Role-based Interaction Infrastructures for Internet Agents

Giacomo CABRI†, Letizia LEONARDI†, and Franco ZAMBONELLI††, Nonmembers

SUMMARY  With no doubt the Internet will achieve advantages in exploiting software agents for applications, thanks to their autonomy in carrying out tasks. In such a scenario, appropriated methodologies are needed to manage the interactions among agents. The BRAIN framework proposes an interaction model based on roles, each one composed by a set of capabilities and an expected behavior. The achieved advantages are agent-oriented features, separation of concerns and reuse of solutions and experiences. In this paper we present two interaction infrastructures, Rolesystem and RoleX, which rely on the above mentioned role-based interaction model. These infrastructures allow agents to assume roles and to interact. An application example and the comparison with other approaches show the effectiveness of our approach.

key words: agents, roles, internet applications, interactions

1. Introduction

Interactions among entities have been carefully taken into consideration since the rising of distributed systems, but, with the advent of software agents, this issue has been more focused, because sociality is one of the key features in agent oriented software engineering. It is exploited in the context of 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 [15]. In addition, the spread of open systems, such as the Internet, has led to a scenario in which also agents of different applications may interact in a competitive way, for example to achieve resources. Besides sociality, we recall that the other main features of agents are proactivity (i.e., the capability of carry out tasks) and reactivity (i.e., the capability of reacting to the environment changes). Furthermore, the feature of mobility, allowing agents to change their execution environment, adds great flexibility at conceptual and implementation levels, but also introduces peculiar issues in interactions, which must be taken into account [13]. There have been different proposals in the area of agent interaction and coordination; they have concerned message passing adapted to agents, “meeting point” abstractions, event-channels, and tuple spaces [5].

From the analysis of these proposals, we argue that they suffer from being adaptations of older approaches traditionally applied in the distributed system area and do not take into account the new agent-based 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, separation of concerns, independence, and promotion of locality.

To these purposes, the BRAIN (Behavioural Roles for Agent INteractions) framework [4] proposes an approach where the interactions among agents are based on the concept of role. A role is defined as a set of capabilities and an expected behavior. 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 modeling 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 enables 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. In particular, roles can also be seen as sort of design patterns [1]: 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 present the BRAIN framework and how two interaction infrastructures can be implemented following the BRAIN model and adopting the BRAIN notation. The paper is organized as follows. Section 2 presents the BRAIN framework and proposes an application example in the field of agent-mediated auctions. Section 3 presents two interaction infrastructures developed inside BRAIN, Rolesystem and RoleX. Section 4 compares our approach with other two, the traditional OO-based and the Aspect Oriented Pro-
gramming, and sketches some related work. Finally, Section 5 concludes the paper.

2. The BRAIN Framework

The BRAIN framework [8] adopts a well-defined model for roles and agent interactions. The BRAIN framework defines also XRole, an XML-based notation for agent roles; 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. The XRole notation is not described in this paper; the interested readers can find more details in [6].

2.1 Role and Interaction Model

We can think at a role as a stereotype of behavior common to different agents in a given situation. Such a behavior is exhibited by the agent, but is also expected by other entities, mainly other agents, organizations [27] and environments. It is useful to deal with roles separately from the agents. In our approach a role is modeled as a set of capabilities and an expected behavior, both applied to the agent that plays such role (see Figure 1). 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 playing a given role can perform. This takes into account the proactiveness feature of agents, since they have to perform actions to carry out their tasks, and so, they must be enabled to do it. The latter point is that an agent playing a given role is expected to exhibit a specific behavior. This accounts for the agent’s reactivity, since 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 behavior” is constituted by the reactions to incoming events; it is “expected” because the agent is supposed, at least, to receive events.

In the BRAIN framework [8], an interaction between two agents occurs when one agent 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 another agent that exhibits the specific behavior (see Figure 1). The underlying interaction system provides for the translation from actions to events. In the following the reader can find a concrete example of roles developed on the base of the BRAIN model. Note that BRAIN provides for interaction mechanisms, disregarding the information content of the interactions, which can be coded accordingly to one of the proposed languages, such as Agent Communication Language (ACL), Knowledge Query Manipulation Language (KQML), and Knowledge Interchange Format (KIF) [21].

2.2 An Application Example

The application we exploit to show the concrete use of our approach is related to agent negotiation, and in particular, addresses auctions. This kind of application has been chosen because it presents several interaction requirements; moreover, auctions can be attended also by mobile agents [22], which introduce relevant issues in coordination and interaction [7].

In an agent-mediated auction there are seller agents that make resources available and bidder agents that are interested in using/acquiring such resources. Usually, there is an intermediate agent, called auctioneer, which actually deal with 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.

In our application example, we develop roles that implement the bidder, the seller and the auctioneer. Referring to the proposed model, the actions of a bidder agent (or, better, of an agent that has assumed the bidder role) are for instance “making a bid”, “asking for the situation” and “talking with another bidder”. The events managed by a bidder agent are “notification about the situation”, “notification of the winner” and “request for talking”. With regard to interactions, the “making a bid” action of a bidder agent is translated into an “accept a bid” event of the auctioneer agent. The roles can be assumed dynamically by agents attending auctions. In this way, the same agent can behave as bidder and as seller in different periods of its life, by assuming the appropriate roles.

Figure 2 reports some fragments of the XRole document defining the bidder role.

2.3 Enforcing Local Policies

Exploiting a BRAIN infrastructure, the administrator of a site can enforce local interaction policies, 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 BRAIN are asymmetric, it may happen that a role A can interact with role B, but role B cannot interact with role A. BRAIN infrastructures can define appropriate GUIs to set permissions. With regard to our application example, we can figure out that there are auction sites where bidders are not allowed to exchange information, to avoid collusion; in this case, the local administrator have to set permissions in the appropriate way, denying interactions where the sender and the addressee are both bidder agents.

3. Interaction Infrastructures

In this section we present two interaction infrastructures developed inside BRAIN, to support the management of roles in mobile-agent based applications. The former, called Rolesystem, relies on an interaction system that enables agents to register themselves in one of the local contexts as playing one or more roles, to perform actions and to receive role-related events [9]. The latter infrastructure, called RoleX, is more dynamic because it enables the dynamic addition of the role features to the agents, via a modification of the Java bytecode of the agent classes. The exploitation of descriptors for roles allows the uncoupling between a role and its actual implementation, and this enables agents to assume roles unknown at compilation time [10].

3.1 The Rolesystem Infrastructure

Rolesystem is an interaction infrastructure that implements the interaction model of BRAIN [9]. Being part of BRAIN, Rolesystem exploits the XRole notation to describe roles. It is completely written in Java to grant high portability and to be associated with the main agent platforms. As shown in Figure 3, 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. The concrete platform we chose to implement Rolesystem is Jade [3], a FIPA compliant agent platform, which allows also mobility of agents. We remark that it is not possible to have a complete platform-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 for each context/environment.

In our implementation, we developed several Java classes, either platform-independent or related to Jade. The connection between the subject layer and the wrapper layer is granted by two Java objects provided by Rolesystem, instances of classes implementing respec-
tively the RoleSystem and RoleRegistration interfaces; the former provides methods to register agents with roles and to search for agents playing a given role; the latter provides methods to listen for incoming events and to perform actions.

To play a role, the first step to be performed by an agent is to obtain an object that implements the RoleRegistration interface, which represents the association between the agent and the specific assumed role. Then, such object can be exploited to perform actions and manage events, as better described in the next subsection. For instance, in the application example, an agent can assume the role of bidder by executing the following code:

```java
RoleRegistration registration = roleSystem.reqRegistration(Bidder.ROLE_ID);
```

### 3.1.1 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 4 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 (i.e., the context) for such role.

Each action defined in a role is represented 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. The following code reports the bid action of the bidder role:

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

Once an agent playing a given role has obtained the appropriate RoleAction instance, it can actually perform the related action by invoking the doAction method of RoleRegistration. 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.

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). The names of the KnownEvent instances correspond to the names of the events they represent, preceded by the prefix "KE_". The following code is an example of definition of an event, in particular the event received by the bidder that has won the auction:

```java
public static final KnownEvent KE_youWon =
    new KnownEvent("youWon", Auctioneer.ROLE_ID, Price.class);
```

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 behavior.

### 3.2 The RoleX Infrastructure

The RoleX infrastructure is a good implementation of the BRAIN model, but its degree of dynamism is limited by the fact that agent programmer must know the role classes at the implementation phase; moreover, role features are not embodied in the agents, and at runtime roles and agents remain separated components. To overcome these limitations, we have implemented a further infrastructure [10], called RoleX (Role eXtension), where roles are conceived as dynamic extensions of agents. The current implementation of RoleX relies on the IBM Aglets agent platform [20], but it can easily associated to other Java platforms by adapting the few platform-dependent classes.

#### 3.2.1 Role, Action and Event Descriptors

To grant a high level of abstraction in assuming, playing and discarding roles, RoleX exploits descriptors for roles, actions and events. A descriptor is an object that describes a role, an action or an event, for example with
some keywords, an aim, a version, a creation date and any further needed piece of information. Action descriptors are exploited to associate specific methods to operations. Event descriptors tell the kind and the context of the occurred event, but not how to manage it. Role descriptors describe what such role does but not how (with which operations) it is done. A role descriptor includes also the descriptors of the corresponding actions and events.

Using descriptors, the agent programmer does not need to know which is the physical class that implements a role, but only the descriptor of the role to be searched for. For example, if the agent must assume the bidder role, the programmer can write code that searches not directly for a bidder role but for a role with a bidder description. The agent can further verify the retrieved descriptor(s) to be sure that the role is the right for it. The descriptors are useful also for hiding to the agent the physical location of the role implementation, allowing agent programmers to disregard about the work of role programmers, and vice versa, because the role behavior is described in a separate way. Descriptors are stored in an appropriate database which can be queried by agents.

As for the previous infrastructure, roles are described in XML exploiting the XRole notation of BRAIN. From these XML documents, we can derive the code of Java classes that concretely implement descriptors.

3.2.2 Extending Agents with Roles

While the descriptors provide a high-level description of the roles, we need also the corresponding code, which will be the concrete extension of the agents. In our approach the Java code of a role is composed of two parts: a Java interface (called role interface) and a Java class (called role implementation).

The fact that an agent assumes a role means that the infrastructure dynamically adds each role implementation member (both methods and fields) to agent members, in order to add the set of capabilities of the role, thus modifying and extending the agent class bytecode. Moreover, the infrastructure forces the agent class to implement the role interface, in order to modify its expected behavior and to allow other agents to recognize it as playing that role (for instance, by means of the instanceof operator).

To implement the dynamic role extension, our system is based on a special class loader, called “role loader”, that can change agent behavior and external appearance. The idea is simple: an agent that wants to assume a new role, after querying the role descriptor database, asks the role loader to reload itself with the new role (see Figure 5), as shown in the following piece of code:

```java
RoleDescriptor rd = new RoleDescriptor("bidder");
RoleLoader.addRoleToMySelf(rd);
```

If everything is right, the role loader sends the agent an event to indicate that the agent has been reloaded. After the reload event the agent can resume its execution. To release a role, the process is the same, but this time the agent is unloaded without that role.

The role loader performs bytecode manipulation to extend the agent with the role. This manipulation is completely made in memory without recompilation. The manipulation is needed to work with and to modify class definitions. Note that this manipulation is not dangerous for code portability and is compliant with the Java security manager. Our implementation of role loader is based on Javassist bytecode manipulation engine [23]. The addition of the role's members to the agent class is performed at each level of the inheritance chains, as depicted in Figure 6. Interested readers can find more technical details in [11].

3.2.3 Role Use by Agents

The programmers do not know anything about the role implementation but know by the descriptors, which actions can be used, and which events can occur. In the following, we focus on the action use, because the management of the events is similar and simpler.

The use of descriptors means that the programmer cannot write code that invokes methods corresponding to role actions in the usual way, because a compile-time error will occur. Therefore, there must be an invocation translator that do introspection on the extended agent to dynamically find which method must be call in response to an invocation on an action description. When the agent invokes a role action, it specifies to the invocation translator a descriptor of the action that wants to perform, the translator searches for a method that corresponds to the description and then invokes it. For instance, in the application example, the bidder agent can search for a descriptor of an action that performs a bid, and provides the invocation translator with the found descriptor to actually bid.
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);
}

4. Comparison with other Interaction Approaches

4.1 Traditional OO Approach

A traditional object-oriented 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.

Figure 7 shows an example of method that manages 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, returning 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.
- From the Internet 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.

4.2 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 [12], and was exploited by E. Kendall in her work [18]. AOP starts from the consideration that there are behaviors and functionalities that are orthogonal to the algorithmic parts of the objects [19]. 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 behavioral issues. Since an aspect is “a property that affects the performance or semantics of the components in systemic way” [19], it is evident the similarity with a role. AOP is thought to achieve performance and maintainability in software development. Figure 8 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 8 they are added to the ag instance of the class MyAgent.

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

- First, the role/aspect must known the class which is going to modify, for instance, in Figure 8 the aspect Bidder must known the MyAgent class to add the appropriate methods.
- 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.

- 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.

4.3 Other Role-based Approaches for Agents

AALAADIN [16] is a meta-model to define models of organizations. It is based on three core concepts: agent, role and group. The last is a set of agent aggregation, which can be considered atomic with regard to a task to accomplish or to dependencies between agents. Even if our approach is quite similar to the AALAADIN one, it differs for a couple of reasons. First, we disregard the concept of group, while focusing on the interactions among agents and between agents and environments. Then, AALAADIN roles are tightly bound to the notion of agent, while our aim is to describe roles in a more independent way, both of applications and environments.

The ROPE project [2] recognizes the importance of defining roles as first-class entities, which can be assumed dynamically by agents. It proposes an integrate environment, called Role Oriented Programming Environment, which can be exploit to develop applications composed by several cooperating agents. In our opinion, ROPE lacks to address the interaction between competitive agents, while BRAIN overcomes this limitation. In fact, our definition of roles can be exploited not only to let agent dynamically assume roles, but also for interactions among agents that do not belong to the same application; this is a relevant aspect in the design of applications for wide-open environments, such as the Internet.

Yu and Schmid [26] exploit roles assigned to agents to manage workflow processes. They traditionally model a role as a collection of rights (activities an agent is permitted on a set of resources) and duties (activities an agent must perform). An interesting issue of this approach is that it aims to cover different phases of the application development, proposing a role-based analysis phase, an agent-oriented design phase, and an agent-oriented implementation phase. Differently from BRAIN, Yu and Schmid do not take into particular consideration the implementation phase, suggesting the exploitation of existent agent platforms, which however do not implement role concepts. Finally, the fact that this approach focuses on the workflow management makes it quite close, but in our opinion there are some interesting cues that can be exploited in a wider range of application area.

RoleEP (Role based Evolutionary Programming) [24] aims at supporting the development of cooperative applications based on agents. In particular, it addresses applications where different agents collaborate to achieve a goal, defining an application as a set of collaborations among agents. RoleEP proposes four model constructs: environments, objects, agents and roles. A role is an entity that belongs to an environment and is composed of attributes, methods, and binding-interfaces. Role attributes and methods can be whatever needed to accomplish the related tasks. The binding-interfaces are exploited to dynamically associate role functions to objects; they can be thought as abstract methods, and are in charge of invoking the concrete methods of the objects they are bound. In this way, RoleEP enable agents to assume roles at runtime, granting high flexibility. The RoleEP approach is quite similar to the BRAIN one, since it enables separation of concerns and role assumption at runtime, but RoleEP seems to focus on the implementation phase only, without providing support during the other development phases, as BRAIN does.

There are also other role-based approaches, not compared here because they disregard the implementation phase (such as Gaia [25]), or because they are quite different from ours (such as TRUCE [17]).

5. Conclusions and Future Work

In this paper we have presented the BRAIN framework and two related interaction infrastructures, Rolesystem and RoleX. In this framework, 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 infras-

```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("A1");
Bidder bidderAspect = newBidder();
bidderAspect.addObject(ag);
// ag makes a bid
ag.make_bid(ag.bid);
...
```

Fig. 8 The bidder agent in AOP
structures, agent-based application developers can focus separately on algorithmic issues and interaction issues, leading to a more modular approach and to separation of concerns. In addition, our approach promotes locality, since each context can specify the allowed roles and consequently can rule the local interactions. Finally, the RoleX infrastructure exhibits a high degree of dynamism, enabling agents to assume also roles unknown by their programmers.

With regard to future work, we point out some research directions. An interesting issue is security: appropriate mechanisms can be defined to control the requests for assuming roles, to revoke roles by administrators, and to specify a lease for each role (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.

Acknowledgement

This work has been supported by the NOKIA Research Center of Boston and by the Italian Research Council (CNR) within the project “Mobile software agents to enable access to multimedia services by mobile users and devices”. We would like to thank Silvio Corciolani and Luca Ferrari for their precious implementation work.

References


Giacomo Cabri

Giacomo Cabri is a research associate in Computer Science at the University of Modena and Reggio Emilia, since 2001. He received the Laurea degree in Electronic Engineering from the University of Bologna in 1995, and the PhD in Computer Science from the University of Modena and Reggio Emilia in 2000. His affiliation is the Department of Information Engineering at Modena. His research interests include methodologies, tools and environments for agents and mobile computing, wide-scale network applications, and object-oriented programming.

Letizia Leonardi

Letizia Leonardi is a full professor in Computer Science at the University of Modena and Reggio Emilia, since 2001, where she teaches basic and advanced computer science courses. She received the Laurea degree in Electronic Engineering in 1982 and the PhD in Computer Science in 1989, both from the University of Bologna. Her affiliation is the Department of Information Engineering at Modena. Her research interests include design and implementation of coordination infrastructures for mobile agent systems, object-oriented programming environments, and parallelism and distribution issues, especially as they apply to object systems.

Franco Zambonelli

Franco Zambonelli is an associate professor in Computer Science at the University of Modena and Reggio Emilia, since 2001. He obtained the Laurea degree in Electronic Engineering in 1992, and the PhD in Computer Science in 1997, both from the University of Bologna. In 1996, he obtained a fellowship to do research activity in distributed systems at the Brown University (Rhode Island, USA). In 1999, he has been visiting researcher at the University of Southampton (UK), where he has done research activity in the area of intelligent agents. His current affiliation is the Department of Engineering Methods and Science at Reggio Emilia. His current research interests include: middleware for mobile and embedded systems, agents and mobile agents technologies, agent-oriented software engineering.