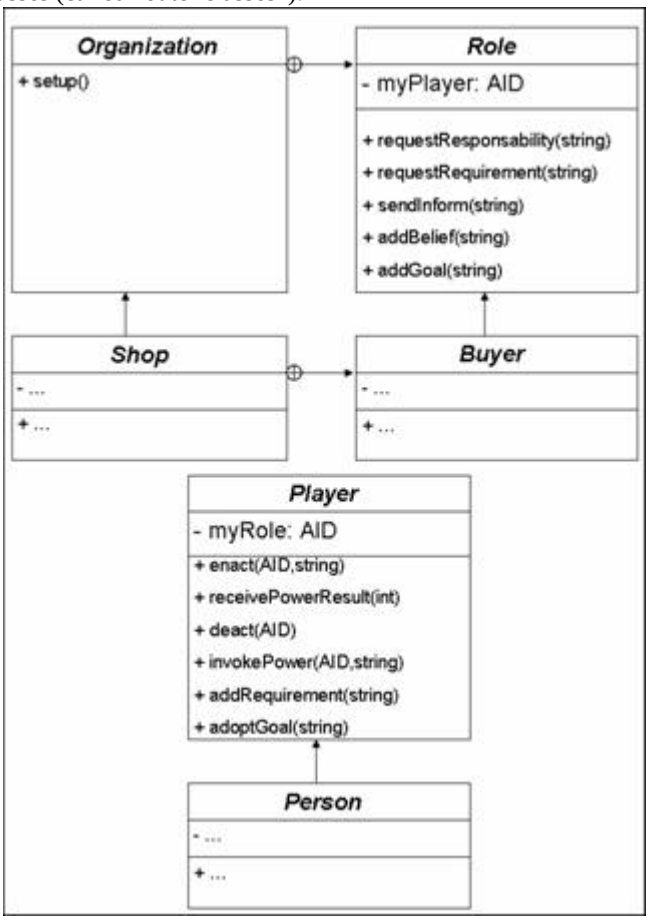

Fig. 1 - UML representation

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. Thus 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 (see Figure 1). E.g., the *Buyer* class

extending the *Role* class must be an inner class of the *Shop* class extending the *Organization* class. In this way the role can access the private state of the organization and of the other roles. Note that this access is not unregulated, since a specific role cannot access the entire private state of the organization and other roles, but only those features which have been decided by the programmer of the role (who is the same as the programmer of the organization, since the role is contained in it). E.g., the methods of the *Buyer* class can access the private state of both the *Shop* class and of other roles of *Shop*, like *Seller*.

If roles are implemented as inner classes, albeit extension of the *Agent* class, this means two facts: first, the role instance must be on the same platform as the organization instance it belongs to; second, 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 [3]. 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.

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 data structures and methods to create new role instances and to keep the list of the AIDs (Agent ID) of role instances which have been created, associated with the AIDs of their players. However, these methods are private, to avoid the misuse by the programmer who could violate the organization invariants (e.g., to instantiate a role first there must be a player, etc.).

The *Enact* protocol allows starting the interaction between player and organization. What happens is that the player sends a message to the 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 to do). At this point, the player can compare his requirement list with the one sent from organization and communicate back if he can play the role or if he can't. Here, in fact, we can also consider that a not honest player could lie to play a role for which he doesn't have all the requirements. Only in a second time, and only if the player will show his limits, some player recovering a controlling role could apply sanctions to the dishonest one. It's important to note that only the player can begin the Enact protocol; the organization, in fact, is intended as a collection of roles that players can play. The only way in which an organization could begin an enactment protocol is to be a player that want to play a role inside another organization.

Each organization has to maintain the list of agents playing roles in it, associating with the player agent AID the role type and the role AID. While the association of a player and a role is done automatically during the enactment of a role (see next section), the programmer may query which role instances an agent is playing (specifying the role type) or which is the player of a given role. Finally, the operation of leaving a role (deact), is asked by the player to the role itself, so the class organization does not offer any method or protocols for that.

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. For helping players to find quickly one or more organizations offering a specific role, Yellow Pages, a JADE feature, 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.

Then the candidate player can start the enactment protocol with the selected organization.

Roles. As discussed above, a role is implemented by extending the *Role* class, thus, inheriting the facilities offered by that class. In particular, the *Role* class offers the protocols to communicate with the player agent and the methods for the role programmer to use these protocols.

To program a role, it is necessary to extend, inside a class extending the *Organization* class, the inner class *Role* of the *Organization* class.

In particular, the communication protocol with the player essentially allows (see Figure 2): (1) To receive the request of invoking powers. (2) To receive the request to deact the role. (3) To send to the player the request to execute a requirement. (4) To receive from the player the result of the execution of a requirement. (5) To notify the player 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 methods which are 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.

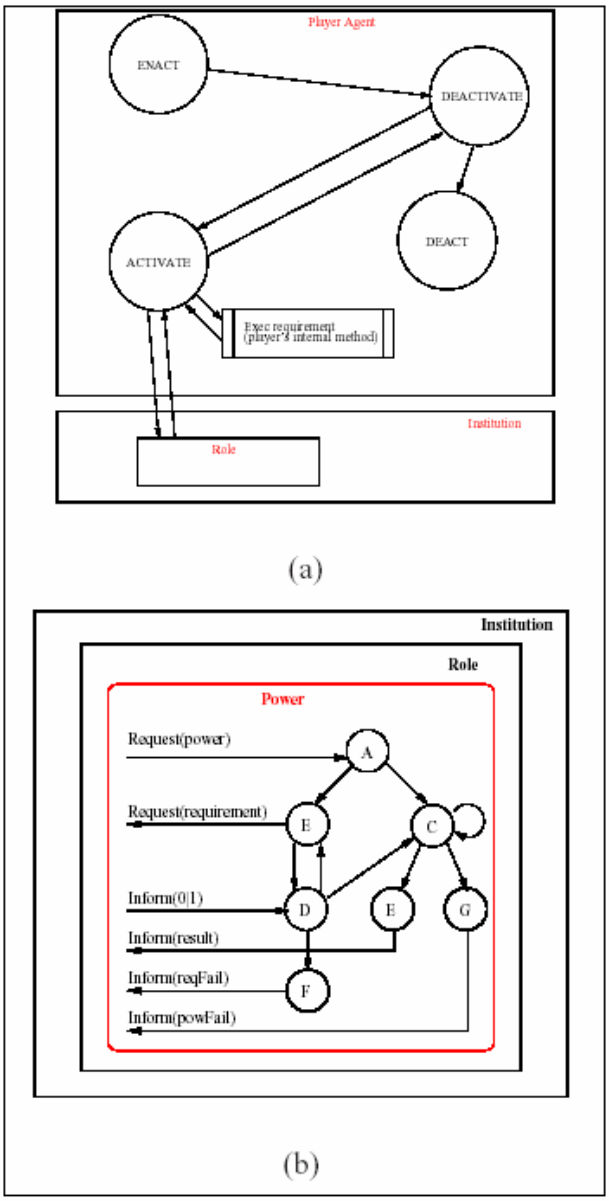

Fig. 2 - (a) The states of role playing. - (b) The behaviour of roles - A: ManagePwRequest (manages the request from player); B: ManageReqRequest (if a requirement is needed); C: Execute (executes the called power); D: MatchReq (checks if all requirements are ok); E: InformResult (sends results to player); F: InformFail (sends fail caused by requirement missing); G: Inform-PowerFail (sends fail caused by power failure)

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, since it seems that the player delegates to the role the execution. However, two things must be noted: (1) The powers are the only way the players have to act on an organization. (2) The execution of an invoked power may request, in turn, the execution by the player of requirements needed to carry on the power, so the result of the power execution still depends on the player.

The second point is important, since the player may refuse to execute a requested requirement, and the requirements determine the outcome of the power, which thus varies from invocation to invocation and from player to player.

Since roles 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 expression "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, as in [20].

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.

- *requestResponsibility(String)*: this method asks to the player to invoke a power of the role.
- *requestRequirement(String)*: this method 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 the *requestRequirement* method and the *requestResponsibility* 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 endowed with beliefs represented, e.g., as Jess knowledge bases:

- *sendInform(String)*: this method is used to inform the player that the beliefs of the roles are changed. This does not assume that the player adopts the conveyed beliefs as well.
- *addBelief(Belief)* and *addGoal(Goal)*: they are invoked by the role's behaviours or by other roles' to update the state of the role.

Besides the connection with its player, which is regulated by the protocols described in the next section, 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.

Players. Players of roles in organizations are JADE agents, which can reside on different platforms with respect to the organization. However, since to play a role special behaviour is needed, 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 allows to model 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. Thus, an instance of the protocols and behaviours is needed for each role played. The list of roles played by the agent, and the state of each role (activated, deactivated), is kept in an hashtable.

The enactment procedure, described in the next section, 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 for 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 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 [11]:

- *Enact*. The communication protocol (which contains another finite state machine 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 finite state machine 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 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 to invoke powers arrives from the role agent it plays, the agent will decide whether 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 finite state machine, that can be seen in Figure 2, 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 activated, otherwise the messages remains in the buffer. More sophisticated solutions can be implemented as needed, but they must be aware of the synchronization problems (e.g., what happens if in the same moment a role sends a request to its player and the player puts the role in a deactivated state?).

The *Player* class offers some methods. They can be used in programming the other behaviours of the agent when it is necessary to make changes in 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 if necessary. It is left to the programmer how to manage the necessity of blocking of the behaviour till an answer is returned with the usual block instruction of JADE. This solution is coherent with the standard message exchange of JADE and allows to avoid using more sophisticated

behaviours based on threads. The methods offered by this class are the following.

- *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.
- *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 represents the id of the invocation.
- *receivePowerResult(int)*: to receive, if needed, the result of the invocation of a power (which is identified by means of the id).
- *deact(roleAID), activate(roleAID), deactivate(roleAID)* respectively definitively deacts the role, killing it and managing the data structure to remove all references to it, activates a role agent that is in the deactivate state, and temporarily deactivates the role agent (e.g., immediately after a successfully enact, the role agent goes in the deactivate state).
- *addRequirement(String)*: when extending the *Player* class it is necessary to specify which of the behaviours defined in it are requirements. I.e., the list of behaviours which can be requested by a role which is played by the agent. 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, defining a player requires to implement an abstract method to decide whether to execute the requirements upon request from the roles. The decision about the implementation of the method is

- *adoptGoal(String)*: it is used to make the player autonomous with reference to requests of role he plays: when the execution of a requirement is requested by the role, this method returns true if the agent decides to execute it.

III. Interaction between our actors

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. Note that the protocols are always split in two part: the side of the initiator and the one of the responder. While the organization is only the responder of a protocol, roles and players can be both initiators and responder.

We refer to Figure 3, that describes the sequence diagram for interaction.

Agents and the organization. Behind the enacting state of the player described in the previous section, there is an enactment protocol inherited, respectively, as concerns the initiator and the receiver, from 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 (e.g., *Buyer*) 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 (since it is implemented as an agent it has an 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. The final part of the protocol shows how the player and the role communicate, for example, for a power request. The role can require the execution of one or more requirements, from which can depend the power execution.

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. [11], 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.

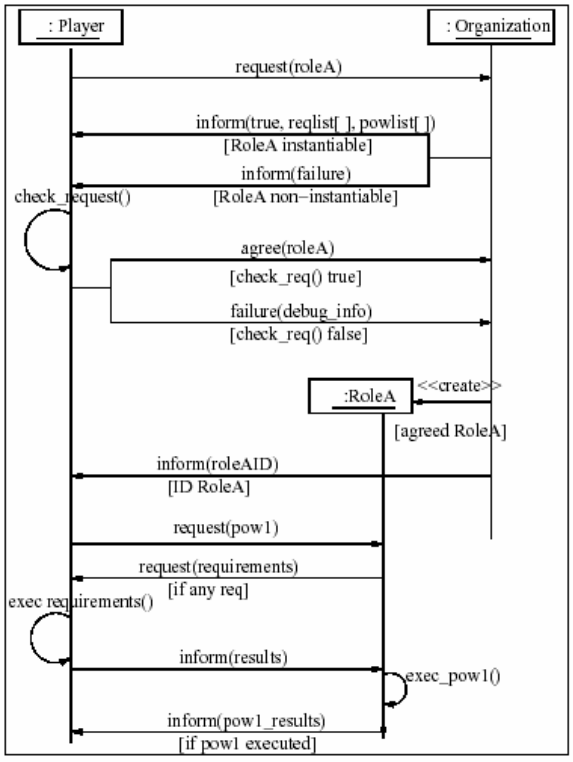

Fig. 3 - The interaction protocol

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 2 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, removing the couple (*Player*, *Role*) from its data structures.

IV. Future developments

Our extension of JADE is inspired to the model of [7] which has been implemented in Java creating the language powerJava [3]. With respect to powerJava, there are similarities and differences, which are due to the fact that in powerJava agents have been reduced to objects, losing some features.

Few agent languages are endowed with primitives for modeling organization. MetateM [14] 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 forget the difference between the play relation and the role-of relation which have different properties (see, e.g., [23]).

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 SMoise+ features [20], we will improve our system with agent sets and subsets 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. Concerning J-Moise+ [19], that is a combination of Jason [8] and Moise+ [18], we will enrich our platform integrating it with a rule engine (like Jess), becoming able to write beliefs and goals as rules. Very interesting is the matter of groups and schemes, that we will consider to implement.

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* [20]; 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.

V. CONCLUSIONS

In this paper we use the ontological model of organizations proposed in [7] to program organizations. We use as agent framework JADE since it provides the primitives to program the MAS in Java. We define a set of Java classes which extends the agent classes of JADE to have further primitives for building organizations structured into roles.

Organizations and roles are implemented as extension of the Agent JADE class: this allows to communicate remotely with them using protocols based on FIPA speech acts, to identify organizations and roles with an AID rather than with a memory reference, to put on yellow-pages services the information about them, and to program them using JADE *Behaviours*.

Roles, modelled as inner classes in Java, can be viewed as agents and as objects at the same time, depending on the perspective: for the external world they are agents with an AID and communication capabilities, from the perspective of the organizations they are objects which can be programmed in the traditional way. Being inner classes, all roles of an organization can be programmed as a single program, since they all belong to the same namespace.

Organizations and roles are agents with a limited autonomy, since they can act only via the actions of the players of the roles. The interaction between them is made via a set of protocols: a protocol between the organization and an agent to start enacting a role, a protocol between the player and its role to invoke powers of a role, a protocol between the role and its player to invoke the requirements, etc. The interaction among roles can happen via normal agent protocol or directly by method invocation, since they belong to the same namespace.

To play a role, an agent has first to contact the organization and then to execute the requirements requested by the role. These requests for requirements represent the expected behaviour of an agent in its role. The agent who plays a role can be in different states with respect to the roles: enacting it, activated, deactivated and deacted.

It is possible to verify that a player fulfills the requirements during the execution and not only before the enactment of a role. In the JADE framework, it is up to the programmers to decide which requirements to provide to an agent implementation at design time by extending the class that contains the role specification. The advantage of our proposal is that we only verify the presence of those requirements that are actually used by the agent during the specific execution, without considering all those requirements that are not necessary to achieve the current goal. In perspective, if the protocol specification were available for inspection before the decision of playing a role, an agent could verify a priori if it owns all the requirements needed for achieving its goal, disregarding all the others, in the line of what presented in [1]. This approach is also important whenever an agent is allowed to decide whether satisfying a requirement depending on the context of the execution. In some cases, it might either decide not to use a requirement that it actually has, or select which requirement to use among a set of available requirements.

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 [3] to implement roles as inner classes. Moreover, being based on Java 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. Under the programming and debugging point of view, an objection could be that the (possible) execution of player and role on different platform (and, then, in a separate way), can create difficult during the debugging phase. To solve this problem it is necessary to develop a new debugging protocol, based on message passing, that will be used to communicate, for example, when an exception is raised by a power. This solution is perfectly lined up to the JADE philosophy.

In this paper we do not consider the problem of structuring organizations in suborganizations nor to model federated organizations residing on different platforms. A prototype implementation has been constructed, as described in the paper. The requirements mechanism will be used to implements many other important features for MAS: obligations and permissions, groups (sets) and subgroups (subsets), plans.

REFERENCES

[1] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Reasoning on choreographies and capability requirements. International Journal of Business Process Integration and Management IJBPIM, 2(4), 2007.

[2] M. Baldoni, G. Boella, and L. van der Torre. Modelling the interaction between objects: Roles as affordances. In Knowledge Science, Engineering and Management, First International Conference, KSEM 2006, volume 4092 of LNCS, pages 42–54. Springer, 2006.

[3] M. Baldoni, G. Boella, and L. van der Torre. Interaction between Objects in powerJava. Journal of Object Technology, 6(2):7 –12, 2007.

[4] F. L. Bellifemine, G. Caire, and D. Greenwood. Developing Multi-Agent Systems with JADE. Wiley, 2007.

[5] G. Boella, R. Damiano, J. Hulstijn, and L. van der Torre. ACL semantics between social commitments and mental attitudes. In International Workshops on Agent Communication, AC 2005 and AC 2006, volume 3859 of LNAI, pages 30–44. Springer, Berlin, 2006.

[6] G. Boella, V. Genovese, R. Grenna, and L. der Torre. Roles in coordination and in agent deliberation: A merger of concepts. In Proceedings of Multi-Agent Logics. PRIMA 2007. Lecture Notes in Computer Science, Springer, 2007.

[7] G. Boella and L. van der Torre. Organizations as socially constructed agents in the agent oriented paradigm. In Engineering Societies in the Agents World V, 5th International Workshop (ESAW'04), volume 3451 of LNAI, pages 1 –13, Berlin , 2005. Springer.

[8] R. H. Bordini, J. F. Hubner, and M. Wooldrige. Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley, 2007.

[9] 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.

[10] A. Colman and J. Han. Roles, players and adaptable organizations. Applied Ontology, 2007.

[11] 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.

[12] J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: an organizational view of multiagent systems. In Agent-Oriented Software Engineering IV, 4th International Workshop (AOSE'03), volume 2935 of LNCS, pages 214–230, Berlin, 2003. Springer.

[13] FIPA. FIPA ACL message structure specification. Technical Report XC00061, Foundation for Intelligent Physical Agents, 2001.

[14] M. Fisher. A survey of concurrent metatem - the language and its applications. In ICTL, pages 480–505, 1994.

[15] M. Fisher, C. Ghidini, and B. Hirsch. Organising computation through dynamic grouping. In Objects, Agents, and Features, pages 117–136, 2003.

[16] 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.

[17] O. Gutknecht and J. Ferber. The madkit agent platform architecture. In Agents Workshop on Infrastructure for Multi-Agent Systems, pages 48–55, 2000.

[18] J. F. Hubner, J. S. Sichman, and O. Boissier. Developing organised multi-agent systems using the moise+ model: Programming issues at the system and agent levels. International Journal of Agent-Oriented Software Engineering., 2007.

[19] J. F. Huebner. J-Moise+ programming organizational agents with Moise+ and Jason. In http://moise.sourceforge.net/doc/tfg-eumas07-slides.pdf, 2007.

[20] J. F. Huebner, J. S. Sichman, and O. Boissier. S-moise+: A middleware for developing organised multi-agent systems. In O. Boissier, J. A. Padget, V. Dignum, G. Lindemann, E. T. Matson, S. Ossowski, J. S. Sichman, and J. Vzquez-Salceda, editors, AAMAS Workshops, volume 3913 of Lecture Notes in Computer Science, pages 64–78. Springer, 2005.

[21] A. Jones and M. Sergot. A formal characterisation of institutionalised power. Journal of IGPL, 3:427–443, 1996.

[22] 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.

[23] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. Data and Knowledge Engineering, 35:83–848, 2000.

[24] N. Tinnemeier, M. Dastani, and J.-J. C. Meyer. Orwell's nightmare for agents? programming multi-agent organisations. In Sixth international Workshop on Programming Multi-Agent Systems PROMAS'08, 2008.

[25] 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.

[26] 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.

Matteo Baldoni is Associated Professor at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, Torino, 10149, Italy. E-mail: baldoni@di.unito.it .

Guido Boella is Associated Professor at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, 10149, Torino, Italy. E-mail: guido@di.unito.it.

Mauro Dorni is student at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, 10149, Torino, Italy. E-mail: sp064535@educ.di.unito.it.

Andrea Mugnaini is student at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, 10149, Torino, Italy. E-mail: sp064278@educ.di.unito.it.

Roberto Grenna is PhD student at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, 10149, Torino, Italy E-mail: grenna@di.unito.it.