# A Middleware for modeling Organizations and Roles in Jade

Matteo Baldoni[1], Guido Boella[1], Valerio Genovese[1],
Andrea Mugnaini[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; {mugnaini81,valerio.click}@gmail.com
[2]Computer Science and Communications, University of Luxembourg, Luxembourg
E-mail: leon.vandertorre@uni.lu

**Abstract.** Organizations and roles are often seen as mental constructs, good to be used during the design phase in Multi Agent Systems, but they have also been considered as first class citizens in MAS, when objective coordination is needed. Roles facilitate the coordination of agents inside an organization, and they give new abilities in the context of organizations, called powers, to the agents which satisfy the necessary requirements to play them. No general purpose programming languages for multiagent systems offer primitives to program organizations and roles as instances existing at runtime, so, in this paper, we propose our extension of the Jade framework, with primitives to program in Java organizations structured in roles, and to enable agents to play roles in organizations. We provide classes and protocols which enable an agent to enact a new role in an organization, to interact with the role by invoking the execution of powers, and to receive new goals to be fulfilled. Since roles and organizations can be on a different platform with respect to the role players, the communication with them happens via protocols. Since they can have complex behaviours, they are implemented by extending the Jade agent class. Our aim is to give to programmers a middle tier, built on the Jade platform, useful to solve with minimal implementative effort many coordination problems, and to offer a first, implicit, management of norms and sanctions.

## 1 Introduction

Roles facilitate the coordination of agents inside an organization, giving new abilities in the context of organizations, called powers, to the agents which satisfy the requirements necessary to play them. Organizations and roles are often seen as mental constructs, good to be used during the design phase in MAS, but they have also been considered as first class citizens in multiagent systems [13], when objective coordination is needed. No general purpose programming languages for multiagent systems offer yet primitives to program organizations and roles as instances existing at runtime.

So, this paper answers the following research questions:

- How to introduce organizations and roles in a general purpose framework for programming multiagent systems?
- Which are the primitives to be added for programming organizations and roles?

– How it is possible to restructure roles during runtime?

As methodology, we build our proposal as an extension of the Jade multiagent system framework, with primitives to program, in Java, organizations structured in roles, for enabling agents to play roles in organizations. As ontological model of organizations and roles we select [8] which merges two different and complementary views or roles, providing an high level logical specification.

To pass from the logical specification to the design and implementation of a framework for programming multiagent systems, we provide classes and protocols which enable an agent to enact a new role in an organization, to interact with the role by invoking the execution of powers (as intended, in OO programming, in [11], and shortly explained in Section 2.5), and to receive new goals to be fulfilled. Since roles and organizations can be on a different platform with respect to the role players, the communication with them happens via protocols. Since they can have complex behaviours, they are implemented by extending the Jade agent class. Our aim is to give to programmers a middle tier, built on the Jade platform, useful to solve coordination problems with minimal implementative effort.

In this paper we do not consider yet the possibility to have BDI agents, even if both the ontological model (see [11]) and the Jade framework allow such extension. However, we give some hints on the usefulness of introducing BDI into our proposal. This issue will be faced in future work.

The paper is organized as follows. First, in Section 2, we summarize the model of organizations and roles we take inspiration from, and we give a short description of our concept of “role”, “organization”, “power”. In Section 2.6, we describe an example of a typical MAS situation that motivates our proposal; in Section 3 we describe how our model is realized introducing new packages in Jade; in Section 4 we discuss a possible powerJade solution to a practical problem (the manager-bidder one), and Section 5 will finish this paper with related work and conclusions.

## 2 The model of organizations and roles

Since we speak about organizations and roles, we need to specify our definition for both, and to refer to a formalized ontological model, in order to make clear our starting point, and to make understandable to programmers how to use the primitives introduced in the JADE framework. In the following subsections we shortly show a description of our concepts of roles and organizations, based on our previous works ([5]), then we give two different (but complementary) views about roles (see [11] and [14]), and we introduce a unified model starting from these, and define a well-founded meta-model. For last, we explain our concept of “power”, “capability”, and “requirement”.

### 2.1 Our definition for Organizations and Roles

In ontology and knowledge representation (like [9], [10], [23], and [25]), we can find a quite complete analysis for organizational roles. Our intention is to introduce a notion of role which could be well founded, on which there is a wide agreement and which is

justified independently from the practical problems we want to solve using it. We use a metaphor directly taken from organizational management. An organization (or, more generally, an institution) is not an "object" which can be manipulated from the outside, but instead it belongs to the social reality, and all the interactions with it can be performed only via the roles it offers ([12]). The utility of roles is not only for modeling domains including institutions and organizations, because we can consider every "object" as an institution or an organization offering different ways for interacting with it. Each way is offered by a different role. So, *our roles are based on an organizational metaphor*.

### 2.2 The Ontological Model for the Organization

In [11] an ontological analysis shows the following properties for roles:

- *Foundation*: a role instance has always to be associated to an instance of the organization to which it belongs, and to an instance of the player of the role too;
- *Definitional dependence*: the role definition depends from the one of the organization to which it belongs;
- *Institutional powers*: the operations defined into the role can access to the state of the organization, and of the other roles of the organization too;
- *Prerequisites*: to play a role, it is necessary to satisfy some requirements, that means that the player has to be able to do actions which can be used in the role's operations execution.

Also the model of [11] 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. So, they share some properties with agents, and, in some respects, can be modeled using similar primitives.

### 2.3 The Model for the Role Dynamics

[14]'s model focuses on role dynamics, rather than on their structure; four operations to deal with role dynamics are defined: *enact* and *deact*, which mean that an agent starts and ends occupying (playing) a role in a system, and *activate* and *deactivate*, which means respectively that an agent starts executing actions (operations) belonging to the role and suspends their execution. Although it is possible to have an agent with multiple roles enacted simultaneously (and, then, *played*), 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.

### 2.4 The Unified Model

Using the distinction of Omicini [26], we use the model presented in [11] as an objective coordination mechanism, in a similar way, for example, artifacts do: organizations are first class entities of the MAS rather than a mental construction which agents use to coordinate themselves. However, this model leaves unspecified how, given a role, its player will behave. So, we merge it with [14]'s model, to solve the problem of formally defining the dynamics of roles, by identifying the actions that can be done in a *open system*, such that agents can enter and leave. Organizations are not simple mental constructions, roles are not only abstractions used at design time, and players are not isolated agents: they are all agents interacting the one with the others. A logical specification of this integrated model can be found in [8].

Also considering what we can find in [7] about mental attitudes, we can summarize some points of the model:

- Roles are instances with associated beliefs and goals attributed to them. These mental attitudes are public.
- The public beliefs and goals attributed to roles are changed by speech acts executed either by the role or by other roles. The former case accounts for the addition of preconditions and of the intention to achieve the rational effect of a speech act, the latter one accounts for the case of commands or other speech acts presupposing a hierarchy of authority among roles.
- The agents execute speech acts via their roles.

This model has been applied to provide a semantics to both FIPA and Social Commitment approaches to agent communication languages, and this semantics overcomes the problem of the unverifiability of private mental attitudes of agents. The implementation of this model is shown in Section 3.

### 2.5 "Powers" and "capabilities"/"requirements" in our view

We know that roles work as "interfaces" between organizations and agents, and they give so called "powers" to agents. A power can extend agents abilities, allowing them to operate inside the organization and inside the state of the other roles. An example of such powers, called "institutional powers" in [22], is the signature of a director which counts as the commitment of the entire institution.

The powers added to the players, by mean of the roles, can be different for each role and, thus, represent different affordances offered by the organization to other agents to interact with it [4].

Powers are invoked by players on their roles, but they are executed by the roles, since they own both state and behaviour.

Respect to the "requirements", we consider them as *needed capabilities* that a candidate player must have to be able to play a particular role. An example of "capability" can be the ability for a bank employee to log into the software of his bank.

### 2.6 An example

The usefulness of the above proposal, that brings to the development of powerJade, introduced in the next section, can be seen through an example. The scenario we consider involves two organizations: a bank, and a software house. Bob has been engaged as a programmer in a software house. The software house management requires him to own a bank account, in order to directly deposit his salary on it. Bob goes to the bank, where the employee, George, gives him some forms to fill. Once that Bob has finished filling in the forms, George inputs the data on a terminal, creating a new account, which needs to be activated. George forwards the activation request to his director, Bill, who is the only person in the bank, able to activate an account. Once the account is activated, Bob will be a new bank customer.

Years later, become a project manager, Bob decides to buy a little house. He has to obtain a loan, and the bank director informs him that for calling a loan, his wage packet is needed. Bob calls to the management of the software house for his wage packet, and bring it to Bill. After some days (and other filled forms), the bank gives the loan to Bob, who can finally buy his new house.

Each organization *offers* some roles, which have to be *played* by some agents, called, for this reason, *players*. In the bank, Bob plays the *customer* role, while George plays the *employee* one, and Bill the *director* one. Since Bob interacts with both the organizations, he has to play a role also inside the software house: he enters as a *programmer*, but after some years he changes it, becoming a *project manager*. As a bank customer, Bob has some *powers*: to call for an account, to transfer money on it, to request for a loan. As a simple employee, George has only the power to create Bob's account, but the account activation has to be done by Bill, the director. The call for activation is done by mean of a specific George's call to Bill, for the execution of a *responsibility*. Also in the case of the loan request, the director has also to manage the situation, maybe examining Bob's account, and calling him for his wage packet. Another Bob's power is to call for his wage packet into the software house. Speaking about personal capabilities, we can imagine that Bill, in order to access to the bank procedures for which he is enabled, must fill a login page with his ID and password; the same happens for George too, and for Bob, in the moment in which he accesses to his account using Internet. However, Bob has also another capability, that is *requested* when he plays the programmer role (but the same happens for the project manager one): to give his login name and password for entering the enterprise IT system. Finally, the director is required to have more complex capabilities, like evaluating the solvency of a client requesting a loan.

## 3 PowerJade

The main idea of our work is to offer to agents programmers a complete middle tier with the primitives for implementing organizations, roles, and players in Jade (see Figure 1), giving a declarative model based on FSM, which is modifiable also at run-time. We called this middleware *powerJade*, remembering the importance of powers in the interaction between roles and organizations. The powerJade conceptual model is inspired to *open systems*: participants can enter in the system and leave from it whenever they

want. For granting this condition, and for managing the (possible) continuous operations for enacting, activating, deactivating, and deactivating roles (in an asynchronous and dynamic way), many protocols have been implemented. Another starting point was the re-use of the software structure already implemented in powerJava [5], based on an intensive use of so-called *inner classes*.

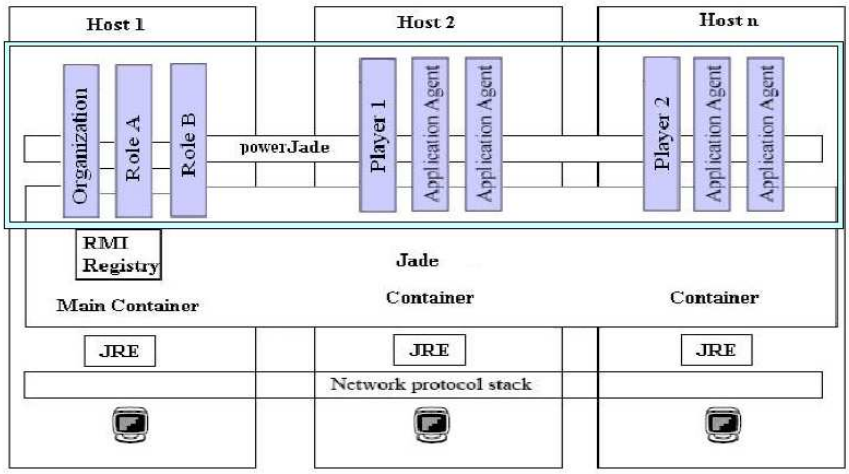


**Fig. 1.** The Jade architecture and the powerJade middle tier.

In order to give an implementation based on the conceptual model we discussed in Section 2.4, not only the three subclasses of the Jade `Agent` class (`Organization`, `Role`, and `Player`) have been realized (they will be described in Sections 3.1, 3.2, 3.3), but also classes for other central concepts, like `Power`, and `Requirement` were implemented (and shown in Sections 3.2, 3.3). For representing the dynamics of the roles, we also implemented all the needed communication protocols, that will be described in Section 3.4.

It can sound strange that we implemented `Organization`, `Role`, and `Player` as subclasses of the Jade `Agent` class. We adopted this solution for taking advantage of the possibility for agents in Jade to communicate by mean of FIPA messages. The only agent having the full autonomy is the `Player`: it can decide whether enact a role, or activating/deactivating it, and also to deact it. The `Role`, instead, has a very limited autonomy, due, for example, to hierarchical motivation (but, however, it's always the `Player` which decides...). Respect to the `Organization`, we would like to extend our middleware giving the opportunity to an organization to play a role inside another one, like in the real world. This consideration also answers the question: what does it bring to program roles and organizations as instances? More details can be found in our previous works, where we started from a theoretical well-formed model for organizations and roles ([8]) applied it to object-oriented programming (with the language

powerJava [5]), then to multiagent systems [3], up to the web world (currently under implementation [6]).

`Organization`, `Role`, and `Player` have similar structures: they contain a finite state machine behaviour instance which manages the interaction at the level of the new middle tier by means of suitable protocols for communication.

To implement each protocol in Jade two further FSMBehaviour are necessary, each one dealing the part of the protocol of the two interacting parties; for example, the enactment protocol between the organization and the player requires two FSMBehaviours, one in the organization and one in the player.

### 3.1 The `Organization` Class

The `Organization` class is structured as in Figure 2. The `OrgManagerBehaviour` is a finite state machine behaviour created inside the `setup()` method of `Organization`. It operates in parallel with other behaviours created by the programmer of the organization, and allows the organization to interact via the middle tier. Its task is to manage the enact and deact requests done by the players. At each iteration, the `OrgManagerBehaviour` looks for any message having the *ORGANIZATION_PROTOCOL* and the performative *ACLMessage.Request*. `EnactProtocolOrganization` and `DeactProtocolOrganization` are the counterpart of the respective protocols inside the players which realize the interaction between organizations and players: instances of these two classes are created by the `OrgManagerBehaviour` when needed.

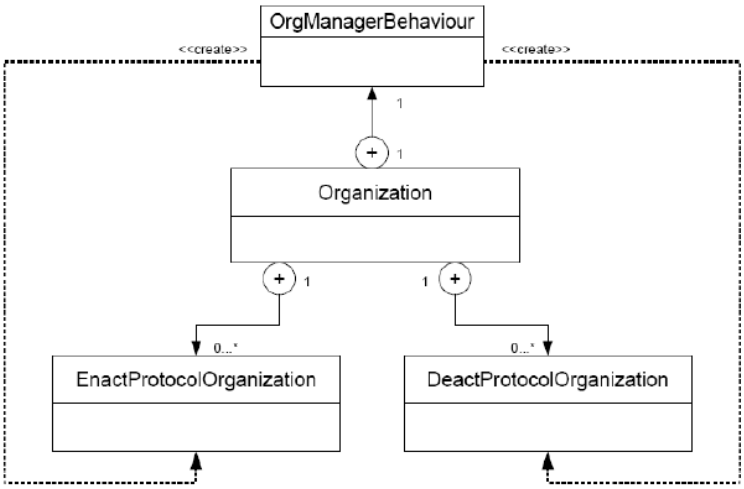


**Fig. 2.** The Organization diagram.

When the `OrgManagerBehaviour` detects a message to manage, it extracts the sender's AID, and the type of request required. In case of an *Enact* request (and whether

all the controls described on Subsection 3.4 about the *Enact* protocol succeeded), a new instance of `EnactProtocolOrganization` behaviour is created, and added to the queue of behaviours to be executed. The same happens (with a new instance of the `DeactProtocolOrganization` behaviour) if a *Deact* request has been done, while if the controls related to the requested protocol will not succeed, the iteration terminate, and the `OrgManagerBehaviour` takes again its cycle. In the behavioural part of this class, programmers can add a "normative" control on the players' good intentions, and managing the possibility of discovering lies before enacting the role, or immediately after having enact it (and before w.r.t. its activation). Primitives implementing these controls are ongoing work.

### 3.2 The `Role` Class

As described in [3], the `Role` class is an `Agent` subclass, but also an `Organization` *inner class*. Using this solution, each role can access to the internal state of the organization, and to the internal state of other roles too. Like the `Organization` class has the `OrgManagerBehaviour`, the `Role` has the `RoleManagerBehaviour`, a finite state machine behaviour created inside the `setup()` method of `Role`. Its task is to manage the commands (messages) coming from the player: a power invocation, an *Activate*, or a *Deactivate*.

Inside the role, an instance of the `PowerManager` class is present. The `PowerManager` is a `FSMBehaviour` subclass, and it has the whole list of the powers of the role (linked as states of the FSM). It is composed as follows:

- a first state, the `ManagerPowerState`, that must understand which power has been invoked;
- a final state, the `ResultManager`, that has to give the power result to its caller;
- a self-created and linked state for each power implemented by the role programmer.

All the transitions between states are added at run-time to the FSM, respecting the code written by the programmer.

**The Powers** Powers are a fundamental part of our middleware. They can be invoked by a player on the active role in the particular moment of the invocation, and they represent the possibility of action for that role inside the organization. For coherence with the Jade framework and to exploit the scheduling facility, powers are implemented as behaviours, taking also advantage from their more declarative character with respect to methods.

Some times, a power execution needs some requirements to be completed; this is a sort of remote method call dealt by our middleware, since requirements are player's actions. In our example, George, as bank employee, has the *power* for creating a bank account for a customer; to exercise this power, George as player has to input his credentials: the login and the password.

The problem to be solved is that players' requirement invocation must be transparent to the role programmer, who should be relieved from dealing the message exchange with the player.

We modeled the class `Power` as a `FSMBehaviour` subclass, where the complete finite state machine is automatically constructed from a declarative specification containing the component behaviours to be executed by the role and the name of the requirements to be executed by the player; in this way, we can manage the request for any requirement as a particular state of the FSM. When a requirement is required, a `RequestRequirementState` (that is another subclass of `FSMBehaviour`) is automatically added in the correct point invoking the required requirement by means of a protocol: the programmer has only to specify the requirement name.

The complexity of this kind of interaction is shown in Figure 3. The great balloon indicating one of the powers for that particular role contains the final state machine obtained writing the following code:

```
addState(new myState1("S1", "R1", "E1"));
addState(new myState2("S2"));
```

where *S1* and *S2* are names of possibly complex behaviours implemented by the role programmer which will be instanced and added to the finite state machine representing the power, *R1* is the name requested requirement, and *E1* is a behaviour representing the error management state. Analyzing the structure of the power, we can see that the execution of the first state *S1* is followed by a macro-state (that is a `FSMBehaviour`), managing the request for a requirement, automatically created by the `addState()` method. This state will send to the player the request for the needed requirement, also managing the possible parameters, waiting for the answer. Whether the answer is positive, the transition to the following state of the power is done (or to the `ResultManager`, if needed); otherwise, the error can be managed (if possible), or the power is aborted. The `ErrorManager` is a particular state that allows to manage all the possible kinds of error, including the case of a player lying about its requirements).

Error management is done via the middle tier. We can detect two kinds of possible errors: (i) the *accidental* ones, and (ii) the *voluntary* ones. Typical cases of the (i) are the "practical" problems (i.e. network too busy and timeout expired), or the ones linked to a player bad working (also, a programming problem); those indicated as (ii) are closely linked to an incorrect (malicious) behaviour of the player, like the case in which an agent lies about on its requirements during an enact protocol, sending a message telling that it owns them, while this is not the truth. The latter case of error managing allows to the organization and roles programmer a first, rough, implicit, normative and sanctioning mechanism: if the player, for any reason, shows a lack of requirements, it could be obliged to the deact protocol w.r.t. that particular role, or it can be "marked" with a negative score, that could mean a lower trust level exercised from the organization to it.

An advantage given by using a declarative mechanism like behaviours (we can use instructions to add or remove states to a FSMBehaviour) for modeling powers is that new powers can be dynamically added or removed from the role. It is sufficient to add or remove (at run-time) transactions linking the power to the `ManagerPowerState` which is a FSMBehaviour too.

This mechanism can be used to model both dynamics of roles in organizational change or access restrictions. In the former case we can model situations like the power of the director to add to the employee the power of giving loans. In the latter case,

we can model security restriction by removing powers from roles, thus avoiding the situation where first a power is invoked and then aborted after controlling an access control list.

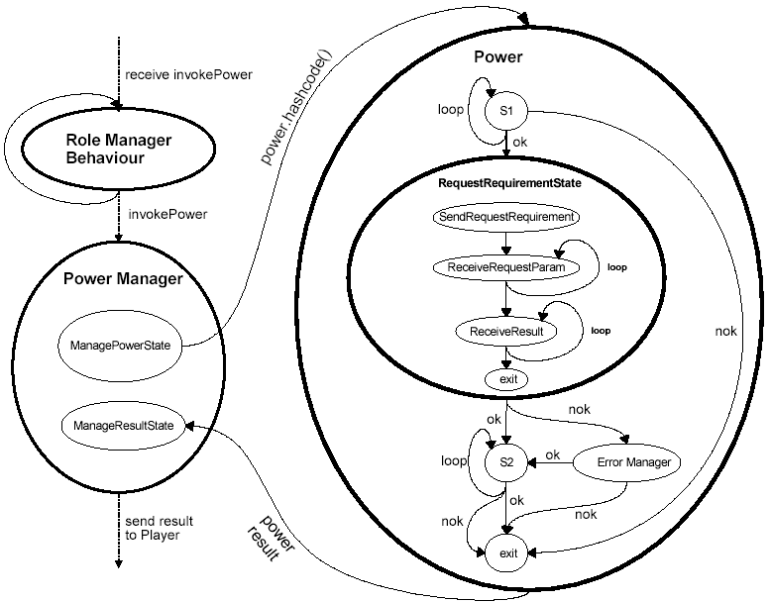


**Fig. 3.** Power management.

### 3.3 The `Player` Class

Analogously to `Organization` and `Role`, also the `Player` class is an `Agent` subclass. Like in the other two cases, we have a `PlayerManagerBehaviour`, a FSM-Behaviour managing all the possible messages that the player can receive. The player is the only agent totally autonomous. It contains other behaviours created by the agent programmer which are scheduled in parallel with the manager behaviour and it can obviously also interact with other agents, not involved in any organization (since the communication protocol existing in Jade always continues working), but it's constrained to interact with any kind of organization using a role offered by the organization itself. Each communication with another agent inside the organization can be done only via roles. Any other activity, communication, or action that both the agents could do without passing through their roles will not have effect on the internal state of the organization at all. Only the player can use all the four protocols described in Subsection 2.3: *Enact* and *Deact* with the organization, *Activate* and *Deactivate* with the role.

It's important to notice that, as we discuss in Subsection 3.4, only the `Player` is able to start the *Enact* protocol, which can be refused by the `Organization`, but it

has the absolute control on activation and deactivation of the roles (once enacted). A role can be deacted because the `Player` wants to deact it (e.g., a clerk which leave his enterprise for a new job), or because the `Organization` force the deacting (the same clerk can be fired).

While the role has to manage powers, the player deals with requirements: this is done by a `RequirementManager`.

The `Player` class offers some methods. They can be used in programming the other behaviours of the agent when it is necessary to make change to the state of role playing or to invoke powers. We assume invocations of powers to be asynchronous via the `invokePower` method from any behaviour implemented by the programmer. The call informs the `PlayerManagerBehaviour` which starts the interaction with the role and 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 behaviour can then consult the `PlayerManagerBehaviour` to get the return value of the power if it is available.

The player, once having invoked a power, stays waiting, i.e., for messages o requests from the active role. When the role needs for some requirements, the `PlayerManagerBehaviour` passes the control to the `RequirementManager`, which execute all the tasks which are needed.

It's important to notice that a player can always grow w.r.t. its capabilities/requirements.

A player can know organizations and roles on the platform by using the *Yellow Pages* mechanism, that is a basic JADE feature.

**The Requirements** Requirements are, for a player, a subset of the behaviours representing its capabilities, and, in some sense, the plans for achieve the personal goals of the agent. Playing a role, an agent can achieve multiple goals (i.e., the goals achievable invoking a power), but, in a general case, the execution of one or more requirements can be needed during the invocation of a power. Referring to our bank example, George can achieve many goals dealing with its employee role (i.e., create a new account), but to do it, it's necessary for him to log in inside the bank IT system. Seen as a requirement, its log in capability denote his "attitude", his "possibility" of playing his employee role.

During the enact protocol, the organization sends (see Section 3.4) to the agent wanting to play one of its roles, the list of requirements to be fulfilled. As we said, the candidate player could lie (e.g., telling that it's able to fulfill some requirement), entering in the role in a not honest way. However, the organization and role programmer has all the possibility for checking the truth of the candidate player's answer before it begins to play the role, not enacting it, or deacting immediately after the enact. Also this kind of choice has been done to grant the highest freedom degree

### 3.4 Communication Protocols

In this Section, an example of a complex communication between a player, an organization, and a role is shown. We have to make some preliminary considerations,

about communication. Each protocol is split in two, specular, but complementary behaviours, one for each actor. In fact, if we consider a communication, two "roles" may be seen: an initiator, which is the object sending the first message, and a responder, which never can begin a communication. For example, when a player wants to play a role inside an organization, an `EnactProtocolPlayer` instance is created. The player is the initiator, and a request for a role is done from its new behaviour to the `OrgManagerBehaviour`, which instantiates an `EnactProtocolOrganization` behaviour. This behaviour will manage the request, sending to the `EnactProtocolPlayer` an `Inform` containing the list of the requirement needed to play the requested role.

The `EnactProtocolPlayer` evaluates the list, answering to the organization part whether it agrees (notice that the player programmer could implement a behaviour that always answers in a positive way, that sounds like a lie). Only after receiving the agreement, the `EnactProtocolOrganization` creates a `RoleManager` instance, and sends the AID of the role just created to the player. The protocol ends with the update by the player of its internal state.

Since the instance of a role, once created, is not yet activated, when the player wants to "use" a role, has to activate it. Only one role at a time is active, while the others, for which the agent finished successfully the enactment protocol, are deactivated. The activation protocol moves from the player to the role instance. The player creates an `ActivateProtocolPlayer`, which sends a message to the role, calling for the activation. This message produces a change into the internal state of the role, which answers with an `inform` telling its agreement.

Once the role has been activated, the player can proceed with a power invocation. As we discussed in [3], this is not the only way in which player and role instance can communicate. We consider it, since it can require a complex interaction, beginning from the `invoke` done by the player on a power of the role. As we shown in Subsection 3.2, the power management can involve the request to the player for the execution of one or more requirements. In this case, the role sends a `request` with the list of requirements to be fulfilled. The player, since autonomous, can evaluate the opportunity to execute the requirement(s), and take the result(s) to the role (using an `inform`, waiting for the execution of the power and for receiving the `inform` with the result. A particular case, not visible in Figure 4, is the one in which the player, for any reason, does not execute the required requirements. This "bad" interaction will finish with an automatic deactment of the role.

## 4 The CNP scenario in powerJade

In Section 2.6, we discussed the bank example, trying to focus on roles' powers, players' requirements, responsibility calls, and all that has a place in our middleware. In this Section, we want to show a more technical example: the CNP one, or manager-bidder problem. In the left part of Figure 5, the inside of a player is shown, with its `Player Manager`, and its `RequirementManager` too.

Attached to the `RequirementManager`, we can see two possible requirements which can be involved in our example: they can be requested by a role, in this case, the

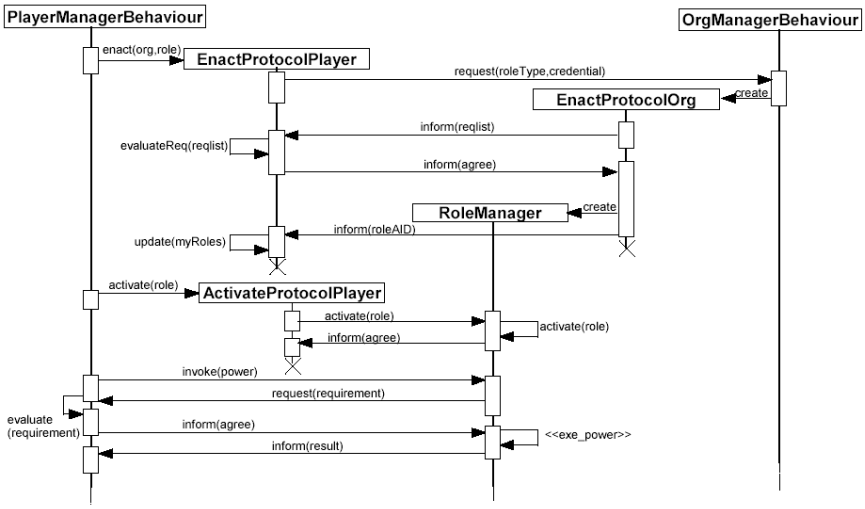


**Fig. 4.** The Sequence Diagram for a complex communication.

`M_CNP`, which is shown in the right part of the same Figure. The double rounded states are the ones created at run-time for the power called *CNP*. By figure 6 we try to explain a little part of the interactions between the player playing the manager role, and its role.

Our scenario considers an agent doing one of its behaviours. In a particular moment, a task has to be executed, but the agent knows that it cannot execute it, since this job is not achievable with its capabilities. The only solution is to find someone able to execute the task, possibly paying the least is possible. The agent has no knowledge about the ContractNet Protocol, but it knows that there is an organization that offers the CNP by mean of its roles. The (candidate) player contacts the organization, starting the enact protocol for the role of manager in the CNP `M_CNP`. The organization sends the list of requirements to be fulfilled, composed by the "task" requirement (that is the ability to send a task for a call for proposal operation), and the "evaluate" task (that is the ability to evaluate the various bidders' proposals, choosing the best one). The candidate player owns the requirements, so the role is created. When the player come to execute once again the behaviour containing the not executable task is, an `invokePower()` is executed, calling for the power with name *CNP* (the bold arc labeled with letter *a*, starting from the state with label *1* in the left part and arriving in the right part of Figure 6, to the `RoleManager`, which has label *2*).

The `RoleManager` passes the control to the `PowerManager` (label *3*), which finds the appropriate power in its list, and begins its execution (the first state is the one with label *4*. The first step is the call for the task which will be evaluated by bidders, but the task is known only by the player of the role, so a request for a requirement is done (label *5*). This means, a message is sent to the player, calling for the execution of a requirement (in this case, "task"). The dotted arc with label *b* simulates this message, which is received and managed by the `PlayerManager` (label *6*, which passes the

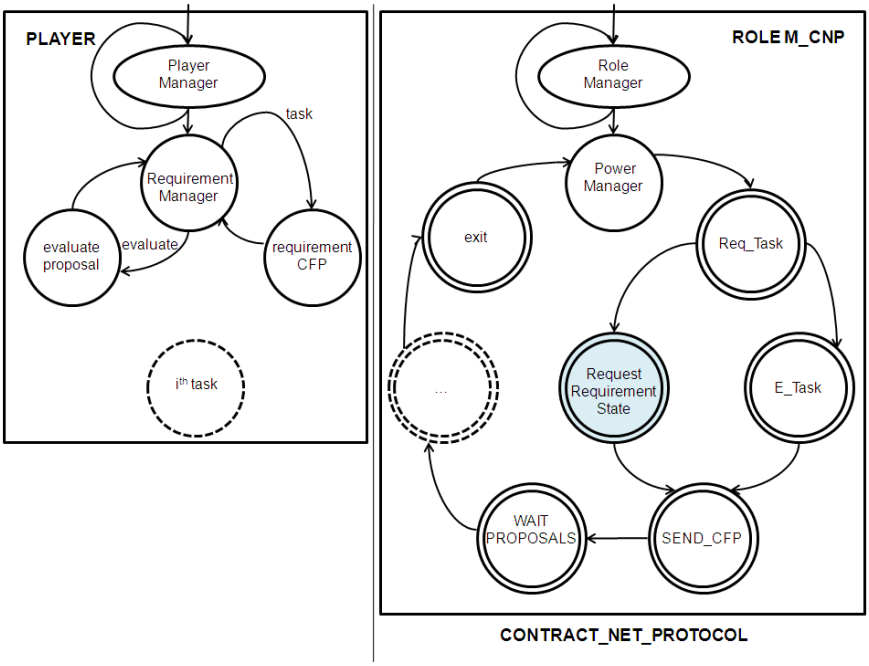


**Fig. 5.** Player and Manager role for the CNP example. The player is the candidate to play the Manager role in the CNP. The interaction after a successfully enactment is shown in Figure 6.

control to the `RequirementManager` (*7*). Once identified the requirement required, the it's reached the correct state (label *8*). After the execution, the control is passed again to the `RequirementManager` (label *9*), which passes, through an appropriate message, the result(s) to the role's `RequestRequirementState` (simulated by the dotted arc with label *c*). Supposing that a valid result has been returned, the power execution goes on, arriving (label *10*) to the `SEND_CFP` state, that provides the call for proposal to any bidder known inside the organization (dotted arc with label *4*, we assume that some agents already enacted the bidder role), going directly to add the appropriate behaviour to the `PowerManager` of the `B_CNP` instances found. The bidder roles will send messages back to the manager role, after requesting to their players the requirement to specify or not a price for the task to be delegated (with a process that is quite the same of the one just described).

The complex interaction between players and their roles, and between role and role, is executed *without* that players have to know the CNP dynamics, since all the complexity has been introduced in the roles. For the player playing the manager role, and for the ones playing the bidder role, the organization is a kind of black box; roles are the "wizards" managing the communication logics, and opportunely calling operations to be done by the players (that are absolutely *autonomous*: they are the only agents able to take decisions).

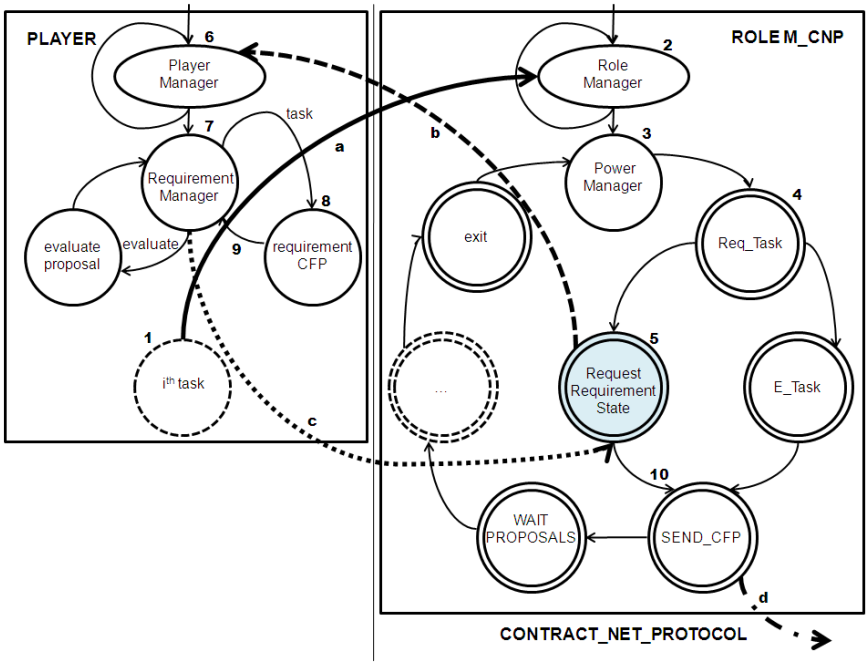


**Fig. 6.** The various interactions between player and role. The double circled states inside the role `M_CNP` are the ones composing the ContractNet power.

## 5 Related and future works, and conclusions

Speaking about organizations and roles, we can find very different approaches and results: models like the one in [17], applications modeling organizations or institutions like in [26], software engineering methods using organizational concepts like roles (GAIA, in [32]). GAIA is a general methodology which can be applied to a wide range of MAS, but also deals with social (macro) level, and agent (micro) level. Under the GAIA vision, a MAS can be modeled as a computational organization composed by many interactive roles. Regarding the analysis of organizations, in [30] can be found what is called the perspective of computational organization theory and artificial intelligence, in which organizations are basically described at the role, and group, composed of roles, levels.

Under the point of view of programming languages, an example is 3APL [31]. 3APL is a programming language developed for implementing cognitive agents and also for programming constructs for implementing cognitive agents and provides programming constructs for implementing agents' beliefs, goals, basic capabilities (e.g., beliefs update, etc.), and a set of practical reasoning rules for updating or revising agents' goals, but it has not primitives for modeling organizations and roles. Another language is the Normative Multi-Agent Programming Language in [29], which is more oriented to model the institutional structure composed by obligations, more than the organiza-

tional structure composed by roles. ISLANDER [15], is a tool for the definition and verification of agent mediated electronic institutions. The declarative textual language of ISLANDER can be used for specifying the institution components, and a graphical editor is also available. The definition of organizations as electronic institutions is done mainly in terms of norms and rules.

Speaking about frameworks, MetateM is a multi-agent framework using a language in which each agent is programmed using a set of (augmented) temporal logic specifications of the behaviour it should adopt. The behaviour is generated by the direct execution of the specifications. The framework is based on the notion of group, and is BDI oriented. Even if the language is not general purpose, the idea of groups can be considered similar to the one of roles. Moise [19] is an organizational model that want help the developer to cope with the agent-centered and the organizational-centered approaches. The MOISE model is structured along three levels: the individual level (in which, for each agent, is present the definition of its responsibilities), the agency level (in which are present aggregations of agents in large structures), and the society level (in which are defined global structuring and interconnections of the agents, and their structures with each other). SMoise+ [21] (which ensures that agents will follow the organizational constraints, is suitable for open systems, and supports for reorganisation) and J-MOISE+ [20] (which is more oriented to programming *how* agents play roles in organizations) are framework based on the MOISE model, but seem to be limited for programming organizations. MadKit [18] is a modular and scalable multiagent platform written in Java. It's built upon the AGR (Agent/Group/Role) organizational model: agents are situated in groups and play roles. It allows high heterogeneity in agent architectures and communication languages, and various customizations, but seems to be also limited in programming organizations.

With respect to organizational structures, Holonic MAS [28] present particular pyramidal organizations in which agents of a layer (under the same coordinator, also known as the holon's *head*) are able to communicate and to negotiate directly between them [1]. Any holon that is part of a whole is thought to contribute to achieving the goals of this superior whole. Apart from the head, each holon consist of a (possibly empty) set of other agents, called body agents. Roles and groups can express quite naturally in Holonic structures, under the previously described perspective.

Looking at agent platforms, there are two other—other than JADE—which can be considered relevant in this context. First, JACK Intelligent Agents [2] supports organizational structures through its Team Mode, where goals can be delegated to team member in order to achieve the team goals. JADEX [27] presents another interesting platform for the implementation of organizations, even if it does not currently have organizational structures.

[24] make a very similar proposal to powerJade. However, it does not propose a middle tier supported by a set of managers and behaviours making all the communication transparent to agent programmers. It presents a simpler approach that relies mostly on the extension of agents through behaviours and represents Roles as components on an ontology, while our approach presents a slightly more complex approach, in which roles are implemented as agents that provide further decoupling by brokering between

organizations and players, and provides a state machine that permits precise monitoring of the state of the roles.

In this paper we introduce organizations and roles as new classes in the Jade framework which are supported by a middle tier offering to agents the possibility to enact roles, invoke powers and to coordinate inside an organization.

The framework is based on a set of FSMBehaviours which realize the middle tier by means of managers keeping track of the state of interaction and protocols to make the various entities communicate with each other.

Powers offered by roles to players have a declarative nature that does not only make them easier to be programmed, but allows the organization to dynamically add and remove powers so to have a restructuring of the roles.

The normative part of our work has to be improved, since, at the moment, only a kind of "implicit" one is present. It can be seen, for example, in the constraints which make possible to play a role only if some requirements are respected. We are also considering possible merge with Jess (in order to use an engine for goals processing), and Jason, and some works using defeasible logic [16], in order to obtain the BDI part which is not present at this moment. We are also applying our model to the web, in order to use roles and organizations, and to improve the concept of *session*, introducing a typed, and permanent, session.

## References


1. E. Adam and R. Mandiau. Roles and hierarchy in multi-agent organizations. In *CEEMAS*, pages 539–542, 2005.
2. AOS. JACK Intelligent Agents, The Agent Oriented Software Group (AOS), http://www.agent-software.com/shared/home/, 2006.
3. M. Baldoni, G. Boella, V. Genovese, R. Grenna, and L. van der Torre. How to Program Organizations and Roles in the JADE Framework. In *MATES*, pages 25–36, 2008.
4. 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.
5. M. Baldoni, G. Boella, and L. van der Torre. Interaction between Objects in powerJava. *Journal of Object Technology*, 6(2):7–12, 2007.
6. G. Boella, A. Cerisara, and R. Grenna. Roles in building web applications using java. In *Procs. of RAOOL'09*, ECOOP'09 - Genova, 2009.
7. 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.
8. G. Boella, V. Genovese, R. Grenna, and L. der Torre. Roles in coordination and in agent deliberation: A merger of concepts. *PRIMA 2007*, 2007.
9. G. Boella and L. van der Torre. An agent oriented ontology of social reality. In *Procs. of Formal Ontologies in Information Systems (FOIS'04)*, pages 199–209, Amsterdam, 2004. IOS.
10. G. Boella and L. van der Torre. The ontological properties of social roles: Definitional dependence, powers and roles playing roles. In *Procs. of LOAIT workshop at ICAIL'05*, 2005.

11. 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.
12. G. Boella and L. van der Torre. A foundational ontology of organizations and roles. In *Declarative Agent Languages and Technologies IV, 4th International Workshop (DALT'06)*, volume 4327 of *LNCS*, pages 78–88, 2006.
13. A. Colman and J. Han. Roles, players and adaptable organizations. *Applied Ontology*, 2007.
14. 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.
15. M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In *AAMAS*, pages 1045–1052, New York, NY, USA, 2002. ACM.
16. G. Governatori, M. J. Maher, G. Antoniou, and D. Billington. Argumentation semantics for defeasible logic. page 675, 2004.
17. 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.
18. O. Gutknecht and J. Ferber. The madkit agent platform architecture. In *Agents Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, 2000.
19. M. Hannoun, O. Boissier, J. S. Sichman, and C. Sayettat. Moise: An organizational model for multi-agent systems. In *IBERAMIA-SBIA*, pages 156–165, 2000.
20. J. F. Huebner. J-Moise$^+$ programming organizational agents with Moise$^+$ and Jason. In *http://moise.sourceforge.net/doc/tfg-eumas07-slides.pdf*, 2007.
21. 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.
22. A. Jones and M. Sergot. A formal characterisation of institutionalised power. *Journal of IGPL*, 3:427–443, 1996.
23. F. Loebe. Abstract vs. social roles - a refined top-level ontological analysis. In *Procs. of AAAI Fall Symposium Roles'05*, pages 93–100. AAAI Press, 2005.
24. C. Madrigal-Mora, E. León-Soto, and K. Fischer. Implementing Organisations in JADE. In *MATES*, pages 135–146, 2008.
25. C. Masolo, L. Vieu, E. Bottazzi, C. Catenacci, R. Ferrario, A. Gangemi, and N. Guarino. Social roles and their descriptions. In *Procs. of Conference on the Principles of Knowledge Representation and Reasoning (KR'04)*, pages 267–277. AAAI Press, 2004.
26. 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.
27. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a bdi-infrastructure for jade agents. *EXP*, 3(3):76–85, 9 2003.
28. M. Schillo and K. Fischer. A taxonomy of autonomy in multiagent organisation. In *Agents and Computational Autonomy*, pages 68–82, 2003.
29. 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.
30. E. L. van den Broek, C. M. Jonker, A. Sharpanskykh, J. Treur, and P. Yolum. Formal modeling and analysis of organizations. In *AAMAS Workshops*, pages 18–34, 2005.
31. 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.
32. 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.