An Organisational Platform for Holonic and Multiagent Systems

Nicolas Gaud, Stéphane Galland, Vincent Hilaire, and Abderrafiâa Koukam

Multiagent Systems Group,
System and Transport Laboratory
University of Technology of Belfort Montbéliard
90010 Belfort cedex, France
{nicolas.gaud,stephane.galland,
vincent.hilaire,abder.koukam}@utbm.fr
http://set.utbm.fr

Abstract. JANUS is a new multiagent platform that was specifically designed to deal with the implementation and deployment of holonic and multiagent systems. It is based on an organisational approach and its key focus is that it supports the implementation of the concepts of role and organisation as first-class entities. This consideration has a significant impact on agent implementation and allows an agent to easily and dynamically change its behaviour. The platform also natively manages the concept of holon to facilitate the deployment of holonic multiagent systems and thus contributes to fill the gap between conception and implementation phases in this domain. This article draws a complete description of JANUS and its main characteristics. A small example of a market-like community is also provided with the associated code review to illustrate the impact of a full organisational approach in terms of code modularity and reusability.

Keywords: Agent Oriented Software Engineering, Holonic Modelling, Multiagent systems implementation and deployment, Holonic multiagent systems.

1 Introduction

Dastani and Gomez-Sanz [1] consider that agent-oriented applications will only be taken up by industry if the gap between multiagent systems specification and design on the one hand and multiagent systems implementation on the other hand is bridged. Our approach consists in filling the gap between design and implementation metamodels, and thus facilitate the transformation between them. Filling that gap requires a platform, whose metamodel offers an implementation as straight as possible of the concepts used for the design of the solution.

This article deals with the last steps of the software development process, dedicated to the implementation and deployment of Multi-Agent Systems and Holonic Multi-Agent Systems applications (MAS and HMAS from now on). It introduces the JANUS platform, that is specifically designed to deal with MAS and HMAS. The metamodel of this platform corresponds to a fragment of the CRIO metamodel [2,3,4] that aims at providing a full set of abstractions to model MAS and HMAS under an organisational perspective. CRIO adopts the system development approach defined in the Model Driven Architecture (MDA) [5] and the elements of this metamodel are organised in three different domains. (i) The Problem domain (CIM[1]) deals with the user's problem in terms of requirements, organisations, roles and ontologies. (ii) The Agency domain (PIM[2]) addresses the holonic solution to the problem described in the previous domain. (iii) Finally, the Solution Domain (PSM[3]) describes the structure of the code solution in the chosen implementation platform. This last domain thus corresponds to the Platform Specific Model, and it is dependant of the JANUS platform presented in this paper.

Holonic multiagent systems are based on self-similar and recursive entities, called Holons. In Multiagent systems, the vision of holons is closer to the one that MAS researchers have of *Recursive* or *Composed* agents. An holon is thus a self-similar structure composed of holons as sub-structures and the hierarchical structure composed of holons is called an *holarchy*. An holon can be seen, depending on the level of observation, either as an autonomous "atomic" entity or as an organisation of holons (this is often called the *Janus effect*). Using a holonic perspective, the designer can model a system with entities of different granularities. He can recursively model sub-components of a bigger system until he achieves a stage where the requested tasks are manageable by atomic easy-to-implement entities. Implementing holonic models requires a platform able of managing the concept of nested hierarchy, but most MAS platforms consider agents as atomic entities. It is therefore difficult to implement the concept of holon, and provide an operational representation of models combining several levels of abstraction, using such platforms.

In the CRIO metamodel, organisations are considered as independent modelling units and blueprints easily reusable in various applications. The key point of this modular definition of organisations is based on the concepts of role and capacity [6]. In order to easily implement models based on CRIO requires a platform whose metamodel considers the role as a first-class entity, independent of the agent. On this aspects, the Madkit[4] platform [7] and its extension MOCA [8] have come to our attention, as they both manage the concept of role. However, Madkit does not consider the role as a first-class entity. Indeed, the behaviour associated with the role is directly implemented in the agent who plays it. Roles are strongly linked to agents architecture. This approach harms organisations reusability and modularity. MOCA considers roles as first-class entities, but sets strict constraints on their implementation. For example, an agent may not play several times the same role. These two platforms do not provide concepts to easily implement MAS designed with an organisational approach. Moreover, neither

[1] Computation Independent Model, first level of model in MDA.
[2] Platform Independent Model, second level of model in MDA.
[3] Platform Specific Model, third level of model in MDA.
[4] http://www.madkit.org/

of them manage the concept of holon. JANUS was specifically designed to deal with the holonic and organisational aspects. Its goal is to provide a full set of facilities for launching, displaying, developing and monitoring holons, roles and organisations. The heart of the implementation of its organisational model was inspired by the approaches adopted in the CRIO metamodel, Madkit and MOCA platforms. And it also integrates all the concepts necessary for an easy implementation of holonic multiagent systems.

This paper is organised as follows. Section 2 describes the metamodel of the JANUS platform, then its general architecture is presented in section 3. Section 4 details the key characteristics of this platform, and especially the implementation of the communication between roles modelled as first-class entities. This section also outlines the key points behind the implementation of the concept of holon. The implementation of a market-like community is described in section 5, to emphasize the advantage of considering a role as a first-class entity. Finally section 6 provides some conclusion statements.

2 Metamodel of the Janus Platform

This section is dedicated to the presentation of the metamodel of the JANUS platform. Its main concepts are described in the UML diagram, presented in figure 1.

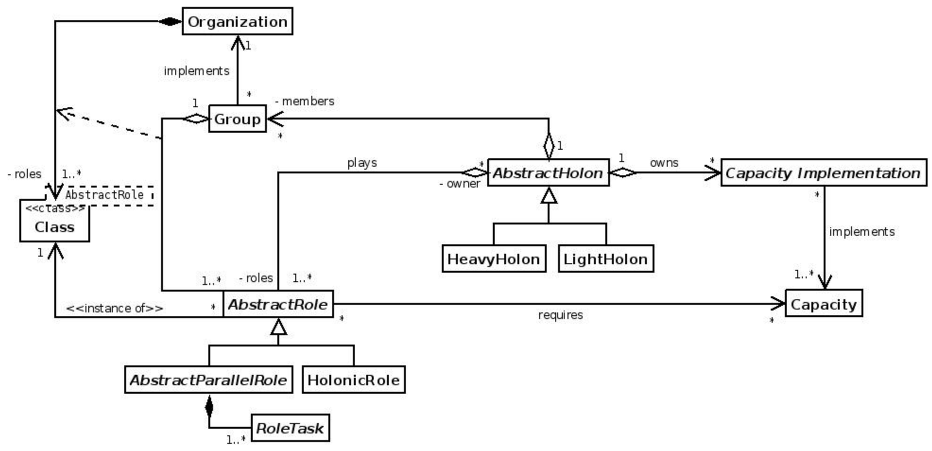

Fig. 1. UML diagram of a part of the metamodel of the JANUS platform

JANUS was designed to facilitate the transition between design and implementation phase. It thus provides a direct implementation of the five key concepts used in the design phase : organisation, group, role, holon and capacity.

The organisation is implemented as a first-class entity (a class in the object-oriented sense), which includes a set of roles classes. An organisation can be instantiated in the form of groups. Each group contains a set of instances of different classes of roles associated with the organisation which it implements. The number of authorized instances for each role is specified in the organisation. A role is local to a group, and provides holons playing the role the means to communicate with other group members. One of the most interesting aspects of JANUS covers the implementation of roles as first class entity. A role is seen as a full-fledged class, and the roles are implemented independently of the entities that play them. Such an implementation facilitates the reuse of organisations in other solutions, but also allows a wide dynamic for roles.

An agent is represented by an atomic holon (a non-composed one). Janus defines two main types of holon: *HeavyHolon* and *LightHolon*. A *HeavyHolon* has its own execution resource (one thread per holon), and can therefore operate independently. The *LightHolon* is associated with synchronous execution mechanisms and it is very useful to develop multiagent-based simulations (an approach similar to the synchronous engine of Madkit[5]). This architecture fits into a synchronization model, which defines the various execution policies of the system and holons in charge of their implementation. A holon can play simultaneously multiple roles in several groups. It can dynamically access to new roles and leave ones that are no longer in use. When an holon accesses a role, he obtains an instance of the class of this role that it stores in its roles container. Respectively, when it leaves a role, the corresponding instance is removed. The size of a holon, in terms of code, is always minimal because it only contains the instances of the roles he plays at a given moment. To access or leave a role, a holon must meet the access and liberation conditions of the role and those of the corresponding group. This mechanism provides many advantages in terms of security, since holons have access to the behaviour of a role (and thus get the corresponding executable code) only if it fulfills these conditions. Each instance of an organisation (or groups) can have specific access and liberation rights. The access and leave conditions of roles are in contrast defined at the organisation level, and cannot be changed at runtime. To model composition relationships between holons, an organisational approach is also adopted. A composed holon is called super-holon. A super-holon do not directly maintain references on its members, it is composed of a set of groups where its members play various roles and contribute to the fulfillment of the goals assigned to their super-holon(s). The approach used to model a super-holon is detailed in section 4.2.

The notion of capacity enables the representation of holon competences. Each holon has, since its creation, a set of basic skills, including the ability to play roles (and therefore communicate), to obtain information on existing organisations and groups within the platform, create other holons, and obtain new capacities. The capacity concept is an interface between the holon and the roles it plays. The role requires some capacities to define its behaviour, which can then be invoked in one of the tasks that make up the behaviour of the role. The set of capacities required by a role are specified in the role access conditions. A capacity can be implemented in various ways, and each of these implementation

[5] See http://www.madkit.net/site/madkit/doc/devguide/synchronous.html

is modelled by the notion of *Capacity Implementation*. This concept is the operational representation of the concept of service defined in the Agency domain. Currently JANUS does not implement a matchmaking procedure for capacities. This aspect is under development, and our approach is inspired by the works of [9,10].

In addition to these concepts, Janus provides a range of tools to facilitate the work of the developer. The various features offered by Janus will be described in the next section.

3 Kernel and General Architecture of Janus

The architecture of the JANUS platform is shown in Figure 2. Janus is developed in Java 1.5. The heart of the platform is embodied by its kernel, which provides the implementation of the organisational model and of the concept of holon. The kernel was then extended to integrate simulation module and holons in charge of the operation of the platform and its integration with the applications.

The various features provided by the JANUS kernel are described below:

– The **Organizational Management System** manages organisations and their instantiations in the form of groups. It also provides mechanisms for the

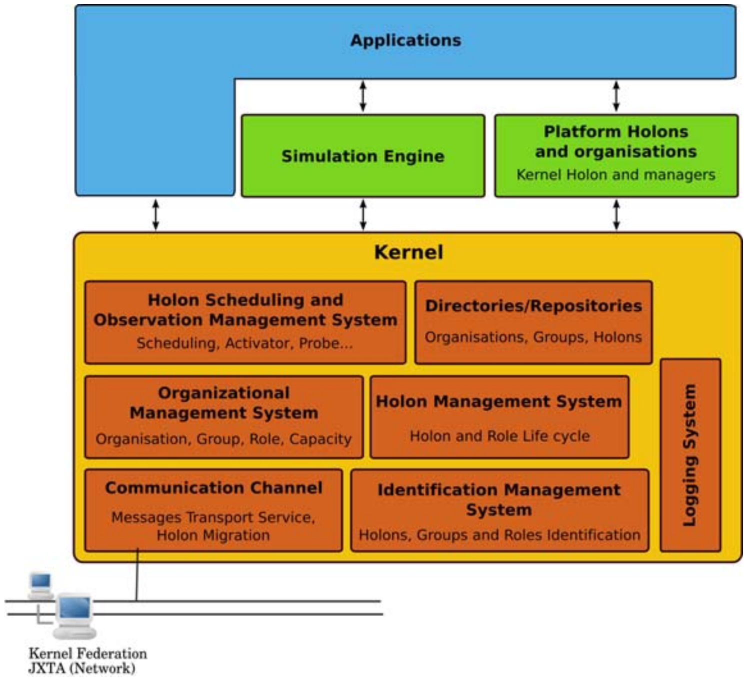
Applications
Simulation Engine
Platform Holons and organisations
Kernel Holon and managers
Kernel
Holon Scheduling and Observation Management System
Scheduling, Activator, Probe...
Directories/Repositories
Organisations, Groups, Holons
Organizational Management System
Organisation, Group, Role, Capacity
Holon Management System
Holon and Role Life cycle
Logging System
Communication Channel
Messages Transport Service, Holon Migration
Identification Management System
Holons, Groups and Roles Identification
Kernel Federation
JXTA (Network)

Fig. 2. General Architecture of the JANUS platform

dynamic acquisition, instantiation and liberation of roles, as well as mechanisms for the dynamic acquisition and execution of capacities. Organizational aspects are managed at the lowest level in the platform so that everything holon, including the platform ones, have access to this functionality. This module is linked to the kernel holon in charge of maintaining this information with other remote kernels (through the Communication Channel).

- **Holon Management System** : The kernel also provides all the tools necessary for the holon life cycle management: identify, launch, stop, etc. Each type of holon (*HeavyHolon* and *LightHolon*) natively provides a set of execution policies for its roles, as well as various policies for the messages management.
- The **Communication Channel** control the exchange of messages within the platform and also with remote kernels (inside the kernel federation). Considering a role as a first-class entity affects this aspect of the platform. Communication Management within JANUS will be detailed in the section 4.1.
- The **Identification Management System** provides all the necessary mechanisms for assigning a unique address (GUID) to all elements of the model which need it. Thus, the holons, groups, and roles have a unique address within a kernel federation.
- **Directories/Repositories** maintains a directory all the groups (GroupAddress $\mapsto$ Group), organisations (Class<*? extends Organization*> $\mapsto$ Organization) and holons (HolonAddress $\mapsto$ Holon) defined in the kernel. A capacity directory associating existing capacities and their available implementations is under development.
- **Holon Scheduling and Observation Management System**: JANUS provides two basic policies for holons scheduling: a concurrent execution model and a synchronous engine inspired by the Madkit one. This module also provides instrumentation based on probes allowing a role to observe another role. Unlike Madkit which manages the observation rights at the agent-level (agent who implements or not the interface *ReferencableAgent*), JANUS manages it at the role-level. It allows a more refined management of the observation rights. A holon may permit observation of one of its roles and prohibit it for the others.
- **Logging System**: All applications based on JANUS have access to a logging system integrated to the platform, which facilitates the debugging process. Logs may be directly displayed or stored in a file. This system may be changed and integrated into existing systems. The current implementation of this feature is based on log4j [6] provided by the Apache Software Foundation.

As Madkit, JANUS exploits its own model in the design of the platform, and all services are managed by holons. The kernel is thus linked to a *KernelHolon*, which contains all the local organisations responsible for managing the platform, and represents its kernel in the federation distributed over the network. A kernel federation is an organisation in charge of managing the various exchanges

[6] More detail at the following address: http://logging.apache.org/log4j/docs/index.html

between kernels and spreading information about the organisational model such as the creation of a new organisation or a new group, migration of a holon, etc.

The architecture of JANUS respects the overall FIPA reference architecture[7]. Only ACL[8] related features are not yet fully implemented. To compensate for this gap, it is foreseen in short term to integrate the relevant part of JADE to ensure a full compatibility of JANUS with the FIPA standard.

4 Main Characteristics of Janus

This section is devoted to the presentation of the main characteristics of the JANUS platform. Issues related to the implementation of the concept of holon and communication mechanisms between roles are specifically focused.

4.1 Communication

To communicate, holons must belong to a common group and play a role in this group. If a group is distributed among several kernels, an instance of this group exists in each kernel, and all the instances have the same name.

Communication in JANUS is based on the roles. Messages are delivered to agents according to their roles. This mode of interaction allows the implementation of communications between an emitter and several receivers (one-to-many communication [11]). The address of the receiver agents is dynamically discovered according to the roles they play. When several agents play the same role within the same group, a mode of communication based on role and agents identifier may also be used to distinguish which role player will receive the message. Although the explicit agent identifier is known, messages always require a recipient role to be handled. This kind of interaction allows the implementation of communication between two identified agents (one-to-one communication).

Each holon owns a personal mailbox for sending and receiving messages. A holon may simultaneously play multiple roles and dynamically acquire new ones. The role is the way for a holon to interact in the particular interaction context represented by the group. Roles constitute the basis of all interactions. Each role thus has its own mailbox (cf. figure 3) and the mailbox of a holon is just the arrangement of all the mailboxes of its roles. A holon can therefore receive messages only through its roles.

4.2 Implementation of the Concept of Holon

In addition to its role model, one of the main contributions of JANUS is embodied in the native management of the concept of holon. Two main aspects have to be distinguished to implement a holon with JANUS : (i) The first aspect deals with the implementation of a non-composed holons and the conception of a

[7] FIPA Abstract Architecture: http://fipa.org/specs/fipa00001/SC00001L.html

[8] Agent-Communication-Language.

general holon architecture able to integrate the capacities owned by the role and the roles he's currently playing. This architecture have to provide means to manage roles and capacities life cycle and dynamically acquire new ones. (ii) The second aspect concerns the manner of implementing a composed holon to ensure communication between a super-holon and its members, located at two different levels of abstraction. Both aspects will be detailed in the following subsections.

Atomic Holon Architecture. An atomic holon is primarily a roles and capacities container. The roles container provides the necessary means for the roles of a holon to interact in the internal interaction context of a holon. The local mechanism of interaction inside a holon is called influence and it is implemented using an event-based communication. Each role can register itself to inform its holon that it wishes to receive all the influences of a given type. Figure 3 describes the architecture of an atomic holon in JANUS. Section 5 will detail a concrete example of the influence mechanism.

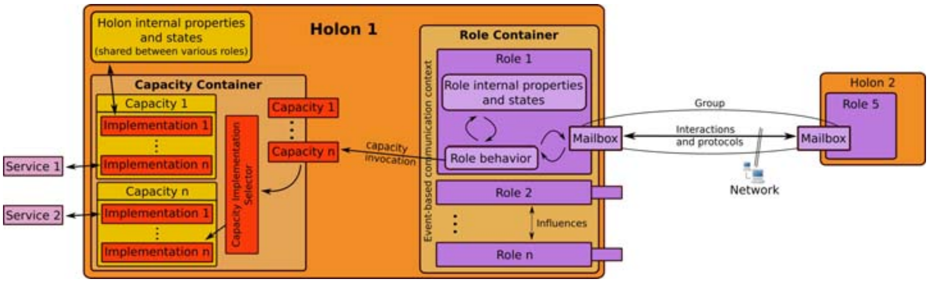

Fig. 3. Architecture of an atomic holon in JANUS

The capacity container stores all the capacities owned by the holon and all the available implementations for each of them. It also ensures their execution when a role invokes a capacity. Two main types of capacities execution are available in JANUS: synchronous or asynchronous. In the first mode, the capacity is directly executed when it is invoked by the role. The execution of the capacity temporarily interrupts the execution of the current role that is waiting for capacity termination. In the second mode, the execution of the role is not interrupted, the capacity is executed after the current role, after all the roles of holon or in parallel. The corresponding role is then informed of the outcome. This last mode is particularly interesting when the capacity is realized by a service provided by the members of a super-holon. It avoids blocking the execution of the super-holon while its members perform a given task, the holon can thus continue the execution of its other roles.

Composed Holon Implementation. Two overlapping aspects have to be distinguished in composed holons: (i) the first is directly related to the holonic nature of the entity (a holon, called super-holon, is composed of other holons, called sub-holons or members) and deals with the government and the administration of a super-holon. This aspect is common to every holon and thus called the *holonic* aspect. (ii) The second aspect is related to the problem to solve and the work to be done. It depends on the application or application domain. It is therefore called the *production* aspect. A composed holon (super-holon) thus contains at least a single instance of a *holonic organisation* to precise how members organise and manage the super-holon and a set (at least one) of *production organisations* describing how members interact and coordinate their actions to fulfill the super-holon tasks and objectives.

The *holonic organisation* is the basis of the holon government and it represents a *moderated group* (see [12]) in terms of roles (called *holonic roles*) and their interactions. In a moderated group, a subset of the members will represent all the sub-holons in the outside world. This management structure was adopted due the wide range of configurations it allows. Four *holonic roles* are defined to describe the status of a member inside a super-holon: (i) **Head**, decision maker: it represents a privileged status conferring a certain level of authority. (ii) **Representative**, interface of the holon: it is an externally visible part of a super-holon, it is an interface between the outside world (same level or upper level) and the other holon members. It may represent other members in taking decisions or accomplishing tasks (i.e. recruiting members, translating information, etc). The *Representative* role can be played by more than one member at the same time. (iii) **Part**: Classical members. Normally in charge of doing tasks affected by head, a *Part* can also have an administrative duty, and it may be employed in the decision making process. It depends on the configuration chosen for modelling the super-holon. The *Part* role represents members belonging to only one super-holon. (iv) **Multi-Part**: extension of *Part*. This role is played by sub-holons belonging to more than one super-holon.

To manage a super-holon, members have to be able to communicate with their super-holon, located at a higher level of abstraction. One of the main problems in implementing this inter-level communication comes from the fact that in JANUS and considering the role as first-class entity, two holons can communicate only if they belong to a common group. This rule implies that a super-holon must share at least one group with its members to enable the transfer of information between two adjacent levels of abstraction. Several alternatives may be considered to implement the notion of super-holon. Our study is limited to the following three alternatives:

1. The first alternative is to appoint one member to represent the rest of the community to the upper level. This approach is described in figure 4. The holon H_3 plays the *representative* role and represents the community at level $n+1$.

   Members playing the *representative* role appear as the most suitable to perform this role of representation. But in a super-holon, several members may play this role, which means that one of the representatives in particular

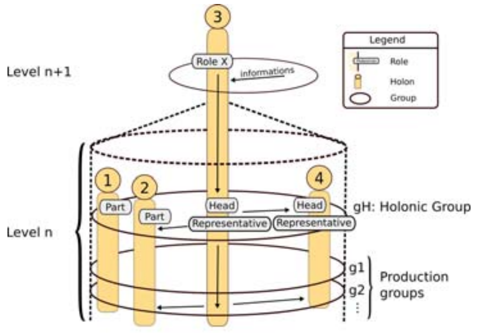

Fig. 4. One way to implement the structure of a composed holon

is identified (or elected). The community can be represented at the upper level by only one member. While this approach appears to be the simplest, it introduces a hierarchical distinction between representative members who may be elected, and therefore it is not completely consistent with the definition provided for the *representative* role in the CRIO metamodel. Moreover, this approach raises two other problems. First, from an abstract point of view, a super-holon is a separate entity distinct from its members. The fact that one of the members represent the community at the upper level means that it may play roles defined by organisations at a higher level of abstraction. Thus, the holon H_3 plays both the role X at the level $n+1$, and various roles in the production groups at level n. To be consistent, the super-holon must be clearly distinguished from its members. Playing roles at the different levels of abstraction could create interference problems or conflicts between these roles.

In addition, the *representative* role is not exclusive of other holonic roles. Thus, it can be played by a holon who also plays the *Multipart* role. Such an holon would then be shared between two holons of level n, and may represent them at the level $n+1$. The problems of confusion and conflict between levels of abstraction would be very important. This approach is the simplest, but it is not optimal.

2. To clearly distinguish the different levels of abstraction and to avoid possible interference between roles defined at different levels, another approach is possible. The latter is depicted in figure 5(a). In this approach, the super-holon is clearly distinguished from its members. A new entity is created at level $n+1$. A new group $g0$ is introduced at the level n to make the interface between the various members' representatives and the super-holon. Indeed, a super-holon must be clearly separated from its members and can only communicate with their representatives. A new role is so introduced: the role *Super*, to enable the super-holon to communicate with the representatives of its members in the group $g0$. This approach is the most consistent with

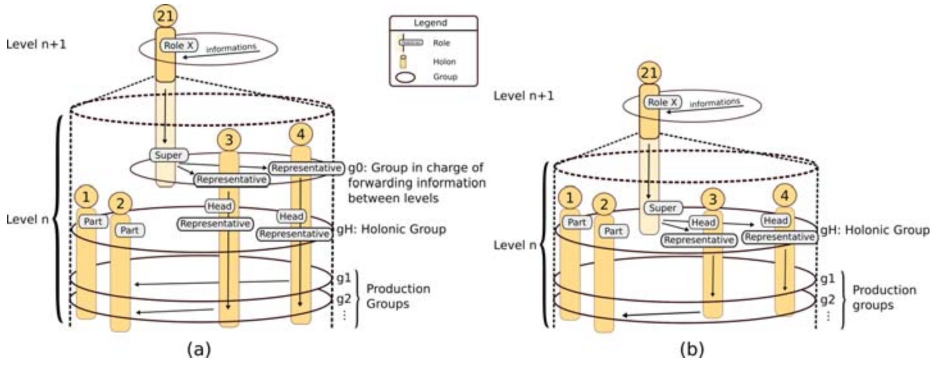

Fig. 5. Theoretical model (a) and concrete structure (b) of a composed holon in JANUS

the CRIO metamodel, but requires the creation of an additional group in all composed holons, and the implantation of a holon becomes more complex.

3. In order to keep the benefits of the previous alternative, while avoiding the additional costs due to the creation of a new group, a third approach based on a compromise between the two previous ones, has been adopted. The implementation of the holonic organisation adopted in Janus is based on this alternative and it is presented in figure 5(b). In this approach, the group $g0$, previously described, is merged with the holonic group. This group is indeed present in all composed holons. A new role is introduced in the holonic organisation, the role *Super*, to represent the upper level and thus allow the transfer of information between the super-holon and representatives of its members. This approach offers the best compromise between compatibility with the CRIO metamodel and implementation performances.

5 A Market Organisation Example

To clarify the implementation of roles as first-class entity, role dynamics and role communication, a short example of a market-like community designed using the CRIO metamodel and implemented with the JANUS platform is presented. It is a classical case study, already used to illustrate the AALAADIN metamodel and the Madkit platform [13].

All organisations, groups, roles and holons required to implement this example are shown in Figure 6. This example is applied to the domestic travel market. A customer, modelled by the *Client* role, who wishes to obtain the best available travel offer, either in terms of price or in terms of travel time, makes its proposal and sends it to the *CBroker*. This latter will forward the information to the *PBroker* role, who broadcasts it to the various available *Provider*s. Depending on the criterion chosen by the customer (time or price), the *PBroker* determines the best proposal and inform the *Client*. *Client* and the best *Provider* then create an instance of the contracting organisation to finalize the order and make

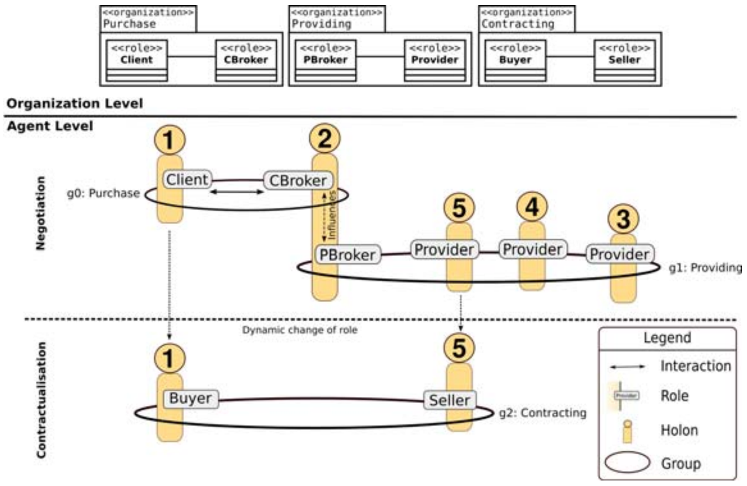

Fig. 6. The organisations and groups of a market-like community in Janus

the payment. Specifically, the proposed example may implemented using three kinds of holons : *Client* (Holon 1), *Provider* (Holons 3, 4 and 5), *Broker* (Holon 2), three organisations : *Purchase*, *Providing*, *Contracting*, and six roles. Each organisation is stored in its own java package containing its java class and those of its roles. The source code of the *Purchase* organisation is provided below:

```java
public class PurchaseOrganization extends Organization {
    private static Organization instance=new PurchaseOrganization();
    //Each organisation is a singleton inheriting from Organization class.
    protected PurchaseOrganization(){
        super();
        //Add classes of roles that are defined on this organisation
        addRole(Client.class);
        addRole(CBroker.class);
    }
    public static Organization getInstance() { return instance; }
}
```

Each kind of holon is also defined in its own class. The source code of the Holon 2 playing the *CBroker* and *PBroker* roles, is described below:

```java
public class BrokerHolon extends HeavyHolon {
    @Override
    public void live() {
        //static holon capacities initialization : adding those required by the PBroker role
        addCapacity(FindLowestCostProposalCapacity.class, new
             FindLowestCostProposalCapacityImpl(this));
        addCapacity(FindShortestTimeProposalCapacity.class, new
             FindShortestTimeProposalCapacityImpl(this));

        GroupAddress clientGA = getOrCreateGroup(PurchaseOrganization.getInstance());
        //Request the CBroker role
        if(requestRole(CBroker.class,clientGA)){ println("role CBroker assigned"); }

        GroupAddress providerGA = getOrCreateGroup(ProvidingOrganization.getInstance());
        //Request the PBroker role
        if(requestRole(PBroker.class,providerGA)){ println("role PBroker assigned"); }

        //Simplest role scheduling
        while(true) { for (Role role : getRoles()) role.behavior(); }
    }
}
```

In the remainder of this section, the implementation of the *PBroker* role is detailed. In the proposed implementation, the *PBroker* role requires two capacities to defined its behaviour : *FindShortestTimeProposalCapacity* and *FindLowestCostProposalCapacity*. These capacities are used to determine the best proposal among those offered by the various providers. This determination is done according the criterion chosen by the customer. If the choice criterion is the overall travel time, the *FindShortestTimeProposalCapacity* capacity will be used, if the criterion is the cost, it will the *FindLowestCostProposalCapacity* capacity.

In addition, the *PBroker* role is dependent on the *CBroker* role. This latter enable the transfer of information between the *Purchase* and *Providing* organisations. To access to the *PBroker* role, a holon have to prior possess the *CBroker* role. These constraints of capacity and dependencies between roles are implemented using special types of access conditions for obtaining a role

```
public class PBroker extends AbstractRole {
    // ...Attributes of the role ...
    private int current = 1; // the current state
    public PBroker() {
        super();
        //Definition of the dependencies of role
        List<Class<? extends Role>> requiredRoles = new LinkedList<Class<? extends Role>>();
        requiredRoles.add(CBroker.class);
        SatisfyRoleDependenciesCondition roleCondi = new SatisfyRoleDependenciesCondition(
             requiredRoles);
        //Definition of the capacities required by the role
        List<Class<? extends Capacity>> requiredCapacities = new LinkedList<Class<? extends
             Capacity>>();
        requiredCapacities.add(FindShortestTimeProposalCapacity.class);
        requiredCapacities.add(FindLowestCostProposalCapacity.class);
        HasAllRequiredCapacitiesCondition capCondi = new HasAllRequiredCapacitiesCondition(
             requiredCapacities);
        //Addition of role access conditions
        addObtainCondition(capCondi);
        addObtainCondition(roleCondi);
    }
    //The core of the behavior of the role.
    public void behavior() {
        current = Run();
    }
    //This methods correspond to the translation in java code of the statechart describing
    //the behavior of the role. This statechart was defined during the design phase
    private int Run() {
        switch (current) {
    //The role register itself to signify that it want to receive a particular type of
    //influence, in this case those from the role CBroker
        case 1 : registerForRoleInfluence(TravelRequestInfluence.class);
            return 2;
    //Waiting for the arrival of the influence from the CBroker role
        case 2 : influence = (TravelRequestInfluence)getNextInfluence();
            if (influence != null) return 3;
            return 2;
        // ...
        //Get the next message in the mailbox associated to the role
        case 5 :m = getNextMessage();
        //...
            List input = new ArrayList();
            input.add(proposalList);

            if (requestType == TravelRequestType.LowestCost) {
                selected = FindLowestCostProposalCapacity.class;
        //Synchronous Execution of the FindLowestCostProposalCapacity capacity
                callAndExecuteCapacity(selected, id, input);
            } else if (requestType == TravelRequestType.ShostestTime) {
                selected = FindShortestTimeProposalCapacity.class;
        //Synchronous Execution of the FindShortestTimeProposalCapacity capacity
                callAndExecuteCapacity(selected, id, input);
            } else println("Error in Request Type");
            return 7;
        //Awaiting the result of the execution of the capacity
        case 7 :if (isResultAvailable(selected, id)) {
            output = getResult(selected, id);
            return 8;
            }
        return 7;
        //Selecting the best offer and inform the selected provider
        case 8 :Best = (Proposal)output.get(0);
            BestProvider = proposals.get(Best);
            sendMessage(BestProvider, Provider.class, new StringMessage("Proposal Accepted"));
            return 9;

        case 9 :m = getNextMessage();
            if ((m != null) && (m instanceof TransfertMessage)) {
                return 10;
            }
            return 9;
        //The role emits an influence and informs all the roles that are listening to
        //this kind of influence (i.e. CBroker)
        case 10 : influenceHolon(new TransfertInfluence(((TransfertMessage)m).getGroupAddress
              ()));
            println("PBroker Finish");
            return 2;//Return in the waiting state
        default : return 1;
        }
    }
}
```

(*ObtainConditions*). Consider now the source code of the *PBroker* role in order to clarify these different aspects.

Figure 7 provides a part of the UML sequence diagram describing mechanisms associated to the creation of a group and the access to a role in this group. The *Broker* Holon (Holon 2) requests the address of a group implementing the *Purchase* organisation. In this example, no instance already exists, a new one is thus created, and its address is returned to the holon. Then the holon requests for access to this group and to the *PBroker* role in this group. The group and role access conditions are verified. In this case, the holon fulfills all required

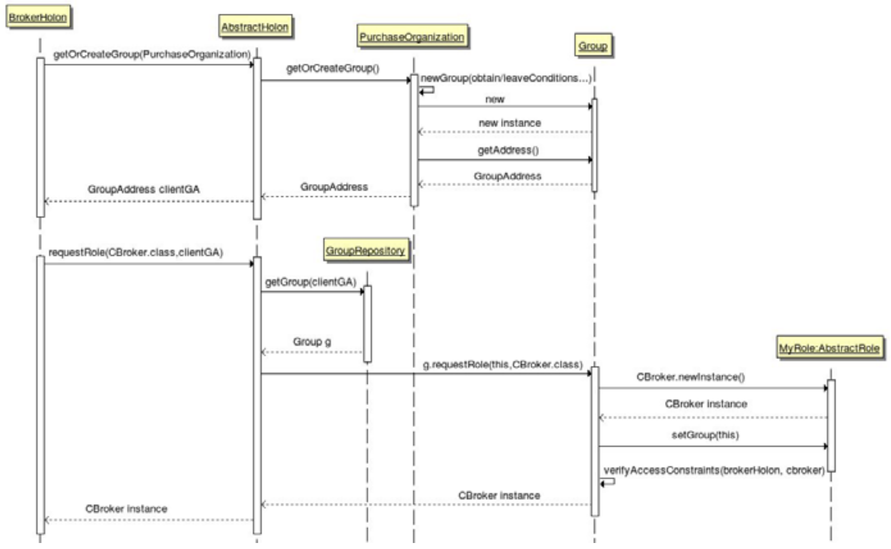

Fig. 7. UML sequence diagram of an access request to a role in a given group

conditions, it thus obtains an instance of the *PBroker* role that is added to its role container.

6 Conclusion

In this article, we have introduced the Janus platform dedicated to the implementation and deployment of MAS and HMAS. In JANUS, the notion of organisation is considered as a true Java module in its own right. The native management of the concept of capacity allows to implement a role, without making any assumption on the architecture of holons playing it, and thus promotes the reuse of organisations in various applications. However, this approach should be relativized, because it requires the definition of a significant number of classes, even for small applications (one class for each organisation, role, holon or agent architecture). So Janus aims primarily at developing large applications where modularity is essential. This aspect confirms the need to associate JANUS with a CASE tool to automatically generate significant portions of code, thus simplifying the intervention of a programmer. This CASE tool is currently under development in our Lab. In addition JANUS, providing a direct implementation of the four concepts at the base of CRIO (capacity, role, organisation and holon), contributes to reduce the gap between design and implementation phases. This platform is part of a larger effort aiming at providing a complete software tools suite for the development of complex applications in an industrial context.

References

1. Dastani, M., Gomez-Sanz, J.J.: Programming multi-agent systems (promas), a report of the technical forum meeting (April 2005), http://people.cs.uu.nl/mehdi/tfg/ljubljanafiles/report.pdf
2. Rodriguez, S., Hilaire, V., Koukam, A.: Fomal specification of holonic multi-agent system framework. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3516, pp. 719–726. Springer, Heidelberg (2005)
3. Cossentino, M., Gaud, N., Hilaire, V., Galland, S., Koukam, A.: A Holonic Metamodel for Agent-Oriented Analysis and Design. In: Mařík, V., Vyatkin, V., Colombo, A.W. (eds.) HoloMAS 2007. LNCS, vol. 4659, pp. 237–246. Springer, Heidelberg (2007)
4. Cossentino, M., Gaud, N., Hilaire, V., Galland, S., Koukam, A.: A Metamodel and Implementation platform for Holonic Multi-Agent Systems. In: The fifth European Workshop on Multi-Agent Systems (EUMAS 2007), Hammamet, Tunisia (December 2007)
5. Object Management Group (OMG): MDA Guide, v1.0.1, OMG/2003-06-01 (June 2003)
6. Rodriguez, S., Gaud, N., Hilaire, V., Galland, S., Koukam, A.: An analysis and design concept for self-organization in holonic multi-agent systems. In: Brueckner, S.A., Hassas, S., Jelasity, M., Yamins, D. (eds.) ESOA 2006. LNCS, vol. 4335, pp. 15–27. Springer, Heidelberg (2007)
7. Gutknecht, O., Ferber, J.: Madkit: a generic multi-agent platform. In: The 4th International Conference on Autonomous Agents (AGENTS 2000), Barcelona, Spain, pp. 78–79. ACM Press, New York (2000)
8. Amiguet, M., Müller, J.P., Baez-Barranco, J.A., Nagy, A.: The MOCA Platform, Simulating the Dynamics of Social Networks. In: Sichman, J.S., Bousquet, F., Davidsson, P. (eds.) MABS 2002. LNCS, vol. 2581, pp. 70–88. Springer, Heidelberg (2003)
9. Sycara, K., Klusch, M., Widoff, S., Lu, J.: Dynamic service matchmaking among agents in open information environments. SIGMOD Record (ACM Special Interests Group on Management of Data) 28(1), 47–53 (1999)
10. Sycara, K., Lu, J., Klusch, M., Widoff, S.: Matchmaking among heterogeneous agents on the internet. In: Proceedings of the 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace (March 1999)
11. Gouaich, A., Michel, F., Guiraud, Y.: MIC*: a deployment environment for autonomous agents. In: Weyns, D., Van Dyke Parunak, H., Michel, F. (eds.) E4MAS 2004. LNCS (LNAI), vol. 3374, pp. 109–126. Springer, Heidelberg (2005)
12. Gerber, C., Siekmann, J.H., Vierke, G.: Holonic Multi-Agent Systems. Technical Report DFKI-RR-99-03, Deutsches Forschungszentrum für Künztliche Inteligenz - GmbH, Postfach 20 80, 67608 Kaiserslautern, FRG (May 1999)
13. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multi-agent systems. In: Demazeau, Y., Durfee, E., Jennings, N.R. (eds.) Third International Conference on Multi-Agent Systems (ICMAS), Paris, France, july 1998, pp. 128–135 (1998)