Implementing Role-based Interactions for Internet Agents

Giacomo Cabri, Letizia Leonardi
Dipartimento di Ingegneria dell’Informazione
Università di Modena e Reggio Emilia
Via Vignolese, 905 – 41100 Modena – ITALY
Phone: +39-059-2056190 – Fax: +39-059-2056126
{giacomo.cabri, letizia.leonardi}@unimo.it

Franco Zambonelli
Dipartimento di Scienze e Metodi dell’Ingegneria
Università di Modena e Reggio Emilia
Via Allegri 13 – 42100 Reggio Emilia – ITALY
franco.zambonelli@unimo.it

Abstract

Large-scale distributed environments, such as the Internet, achieve advantages in exploiting software agents for applications, thanks to their autonomy in carrying out tasks. In such a scenario, interactions among agents are an issue that must be faced in an appropriate way. In the BRAIN project, interactions among agents are fruitfully modelled on the basis of roles, which define a set of capabilities and an expected behaviour. This approach achieves several advantages. First, it is an agent-oriented approach since it respects all features of agency. Then, it enables separation of concerns between the algorithmic issues and the interaction issues. Finally, roles promote the reuse of solutions and experiences in different applications. In this paper we propose an interaction infrastructure, called Rolesystem, which relies on the above mentioned role-based interaction model. This system allows agents to assume roles and interact accordingly. An application example and the comparison with other approaches show the effectiveness of our approach.

Keyword: Agents, Roles, Internet applications, Interactions

Technical areas: Internet Agents and Collaboration Technology

1. Introduction

Interactions among agents are an important issue to be taken into consideration in the development of agent-based Internet applications. Since the rising of distributed systems, the interaction among entities has been an issue to be faced carefully; different approaches have been proposed, among which the most popular is the message-passing [Cha90, Wal94]; it enables the exchange of messages between two entities, and it is simple and suits a wide range of application requirements. Another interaction approach that is worth being mentioned is Linda [AhuCG86], which is based on an uncoupled coordination model and relies on shared dataspaces.

With the advent of agents, the problem of dealing with interactions has been more focused. In fact, one of the main features of agents is sociality, i.e., the capability of interacting each other. Besides sociality, we recall that the other main features of agents are proactiveness (i.e., the capability of carry out tasks) and reactivity (i.e., the capability of reacting to the environment changes). The sociality feature is exploited in Multi Agent Systems (MAS), where the main task is divided into smaller tasks, each one delegated to a single agent; agents belonging to the same application have to interact in order to carry out the task [FalHM99]. Moreover, the diffusion of open systems, such as the Internet, has led to a scenario in which not only agents of the same application interact in a cooperative way, but also agents of different applications may interact in a competitive way, for example to achieve resources. The feature of mobility, which allows agents to change their execution environment, adds great flexibility at both conceptual and implementation levels, but also introduces peculiar issues in interactions, which must be taken into account [DomLD97].

Pag. 1
There have been different proposals in the area of agent interaction and coordination [CabLZ00a]. They have concerned message passing adapted to agents, “meeting point” abstractions [Whi97], event-channels [Bau98], and tuple spaces [CabLZ00c]. Another interesting approach is the one adopted in Agentis [InvKL98], which models interactions in terms of services and tasks; it clearly points out the need of tailoring interactions on agent features. A related issue concerns the notation that can be exploited to represent agent interaction: it is likely that a notation based on the UML language will have a more widespread success than other proposals [OdePP01, Lin01].

From the analysis of the above-described proposals, we argue that these approaches to agent interactions suffer from being adaptations of older approaches traditionally applied in the distributed system area and do not take into account the new scenario. In particular, traditional approaches often consider agents as bare objects; this has implied that the different features of agents have been managed in different ways, leading to fragmented approaches.

Our aim is modeling agent interactions on the basis of the following requirements:

- **Agent oriented features.** To overcome the limitations due to the fact that proposals are adaptations of older models, the most important requirement is that interactions are modeled following an agent-oriented approach, i.e., all the peculiar features of agents must be taken into account.
- **Separation of concerns.** We argue that an important aspect in the development of agent applications, and in particular of MAS, is the separation of concerns between algorithmic issues and interaction issues. This helps in developing applications because allows facing the two issues separately, leading to a more modular approach.
- **Independence.** It is quite obvious that a model is independent of platforms and applications; moreover, we require a model that is independent of environments, and that enables a useful notation, which allows interoperability and independence.
- **Concrete usability.** We are not interested in producing a formalism for interactions, instead, we would like to promote a way to support and simplify the development of interactions in agent-based applications.
- **Promotion of locality.** A trend that has recently appeared and seems to be accepted in the agent area is the adoption of locality in agent interactions, i.e. the environment is modeled as a multiplicity of local interaction contexts, representing the logical places where agent interaction activities occur, which can change depending on the agents’ movement. In fact, depending on its current location, an agent is situated on a given interaction context and there will be enabled to access local resources and to interact with local executing agents [CabLZ02c].

To meet the above requirements, the BRAIN (Behavioural Roles for Agent INteractions) project [BRAIN02] proposes an approach that respects the agent-oriented features, enables separation of concerns [CabLZ02b] and concrete usability, and promotes locality in interactions. In this project, the interactions among agents are based on the concept of role. A role is defined as a set of capabilities and an expected behaviour. The former is a set of actions that an agent playing such role can perform to achieve its task. The latter is a set of events that an agent is expected to manage in order to “behave” as requested by the role it plays. Interactions among agents are then represented by couples action-event, which are dealt with by the underlying interaction system, which can enforce local policies and rules. There are different advantages in modelling interaction by roles and, consequently, in exploiting derived infrastructures. First, it enables a separation of concerns between the algorithmic issues and the interaction issues in developing agent-based applications. Second, it permits the reuse of solutions and experiences; in fact, roles are related to an application scenario, and designers can exploit roles previously defined for similar applications; for instance, roles can be exploited to easily build agent-oriented interfaces of Internet sites [Cab01]. Finally, roles can also be seen as sort of design patterns [AriLan98]: a set of related roles along with the definition of the way they interact can be considered as a solution to a well-defined problem, and reused in different similar situations.
In this paper, we show how the interaction model of BRAIN can be implemented; the paper presents an interaction infrastructure, called Rolesystem, which is based on the proposed models for roles and interactions.

The paper is organised as follows. Section 2 introduces the adopted models of roles and interactions. Section 3 presents an application example that will be used to concretely show the features of the infrastructure. Section 4 shows the implementation of Rolesystem, the interaction infrastructure based on the previously presented models. Section 5 makes a comparison with other implementation approaches. Section 6 concludes the paper and sketches some future work.

2. Role and Interaction Models

In this section we present the models of roles and of interactions, adopted in the BRAIN project, on which the Rolesystem interaction infrastructure presented in the Section 4 is based. Inside the BRAIN project we have defined also XRole, an XML-based notation for agent roles [CabLZ02a]. It aims at describing roles in a way that supports portability and interoperability, and allows the exploitation of roles at different phases of the application development. See [CabLZ02d] for further details about the models and the related advantages.

2.1 Modeling Roles

In our approach, a role is modeled as a set of the capabilities and an expected behavior, both related to the agent that plays such role (see Figure 1). There are some characteristics of roles that lead to deal with them separately from the concept of agent. The role is temporary, since an agent may play it in a well-defined period of time or in a well-defined context. Roles are generic, in the sense that they are not tightly bound to a specific application, but they express general properties that can be used in different applications and then for different agents. Finally, roles are related to contexts, which means that each environment can impose its own rules and can grant some local capabilities, forcing agents to assume specific roles. As mentioned before, roles represent behaviors that agents are expected to show; who expects such behavior are entities external to agents themselves, mainly organizations [ZamJW01] and environments. This model of role leads to a twofold viewpoint of the role: from the application point of view, the role allows a set of capabilities, which can be exploited by agents to carry out their tasks; from the environment point of view, the role imposes a defined behavior to the entities that assumes it.

The former point is that a role is a set of capabilities, i.e., a set of actions that agents assuming a given role can perform. Since the proactiveness feature of agents, they have to perform actions to carry out their tasks, and so, they must be enabled to do it. Note that our aim is not to limit the agents’ possibilities, but to limit them in order to manage them easier, granting the actions needed to carry out tasks related to the specific role.

The latter point is that an agent assuming a given role is expected to exhibit a specific behaviour. Since the agents are reactive, they are sensible to what happens in the environment where they live, no matter if it is a specific request rather than a change occurred. The “expected behaviour” is the reactions to incoming events; it is “expected” because the agent is supposed, at least, to receive events. We think that a behaviour in terms of managed events can well shape the reactivity feature of agents.
2.2 Modeling Interactions

In the BRIAN project, we propose a three-level model for agent interactions, thought to overcome the complexity of interactions in multi-agent applications and the uncertainty that must be faced in the Internet [CabLZ02d]. In this model (see Figure 2), the application level is represented by the agents; the lowest level concerns the environment, which defines its own policies to rule agent-to-agent and agent-to-resources interactions. The middle level is the one focused in this paper, and enables the interactions among agents and between agents and the environment, by using the concept of role, which is modelled as explained in the previous subsection.

![Fig. 2. The model for interactions with roles](image)

An interaction between two agents occurs when the former one performs an action (chosen among the set of capabilities belonging to the role it plays) and such action is translated into an event that is notified to the latter agent that exhibits the specific behaviour. The underlying interaction system provides for the translation from actions to events (see Figure 3). In the following we first present an application example and then we will focus on the implementation of the interaction system.

![Fig. 3. Interaction between two agents](image)

3. An Application Example

In this section we present an application example that will be used in the following to show the concrete exploitation of the Rolesystem infrastructure. This application is related to agent negotiation, and in particular, addresses auctions. This kind of application presents several interaction requirements that are dealt with by the Rolesystem infrastructure; moreover, auctions can be attended also by mobile agents [SanH00], which introduce relevant issues in coordination and interaction [CabLZ02c].

In an auction there are agents that make resources available and agents that are interested in using/acquiring such resources. The former ones are usually called sellers, while the latter ones are called bidders. Usually, there is an intermediate agent, called auctioneer, which actually performs the negotiation. The price of the resources sold by sellers via an auction is not fixed, but it is dynamically determined by the interest of the bidders. The seller can set a reserve price, i.e., a price under which it does not want to sell the resource. There are several different forms of auction, depending on the number of participants, on the criteria with which the resources are assigned, and so on. We focus on the auctions with one seller and multiple bidders at a time, ruled by several mechanisms: for example, English, Dutch, first-price and Vickery [Ago96].
In our application, we develop roles that implement the bidder, the seller and the auctioneer. Such roles can be assumed by agents attending auctions. In this way, the same agent can behave as bidder and as seller in different period of its life, by assuming the appropriate roles.

4. Rolesystem Implementation

Rolesystem is an interaction infrastructure that implements the interaction model of BRAIN. It is completely written in Java to grant high portability and to be associated with the main agent platforms. The concrete platform we chose to implement Rolesystem is Jade [BelICTR01], a FIPA compliant agent platform, which allows also mobility of agents.

As shown in Figure 4, the Rolesystem infrastructure is divided into two parts: the upper one is independent of the agent platform, while the lower part is bound to the chosen agent platform, in this case Jade. We remark that it is not possible to have a complete independent implementation, but our effort was in the direction of reducing the platform-dependent part.

In applications exploiting the Rolesystem infrastructure, agents can be seen by two points of view: they are both subjects of the role system and agents of the Jade platform. Accordingly, an agent is composed by two layers: the subject layer, representing the subject of the role system independent of the platform, and the wrapper layer, which is the Jade agent in charge of supporting the subject layer. A specific agent, called server agent, is in charge of managing the roles and their interactions. It interacts with the wrapper layer of agents by exchanging ACL messages formatted in an appropriate way.

![Fig. 4. Domain separation in the Rolesystem infrastructure](image)

In our implementation, we developed several Java classes, both platform-independent and related to Jade. The main platform-independent classes are reported in the UML class diagram of Figure 5. Such classes have been grouped in the rolesystem.core and rolesystem.roles packages.

The connection between the subject layer and the wrapper layer is granted by two Java objects, instances of classes implementing respectively the RoleSystem and RoleRegistration interfaces, which provide methods to register agents with roles, to search for agents playing a given role, to listen for events and to perform actions. Such interfaces are described in detail in the next subsection.
4.1 RoleSystem and RoleRegistration Interfaces

The RoleSystem interface enables agents to perform preliminary operations needed to assume a role; since agents do not play roles yet, such operations are called “anonymous”. The following methods are available:

- `reqRegistration()`, to register in the system with a specified role. The server agent registers the requiring agent (or, better, the subject); this method returns an object that implements the RoleRegistration interface, detailed later, which enables the use of the role.
- `searchForRoleAgent()` and `searchForRoleAgentNoWait()`, to search for agents playing a given role. In the case of blocking operation, the method returns if there is a matching registration or as soon as an agent performs a matching registration. In any case, the methods return an array (possibly empty) of registrations.

Note that, when an agent asks for registration, it already has the class implementing the role; the registration is needed by the role system to know which role such agent plays.

```java
RoleRegistration registration;
try {
    registration=roleSystem.reqRegistration(Bidder.ROLE_ID);
} catch(RoleException re) {
    System.err.println(re);
    return;
}
```

With regard to the presented application example, Figure 6 shows the request for registration of an agent A1 that wants to assume the bidder role. The request is made by invoking the reqRegistration method, supplying the name of the role (the static field ROLE_ID of the Bidder class,
see Figure 8) as parameter. This invocation occurs in the code of the class that implements the subject layer of the agent A1.

The latter interface is RoleRegistration, which enables agents to perform operations on the system via a specific registration (i.e., after that the agent has assumed a role). The following methods are defined:

- listen, with and without timeout, and listenNoWait, to listen for occurring events. The operation without timeout is blocking, and returns as soon as an event occurs. The methods return an object of event class, or null.
- doAction, to perform a given action; this method will be detailed later.
- whoAmI, to know the identifier associated to the role registration.
- dismiss, to cancel the registration and leave the corresponding role.

Examples of use of this interface will be shown in the Subsection 4.3.

To play a role, the first step to be performed by an agent is to obtain an object that implements the RoleRegistration interface, by the invocation of the reqRegistration method. The returned object represents the association between the agent and a specific role, i.e., the role assumed. Then, such object can be exploited to perform actions and manage events, as better described in the next subsection. As soon as an agent does not need to play the assumed role, it can release the role registration via the dismiss method. If the agent wants to assume a role again (the same or another one), it has to require another registration via the reqRegistration method.

### 4.2 Roles, Actions and Events

A role is implemented by an abstract class, where the features of the role are expressed by static fields and static methods. Figure 7 reports the UML class diagram that explains the relationships among the classes that represent roles, actions and events.

The class that implements a role has the same name of the role, and is part of a package that represents the application scenario for such role. A static and final field called ROLE_ID, of type String, is exploited to identify the role; to avoid name conflicts, the value of this field is the concatenation of the name of the package with the name of the role, in the form package.Role. In the following, we report some code fragments of the classes belonging to the auction application example. Figure 8 reports the beginning of the class that represents a bidder. In particular, we can outline the presence of the ROLE_ID field, which defines the name of the role, used in the reqRegistration method to require the registration as a bidder.

![Fig. 7. The diagram of an example of role](image-url)
package rolesystem.roles.auction;
import rolesystem.core.RoleAction;
import rolesystem.roles.KnownEvent;

/**
 * This role is the bidder of an auction.
 * Keywords: auction, bidder.
 */
public abstract class Bidder {
    /**
     * Role identifier.
     */
    public static final String ROLE_ID="auction.Bidder";

    public static RoleAction askSituation(int addressee) {
        return new RoleAction("askSituation", addressee);
    }

    public static RoleAction bid(Id addressee, Price content) {
        return new RoleAction("bid", addressee, content);
    }

    public static RoleAction talk(Id addressee, String content) {
        return new RoleAction("talk", addressee, content);
    }

    Fig. 8. The beginning of the Bidder class

    Each action defined in a role is built by a static method, which is in charge of creating an appropriate instance of the class RoleAction and returning it to the caller. Such a static method has the same name of the corresponding action and one or two parameters: the former one is the agent addressee of the event corresponding to the action; the optional latter parameter is the information content to perform the action. For example, Figure 9 reports some actions defined in the Bidder class of the application example.

    Fig. 9. Some actions defined in the Bidder class

    To perform an action, an agent playing a given role must obtain the appropriate RoleAction instance, invoking the corresponding static method of the role class. Then, it has to invoke the doAction method of RoleRegistration to actually perform the action, supplying the previously created
instance of RoleAction (see Figure 11 for an example of code). Then, when the server agent receives
the request to perform the action via the wrapper layer, translates it into a known event, and sends it
to the addressee agent.

To find partners to interact with (i.e., addressee agents), agents exploit the
searchForRoleAgent methods made available by the RoleSystem interface, by which the agent can get
a list of the registrations related to a specific role, each one specified by an identifier.
The addressee agent waits for incoming events by invoking the listen method of the
RoleRegistration interface. When an event for this agent arrives, the listen method returns an instance
of the class RoleEvent, and then the agent can evaluate whether the incoming event is among the
recognized ones, which are defined in the role class as instances of the class KnownEvent. This class
describes the name of the event, the role assumed by the sender of the event, and the class of the
information content of the event. Thanks to the match method of this class, agents can compare the
known events (instances of KnownEvent) with the occurred event (instance of RoleEvent); this
method returns true if and only if:
  o The name of the occurred event is the same of that of the known event.
  o The role of the sender agent is the same of that of the known event.
  o Both the events are without any information content; otherwise the information content of the
    occurred event can be casted to the class of the information content of the known event.

The names of the KnownEvent instances correspond to the names of the events they
represent, preceded by the prefix ‘KE_ ’. Figure 10 reports some recognized events for the Bidder
class of the application example. Each event is represented by a static field initialized with the
appropriate instance of KnownEvent.

```java
/**
 * An auctioneer notifies the current situation of the auction.
 * Sender role: Auctioneer
 * Content: Situation of the auction, i.e. proposed price, winning bidder
 * and countdown state.
 */
public static final KnownEvent KE_notifySituation=new
KnownEvent("notifySituation", Auctioneer.ROLE_ID, Situation.class);

/**
 * An auctioneer notifies that I have won the auction and thus I must pay.
 * Sender role: Auctioneer
 * Content: Price that must be paid.
 */
public static final KnownEvent KE_youWon=new KnownEvent("youWon",
Auctioneer.ROLE_ID, Price.class);

/**
 * Another bidder talks.
 * Sender role: Bidder
 * Content: Message.
 */
public static final KnownEvent KE_talk=new KnownEvent("talk",
Bidder.ROLE_ID, String.class);
```

Fig. 10. Some events defined in the Bidder class

4.3 An Interaction Example

An interaction between two agents occurs when the former one performs an action (chosen
among the available to the role it plays) and such action is translated into an event that is notified to
the latter agent that exhibits the specific behaviour. The Rolesystem infrastructure provides for the 
translation from actions into events.

As an example of interaction in the proposed auction application, we show how a bidder 
agent interacts with an auctioneer agent to make a bid. We suppose that both agents are registered 
with the respective roles, and the bidder agent is interested in the good on sale by the auctioneer. 
Also, the bidder agent has limited money to spend, so its budget field expresses the maximum 
affordable bid. Figure 11 reports a fragment of the code of the bidder agent. Since the kind of the 
auction is English, the agent retrieves the price currently proposed by the auctioneer (which is more 
than the latest bid), and, if it is less than or equal to its money availability, perform the action of 
bidding.

```java
... 
registration=roleSystem.reqRegistration(Bidder.ROLE_ID); } 
... 
// get the price proposed by the auctioneer and set my bid 
myBid=situation.getPrice(); 
// if I can afford such bid 
if( myBid.getAmount()<=budget.getAmount() ) 
  // do bid by performing the related action, 
  // specifying the auctioneer addressee and my bid 
  RoleAction action = Bidder.bid(auctioneer, myBid); 
  registration.doAction(action); 
}
...
```

**Fig. 11. How a bidder agent performs a bid**

The bidding action performed by the bidder agent is received by the server agent of the 
Rolesystem infrastructure, which translates it into an appropriate event. In particular, the bid action 
of the bidder is translated into a KE_bid event and sent to the auctioneer agent specified as addressee 
by the bidder.

```java
... 
// listen for incoming events 
event=registration.listen(); 
// if the occurred event matches a bid action 
if(Auctioneer.KE_bid.match(event))
{
  // get the bid from the event 
  Price tmp=(Price)event.getContent(); 
  // if the bid is more than or equal to the proposed price 
  if( tmp.getAmount()>=proposed.getAmount() )
  {
    // record the bid 
    reached=tmp; 
    // and the bidder agent 
    winnBidder=event.getSender();
  }
}
...
```

**Fig. 12. How an auctioneer agent manages a bid**

Figure 12 shows the behaviour of the auctioneer agent. When it receives an event, it 
evaluates whether one of the known events matches with the incoming event (following the rules 
reported in the previous subsection). In the depicted case, the received event matches with the 
KE_bid event, which means that another agent has performed the bid action. The auctioneer agent,
by managing the incoming event, knows the bid price and, if it is a valid bid, retrieves the bidder agent, recording it as the current winner.

The administrator of a site can enforce local interaction laws, by defining which interactions are allowed in the site. In particular, (s)he can set a grid of permissions, which tells who can interact with who, and each interaction permission must specify the sender and the addressee; since the interactions in Rolesystem are asymmetric, it may happen that a role A can interact with role B, but role B cannot interact with role A. An appropriate GUI can be exploited to set permissions. With regard to our application example, we can figure out that there are auction sites where bidder are not allowed to exchange information, to avoid collusions; in this case, the local administrator have to set permission in the appropriate way, denying interactions where the sender and the addressee are both bidder.

5. Comparison with other Approaches

5.1 Traditional Approach

A traditional approach implies the definition of methods in the agent code, to interact with other agents (or entities). The mostly adopted model is the message-passing one, usually exploited in a “procedure call” fashion, i.e., the agent invokes a method and expects an answer as return value.

```java
... public Situation bid(int price, Agent sender) {
    // if the bid is more than or equal to the proposed price
    if(price.getAmount()>=proposed.getAmount() )
    {
        // record the bid
        reached=price;
        // and the bidder agent
        winnBidder=sender;
    } // return the situation with the current winner
    return new Situation(winnBidder);
} ...
```

Fig. 13. The fragment of the auctioneer in the traditional approach

Figure 13 shows an example of method that manage an interaction between a bidder and an auctioneer: the method bid of an auctioneer is invoked by a bidder that wants to make a bid; moreover, the method returns the current situation.

Even if this approach is the simplest one, its main limitation is that the specific features concerning the role played by the agent are not separated from the general features, for instance from the mobility or the planning features. This leads to some important drawbacks:

- Interactions are not well integrated with the agent characteristics, since it is difficult to clearly point out how they fulfill the proactiveness and the reactivity. Interactions are more object-oriented than agent-oriented: in particular, interactions and events are dealt with in a different way, leading to a fragmented implementation without a global management of the possible situations.
- At the design phase, the traditional approach does not enable the definition of roles as well-defined entities, which can be exploited in the further development phases, leading to fragmented and more error-prone solutions.
- The software is more difficult to be maintained, because modifications or additions often influence the whole code, and cannot be isolated in the related sections.
o From the site point of view, specific site-dependent code of actions cannot be provided, and agents must embody the code from the beginning. Moreover, it is not possible to enforce local laws and to control the interactions among them.

5.2 Aspect-Oriented Approach

Even if it has not been designed in connection with roles, Aspect Oriented Programming (AOP) seems to provide interesting mechanisms to support the management of roles for agents [CACM01, Ken00]. AOP starts from the consideration that there are behaviours and functionalities that are orthogonal to the algorithmic parts of the objects [Kic97]. So, it proposes the separate definition of components and aspects, to be joined together by an appropriate compiler (the Aspect Weaver), which produces the final program. The separation of concerns introduced by AOP permits to distinguish the algorithmic issues from the behavioural issues. Since an aspect is “a property that affects the performance or semantics of the components in systemic way”, it is evident the similarity with a role. AOP is thought to achieve performance and maintainability in software development.

Figure 14 reports an example of use of AOP in our application. The Bidder aspect implements the role, and provides the appropriate methods that are embodied in the agent code by the Aspect Weaver; for instance, in Figure 14 they are added to the ag instance of the class MyAgent.

Even if the AOP approach is similar to our, in our opinion it has some limitations:

o First, the role/aspect must know the class which is going to modify, for instance, in Figure 14 the aspect Bidder must know the MyAgent class to add the appropriate methods.

o As a consequence of the first point, this approach lacks flexibility in the definition and usage of aspects, and this is due to the fact that AOP focuses on software development rather than addressing the uncertainty and complexity issues of wide-open environments, such as the Internet.

o Finally, interoperability among agents of different applications is hard to be achieved, since different applications are likely to use different implementations of the agents, so the roles defined in an application scenario may fit only one implementation.

```java
public class MyAgent
{
    // intrinsic members of the class
    private Price bid;
    ...
}

aspect Bidder extends Role
{
    ...
    // introduce extrinsic member to Agent
    introduce public void MyAgent.make_bid() {}
    // advise weaves impact extrinsic members
    advise public void MyAgent.make_bid()
    { // code of the bidding action}
    ...
}

// Java code to instantiate MyAgent and Bidder
// and to attach ag to the aspect
MyAgent ag = new MyAgent("Shopper");
Bidder bidderAspect = new Bidder();
bidderAspect.addObject(ag);
ag.make_bid(ag.bid); // ag makes a bid
...
```

Fig. 14. The fragment of the bidder agent in the AOP approach
As a last note, not bound to the implementation, we can say that AOP does not provide a support for the designer as effective as the one provided by XRole, our XML-based notation that allows a rich description of the roles.

6. Conclusions

This paper has presented the implementation of an interaction system for agents, based on roles. Our approach, whose aim is to concretely support the development of interactions in agent-based Internet applications, has several advantages. Differently from traditional proposals in the area, which are mainly adaptations of older models, in our approach interactions are modelled and implemented following an agent-oriented approach, i.e., all the peculiar features of agents are taken into account. Exploiting our interaction system, agent-based application developers can focus separately on algorithmic issues and interaction issues, leading to a more modular approach and to separation of concerns. To obtain independence, our implementation relies on two parts: the former one is independent of the agent platform adopted, while the latter (very small) is related to the adopted agent server. So, when a developer chooses to exploit the interaction system in his/her applications, (s)he has to write only the server-dependent part. Finally, our approach promotes locality, since each context can specify the allowed roles and consequently can rule the local interactions.

With regard to future work, we point out some research directions. An interesting issue to be faced is the exploitation of roles at runtime by agents. In fact, it may require appropriate mechanisms and constructs, possibly provided by the implementation language. In particular, we are extending our infrastructure to dynamically apply roles to mobile agents, to perform tests in a very dynamic environment. Another interesting issue is security: appropriate mechanisms can be defined to control the requests for registration, to revoke roles by administrators, and to specify a lease for each registration (i.e., a timeout after which the registration expires). Finally, even if the shown application example involves general interaction issues, other application areas must be assumed as test bed to evaluate the usability of our approach, and designers and development must be involved to evaluate our approach.

References


