# Role-based Approaches for Engineering Interactions in Large-scale Multi-Agent Systems

Giacomo Cabri[1], Luca Ferrari[1], Franco Zambonelli[2]

[1]Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia
41100 Modena, Italy
{cabri.giacomo, ferrari.luca}@unimo.it
[2]Dipartimento di Scienze e Metodi dell'Ingegneria – Università di Modena e Reggio Emilia
42100 Reggio Emilia, Italy
zambonelli.franco@unimo.it

**Abstract**. This chapter discusses how the concept of role can be exploited in engineering large-scale multi-agent systems, in order to support the development of the applications and to simplify the related tasks. A role can be considered as a stereotype of behavior common to different classes of agents. Role-based approaches can give several advantages, the most important of which is the separation of concerns between different issues (for instance, algorithmic ones and interaction-related ones), which is very useful to simplify the development of large-scale agent-based applications. A survey of different approaches shows the advantages and the peculiarities of introducing roles in agent-based application development. Moreover, we present in detail the BRAIN framework, developed at the University of Modena and Reggio Emilia, which is an approach based on roles to support agent developers during their work.

## 1 Introduction

Agent sociality leads to autonomy in interactions, allowing scalable decompositions of large-scale applications in terms of decentralized multi-agent organizations [15], and enabling interaction among agents not only belonging to the same application but even to different ones, as happens in the people real world. Therefore, interactions are an important issue to be faced in agent-based applications, both whether they occur between cooperating agents of the same application and between competitive agents belonging to different applications. *Sociality*, *proactiveness* (i.e., the capability of carrying out their goals) and *reactivity* (i.e., the capability of reacting to environment changes) are considered the main features of the agents [22], and in our opinion must all be taken into account in developing interactions between agents. The fact that agents can move from host to host adds further flexibility, but also introduces peculiar issues in interactions [13]. In the following we consider two scenarios, ubiquitous computing and the Web, to discuss about interaction requirements in large-scale distributed environments.

An approach that seems to be useful in this field is the exploitation of the abstraction concept of *roles*. Roles represent a concept that have been exploited in several software engineering fields, such as UML [37] or RBAC [36], and this emphasizes how roles can be useful in engineering interactions; of course, they must be adapted to be applied to the agent scenarios. A role is a stereotype of behavior common to different classes of agents [17, 26] and can generally be considered as a set of behavior, capabilities and knowledge that agents can exploit to perform their task(s). There are different advantages in introducing roles in interactions and, consequently, in exploiting derived infrastructures. The most important one for large-scale development is that the use of roles enables a separation of concerns between the algorithmic issues and the interaction ones. In the literature we can find different role-based approaches addressing different phases of the multi-agent system development. This chapter presents a survey of different approaches and point out their main limitations; in our opinion, the most important limit is that the presented approaches address only single aspects of the agent-based software development (for instance, some propose a model, while other focus on implementation) and a more complete approach is missing.

After a survey of existing role-based approaches, we present in detail the BRAIN (Behavioral Roles for Agent INteractions) framework, developed at the University of Modena and Reggio Emilia [10]. BRAIN proposes an approach based on roles whose aim is to support the development of interactions in agent-based applications, overcoming the limitations of existing approaches. To this purposes it provides for (i) a simple yet general *model of interactions* based on roles, (ii) an XML-based *notation* to describe the roles, and (iii) implementations of interaction *infrastructures* based on the previous model and notation, which enables agents to assume roles and interact. In particular, BRAIN proposes a specific definition of role, which is not in contrast with the previously mentioned general definition, but it is quite simple and can be easily exploited in concrete implementations. In BRAIN, 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 infrastructure, which can enforce local policies and rules.

The structure of this chapter is the following. Section 2 proposes two large-scale application scenarios that propose challenging requirements that multi-agent application developers are going to meet. Section 3 reports on different role-based approaches that can be found in the literature. Section 4 describes the BRAIN framework. Section 5 shows the usefulness of roles, and of BRAIN in particular, by means of an application example. Finally, Section 6 concludes.

## 2 Background and Motivations

Two significant examples of complex software systems that introduce challenges in large-scale application development are *pervasive computing* and *Web-based services*. In this section we list the main requirements emerging in such scenarios and sketch how role exploitation can meet such requirements.

Pervasive (or ubiquitous) computing aims at assist users in carrying out their tasks by means of a computing infrastructure that is ubiquitous and invisible [39]. Such infrastructure is composed of several connected entities [4, 14], mainly sensors (which keep track of the evolution of the physical environment) but also devices that interact with people. Such a scenario is *highly dynamic*, both because the entities belonging to the same context may change very quickly and because there is a lot of pieces of information that must be evaluated. The different entities are *heterogeneous* (i) as kind, ranging from the so-called *smart dust* (small networked sensors) [25] to powerful notebooks, (ii) as devices' capabilities, and (iii) as running OSs and applications. Moreover, in such a world the failures are not exceptions, but normally occur, leading to *unpredictability* that must be faced by entities in an autonomous way. From our point of view, one important requirement of ubiquitous computing is *spontaneous interoperation* [28], i.e., the capability of different entities of interacting, without human intervention, to carry out a task. Very different from the current interaction among computers, it requires autonomy in discovering other entities, adaptability in talking, and dynamism in assuming appropriate roles [32].

The other challenging scenario is constituted by Web-based services. Internet and the Web have rapidly spread all over the world, and are clearly influencing the everyday life of millions of people. It is a scenario more mature and spread than pervasive computing, and the exploited technologies are quite settled. However, it is still evolving, shifting from a bare information container to a provider of several kinds of service. This does not calm the call for new methods and tools to face the application requirements introduced by such an evolution. The content of the Web is required to be more *dynamic*, to suit on demand services. The willing of exploiting the Internet everywhere has led to the need of facing the *heterogeneity* (in terms of capabilities, OSs and applications) of devices, such as cellular phone, PDA, and whatever the device technology will propose. Today's and future Internet applications are component based, and must be open and adaptable. Also in this scenario, the interaction among different components is a relevant issue because have to take into consideration heterogeneity, dynamism, and unexpected situations.

Pervasive computing and Internet scenarios are converging, because they exhibit similar properties and both call for software development methodologies based on components, capable of dealing with the described characteristics in an autonomous way while carrying out tasks. A suitable recognized approach is the agent-oriented one, which is based on the definition of autonomous software entities that carry out tasks on behalf of users, reacting to the changes of the environment where they live [20, 43]. So the agent-based approaches seem to provide the adequate abstractions for building complex

software systems [21], thanks to their autonomy, which help application developers in identifying responsibilities, and their adaptability, which enables them to face dynamic situations [27].

Moreover, as mentioned, in the two challenging application scenarios, interactions are an important issue to be faced. In the multi-agent field 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; interacting system components can be thought as organizations of agents. Multi-agent system massively exploit the feature of sociality, since 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]. Moreover, scenarios such as pervasive computer and the Web-based services imply that 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, such as localization, site and platform dependences, which must be taken into account [13].

We argue that an important aspect in the development of multi-agent applications is the *separation of concerns* between algorithmic issues and interaction issues. This facilitate developing applications because allows facing the two issues separately, leading to a more modular approach. This is one of the main aims of our approach. Besides this, the depicted scenarios point out the following non-functional *requirements* for the development of agent interactions in large-scale applications:

- *Generality*. Approaches should be quite general, and not related to specific situations, to be effectively adaptable and flexible.
- *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. Due to its movements, 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 [8].
- *Reusability*. Developers should not reinvent the wheel for any new application. A valid approach must enable an easy and wide reuse not only of code but also of solutions.
- *Agent oriented features*. In our opinion, the most important requirement is that interactions have to be modeled following an agent-oriented approach, i.e., all the peculiar features of agents must be taken into account.
- *Practical usability*. Besides formalisms for modeling interactions (such as UML-based approaches to AOSE [30, 34]), we argue that the agent interaction development must be supported and simplified in a concrete way.

To meet the above requirements, a clear and simple *model* must be adopted, and, on the base of it, *tools* and *infrastructures* must be built to support the developers of agent-based applications.

There have been different proposals in the area of agent interaction and coordination. They have concerned message passing adapted to agents, "meeting point" abstractions [40], event-channels [2], and tuple spaces [6]. However, 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 peculiar features of agents (mainly proactiveness, reactivity and sociality), leading to fragmented approaches. Some efforts in this direction model interactions in terms of services and tasks [18], tailoring interactions on agent features. Moreover, traditional approaches often consider agents as bare objects; this implies that the different features of agents are managed in a not uniform way, leading to fragmented approaches. Finally, there is no approach that covers several phases of the software development, leading to fragmentation of the solutions.

The approaches that seem to overcome the mentioned limitations are the one based on the concept of *role*. As mentioned above, a role is a stereotype of behavior common to different classes of agents [17] and can generally be considered as a set of behavior, capabilities and knowledge that agents can exploits to perform their task(s). There are different, well-recognized advantages in modeling interactions 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 [10]. Second, being a high-level concept, roles allow independence of specific situations, and promote quite general approaches. Third, roles can be developed according to local needs and laws, promoting locality in the development of large-scale distributed applications. Fourth, roles permit 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; roles can also be seen as a sort of design patterns [1, 31]: 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. The last two requirements previously mentioned (the *agent oriented features* and the *concrete usability*) do not derive from roles, but must be met by specific approaches.

## 3 A Survey of Existing Approaches

In this section we propose a survey on role-based approaches in the area of agent-based application development. The concept of role is not exploited at same way in all approaches, and usually it is used in only one phase of the application development. For instance, the first two approaches uses roles at the model level, others at the implementation level, and further ones in a mixed way.

Gaia [41] is a methodology for agent-oriented analysis and design. Its aim is modeling multi-agent systems as *organizations* where different roles interact. In Gaia, roles are used

only in the analysis phase and are defined by four attributes: *responsibilities*, *permissions*, *activities*, and *protocols*. Gaia exploits a formal notation (based on the FUSION notation [11]) to express permissions of a role. Even if it is not a notation to completely describe roles, it can be helpful because enables the description of roles in terms of what they can do and what they cannot. In addition, Gaia proposes also an interaction model that describes the dependencies and the relationships between the different roles of the system, and it is considered a key point for the functionalities of the system. In Gaia, the interaction model is a set of *protocol definitions*; each protocol definition is related to a kind of interaction between roles. A possible limitation of Gaia is that roles are considered only in the analysis phase.

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. The group concept can be useful when a large number of agents are involved, as may happen in large-scale applications: in fact, it enforce modularity and allows an easy decomposition of tasks, still keeping the single agents simple. In AALAADIN roles are tightly bound to the notion of agent, and this can become a drawback if developers' aim is to describe roles independently of agents.

The ROPE project [3] 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. Since this approach strictly focuses on collaboration, in our opinion, it lacks to address the interaction between competitive agents, while other approaches are more flexible and provide roles also for interactions between agents that do not belong to the same application; this is a relevant aspect in the design of applications for large-scale and open environments, such as the Internet.

Yu and Schmid [42] 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. Anyway, 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 (such as the exploitation of roles and the covering of different phases) that can be exploited in a wider range of application area.

RoleEP (Role based Evolutionary Programming) [38] 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 and enabling separation of concerns and role assumption at runtime. Nevertheless, RoleEP seems to focus on the implementation phase only, without providing support during the other development phases.

TRUCE (Tasks and Roles in a Unified Coordination Environment) is a script-based language framework for the coordination of agents [19]; the main requirements it aims to meet are *adaptability*, *heterogeneity* and *concurrency*. To these purposes, TRUCE allows the definition of coordination rules that are applied to roles that are assumed by agents, disregarding which actual agents will be implied in the coordination. In TRUCE, a role describes a *view* of an agent, hiding the other details, and letting the agent reveal only the properties concerned by the assumed role. The fact that TRUCE enables agents to adapt to various coordination protocols dynamically, makes it an open collaboration framework. As in the case of RoleEP, TRUCE addresses only the implementation level, by defining collaboration scripts that are interpreted by the agents.

The evaluation of the different approaches confirms that the exploitation of roles in agent-based applications provides different advantages. The main one is the separation of concern that they enable at different levels. In addition, they allow reuse of solutions, more agent-oriented views, and flexibility. However, none of the proposed approach exploits the concept of role to support the whole application development, leading from fragmented solutions. Starting from the above advantages and from the analysis of the limitations of the existing approaches, we propose the BRAIN framework detailed in the next section.

## 4 The BRAIN Framework

BRAIN (Behavioral Roles for Agent INteractions) is a role-based developing framework whose aim is to cover the agent-based application development process at different phases [10]. As shown in Fig. 1, the BRAIN framework provides three different components, structured in as much layers: (i) a *model of interactions* that is based on roles, (ii) an XML-based *notation* to describe the roles, and (iii) *interaction infrastructures* supporting agents in the management of roles. Such infrastructures are based on the adopted model and rely on the defined XML notation. Our framework provides for more than one interaction infrastructure, since different environments can have different needs in terms of implementation, while it is important that all infrastructures relies on the same model and notation, so that high-level descriptions of solution can be easily shared and managed by developers.

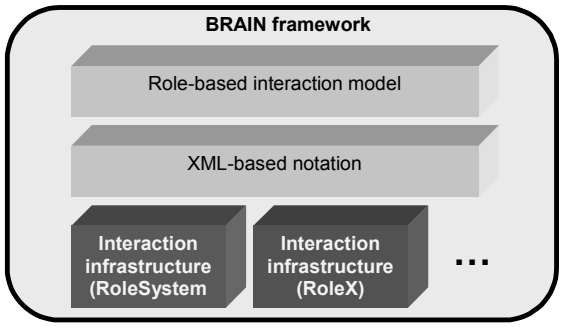


**Fig. 1.** The BRAIN framework

### 4.1 BRAIN Model and Notation

In BRAIN, 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. Fig. 2 shows how an interaction between two agents occurs. When the Agent A wants to interact with the Agent B, it performs an action chosen among the capabilities provided by the role. This action is translated into an event by the interaction system, which is delivered to the Agent B, which is in charge of managing it in the appropriate way.

We chose this model of interactions because it is very simple and very general, and well suits the main features of the agents: the actions can be seen as the concrete representation of agent *proactiveness*, while the events reify the agent *reactivity*.

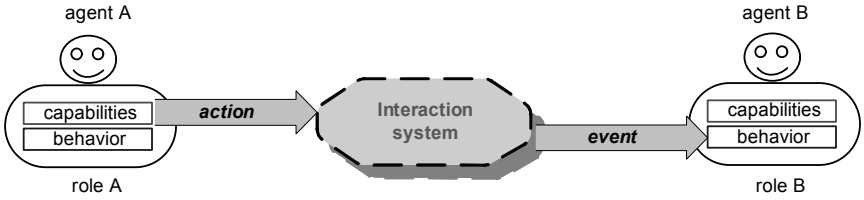


**Fig. 2.** The Interaction model in BRAIN

The notation proposed by BRAIN, called XRole [7], enables the definition of roles by means of XML documents. The use of XML grants high interoperability, since it leads to the capability of correctly parsing and understanding role definitions even by different software entities, possibly developed with different technologies. Furthermore, thanks to XML, developers can get different views on the same document, for example using a XSL

style sheet. This leads to the capability of using only information really needed. An example of XRole document is reported in Subsection 5.1. It is worth noting that each different representation derives from the same information, so the different phases of the development of applications rely on the same information, granting continuity during the entire development. For instance, during the analysis phase, the analysts create XRole documents following the XML Schema proposed in BRAIN XML-based notation, which guides them in the definition of the role features. These XRole documents can be translated into HTML documents to provide high-level descriptions also for further uses. In the design phase, the same XRole documents can be translated into more detailed HTML documents to suggest functionalities of the involved entities. Finally, at the implementation phase, again the same XRole documents can be exploited to obtain Java classes that implement the role properties.

### 4.2 BRAIN Interaction Infrastructures

As shown in Fig. 1, BRAIN allows the implementation of different interaction infrastructures, which can be plugged-in into the agent system by the platform administrator. The interaction infrastructures enable the assumption of roles by the agents, possibly in different ways as shown in the following. Moreover, they manage the interactions between agents in the sense that they actually perform the translation of actions into events, and the delivery of the latter. The interaction infrastructures can control interactions and enforce local policies, such as allowing or denying interactions between agents playing given roles; this is done in addition to the security policies enforced by the adopted programming language, for instance the Java security manager.

This feature of BRAIN allows a great adaptability, since each platform can use the more efficient interaction infrastructure for its purposes. In fact, since we believe that there is not a single optimal implementation, the use of such architecture allows us to develop several implementations, testing them to find which are better and for which purposes.

It is important to note that the implementation infrastructure is the bottom layer of the BRAIN framework (see Fig. 1), and this means that the two top layers do not change depending on the interaction infrastructure. This leads to more reusable software, since all the analysis and the XRole documents can be applied to different interaction infrastructure implementation. In other words, the use of such architecture, allows the developers to concentrate on the analysis phase discarding each detail of the interaction infrastructure, because their results will be always portable on the BRAIN framework. A further consideration relates is the use of events in BRAIN, which enables different implementations to communicate using a common mechanism (i.e., events themselves). This is really important, since the two agents could be in platforms with different BRAIN implementation, and still able to communicate each other by exchanging events.

At the moment we provide two main implementations of the BRAIN interaction infrastructure, as shown in Fig. 1.

The first one, called *RoleSystem* [9], relies on abstract classes which represent roles. In this approach, when an agent wants to assume a role, it must register itself to the core of the system. When the *RoleSystem* core receives a registration request, it evaluates if the agent can assume the role and then, in case of success, returns to the agent a *registration certificate*. This certificate allows the agent to play the requested role, which means to execute role actions. In fact, the agent executes actions and waits for events through its registration certificate, which in other words is simply a connection between the agent and the role itself. When the agent decides to release a role it simply communicate to the *RoleSystem* core to dismiss the role. The core invalidates the registration certificate so that the agent can no longer use it, and this corresponds to the exit from the interaction context.

The main advantage of this approach is that it is quite simple, both to use and to implement. This leads to a quite light architecture which can be easily exploited from devices with limited resources (such as PDAs, mobile phones, etc.). Nevertheless this approach requires a central unit, the *RoleSystem* core, which overhangs all the role assumption/release process. This means that (i) there is a single point of failure in the whole system, and (ii) agents do not *assume* and *release* roles, but simply take connections to them.

Starting from the above consideration we developed a new and more innovative infrastructure, called *RoleX* (Role eXtension for agents) [5]. In RoleX a role is composed of a full-implemented class, whose members are added to the agent when it assumes the role, and of an interface, which agents can implement in order to appear as playing the assumed role (*external visibility*). RoleX relies on a special component, called *RoleLoader*, which performs bytecode manipulation fusing the agent and the request role in a single entity. After this, the agent can exploit role members, such as method or variables, in a direct way, since they are embedded in the agent itself. Furthermore, during the manipulation process, the *RoleLoader* forces the agent to implement (in Java sense) the interface of the role, so that the agent will appear, from an external point of view, as playing such role. In other words, no central services maintain the role played by the agent, because a simple instanceof will give the role itself. When the agent decides to release a role, the *RoleLoader* performs a new bytecode manipulation in order to discard all role members, so that the agent regains its original structure.

RoleX exploits better the XRole notation, using XML documents that describe each role in a finer grain than in RoleSystem (still keeping the two infrastructures compliant). In fact, in RoleX each role is described by nested sections of XML documents, called respectively *RoleDescriptor, ActionDescription,* and *EventDescription*. Each descriptor describes a particular behavior of the role, starting from its main aim (*RoleDescriptor*), each action the role provides (*ActionDescription*) and which events will be generated by such actions (*EventDescription*). Thanks to this structure an agent can better analyze available roles than in RoleSystem, choosing in a more dynamic way the one that better fits.

The main advantage of this approach is the high dynamism provided: the agent can choose more dynamically the role at run-time, thanks to the more modularized use of the

XRole notation, and the assumption/release changes at run-time the agent behavior and appearance. Nevertheless, the use of the *RoleLoader*, which performs bytecode manipulation entirely in memory, without source code alteration or recompilation, leads to the need of more resources than in *RoleSystem*, and requires a more complex development.

Accordingly to the opening sentences of this section, we stress how is important to provide ad-hoc implementations dealing with the hardware-software context in which the BRAIN framework will be executed. Our implementations can cover both situations with limited resources (*RoleSystem*) and where high dynamism is required (*RoleX*). The two infrastructures proposed here are compliant with regard to the model and the notation, allowing also the communication between agents playing *RoleSystem* roles and agents playing *RoleX* ones.

## 5 An Application Example

This section provides an application example that exploits roles in order to build a more scalable and reusable large-scale software. The main aim of this section is to detail the steps involved in a role-prone development and to focus the advantages achieved by the use of roles.

The application chosen is a flying and hotel reservation system, developed of course as a MAS application in a large-scale environment. We suppose that reservations (for both flights and rooms) can be performed in appropriate Web sites that provide on-line services and can also host mobile agents. For instance, we can consider the hotel Web sites as the places where mobile agents can search for rooms and book them; similarly happens with the flight carrier Web sites for the flights. Our application exploits mobile agents capable of moving to different sites in order to search for rooms/flights and book them. In particular, we propose to split the application main task, which is the reservation of both the flight and the hotel for the user trip, into two smaller separated tasks: the reservation of the flight and the reservation of the hotel. Each task will be performed by a single mobile agent, which is in charge of traveling to find the best reservation for the hotel room or the flight[1]. The two agents will be called respectively *hotelReservator* and *fligthReservator*. Note that in our example we will exploit only two agents for simplicity's sake. In actual applications, a larger number of agents are involved: for instance, both the agents are likely to compete with other agents (belonging to other users) in order to obtain the reservation at a given price; another case is when the application is composed of different agents of the same kind, which perform a search in a parallel way. Nevertheless, the two agents of our example are representative of general interaction issues, and the

[1] We do not specify what "best" means; it could be the lowest price, or the more luxurious class. For the purposes of this section it does not mind, and we assume that the agent knows how to identify the "best" solution, accordingly to the will of its user.

adopted solution can be easily extended to more complex situations defining appropriate roles in a scalable way.

The application works as follows. As shown in Fig. 3, the two agents start from the user's host, which can be a standard workstation, a laptop, a PDA or something else. After that the agents move around all available platforms searching for the reservation.

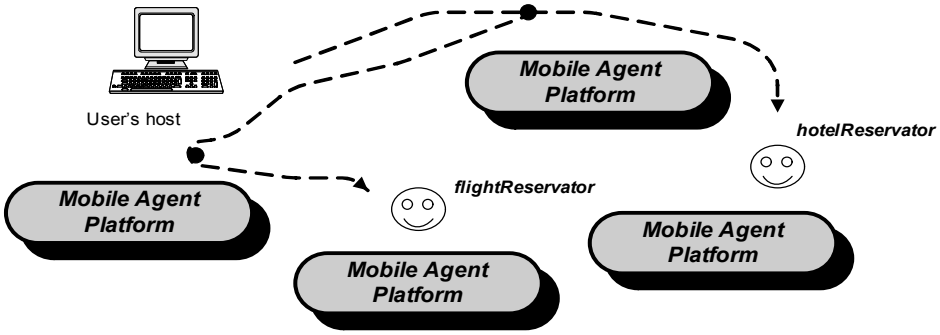


**Fig. 3.** The application scenario

We have chosen this kind of application because it involves several requirements and coordination issues, which are typical of large-scale distributed environments. In particular it is important to note that:

- Both the agents must interact with the external world, such as other agents or the reservation databases, in order to complete the reservation;
- The two agents must interact each other, in order to reserve both the hotel and the flight in a coordinate way.

Please note that, since the two agents are autonomous, they could find the two reservations at different platforms. For example, with regard to Fig. 3, the *hotelReservator* agent can find its best reservation at the bottom right platform, whilst the *flightReservator* can find it at the bottom left one. In this case the synchronization between the two agents must be remote, which means it implies a remote interaction. Instead the interactions among the above agents and the external world (i.e., each platform) are always local. This implies that the two agents require a mechanism to find each other and to communicate, in order to synchronize the reservation. As detailed below, roles can be useful in this situation, leaving the agent developers free to reuse an already existent communication library.

Let us start our application analysis from the interaction among agents and the external world. Since both the agents must travel among different sites (platforms), and each site could have different ways to manage reservations, agents cannot embed in themselves the needed knowledge and behavior to face all possible situations. This suggests embedding this knowledge and behavior in a set of local roles, provided by each platform. Thanks to

this, the agent developer is not in charge of knowing how the platform manages reservation, but it must simply develop the agents so that they will search (and play) an appropriate role. For example, as shown in Fig. 4, the *hotelReservator* agent can play a *room_searcher* role in order to search for and, in case, book the room. The *hotelReservator* agent is not in charge of knowing how the site manages reservations, since, as detailed above, the *room_searcher* role is in charge of knowing it. The agent simply assumes the role and then exploits a service via the role, which means the agent performs a role action. The role supports the agent in the execution of the service, translating the requested service into a site dependent action, and then returning results to the agent. For example, as shown in Fig. 4, the *hotelReservator* searches for all available rooms via a *room_searcher* role operation, and then it can evaluate the best (if there is one) for the user. Fig. 4 a) shows how the *room_searcher* role performs a direct query to a local database, while the figure b) shows a more complex situation, where there is an *administrator agent* which manages and authorizes all accesses to the database. The difference in the above cases is that in the former the access to the booking database is direct, while in the latter it is undertaken to the will of the *administrator agent*. Nevertheless, as written more times, the *hotelReservator* agent does not worry about the access way to the database, since it is the role that translates the requested service to the site dependant action(s).

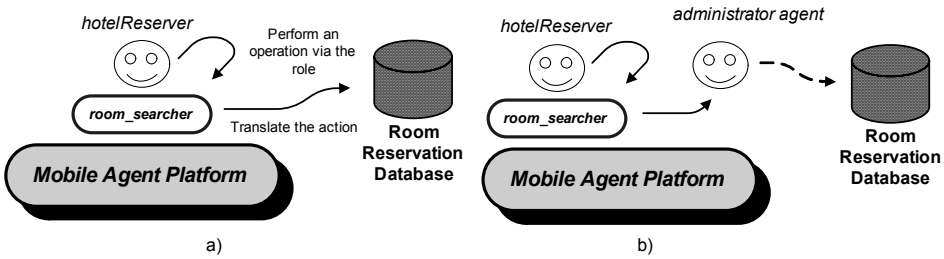


**Fig. 4.** Exploiting a role to perform different platform specific actions

Similar consideration can be done for the *flightReservator* agent, which can exploit a *flight_searcher* role to query the flight-booking database.

The use of the above roles allows the agents to better interface to a lot of different scenarios, but not only: since these roles are provided by each platform, the agents are smaller, in terms of code size, making faster the mobility (i.e., agent transmission through the network).

The above roles can be played by the agents to book effectively a flight or a room, when they judge that the reservation is "good". But as written above, one requirement of this application is to reserve a room in a coordinate way with the flight reservation. To achieve this goal the agents must interact each other. For example, when the *hotelReservator* finds the best available room, it must communicate to the other agent, the

*flightReservator*, that the room has been found, so that starting from this moment the other agent can book the flight. Furthermore, when the *flightReservator* find the best flight, must contact again the *hotelReservator* in order to check if the room is still available (since the *hotelReservator* is waiting for a flight to be found), and if both the agents can successfully book the service, they synchronize themselves and complete the booking. Of course similar consideration can be applied in the case the *flightReservator* finds the flight before the *hotelReservator* finds a room. Both the agents can book the room or the flight using a time-out. They have to confirm the reservation before the time-out expires, else the room (or the flight) will be considered available again. The time-out is required to avoid deadlock problems and to grant to an agent the required time to contact and deal with its partner (i.e., the other agent). For example, supposing that the *hotelReservator* finds the best room, it can temporary lock it, and then it can contact the *flightReservator* in order to establish if the room can be definitively booked. If the *flightReservator* has found the best flight, both the agent can book; if the time-out expires, the application continues. The use of time-out partially solves the communication between the two agents, which is still a distributed-commit problem.

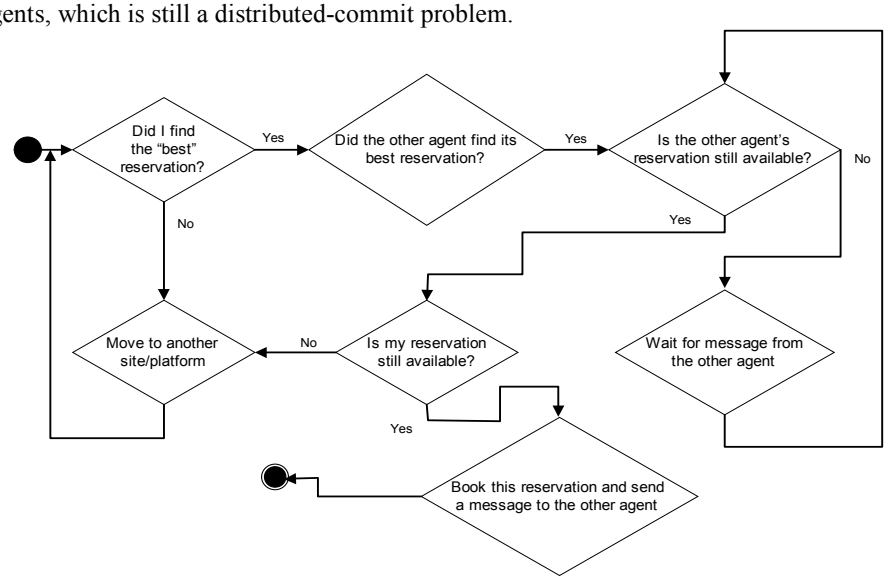


**Fig. 5.** The flow chart for agent synchronization

The flow chart in Fig. 5 shows the application logic needed to achieve this synchronization. From that state chart it is possible to note how, when an agent finds the best reservation, it asks the other agent if it has found the best reservation too (and if it is still available). If this is the case, the booking can be committed for both the agents. In the other case, the agent must wait until the other agent finds the best reservation (i.e., it waits for a message). When this happens the booking can be committed. Please note that, since

the first agent that finds the best reservation must wait until the other finds it too, the reservation could be no more available (maybe booked by an agent of another user). It is for this reason that the agents must synchronize themselves also on the availability of the reservation.

The above considerations suggest the use of a quite complex mechanism to synchronize the two involved agents. Instead of developing this communication mechanism, which implies also the use of a way to track the agent position (i.e., on which host the other agent is), developers can still use ad-hoc roles. The problem of finding another agent running in another platform/site is a quite common problem in all mobile agent applications, and different communication protocols have been built to allow it. Instead of embedding each possible discovering/communication protocol, such as MASIF [33], RMI [35], Jini [23], CORBA [12], JXTA [24], etc., in the agent, this can exploit a *remote_communicator* role to discover and communicate (i.e., exchange messages) with it. This role hides all network specific details, even about proxies and protocol tunneling. Fig. 6 briefly shows how the agents can communicate exploiting the above role. Note that the agents can assume more roles at the same time, such as the one required for the remote communication and the one required for completing the booking. Similarly an agent has to assume the *remote_communicator* role only when this is strictly needed, that means when a (possible) reservation has been found. This emphasizes how roles provide dynamic services, which can be used at run-time on demand.

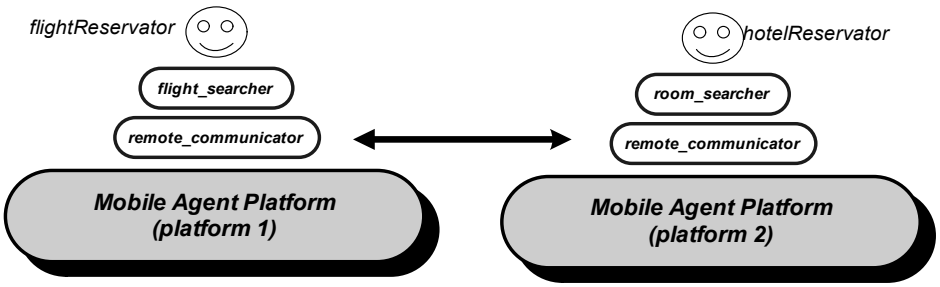


**Fig. 6.** Communication between the two agents through the remote_communicator roles

When both the agents have found their reservations and have rightly booked the room and the flight, they return the results to the user and the application ends.

The above application has shown how useful can be roles in the development of large-scale mobile-agent applications. In particular, roles allow developers to not worry about specific platform-dependant details, providing also code reusability, since the same role can be exploited from different agents in different applications. Furthermore, since roles act as a run-time addition, they can be developed in a separate way from agents, enabling separation of concerns that simplifies application development. This implies also that roles can evolve separately from agents, easing maintenance and modification of the software; agents can continue to play roles without worrying about these changes.

```
<?xml version='1.0'?>

<role xmlns="http://polaris.ing.unimo.it/schema/RoleDescriptionSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:SchemaLocation="http://polaris.ing.unimo.it/schema/RoleDescriptionSchema" >

   <GenericRoleDescription>
     <description>Hotel Reservator Role</description>
     <roleName>HotelReservator</roleName>

     <keyword>hotel</keyword>
     <keyword>reservation</keyword>
     <keyword>room</keyword>
     <keyword>accomodance</keyword>
     <version>1</version>

       <OperationDescriptor>
         <name>Available Rooms</name>
         <aim>see available rooms</aim>
         <keyword>room</keyword>
         <version>1.0</version>

         <methodName>ListRooms</methodName>
         <returnType><className>java.util.List</className></returnType>
       </OperationDescriptor>

       <OperationDescriptor>
         <name>Reserve a room</name>
         <aim>take a reservation</aim>
         <keyword>room</keyword>
         <keyword>reserve</keyword>
         <version>1.0</version>

         <methodName>ReserveRoom</methodName>
         <returnType><className>java.lang.Integer</className></returnType>
         <paramName><className>java.lang.Integer</className></paramName>
         <paramName><className>java.lang.Integer</className></paramName>
         <receivingEvent>
              <eventClassName>example.role.ReservationConfirmEvent</eventClassName>
         </receivingEvent>
       </OperationDescriptor>

       <!—other operations here …. -->

   </GenericRoleDescription>
</role>
```

**Fig. 7.** The XRole document for the *hotelReservator* role

### 5.1 Application Code

This section shows some pieces of code related to the above detailed application; in particular we report the definition of roles in XRole (independent of implementation), and the implementation of roles in the RoleX infrastructure.

Fig. 7 shows the main part of the XML notation related to the *hotelReservator* role. The document is compliant with the http://polaris.ing.unimo.it/schema/RoleDescriptionSchema schema, which is not detailed here due to space limitation. The document can be read by humans, and it defines a role through a *role descriptor* (tag <GenericRoleDescription>), which defines some semantic data related to the role itself. This data can be, for example, a list of *keywords*, an *aim*, the *version*, etc. Nevertheless, even if this semantic data is really useful to search for a particular role and analyze it, it is not enough. In fact, since an agent will use some role services (i.e., operations) to perform its task(s), the description of these services must be included in the XRole document. To achieve this, the *role descriptor* can contain one or more *operation descriptors*. An *operator descriptor* describes a single role operation that the agent can use to perform a specific task; this information is embedded in the tag <OperationDescriptor>. As for the role, an *operator descriptor* collects data such as *keywords, aim, name, version,* etc. and a piece of information that enables the agent to actually request the service to the implementation infrastructure. In particular this binding is achieved through the *method name* and the *events* related to the method (respectively with the tags <methodName> and <receivingEvent>). The former information details which method will be invoked to perform the specific operation, while the latter details which event(s) will be sent from the above method.

With regard to the *hotelReservator* role, Fig. 7 shows the two main operations of the role itself. The first, realized through the method *ListRoom*, simply returns a list of available rooms at a particular host/platform. The second, based on the method *ReserveRoom*, can be used to reserve a specific room. This method accepts also two parameters, both of type integer, that are the credit card number used to confirm the reservation and the room number. The return value, that is an integer too, is the reservation number id.

There is an important difference between the above two operations: the latter, which corresponds to the *ReseveRoom* method, declares also an event, called *ReservationConfirmEvent*, while the other operation does not. The above event is sent back from the system to the agent to inform that its reservation has been accepted. The agent must manage this event in order to know its reservation status[2]. Since there is no need for the *ListRooms* method to get back an event, because it is a simple query to the system, it makes sense for the *ReserveRoom* since the reservation could require some time to be complete, and during this time the agent should be free to do something else (e.g., contacting the other agent).

---

[2] We simply suppose that the *ReservationConfirmEvent* can carry either a confirmation or a reject, so that a single event is needed to inform the agent about its reservation status.

After the XRole document, let us consider how to implement a role compliant to it. Fig. 8 shows an example of a Java implementation for the RoleX infrastructure.

```
public class hotelReservatorRole implements hotelReservatorInterface{
   // variable for the SQL database
   protected String url="jdbc:odbc:RoomDB";
   protected String username="hotelReservator";
   protected String passwd="password";
   protected boolean connected=false;
   protected java.sql.Connection db;

   // method to list all available rooms
   public java.util.List  ListRooms(){
      // contact the room database
      init();
      // query the database
      try{ String query="SELECT * FROM ROOMS WHERE AVAILABLE='true'";
         java.sql.ResultSet result=db.createStatement().executeQuery(query);
         // create a list from the result set
         return ResultSetToList(result);
      }catch(Exception e){return null;}
   }

   // a method to reserve a specific room
   public Integer ReserveRoom(Integer creditCard,Integer roomNumber){
      // reserve the room
      init();
      try{
         String update="UPDATE ROOMS SET AVAILABLE=false WHERE NUMBER="+roomNumber;
         db.createStatement().executeQuery(update);
         // get the reservation number
         String query="SELECT ID FROM RESERVATION WHERE ROOM="+roomNumber;
         java.sql.ResultSet result=db.createStatement().executeQuery(query);
         // convert to an integer
         return ResultSerToInteger(result);
      }catch(Exception e){}
      return new Integer(-1);
   }

// init the database connection
   protected void init(){
      try{ if(!connected){
            Class.forName("sun.jdbc.JdbcOdbcDriver");
            db=DriverManager.getConnection(url,username,password);
            connected=true;
         }
      }catch(Exception e){connected=false;}
   }
}
```

**Fig. 8.** The Java code for the *hotelReservator* role

The two methods *ResultSetToInteger* and *ResultSetToList* are not explained here because their implementation is not important for the purposes of this paper.

The role shown in Fig. 8 is thought for a platform where rooms are managed through a relational database. Nevertheless, as already written in the previous sections, roles allow agents to not worry about such details, so that an agent has simply to use the *hotelReservator*, independently of its implementation. For instance, if the reservations are stored in a file, the code of the role must be adapted, but the interface remains the same and the agent must not be modified.

```
public class hotelAgent extends Aglet{
    protected boolean hasRole=false; // a flag used when reloaded
  …

  public void run(){ // the execution method
    // load the role
    if(!hasRole){
      try{ hasRole=true;
        BRAIN.RoleX.RoleLoader.loadRole("hotelReservator");
        registerEventListener(this); // add myself as event manager
      }catch(Exception e){ hasRole=false;}
    }

    java.util.List rooms=act("ListRooms",null); // use role operations, by name

    int chosenRoom=choose(rooms); // choose the room

    If(chosen |=null && chosen>0){
      // ask to the other agent if ready
      // …..
      if(ready){  // reserve the room
          Object parms[]=new Object[2];
          parms[0]=cardNumber;
          parms[1]=chosen;
          act("ReserveRoom",parms);
      }
    }
  }

    // manage event
    public void notify(Object event){
      if(event instanceof ReservationConfirmEvent){
          // do something
      }
    }
}
```

**Fig. 9.** The agent code

The last piece of code shown in this section is related to the agent that is going to play the above role. Fig. 9 shows an example of Aglet agent, that is an agent for the IBM Aglets platform [29], which exploits the above *hotelReservator* role. In the reported code,

the agent knows the name of the role, but actually the RoleX infrastructure exploits role descriptors to uncouple the needed role from its implementation. In this way, agents can provide keywords to search for roles [5]. The role descriptors are not detailed here due to space limitation.

After the agent has asked the RoleX bytecode-manipulator, called *RoleLoader*, to load the specified role (i.e., to add the role members to the agent itself; for further details see [5]), it starts using the role through the special method *act(..)*. The latter is an added method that the agent can use after a role assumption (i.e., this method is added by the *RoleLoader*). This method performs a kind of reflection in order to find the role operation, and then invokes it and returns back the result.

Please note that the agent, after the role assumption, register itself as an *event listener*. This means that the RoleX implementation scatters generated events to that listener, which is in charge of managing them. In connection to the application example, when the booking system wants to confirm (or reject) the reservation, generates an event (i.e., a Java objects that contains a specific payload), and the agent, since it has registered itself, receives the event and manages it.

## 6 Conclusions

This chapter has presented how the concept of role can be fruitful exploited in engineering agent interactions in large-scale applications. Roles can carry different advantages, in terms of separation of concerns between the algorithmic issues and the interaction issues, generality of approaches, locality, and reuse of solutions and experiences. In dealing with agent interactions, roles can simplify the development because can embody the interaction issues, which are managed separately from the algorithmic issues that are embodied into agents. Roles can be developed separately from agents, implying separation of issues, solution reuse, more scalability and easier maintenance.

The survey of different existing approaches has confirmed such advantages in the development of agent-based distributed applications, but has also outlined a high degree of fragmentation in the evaluated approaches. In fact, the concept of role is usually exploited in only one phase of the application development, while only few approaches take roles into consideration in more than one phase.

Starting from the above consideration, we have proposed BRAIN, a framework to support agent developers during their work. BRAIN exploits the concept of role in different phases of the development, since it is based on a simple yet general role-based model for interactions. The presence of an XML-based notation enables interoperable and flexible definitions of roles, which can be tailored to different needs and managed by humans, agents and automatic tools. Finally, interaction infrastructures based on the BRAIN model and adopting the BRAIN notation provide for a suitable support at

runtime. An application example has shown how BRAIN can be exploited in a large-scale agent-based application.

With regard to future work, we are exploring some directions. First, we want to evaluate our approach in different application areas; we are currently developing a role-based application for e-democracy, to support the participation of citizen in public life. Second, we are going to consider trust and security issues in the use of roles: this will make our approach more feasible for real applications. Finally, we are exploring the development of high-level support for specific phases of the application development, built on top of the XRole notation.

**Acknowledgments:** Work supported by the Italian MIUR and CNR within the project "IS-MANET, Infrastructures for Mobile ad-hoc Networks", and by the MIUR within the project "Trust and law in the Information Society. Fostering and protecting trust in the market, in the institutions and in the technological infrastructure".

## References

1. Aridor, Y., Lange, D. B.: Patterns: Elements of Agent Application Design. Proceedings of Agents '98, St Paul, Minneapolis, USA, 1998.
2. Baumann, J., Hohl, F., Rothermel, K., Straßer, M.: Mole - Concepts of a Mobile Agent System. The World Wide Web Journal, Vol. 1, No. 3, pp. 123-137, 1998.
3. Becht, M., Gurzki, T., Klarmann, J., Muscholl, M.: ROPE: Role Oriented Programming Environment for Multiagent Systems. Proceedings of the Fourth IFCIS Conference on Cooperative Information Systems (CoopIS'99), Edinburgh, Scotland, September 1999.
4. Berlin, A. A., Gabriel, K. J.: Distributed MEMS: New Challenges for Computation. IEEE Computing in Science and Engineering, Vol. 4, No. 1, pp. 12-16, January-March. 1997.
5. Cabri, G., Ferrari, L., Leonardi, L.: Enabling Mobile Agents to Dynamically Assume Roles. The 2003 ACM International Symposium on Applied Computing (SAC), Melbourne, Florida, USA, March 2003.
6. Cabri, G., Leonardi, L., Zambonelli, F.: Mobile-Agent Coordination Models for Internet Applications. IEEE Computer, Vol. 33, No. 2, pp. 82-89, February 2000.
7. Cabri, G., Leonardi, L., Zambonelli, F.: XRole: XML Roles for Agent Interaction. Proceedings of the 3rd International Symposium "From Agent Theory to Agent Implementation", at the 16th European Meeting on Cybernetics and Systems Research (EMCSR 2002), Wien, April 2002.
8. Cabri, G., Leonardi, L., Zambonelli, F.: Engineering Mobile Agent Applications via Context-dependent Coordination. IEEE Transactions on Software Engineering, Vol. 28, No. 11, pp. 1040-1056, November 2002.
9. Cabri, G., Leonardi, L., Zambonelli, F.: Implementing Role-based Interactions for Internet Agents. Proceedings of the 2003 International Symposium on Applications and the Internet (SAINT 2003), Orlando, Florida, USA, January 2003.

10. Cabri, G., Leonardi, L., Zambonelli, F.: BRAIN: a Framework for Flexible Role-based Interactions in Multiagent Systems. Proceedings of the 2003 Conference on Cooperative Information Systems (CoopIS), Catania, Italy, November 2003.
11. Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P.: Object-Oriented Development: The FUSION Method. Prentice Hall International: Hemel Hempstead, England, 1994.
12. CORBA, http://www.corba.org/
13. Domel, P., Lingnau, A., Drobnik, O.: Mobile Agent Interaction in Heterogeneous Environment. Proceedings of the 1st International Workshop on Mobile Agents, Lecture Notes in Computer Science, Springer-Verlag (D), No. 1219, pp. 136-148, Berlin, Germany, April 1997.
14. Estrin, D., Culler, D., Pister, K., Sukjatme, G.: Connecting the Physical World with Pervasive Networks. IEEE Pervasive Computing, Vol. 1, No. 1, pp. 59-69, January 2002.
15. El Fallah-Seghrouchni, A., Haddad, S., Mazouzi, H.: Protocol Engineering for Multi-agent Interaction. Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW '99), Valencia, Spain, June 1999.
16. Ferber J., Gutknecht, O.: AALAADIN: A meta-model for the analysis and design of organizations in multi-agent systems. Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS), Cite des Sciences - La Villette, Paris, France, July 1998.
17. Fowler, M.: Dealing with Roles. http://martinfowler.com/apsupp/roles.pdf, 1997.
18. D'Inverno, M., Kinny, D., Luck, M.: Interaction Protocols in Agentis. Proceedings of the Third International Conference on Multi-Agent Systems, Cite des Sciences - La Villette, Paris, France, July 1998.
19. Jamison, W., Lea, D.: TRUCE: Agent coordination through concurrent interpretation of role-based protocols. Proceedings of Coordination 99, Amsterdam, The Netherlands, April 1999.
20. Jennings, N. R.: On Agent-Based Software Engineering. Artificial Intelligence, Vol. 117, No. 2, pp. 277-296, 2000.
21. Jennings, N. R.: An agent-based approach for building complex software systems. Communications of the ACM, Vol. 44, No. 4, pp. 35-41, 2001.
22. Jennings, N. R., Wooldridge, M. (eds.): Agent Technology: Foundations, Applications, and Markets. Springer-Verlag, March 1998.
23. Jini Network Technology, http://wwws.sun.com/software/jini
24. The JXTA Project, http://www.jxta.org/
25. Kahn, J. M., Katz R. H., Pister, K. S. J.: Mobile Networking for Smart Dust. Proceedings of the ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99), Seattle, WA, August 1999.
26. Kendall, E. A.: Role Modelling for Agent Systems Analysis, Design and Implementation. IEEE Concurrency, Vol. 8, No. 2, pp. 34-41, April-June 2000.
27. Kephart, J. O., Chess, D. M., The Vision of Autonomic Computing. IEEE Computer, Vol. 36, No. 1, pp. 41-50, January 2003.
28. Kindberg, T., Fox, A.: System Software for Ubiquitous Computing. IEEE Pervasive Computing, Vol. 1, No. 1, pp. 70-81, January-March 2002.
29. D. B. Lange, M. Oshima, "Programming and Deploying Java™ Mobile Agents with Aglets™", Addison-Wesley, Reading (MA), August 1998.
30. Lind, J.: Specifying Agent Interaction Protocols with Standard UML. Proceedings of the Second International Workshop on Agent Oriented Software Engineering (AOSE), Montreal (C), May 2001.

31. Lind, J.: Patterns in Agent-Oriented Software Engineering. Proceedings of the Third International Workshop on Agent Oriented Software Engineering (AOSE), Bologna (I), July 2002.
32. Mamei, M., Zambonelli, F.: Spray Computers: Frontiers of Self-Organization for Pervasive Computing. Proceedings of the Workshop dagli Oggetti agli Agenti, tendenze evolutive dei sistemi software (WOA), Cagliari, IT, September 2003.
33. Mobile Agent System Interoperability Facility, http://www.fokus.fraunhofer.de/research/cc/ecco/masif/
34. Odell, J., Van Dyke Parunak, H., Bauer, B.: Representing Agent Interaction Protocols in UML. Agent Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds., Springer-Verlag, Berlin, pp. 121–140, 2001.
35. Remote Method Invocation, http://java.sun.com/products/jdk/rmi/
36. Sandhu, R. S., Coyne, E. J., FeinStein, H. L., Youman, C. E., Role-based Access Control Models. IEEE Computer, Vol. 20, No. 2, pp. 38-47, 1996.
37. Steimann, F.: Role = Interface: a merger of concepts. Journal of Object-Oriented Programming Vol. 14, No. 4, pp. 23-32, 2001.
38. Ubayashi, N., Tamai, T.: RoleEP: role based evolutionary programming for cooperative mobile agent applications. Proceedings of the International Symposium on Principles of Software Evolution, Kanazawa, Japan, November 2000.
39. Weiser, M., Hot Topics: Ubiquitous Computing. IEEE Computer, Vol. 26, No. 10, October 1993.
40. White, J.: Mobile Agents. in Software Agents, J. Bradshaw (Ed.), AAAI Press, pp. 437-472, 1997.
41. Zambonelli, F., Jennings, N. R., Wooldridge, M. J.: Developing Multiagent Systems: the Gaia Methodology. ACM Transactions on Software Engineering and Methodology, Vol. 12, No. 3, July 2003.
42. Yu, L., Schmid, B.F.: A conceptual framework for agent-oriented and role-based workflow modeling. Proceedings of the 1st International Workshop on Agent-Oriented Information Systems, G. Wagner and E. Yu eds., Heidelberg, June 1999.
43. Zambonelli, F., Jennings, N. R., Omicini, A., Wooldridge, M. J.: Agent-Oriented Software Engineering for Internet Applications. in Coordination of Internet Agents, Springer, 2001.