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 requirements necessary 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 [8], when objective coordination is needed. No general purpose programming languages for multiagent systems offer primitives to program organizations and roles as instances existing at runtime, yet.

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?

Another subquestion could be the following: what does it bring to program roles and organisations as instances?

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 [6] 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 [7], and shortly explained in Section 2.4), 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 coordination problems.

We test our proposal on a possible scenario, highlighting the features of our model.

In this paper we do not consider the possibility to have BDI agents, even if both the ontological model (see [7]) and the Jade framework allow such extension.

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 “power”. In Section 3, we describe an example of a typical MAS situation in the real life; in Section 4 we describe how our model is realized introducing new packages in Jade; in Section 5 we discuss a possible powerJade solution to a practical problem (the manager-bidder one), and Section 6 will finsh this paper with related work and conclusions.

2 The model of organizations and roles

Since we speak about organizations and roles, we need to refer to a formalized ontological model, in order to avoid ad hoc solutions imposed by the Jade framework, and to make understandable to programmers how to use the primitives. In the following subsections we shortly show two different (but complementary) views about roles (see [7] and [9]), and we introduce a unified model starting from these, and define a well-founded metamodel. Then, we explain our concept of “power”.

2.1 The Ontological Model for the Organization

In [7] 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 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 requisites, 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 [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. So, they share some properties with agents, and, in some respects, can be modelled using similar primitives.

2.2 The Model for the Role Dynamics

[9]'s model focus 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 finishes to occupy (play) 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, 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.

2.3 The Unified Model

Using the distinction of Omicini [19], we use the model presented in [7] 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 [9]'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 [6].

2.4 "Powers" in our view

We knows 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 other roles. An example of such

powers, called “institutional powers” in [17], 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.

3 An example of MAS in real life

We will start with a real-life example, in order to explain a common situation that could be modeled with a Multi Agent System application. The scenario we want to 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 imposes to him the owning of a bank account, in order to directly deposit his salary on it. Bob goes to the bank, where the employee, George, gives him some templates to fill. Once that Bob finished compiling the modules, George inputs the data on the terminal, creating the new account, which needs to be activated. George forwards the activation request to his director, Bill, who is the only able to activate an account in all the bank. Once that the account will be 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 templates filled), 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. George, being a simple employee, has 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 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 access to his account using Internet. Bob, however, 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.

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

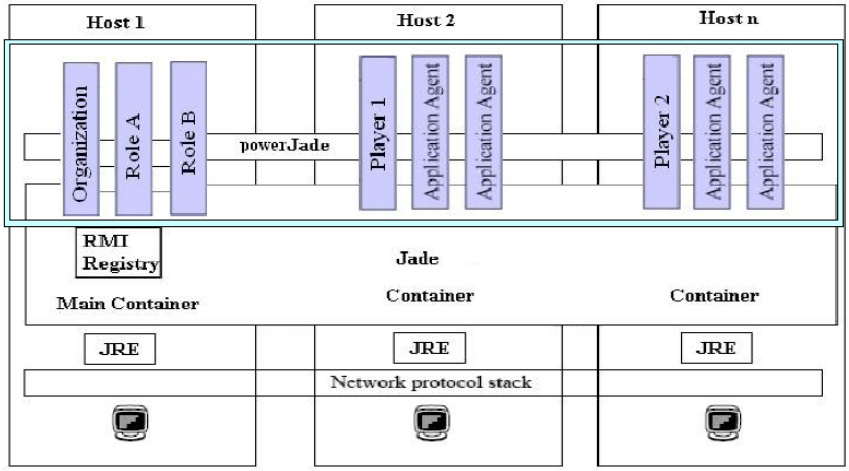

Fig. 1. The Jade architecture and the powerJade middle tier.

In order to give an implementation based on the conceptual model we discussed in Section 2.3, not only the three subclasses of the Jade `Agent` class (`Organization`, `Role`, and `Player`) have been realized (they will be described in Sections 4.1, 4.2, 4.3), but also classes for other central concepts, like `Power`, and `Requirement` were implemented (and showed in Sections 4.2, 4.3). For representing the dynamics of the roles, we implemented also all the needed communication protocols, that will be described in Section 4.4.

`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 interactants; for example, the enactment

protocol between the organization and the player requires two FSMBehaviours, one in the organization and one in the player.

4.1 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.

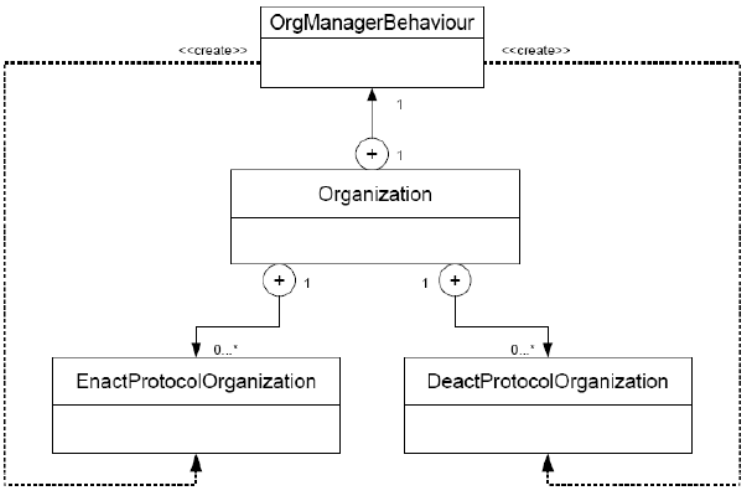

Fig. 2. The Organization diagram.

When the `OrgManagerBehaviour` recognize 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 4.4 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.

4.2 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, getting also advantage of 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* of 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

added automatically 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, also the case in which a player lied about its requirements).

Error management is done via the middle tier. We can individualize 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 behaviour of the player, like the case in which an agent lied on its requirements during an enact protocol. The latter case of error managing allows to the organization and roles programmer a fist, rough, implicit, normative and sanctionative 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 for modelling powers is that new powers can be dynamically added or removed from the role. It is sufficient to add or remove 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, so to avoid the situation where first a power is invoked and then aborted after controlling an access control list.

4.3 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-

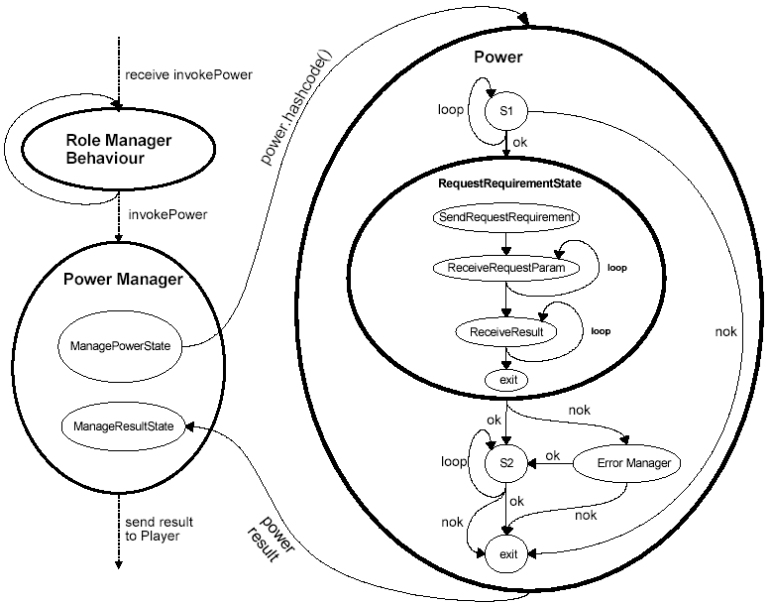

Fig. 3. Power management.

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. In case of a communication with another agent inside the organization, it 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 2.2: *Enact* and *Deact* with the organization, *Activate* and *Deactivate* with the role. 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 in 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 more 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 4.4) 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, entering in the role in a not honest way. The organization and role programmer, however, has all the possibility to check 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

4.4 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" can 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 4.2, 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.

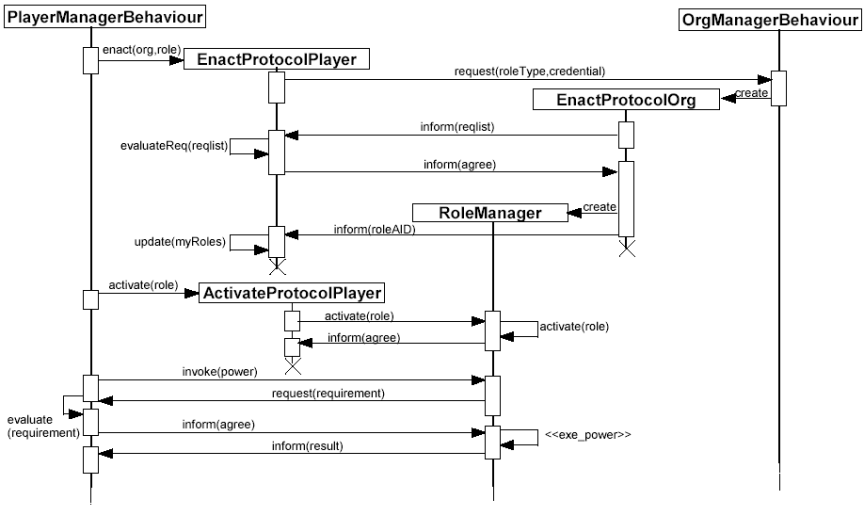

Fig. 4. The Sequence Diagram for a complex communication.

5 The CNP scenario in powerJade

In Section 3, 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 Figure 5, a little part of the interaction between the player for the manager role and its role is shown. Let's consider 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 fullfilled, 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, an `invokePower()` is executed, calling for the power with name *CNP* (the bold arc with number 1 in Figure 5). The role begins the power execution (managed by the `PowerManager`, after the `RoleManager` has passed to it the control). The first state for the power is the request for a requirement: for starting a call for proposal, the task to be delegated must be specified by the player. The `RequestRequirementState` sends a request for requirement to the `PlayerManager` (the bold arc with number 2 in Figure 5), that passes the control to the `RequirementManager`. The correct requirement is executed (the state which entering arc is labeled "task"), and the result is sent back to the `RequestRequirementState` (the bold arc with number 3). The power execution goes on, arriving to the `SEND_CFP` state, that provides the call for proposal to any bidder known inside the organization (bold arc with label 4, we assume that some agents already enacted the bidder role), going directly to add the opportune behaviour to the `PowerManager` of the `B_CNP` instances found. The bidder roles will send messages back to the manager roles, after requesting to their players the requirement to specify or not a price for the task to be delegated.

The complicated 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.

6 Related work and conclusions

On organizations and roles representations, many models have been proposed [12], applications modeling organizations or institutions [19], software engineering methods using organizational concepts like roles [25]. Several agent programming languages (among which 3APL [24]) have been developed, but few of them have been endowed with primitives for modeling organizations and roles as first class entities. Exceptions can be found in MetateM [11] (which is BDI oriented, is based on the notion of group,

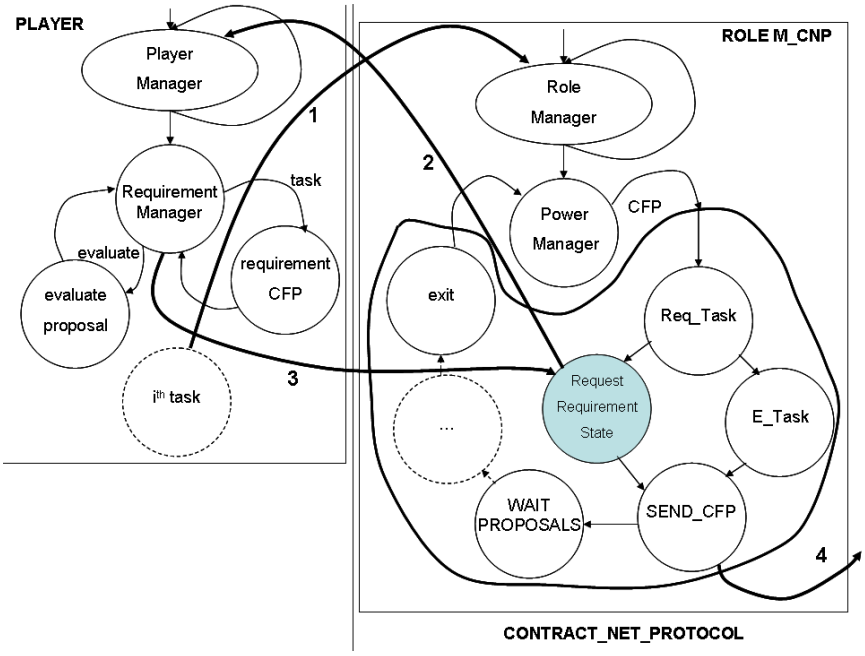

Fig. 5. Part of the solution for the CNP example. We can notice three interactions between different actors: (1) is from a player's behaviour to the active role; (2) is from a role's power to the player; (3) is from a the player to the role, communicating the requirement result; (4) is from a role's power to another role.

and it is not a general purpose language), J-MOISE+ [15] (which is more oriented to programming *how* agents play roles in organizations), and the Normative Multi-Agent Programming Language in [22] (which is more oriented to model the institutional structure composed by obligations, more than the organizational structure composed by roles). Considering frameworks for modelling organizations like SMoise+ [16] and MadKit [13], can be noticed limited possibilities to program organizations.

Regarding the analysis of organizations, in [23] 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 this perspective, works such as GAIA [25] (which is a model for designing MAS, more than a framework) and the already cited (with extensions) MOISE [14] can be found, while other models, such as ISLANDER [10], define organizations as electronic institutions, in terms of norms and rules.

With respect to organizational structures, Holonic MAS [21] 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]. Roles and groups can express quite naturally 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 [20] presents another interesting platform for the implementation of organizations, even if it does not currently have organizational structures.

[18] 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.

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, V. Genovese, R. Grenna, and L. der Torre. Roles in coordination and in agent deliberation: A merger of concepts. *PRIMA 2007*, 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. A. Colman and J. Han. Roles, players and adaptable organizations. *Applied Ontology*, 2007.
9. M. Dastani, B. van Riemsdijk, J. Hulstijn, F. Dignum, and J.-J. Meyer. Enacting and deacting roles in agent programming. In *Procs. of AOSE'04*, pages 189–204, New York, 2004.
10. 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.
11. M. Fisher, C. Ghidini, and B. Hirsch. Organising computation through dynamic grouping. In *Objects, Agents, and Features*, pages 117–136, 2003.
12. 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.
13. O. Gutknecht and J. Ferber. The madkit agent platform architecture. In *Agents Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, 2000.
14. 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.
15. J. F. Huebner. J-Moise$^+$ programming organizational agents with Moise$^+$ and Jason. In *http://moise.sourceforge.net/doc/tfg-eumas07-slides.pdf*, 2007.
16. J. F. Huebner, J. S. Sichman, and O. Boissier. S-moise+: A middleware for developing organised multi-agent systems. In 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.
17. A. Jones and M. Sergot. A formal characterisation of institutionalised power. *Journal of IGPL*, 3:427–443, 1996.
18. C. Madrigal-Mora, E. León-Soto, and K. Fischer. Implementing Organisations in JADE. In *MATES*, pages 135–146, 2008.
19. 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.
20. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a bdi-infrastructure for jade agents. *EXP*, 3(3):76–85, 9 2003.
21. M. Schillo and K. Fischer. A taxonomy of autonomy in multiagent organisation. In *Agents and Computational Autonomy*, pages 68–82, 2003.
22. 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.
23. 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.
24. 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.
25. 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.