Injecting roles in Java agents through runtime bytecode manipulation

G. Cabri
L. Ferrari
L. Leonardi

Agents are problem-solving entities that can be used to develop complex and distributed systems because they are autonomous, mobile, reactive, social, and proactive. Today's trends in agent technology include the development of applications as multi-agent systems, where several agents interact within the same application. In these systems, the interactions among agents must be carefully considered. Roles constitute a powerful paradigm for modeling interactions, allowing algorithmic issues and interaction-dependent issues to be handled independently. In this paper, we present the RoleX interaction infrastructure, which enables Java™ agents to dynamically assume and use roles at runtime. Our approach is based on using bytecode manipulation to add (or remove) Java members from agents, changing their capabilities. We detail the main component of RoleX, the Role Loader, which performs the bytecode manipulation that allows agents to dynamically assume and release roles.

Agents are autonomous entities that can perform their tasks without requiring continuous user interaction.[1] The agent-oriented paradigm is emerging as a sound approach for the development of today's complex software systems.[2] Because of their autonomy, agents can be exploited to build complex systems and applications where they perform actions on behalf of their users. Because they can run in a proactive way and react to environmental changes,[1] agents can naturally provide adaptability and deal with heterogeneity and unpredictability. Moreover, agents can be mobile; that is, they can search and run in different environments during their execution. This mobility makes it very important to take into consideration the interactions between an agent and its surrounding environment.

Agents can interact with other agents or environments in a cooperative or a competitive way. Multi-agent systems represent a powerful way to solve a distributed task, but their use requires agents to routinely use extensive social interactions in order to coordinate among themselves. For this reason,

research activity must take into account interactions among agents, and appropriate tools and paradigms must be developed in order to reduce development effort and to control interactions.

In this paper, we focus on interactions based on the concept of *role*, a stereotype of behaviors common to different agents.[3,4] Roles can be exploited by agents at runtime in order to enhance their capabilities. A role can be thought of as a set of behaviors and capabilities that agents can exploit to perform their tasks in a given context.

There are many advantages to modeling interactions by using roles and exploiting the resulting infrastructures. Firstly, this approach enables a separation between algorithmic and interaction-related issues in developing agent-based applications.[5] The algorithmic issues are addressed within the agents, and the interaction-related issues within the roles. As a consequence, roles can be developed by one developer and agents by another, which promotes software reuse. Secondly, roles permit the reuse of solutions and experiences, and, in fact, because roles are related to application scenarios, designers can exploit previously defined roles for similar applications or situations. As an example, Reference 4 shows how roles can be exploited to easily build agent-oriented interfaces for Internet sites. This implies that roles can be seen as design patterns:[6] a set of roles along with their interaction relationships can be considered as a solution to a well-defined problem and reused in similar situations. Finally, the use of roles promotes locality in interactions: each local interaction context can define its own roles, thus controlling the interactions among them.

Roles have been used in many branches of computer science. Role-Based Access Control[7] (RBAC) allows uncoupling of users and permissions. In Computer Supported Cooperative Work[8] (CSCW), roles support adaptability and the separation of duties. In the area of software development, roles are used in object-oriented programming[9,10] and in design patterns[11] such as the Role Object Pattern.[12]

Applied to the agent scenario, the exploitation of roles has a few limitations. Firstly, many approaches exploit roles only in the design phase, without taking into account the implementation phase. Supporting roles at the design phase only is inadequate for today's programming trends, and in particular for the development of agent-based applications. Powerful implementation support must be provided. Today's agent-based applications need to be very dynamic, adapting themselves to continuously changing environments. For example, adaptability is required in applications for e-commerce, general Internet applications, and those related to pervasive computing. Applications must be able to adapt easily to execution contexts without increasing the complexity of their development. In the context of agents and roles, this necessitates a dynamic way to assume and play roles at runtime, reducing the coupling between agents and their roles. Dynamic support for role assumption frees agents to exploit role capabilities on demand; that is, only when they are really required. As a consequence, agent code can be simple and light because capabilities that can be dynamically assumed through one or more roles need not be embedded.

In order to provide a powerful and, as much as possible, dynamic role system implementation, we have developed the RoleX (Role eXtension) infrastructure,[13] which is the subject of this paper. RoleX has been developed as part of the BRAIN (Behavioral Role Agent Interactions) project,[3,14] a project with the aim of supporting role-based development during different phases (analysis, design, and implementation). To achieve its goal, BRAIN proposes and provides a three-level framework: (1) a model of interaction based on roles, (2) a notation, XRole, based on XML (Extensible Markup Language), to describe the roles, and (3) several possible interaction infrastructures based on this model and notation, which enable agents to assume and play roles. BRAIN supports developers through the entire development process, making the use of roles a homogeneous and natural process. Furthermore, the fact that BRAIN provides different interaction infrastructures (such as RoleX) allows developers to easily migrate an application from one implementation to another.

This paper focuses on the RoleX infrastructure, which is a complex, innovative role-system implementation with highly dynamic and flexible services. The Role Loader[15] is a special component at the center of RoleX that is able to endow agents with role capabilities. RoleX is innovative in its approach because it not only provides for dynamic role assumption, but also because of the way that roles

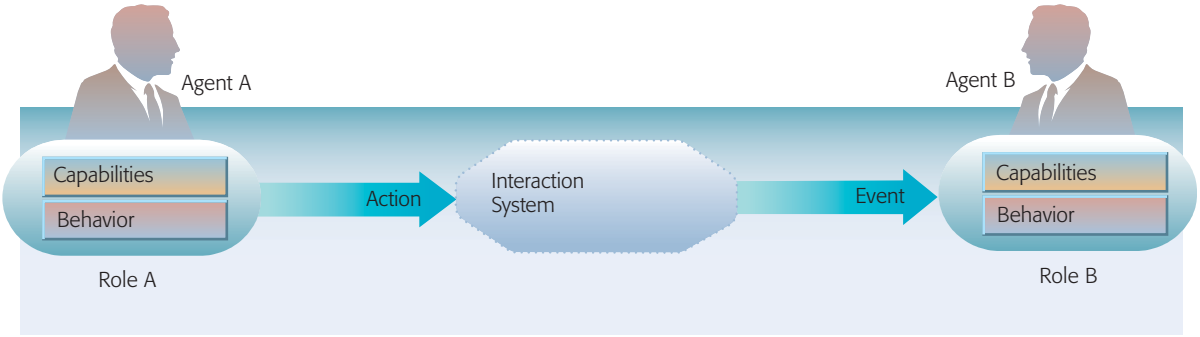

Figure 1
RoleX interaction model

are assumed, fusing the agent and the role in a single entity. In other words, the role can be said to be "injected" into the agent.

RoleX has been developed in Java**, and the Role Loader is expressly designed to work with Java agents. Java was chosen as the programming language for the following reasons: (1) Java is the most popular language for implementing agent platforms (including mobile ones) because of its portability, security, and network-oriented nature; (2) Java relies on an intermediate bytecode, allowing it to be modified at runtime to add new functionalities (without recompilation and with respect for the Java security constraints); and (3) Java is a well-known and widely exploited language. This should aid developers in using and understanding RoleX.

The RoleX implementation modifies the code of the Java agents at runtime, adding the features related to the roles they are going to assume and play. In addition, a descriptor-based mechanism to manage roles is exploited to further uncouple agents and roles and to help agent developers. RoleX allows agent developers to use a completely dynamic approach, granting a high degree of adaptability without requiring an extensive coding effort.

The rest of this paper is organized as follows. The second section presents the RoleX infrastructure, followed by a presentation of the RoleX behavior in the third section. This section also introduces Rolex's main component—the Role Loader—and details how role injection works. The fourth section details how RoleX faces particular conditions that can occur during role assumption and provides a code example. The fifth section compares our approach with others, followed by our conclusions.

THE ROLEX INFRASTRUCTURE

RoleX is a Java infrastructure that enables agents, either mobile or not, to exploit roles at runtime. RoleX enables agents playing roles to interact by means of actions and events. *Actions* implement the capabilities a role provides to agents, and *events* determine the expected behaviors derived from the action's execution. Events are delivered by the RoleX infrastructure to the addressee agent, which exhibits an expected behavior for managing the incoming event (see ***Figure 1***). This model of interaction is simple and very general and is well-suited to the main features of the agents: the actions can be seen as the concrete representation of the agent's proactive nature (i.e., the capability of carrying out its goals), while the events are the concrete representation of the agent's reactivity (i.e., the capability of reacting to environment changes).

The approach of RoleX to roles is inspired by real life, where a human playing a certain role in a given context (e.g., an employee at work) does not own but assumes the role and can release it to assume a new role (e.g., a tennis player in his or her free time). Because software agents can act on behalf of real users,[1] in our opinion their role model must be as similar to the human one as possible. From a software point of view, this necessitates two features: adaptability and external visibility. Instead of conceiving roles as entities separated from agents, as other approaches do,[16,17] our approach conceives of roles as first-class entities, which fuse with the agent that has assumed them by extending its code.

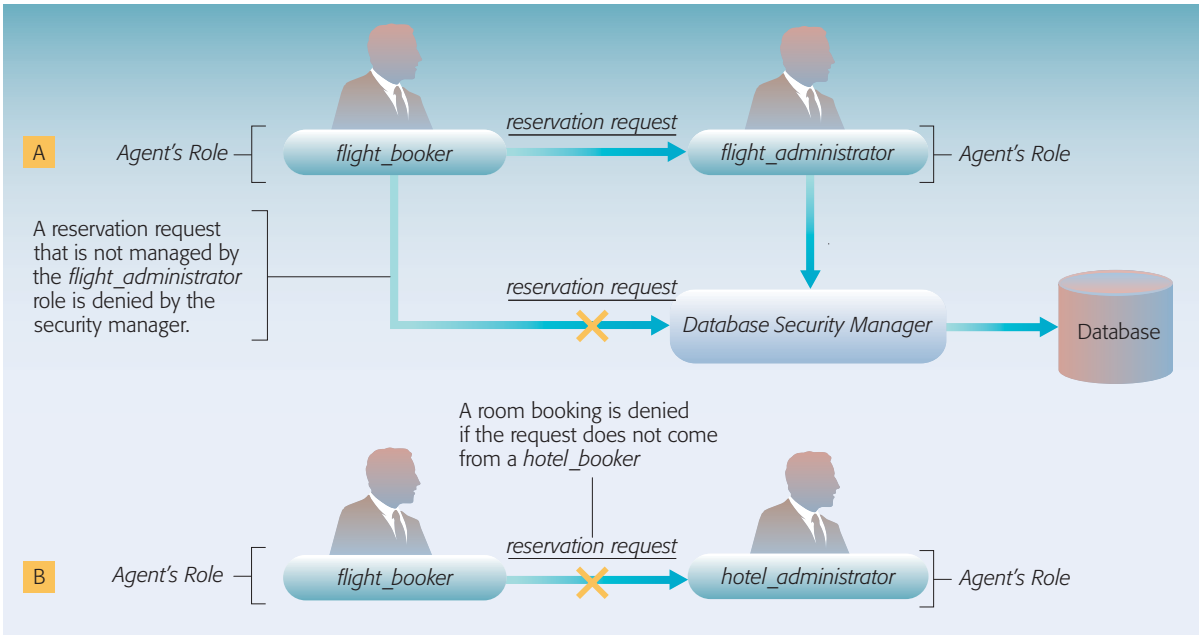

Figure 2
Possible security checks based on role visibility

This is a new way to conceive of roles, and one more similar to real life.

Recalling the example of the person playing the role of employee, we note that the role is incorporated. In fact, the person does not use an external entity to perform any employee action, but simply performs it because all employee capabilities are part of the person's behavior. This leads to another aspect: in real life persons can be recognized by their role. For example, persons can be recognized as employees because they display employee capabilities. Thus, in the real world, a role grants an intrinsic set of capabilities and behaviors as well as giving the behavior an external visibility.

Adaptability is required, as when, at the end of the work day, the person releases the employee role and assumes a new role, for example that of a tennis player. The person assumes and releases roles depending on what he or she wants to do. The same must happen in the agents' world: the agents must be free to assume and release roles in a dynamic way.

To give roles external visibility, RoleX uses Java interfaces: if an agent class is forced to implement a particular interface, all the casting operators (such as instanceof) will recognize the interface, thus making the role externally visible. Nevertheless, Java interfaces cannot define a behavior, which means they cannot include method definitions or mutable variables. To this end, RoleX uses Java classes to implement role behavior.

Defining a role as a few classes and interfaces is not sufficient for an infrastructure that requires adaptability and external visibility. The agent must be forced to implement the role interface when it assumes the role and to discard the interface when it releases the role. To achieve this, RoleX performs a manipulation of the agent class, with the aim of obtaining a new agent class extended by the addition of the role and the appropriate interface. In other words, the agent's basic structure (i.e. the bytecode) is changed at runtime without any source-code alteration or decompiling/recompiling sequences, and a new extended agent is created. The changes made to the agent bytecode add all the role class members and force the agent class to implement the role interface; this manipulation is called the extension process because the role's features extend those of the agent.

To better explain why the use of roles and their characteristics can be useful for agent applications, consider the following example. We use an application inspired by the TabiCan[18] application, in

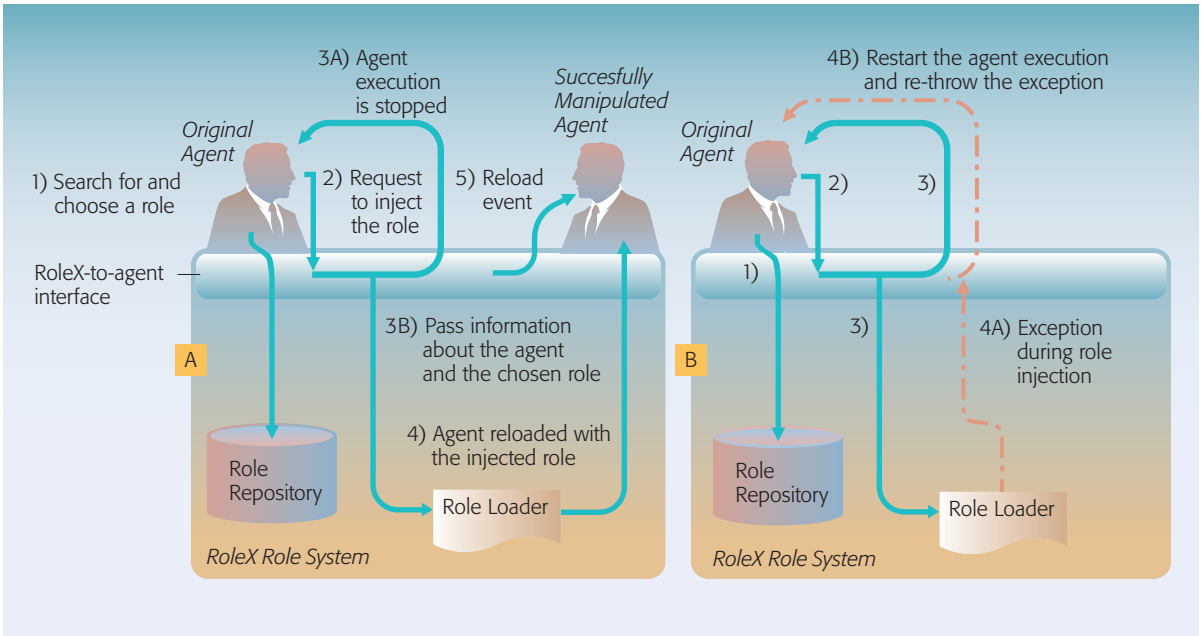

Figure 3
Role system behavior

which agents are in charge of reserving flights and hotel rooms for their users. In such an application, we can provide different roles to model interactions. In particular, the flight_booker role is in charge of managing flight reservations, the flight_administrator role represents the software administrator of a flight booking system, the hotel_booker role is in charge of reserving rooms and, finally, the hotel_administrator role is in charge of accepting room reservations. In this scenario, an agent sent by a user (called the "user agent" in the following) is in charge of exploiting the flight_booker and hotel_booker roles to reserve flights and rooms, and interacting with agents playing the flight_administrator and hotel_administrator roles. Using roles, the user agent can quickly adapt to different scenarios, without embedding all interaction details (i.e., interaction protocols, message schemas, etc.) in itself. Furthermore, because roles are tied to the local interaction contexts and are developed with regard to each other,[19] each "booker" role knows how to interact with the corresponding administrator role in order to make a reservation, so that the user agent is simply in charge of exploiting the capabilities of the role and providing required data (e.g., the user credit card number).

Adaptability allows user agents to change their role, enabling a flexible situation. For example, they can assume the hotel_booker role right after having reserved a flight, or they can reassume the hotel_booker role to cancel a reservation if a flight has been canceled. External visibility allows entities running in the system (either agents or environments) to quickly and definitively recognize an agent by its role, denying or allowing certain operations depending on it. For example, as shown in ***Figure 2A***, a flight database security manager (not strictly related to the Java Security Manager) can prevent an agent from interacting directly with the flight database if it is not playing the flight_administrator role. Similarly, a hotel_administrator role can deny the request for a reservation that comes from an agent that does not play the hotel_booker role because such an agent might be malicious (see ***Figure 2B***).

ROLEX AT WORK

This section describes RoleX internal details, showing how RoleX works and how agents can exploit it to assume, play, and release roles. In short, RoleX works as follows (these steps correspond to the numbers in ***Figure 3***):

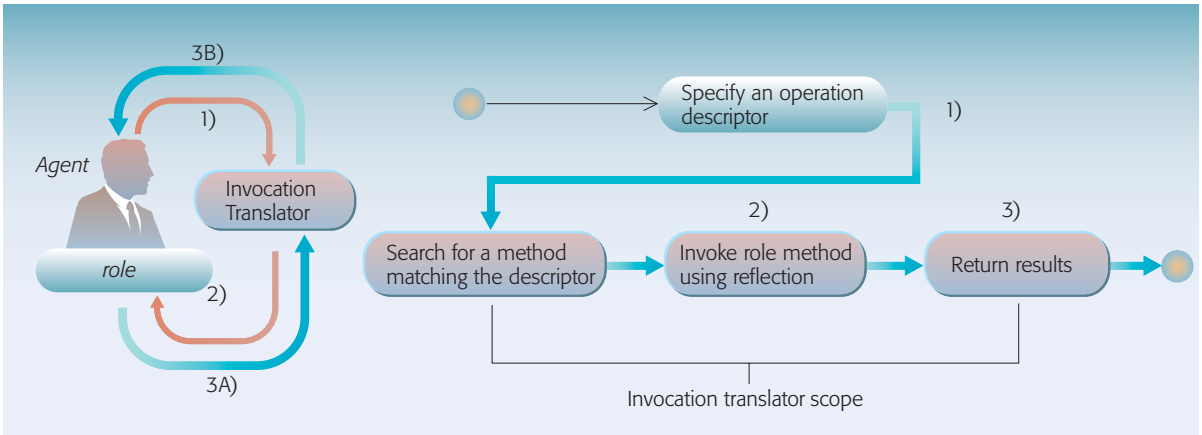

Figure 4
Use of the invocation translator

1. The agent searches for one (or more) roles in a role repository, a container of available roles.
2. When the agent has chosen a role, it contacts the role system and asks to inject the role into itself.
3. The role system transparently stops the execution of the agent (see Figure 3, step 3A), then passes the information about the chosen role and the agent requesting it to the Role Loader (see Figure 3, step 3B).
4. The Role Loader starts a sequence of internal operations to inject the role into the agent, performing the extension process. If, for some reason, the role cannot be injected (see Figure 3, step 4A), the Role Loader notifies the agent about the error by throwing an exception, and then the agent execution restarts transparently (see Figure 3, step 4B). If restarting is not possible, it reloads the agent after having manipulated it.
5. When role injection is successful, the role system sends a reload event to the reloaded agent to indicate that it can restart its execution and that it can now exploit the added role functionalities.

Before describing how the Role Loader performs the bytecode manipulation required to inject the role into the agent, it is worth discussing how the agent can use the assumed role. In fact, after a successful role injection, the agent has new Java members (both variables and methods). The question is, "How can it exploit them?" Of course, it is not possible to use them by direct reference (using the standard Java dot notation) because at compilation the agent does not know the role members and the compiler cannot resolve unknown symbols. This problem can be solved by using *reflection*, which is the capability to analyze, at runtime, the structure of a class by accessing its methods and variables,[20] but reflection complicates the agent's logic by requiring a way to search for and access new members. Instead, we provide support for quick and simple *introspection*, based on the compound use of operation descriptors and an *invocation translator*. Operation descriptors represent single role operations through a "meta" level of information. In other words, an operation descriptor describes how an operation can be used and what results it will produce. An example of a simple operation descriptor is, "set the value of variable X," which means that the execution of this operation will set the role variable X to a given value. Operation descriptors have a powerful structure that includes information related to permissions, parameters, return values, and so forth, and will be treated more in detail in the next section. For now, it is sufficient to know that operation descriptors give an agent the information about the operations provided by the injected role to an agent, and what they do and mean. When an agent searches for a role and asks to inject the role into itself, it gets the set of operation descriptors of the role itself and can keep them to use the second support component that we provide, the invocation translator.

The invocation translator, a component embedded in the agent, is in charge of executing (through reflection) a role operation by using its descriptor. Imagine, for example, that an agent wants to invoke a role-added method. As shown in ***Figure 4***, in step

1 the agent specifies a role operation descriptor to its invocation translator; in step 2 the latter executes the operation by invoking the corresponding method among those derived from the injected role. The role method returns results to its caller, the invocation translator (step 3A). The translator passes them to the agent itself (step 3B). Similarly, if the execution of a role operation raises an exception, the invocation translator propagates it to the agent. In this way the agent has access to all injected members and can rapidly use them. Providing the invocation translator has the advantage that agent developers are not in charge of doing introspection directly, and thus the resulting agent code is simpler and clearer. It is worth noting that because the invocation translator is embedded in the agent, step 1 of Figure 4 corresponds to the invocation of a method in the agent itself.

From operation descriptors to role descriptors

In addition to providing reflection for injected roles, operation descriptors introduce a high degree of abstraction for operations because they are accessed and executed by semantic information and not by syntactic data. For this reason, we have embedded in operation descriptors *event descriptors* as well, the latter being descriptors containing information related to sending and receiving events (if any) resulting from the execution of an operation. Event descriptors let the agents understand the consequences of the role assumption and, in particular, the execution of an operation. In fact, while the invocation translator can only return the return value of an executed operation, event descriptors can make an agent aware of the events that it is sending to other agents and what their reactions may be (i.e., which events can come from other agents). This characteristic is important for a social scenario because it lets an agent know the reactions that should happen after the execution of a role operation and how it is influencing other agents.

To grant flexibility and modularity, we have introduced a third component, the *role descriptor*, which embeds the other two descriptors. Each kind of descriptor has been implemented as a separate Java class, as shown in ***Figure 5***. Because RoleX exploits the BRAIN XML-based notation called Xrole,[3] all three kinds of descriptor are written using the XRole notation. As an example of a role descriptor, ***Figure 6*** shows a fragment of the code for the flight_booker role.

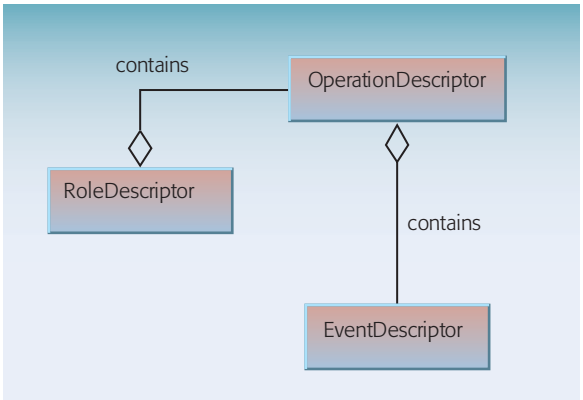

Figure 5
Relationships among the descriptor classes

To make descriptors available to agents in a quick and easy way, the RoleX infrastructure automatically provides a Java version of all installed descriptors generated from the XML document. XML descriptors are, in fact, translated into a set of Java objects, instances of the classes shown in Figure 5. In this way, an agent can directly access the descriptors without needing an XML parser, thus keeping the agent code simpler and smaller. Following the example introduced in the section "RoleX infrastructure," Figure 6 shows a single operation descriptor for the flight_booker role. This descriptor provides information about a single operation, called Book, which is performed through the method book_flight, which allows the agent playing the role to buy a ticket. A Java prototype of such an operation, based on the XML descriptor, is the following:

```
boolean book_flight(String creditCardNumber,
                    Calendar when);
```

where the parameter *creditCardNumber* is the number of the user's credit card for payment, and *when* is the date of the sought flight. The above operation descriptor also contains an incoming event, called *DeletedEvent*, which represents the notice of a flight cancelation. After the agent has executed the book_flight method, the *DeletedEvent* event may be received to indicate that the booked flight has been canceled.

The Java classes used by RoleX to translate an XML descriptor provide several methods in order to allow agents to access the descriptor properties. As an

```
<?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>Flight Booker Role</description>
            <roleName>flight_booker</roleName>
            <keyword>flight reservation</keyword>
            <keyword>travel airplane</keyword>
            . . .
            <version>1</version>
            <OperationDescription>
                  <name>Book</name>
                  <aim>book a flight</aim>
                  <keyword>flight</keyword>
                  <version>1.0</version>
                  <methodName>book_flight</methodName>
                  <returnType>
                        <className>java.lang.Boolean</className>
                  </returnType>
                  <parameter>
                        <className>java.lang.String</className>
                  </parameter>
                  <parameter>
                        <className>java.util.Calendar</className>
                  </parameter>
            </OperationDescription>
            <EventDescriptor>
                  <name>DeletedEvent</name>
                  <aim>Inform that the reserved flight has been deleted</aim>
                  <className>examples.tabican.DeleteEvent</className>
                  . . .
                   <ReceivingEvent>true</ReceivingEvent>
            </EventDescriptor>
           . . .
      </GenericRoleDescription>
</role>
```

Figure 6
Partial code for XML descriptor for *flight_booker* role

example, ***Figure 7*** shows a partial list of methods of the OperationDescriptor class, instances of which are used to encapsulate operation descriptors. All of the methods shown in the figure can be used to get information about the operation described by the descriptor, such as its keywords, goal, and so forth, in order to evaluate the operation by use of its semantic data. Furthermore, each object of type OperationDescriptor contains information about Java methods to invoke on the corresponding role in order to execute the operation. Such information can be used by the invocation translator in order to perform the operation requested. The other two kinds of descriptors provide similar methods to agents.

It should be noted that it is possible to directly write Java descriptors, creating instances of the classes of Figure 5, though this solution is deprecated because it overtakes the middle layer of BRAIN, leading to a possible poor reuse of roles in different scenarios.

Injecting roles into agents

In a role assumption the Role Loader adds each role class member (both methods and variables) to the agent class in order to add the role's set of capabilities. At the same time, it forces the agent class to implement the role interface in order to modify its appearance and to allow other agents to recognize it as playing that role. This mechanism

```
public class OperationDescriptor implements java.io.Serializable{
        public String getName();
        public Class getReturnType();
        public String getReturnTypeAsString();
        public Class[] getParamsType();
        public String[] getParamsTypeAsString();
        public double getVersion();
        public String getAim();
        public String[] getKeywords();
        public boolean matchKeyword(String keyword);
        public int matchKeywords(String[] keywords);
        public Enumeration getPermissionEnumeration();
        public String[] getPermissionsAsString();
        public Calendar getCreationDate();
        public boolean equals(Object descriptor);
        . . .
}
```

Figure 7
A partial list of services provided by the Java class OperationDescriptor

results in the definition of a new agent class. The Role Loader (which is implemented through the class RoleLoader) is simply a special class loader that can change agent behavior and external appearance. After the Role Loader has successfully carried out the role assumption process, it reloads the agent, so that the latter can restart its execution (see Figure 3A).

Releasing a role is similar to the above process, but in this case the Role Loader removes each role member and the role interface, reloading the agent without them.

To perform role assumption or release, the Role Loader uses runtime bytecode manipulation; this manipulation is done completely in memory without needing the source code of the agent or role and without requiring a recompilation. The bytecode alteration needs to work with and to modify class definitions. In fact, to obtain an agent extended with a role, we need to create a new agent class, manipulated with respect to the original one, from which a new agent instance is obtained. Our implementation of the class RoleLoader, which extends SecureClassLoader, is based on the Javassist bytecode manipulation engine,[21] though the simple use of this engine alone is not enough to completely achieve our goals. In fact, our approach takes into consideration code reusability and separation of concerns. For this reason, the assumption mechanism is performed through several steps performed by an instance of RoleLoader:

1. The inheritance stack for the role class and the agent class is defined (i.e., all superclasses of both classes are calculated);
2. For each level of the inheritance stack, all of the members (both methods and variables) are copied from the role class to the agent class; the role interface is then added to the interface list of the manipulated agent class;
3. A new agent instance is created from the obtained manipulated class;
4. The original agent state values are copied to the new agent.

Figure 8 shows the main operations performed by the Role Loader during each step. After the above steps, as mentioned, the RoleX infrastructure starts the execution of the new agent, similar to the restarting of the agent's execution. This operation can be considered an agent execution restart because the new agent has the entire state (i.e., variable values) of the original agent. Due to the work done by the Role Loader, even if an agent and its assumed role have been developed separately, they dynamically become a single entity with the correct external visibility. Each step is detailed next.

Step 1: Calculating the inheritance stack

The first step performed by the Role Loader is needed to grant role-inherited properties to the

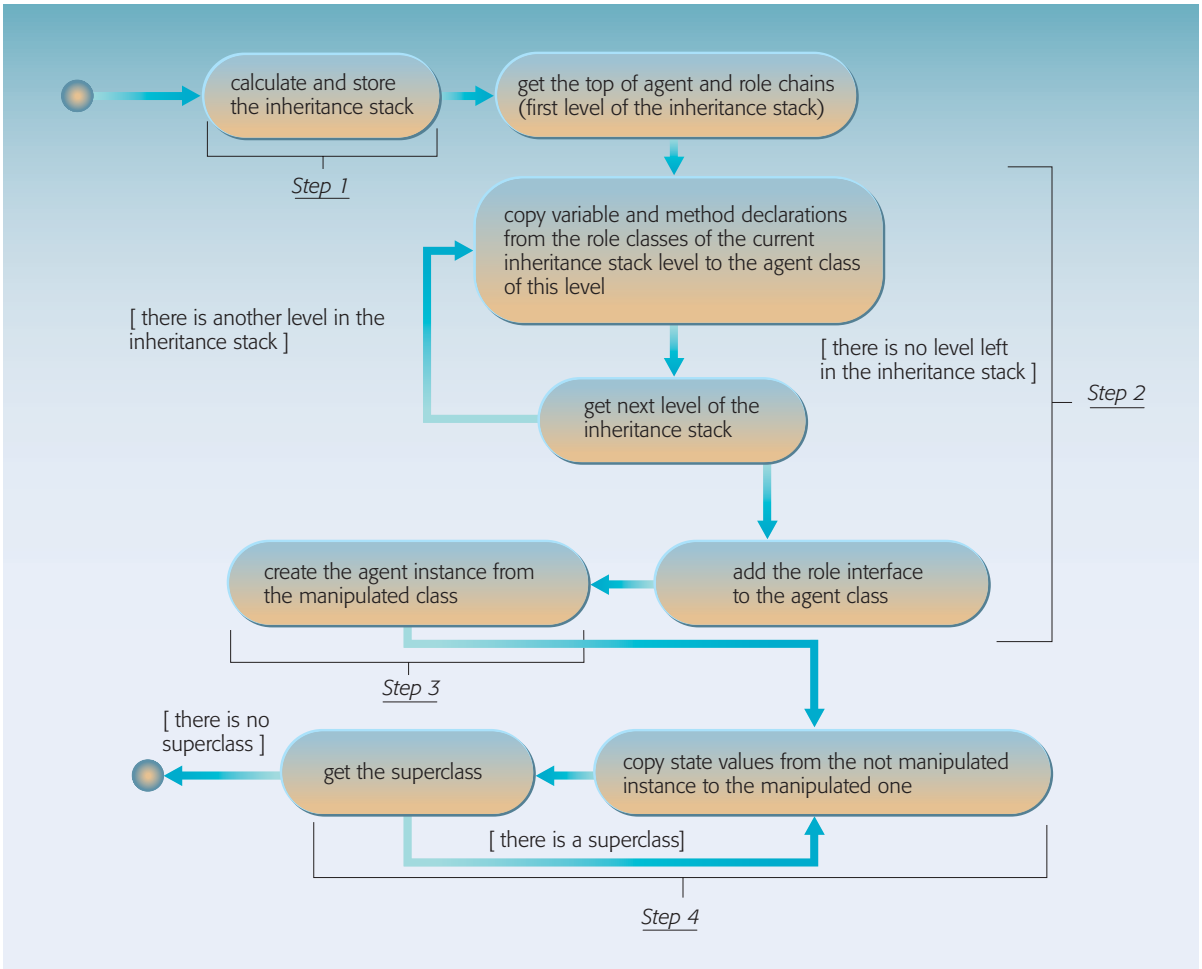

Figure 8
Operations performed by the Role Loader

agent, promoting role code reusability. In fact, the role implementation may be the bottom of an inheritance chain rather than a single class. For example, the flight_booker role could inherit properties from a generic booker role. To ensure that the role will work in the right way, every role superclass (i.e., every class at any level in the role inheritance chain) must be added to the agent superclasses at the corresponding level. In fact, because a role implementation class or subclass expects to find some capabilities in its superclasses, it must be ensured this condition will remain true.

Figure 9 clarifies, by means of a class diagram, the inheritance chains of both an agent and a role that the agent is going to assume. Both the role and the agent classes are represented by the bottom of their respective chains, and this means that the bottom classes must be joined. The superclasses must be joined as well. This must be done for each chain level. In this way, our infrastructure ensures that both the role and the agent, after the extension process, will continue using inherited properties; in other words, Java's "super" operator will work correctly. This step does not do anything but calculate the inheritance stack, which specifies how a role class and an agent class must be joined and at what level. The computed inheritance stack for the example of Figure 9 is shown in *Table 1*. Every row in the stack indicates which classes will be joined, and will be used in the second step to determine from which class the members will be copied in the agent chain and in the fourth step to determine which member values must be copied. Please note that in Table 1, the root object java.lang.Object has

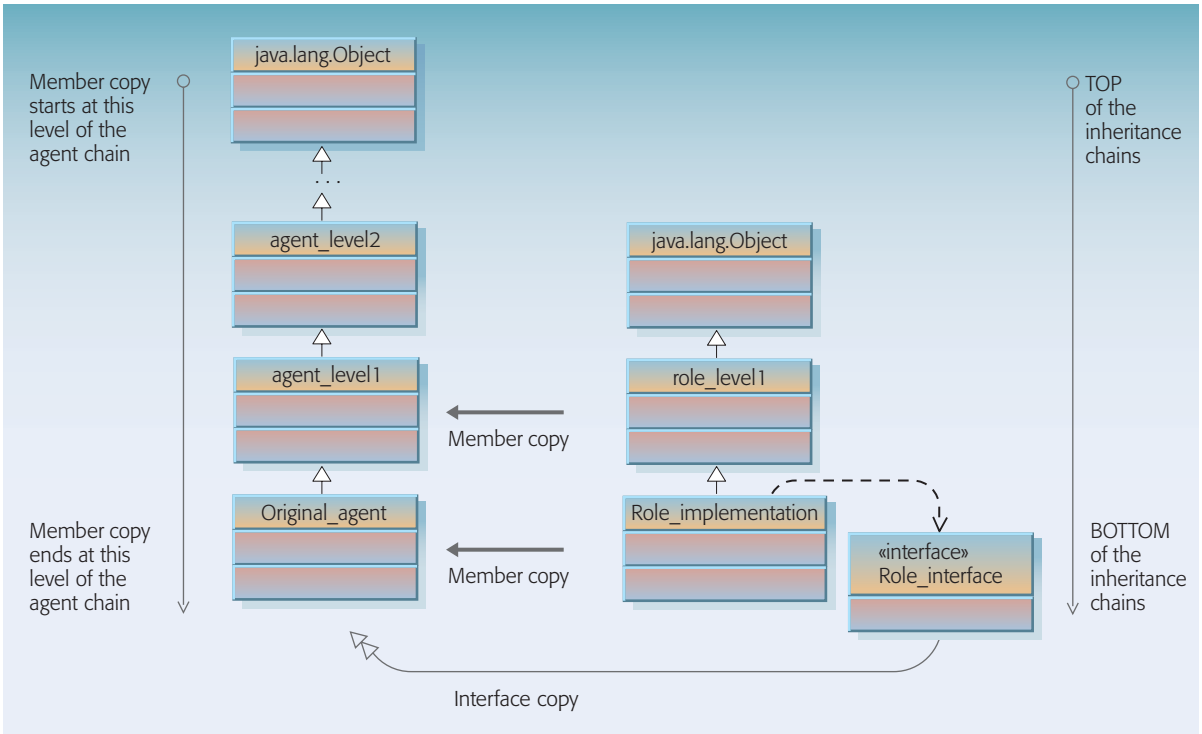

Figure 9
Member copy at all inheritance levels

been introduced only to make the inheritance stack more readable.

Nevertheless, as shown in ***Figure 10***, the java.lang.Object class is never inserted because each Java class inherits, if no other class is specified, from the class Object. The inheritance stack is calculated through the method computeInheritanceStack( ) of the class RoleLoader, as shown in Figure 10. As shown in that figure, the method accepts two parameters, both of type Class: the agent class and an array of role classes to which the agent is to be joined. The use of an array allows the Role Loader to inject multiple roles into the same agent at the same time, improving performance. In the simplest case, that is, a single role assumption, the array of role classes is a one-element array.

Table 1 The inheritance stack calculated by the Role Loader.

Agent's Chain	Role's Chain
java.lang.Object	None
. . .	None
agent_level2	None
agent_level1	role_level1
Original_agent	Role_implementation

The method computeInheritanceStack( ) exploits another internal service of the Role Loader: the method addToStack( ). The latter simply adds the class names specified as parameters to the variable roleStack that is an internal stack object, which represents the inheritance stack the RoleLoader class will use in further steps. Another service exploited by the code in Figure 10 is getRoleNamesFromClass( ), a method that returns a string array with the names of the class or array of classes passed as a parameter.

The method computeInheritanceStack( ) first pushes the first level of the chain, the agent and the role bottom classes, onto the inheritance stack. After that, computeInheritanceStack( ) iterates among all superclass levels. To do this, agentName and role[i] are assigned to the corresponding superclasses, or to null if there are no superclasses or the java.lang.Object is reached. The boolean variable stop, which drives

```
protected final void computeInheritanceStack(Class agent,Class[] roles){

        /* bottom level */
        this.addToStack(agent.getName(),this.getRoleNamesFromClass(roles));

        String agentName=null;
        boolean stop;

        /* iterate other levels */
        do{
              stop=true;
              /* get the agent name for this level */
              if(agent!=null && (agent=(agent.getSuperclass()))!=null){
                agentName=agent.getName();

              /* skip java.lang.Object */
              if(agentName.equals("java.lang.Object")){ agentName=null; }
              }else{ agentName=null; }

              for(int i=0;i<roles.length;i++){
                if(roles[i]!=null &&
                (roles[i]=(roles[i].getSuperclass()))!=null){
                /* skip java.lang.Object */
                if(roles[i].getName().equals("java.lang.Object")){
                roles[i]=null; }

                /* do not stop if at least one role is not null */
                if(roles[i]!=null){ stop=false; }
               }
              }

              /* stop if the agent and roles are both null */
              boolean end=true;

                 if(agentName==null){
                   if(roles!=null){
                     for(int i=0;i<roles.length;i++){
                            if(roles[i]!=null){ end=false; break;}
                      }
                   }
              }else { end=false; }

            if(end==false){
              /* add to the stack this level */
              this.addToStack(agentName,this.getRoleNamesFromClass(roles));
            }
          }
          while(stop==false || agentName!=null);
      }
```

Figure 10
Code for *computeInheritanceStack()* method

the do..while loop is set to false only if no role class has been added to the stack. Finally, the method checks if the agent and the role classes are both null (for example because they were equal to java.lang.Object), pushing them into the stack if they are not.

The inheritance stack is kept as an internal RoleLoader object, of type java.util.Stack. In this stack the Role Loader stores (as a string array) the names of the classes that must be joined at the same level. The use of a stack allows the Role Loader to start introspection from the bottom of the inheritance

chain of the agent and the role (or roles) and to go up, pushing each level onto the stack and retrieving them in the required reversed order in further steps. Why does the Role Loader need to retrieve classes in a reversed order? This requirement stems from the presence of a class loader cache: because the Role Loader is a particular kind of Java class loader, it has, like all other class loaders, a private class cache that stores classes which it has already loaded, in order to speed up the redefinition of those classes, thus reducing the response time. Though this cache is an important component of each class loader, in our approach it can lead to problems if not treated appropriately. In fact, this cache is a private member of the class java.lang.ClassLoader, and this means that a subclass (such as RoleLoader) cannot access it directly. In other words, a user-defined class loader cannot directly remove or add any entry in the cache because this is done transparently by the base class ClassLoader. Because each Java class loader has a unique namespace, each class is uniquely identified by the class loader into its namespace, and it is not possible to define two classes with the same name. This implies that the Role Loader must load the new agent class starting from the top of the chain. The member copy is done starting from the base classes, going down the inheritance chain until the last agent class is reached, as shown in Figure 9. In fact, when the Original_agent class is loaded, all its base classes should be found in the loader cache, because a class-loading definition process is done from the first base class to the last one.

To better understand this problem, let us suppose, for instance, that the copy is made from the bottom of the inheritance chain to the top: the first class manipulated is Original_agent, which is joined with Role_implementation. In this case, when the class Original_agent is loaded by the Role Loader (before the manipulation process is started), it should be linked with its superclass, agent_level1; this class should then be linked with its superclass, agent_level2, and so on until the java.lang.Object class is reached. As described in the Java Language and Virtual Machine specifications,[22] every loaded class must be put into the class loader cache, so after the manipulation of the Original_agent class, at the loading time, the new class will be linked, by the Role Loader, to the class agent_level1 already present in the cache. Because the loader already has the class in its cache, it does not reload it, and the manipulated agent class is linked with a superclass which has not been manipulated. Because the loader cache is untouchable from a subclass and each Java class loader has a single namespace, a class modification that starts from the bottom, going up to the top, produces a LinkageError exception. This is because the class namespace of the Role Loader has two (or more) classes with the same name but different bytecode definitions. By starting from the top of the chain instead, the class loader cache is filled with manipulated classes that act as base classes for the next level; we call this mechanism *reverse class loading*.

To explain these concepts, consider what happens, step by step, during a role assumption like the one shown in Figure 9. In such a situation, when the Original_agent class is loaded, our Role Loader tries to link that class with its base class, agent_level1, searching its cache for it. If the latter class has been loaded and manipulated before the one in the current level, which means it is already in the cache, the link is correctly resolved. Otherwise, if the base class is not in the cache, the class loader must load it (for example from a URL), and then manipulate it. But the manipulation, if made after a linkage operation, causes an error because two classes with the same name (agent_level1) but with different definitions would be present in the cache. Therefore, to allow the manipulation process, the classes must be loaded and manipulated in a separate way, without the dynamic linking provided by the Java language. Only when a class has been manipulated can it be used as a valid linkable base class.

Step 2: Copying members' declarations and adding the role interface

This step performs the member declaration copy by consulting the inheritance stack and then copying every member declaration from the role chain to the agent chain in the classes of the same level. This step uses bytecode manipulation that allows the system to modify the class definition. Note that no members are removed from the Original_agent class. In our implementation, to ensure a correct execution of the agent, in every situation only member adding occurs.

In this step, bytecode manipulation is also used to force the agent class to implement the role interface. Because every class contains a list of implemented interfaces,[22] this is done simply by adding the role interface to that list in the manipulated agent class. The member declaration copy is performed through

```
protected final CtClass copyMembers(CtClass src,CtClass dest)
        throws CannotCompileException,NotFoundException

    /* check params */
        if(src==null || dest==null){
                return dest;
        }

    /* check if the class has already been copied at another level */
        if(this.isAlreadyCopied(src.getName())){ return dest; }

    /* add all the methods from src to dest */
        CtMethod toAdd[]=src.getDeclaredMethods();
        CtMethod copy=null;

    /* add all the methods */
         for(int i=0;toAdd!=null && i<toAdd.length;i++){
                // does the dest class already have the method?
                if(!this.hasThisMethod(dest,toAdd[i]) &&
                   !Modifier.isFinal(toAdd[i].getModifiers())) &&
                   (!Modifier.isStatic(toAdd[i].getModifiers())) ){
                   /* add it */
                   copy=CtNewMethod.copy(toAdd[i],dest,null);
                   dest.addMethod(copy);
                }else
                if((!Modifier.isFinal(toAdd[i].getModifiers())) &&
                   (!Modifier.isStatic(toAdd[i].getModifiers())) &&
                   (!this.isObjectMethod(toAdd[i]))){
                   /* generate a warning */
                addWarning(LoaderWarnings.duplicatedMethod,dest,toAdd[i],toAdd[i].getName(),src);
                }
           }

    /* now add all the variables */
        CtField fields[]=src.getDeclaredFields();
        CtField addingField=null;

    for(int i=0;fields!=null && i<fields.length;i++){
        /* is the variable already present in the dest class? */
        if(!this.hasThisMember(dest,fields[i]) ){
                addingField=new CtField(fields[i].getType(),fields[i].getName(),dest);
                addingField.setModifiers(fields[i].getModifiers());
                dest.addField(addingField);
        }
        else{
        /* generate a warning */
        addWarning(LoaderWarnings.duplicatedVariableName,dest,fields[i],fields[i].getName(),src);
        }
    }

    /* store the class name, thus the role loader will not re-copy existing classes */
        this.alreadyCopied(src.getName());

    return dest;
}
```

Figure 11
Code for the *copyMembers()* method

```
protected final Class generateAgentClass()
throws NotFoundException,CannotCompileException,IOException {
        . . .
                byte code[]=agentPool.write(agentName);
                return this.defineClass(agentName,code,0,code.length);
}
```

Figure 12
Creating the manipulated agent class

the method copyMembers( ) of the class RoleLoader (see ***Figure 11***), a method that exploits the Javassist[23] capabilities to add variables and methods to a class (in this case, the agent class). Because the copy of the declared members is the most important operation done by the Role Loader, the method exploits several ad hoc services to check if a member is already present. These services are not detailed here, but it is important to stress that several checks are required during the member declaration copy. The copyMembers( ) method returns a CtClass object, which is a Javassist object that represents a compile-time class. Similarly, the method accepts two CtClass parameters corresponding to the original agent class and to the role class. Compile-time classes allow developers to interact with the bytecode of the corresponding class by adding methods and variables, changing the access attributes, and so forth. Moreover, the method exploits other Javassist objects, namely CtField, CtMethod and CtNewMethod, to interact with the class structure.

The code of Figure 11 first copies each method declaration. This is done to add role capabilities, but only if there is not already a method with the same signature in the destination class. If the method is already present, a warning is generated to notify the developer of methods that were not copied. After the method declarations are copied, copyMembers( ) copies the variable declarations. It is important to note that variables are always copied, even if the same variable exists in the destination class, which results in the generation of a warning.

While the warnings generated in the case of duplicated methods report a possible error condition during the injection of the role, warnings generated in the case of duplicated variables are used only as information about name conflicts between the original agent and the role classes. More details about duplicated members and warnings can be found in the section "Duplicated members."

The RoleLoader class provides methods to convert CtClass objects into Class objects and vice versa, in order to change from the Javassist model to the Java reflection and back. At the end of this step, the RoleLoader instance loads the current manipulated class, in order to put it into the cache and to make it available for use by subclasses. This operation is performed by exploiting the class loader capabilities that the Role Loader has inherited from the SecureClassLoader. After all classes have been manipulated and loaded, the RoleLoader instance can proceed with the next step, which is the creation of the manipulated agent object.

Step 3: Creating a new agent object

After the previous two steps, RoleX makes available a new agent class, to which the role has been joined. To obtain a new agent, RoleX must create a new instance of the manipulated agent class. This is performed in step 3, which creates an agent instance from the manipulated class, which is linked (directly or indirectly) to all the manipulated superclasses.

The code fragment of ***Figure 12*** shows how the Role Loader exploits the method defineClass( ), inherited from the base class ClassLoader, to define a new manipulated agent class. The agentPool variable is a Javassist ClassPool, used to create the bytecode from CtClass objects. After the call to the method shown in Figure 12, the Role Loader uses the Class capabilities to create a new object (through the method newInstance( )), thus creating the new agent instance.

It is important to note that, as shown in Figure 12, the Role Loader does not exploit the delegation model. Introduced from the Java API (application programming interface) Version 1.2, the delegation model requires a class loader to ask its parent class loader to define a class before it can do so on its own. This leads to efficiency and code portability because a class is defined by only one class loader. For example, system classes, like those in the java.lang package, are defined only by the first class loader. However, as specified earlier, the Role Loader cannot use the delegation model because if it did, different class definitions would be present at runtime, producing namespace clashes. If the Role Loader were to ask its parent to load and define an agent class, this would lead to the definition of an unmodified agent class because the parent class loader has not performed bytecode manipulation on it.

As mentioned earlier, after a successful agent class manipulation, a new instance of the latter is created. This could lead to a situation where old references to the original agent, no longer existing, are not updated to become references to the new agent. To avoid this, RoleX adopts a protection mechanism based on the concept of proxy,[24] which masks the agent itself, thus avoiding dangling references. Of course, only the role infrastructure itself can own a direct agent reference, which is used to substitute original agents with manipulated ones. It is important to note that, as shown in Figure 3, during the manipulation, the agent execution is halted, and thus unable to process incoming events from other agents. For this reason, our proxies store incoming events in a queue, flushing them when the execution of the manipulated agent restarts. Due to the use of these proxies, all other running entities will never own a dangling reference to an agent and will not perceive any difference between agent references before and after a role assumption or release.

Step 4: Copying members' values

In the last step, every variable value is copied from the original agent to the newly created agent. This step ensures that the agent's original state will not be lost during the extension process. The copy is done by iterating for each agent inheritance level and executing the copyMemberValues( ) method of the RoleLoader class (see ***Figure 13***), which accepts the source object (the not manipulated agent), its class type, the destination object (the manipulated agent) and its class type. The method iterates over all variables in the source class, accessing them even if they are private, and copying their values to the destination object. To access private members, the Role Loader requires the permission object java.reflect.ReflectPermission. Granting this permission does not represent a real risk because Role Loaders cannot be created directly by agents, being prevented from doing so by the role infrastructure. This prevents cases where agents maliciously use Role Loaders to access or manipulate other agents. This also means that the administrator is in charge of granting the use of trusted RoleLoader instances.

PARTICULAR CASES AND CODE EXAMPLE

This section extends the discussion of RoleX by taking into account particular conditions that can happen during the injection of the role into the agent. A code example is presented to help readers understand the practical use of the RoleX infrastructure and its components.

Particular cases

As shown in Figure 3, there are conditions where role injection cannot be successfully performed, and an exception is thrown. This could happen when an agent tries to violate rules imposed by the system administrator. For example, two roles can be marked by the administrator as incompatible, meaning that the agent cannot assume both at the same time. Moreover, an agent may not own the required rights to assume a specific role. In these cases, bytecode manipulation does not start, and an exception is thrown to the agent.

Another exceptional case is when the Role Loader cannot run due to various conditions (e.g., JVM** cannot instantiate it, the Role Loader cannot find the role repository, etc.). Because role loading is a very important operation for the agent, the Role Loader performs an autotest before starting the manipulation in order to understand if it can run or not. During the autotest, the Role Loader also tries to load itself before starting the manipulation, as shown in ***Figure 14***.

Aside from the preceding particular cases, which are related to the role-infrastructure runtime environment, there are other particular cases related instead to the role and agent implementations. In these cases, the manipulation can be successfully done, but in a different manner from that detailed above.

```
protected final boolean copyMemberValues(Object src,Class srcClass,
            Object dest,Class destClass)
            throws IllegalArgumentException,IllegalAccessException {
      // parameters check
      . . .

      /* get the variables */
      Field[] srcFields,destFields;
      srcFields=srcClass.getDeclaredFields();
      destFields=destClass.getDeclaredFields();

      if(destFields==null || srcFields==null){ return false; }

      /* used to store the original access type of the fields */
      boolean originalAccessType=true;

      for(int i=0;i<srcFields.length;i++) {
            for(int j=0;j<destFields.length;j++){
              if(srcFields[i].getName().equals(destFields[j].getName())){
                  /* store the access type */
                  originalAccessType=destFields[j].isAccessible();
                  /* make the dest field accesible */
                  destFields[j].setAccessible(true);
                  /* make accessible the src variable */
                  srcFields[i].setAccessible(true);
                  /* copy the variable */
                  if(!Modifier.isFinal(destFields[j].getModifiers()) &&
                     !Modifier.isTransient(destFields[j].getModifiers()) &&
                     !Modifier.isVolatile(destFields[j].getModifiers()) &&
                     !Modifier.isStatic(destFields[j].getModifiers())){
                           destFields[j].set(dest,srcFields[i].get(src));
                  }
                  /* now re-set the accessibility */
                  destFields[j].setAccessible(originalAccessType);
                  srcFields[i].setAccessible(originalAccessType);
              }
            }
    }
    return true;
}
```

Figure 13
Code for the *copyMemberValues( )* method

```
public void autoTest() throws UnusableRoleLoaderException {
      /* try to load the loader itself */
      try {
            this.loadClass(this.getClass().getName(),true);
      }
      catch(Exception e) {
            e.printStackTrace();
            throw new UnusableRoleLoaderException("Did not pass the base auto-test");
      }
}
```

Figure 14
A simple autotest performed by the Role Loader

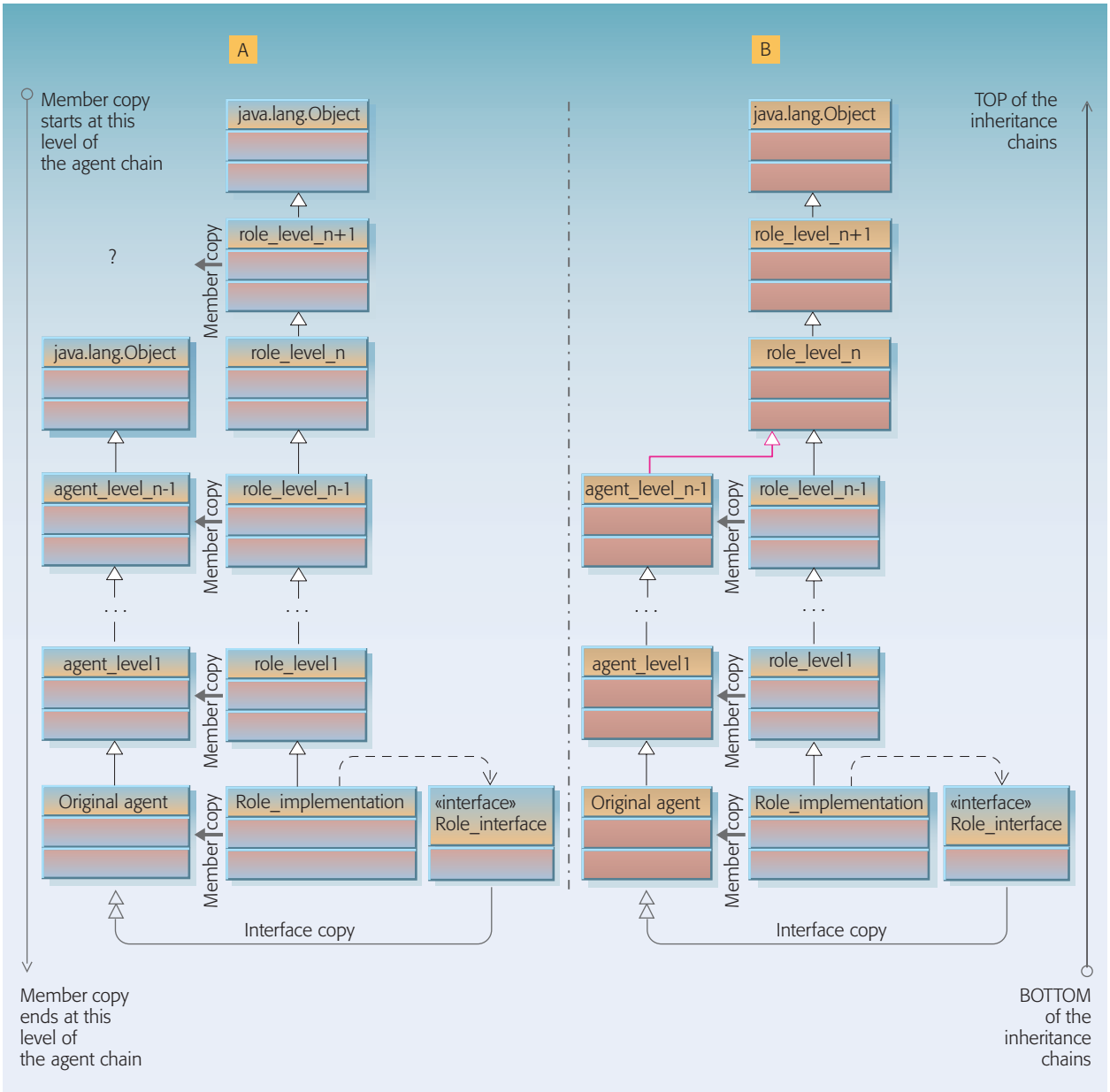

Figure 15
Role inheritance chain longer than agent chain; (A) initial situation; (B) solution

The following subsections describe some of these cases.

Inheritance chain length

This case occurs when the role inherits from a number of classes greater than the number of classes inherited by the agent. In this case, unlike the situation in Figure 9, there are extra classes in the role inheritance chain that cannot be copied to a corresponding level of the agent chain (***Figure 15A***). In this situation, the computed inheritance stack is calculated in a different way: the Role Loader, during the inheritance stack computation, removes the java.lang.Object class from the agent chain and substitutes it with the first extra class of the role chain. In this way the extra classes of the role chain are not lost. This solution implies that the agent inheritance chain changes, because its top is "attached" to the bottom of the extra part of the role chain, as shown in ***Figure 15B***. This leads to a

```
protected void addToStack(String className,String[] roleNames){
        int index=0;
        boolean changedChain=false;

         if(className==null && roleNames==null){
                return;
        }

        // if agent class (className) is null (role chain longer than the agent one)
        // place the first role class as exuberant agent class
        if(className==null ){
                for(int i=0;i<roleNames.length;i++){
                        if(roleNames[i]!=null){
                          className=roleNames[i]; index=i; changedChain=true; break;
                        }
                }
        }

        // if there are no more roles (agent chain longer than the role one)
        // insert null role names
        if(roleNames==null){
              String v[]=new String[2]; v[0]=className; v[1]=null;
              this.roleStack.push((String[])v); return;
        }

        /* now construct a single vector */
        String tmp[]=new String[(1+roleNames.length)];
        for(int i=0;i<tmp.length;i++){ tmp[i]=null; }

        /* copy the names into the array */
        tmp[0]=className;

        for(int i=1;i<tmp.length && i<roleNames.length-index ;i++){
           tmp[i]=roleNames[i-1+index];
        }
        this.roleStack.push((String[])tmp);
}
```

Figure 16
Code for the *addToStack( )* method

situation where the final agent chain is composed of classes of different chains, which are the darker chains in Figure 15B. Of course, while doing this, the Role Loader removes the java.lang.Object class at the top of the agent chain, because it will be substituted by the sub-chain at the top of the role inheritance chain (which is correctly terminated by a java.lang.Object class).

From a code point of view, the preceding solution is implemented by the addToStack( ) method, invoked by the computeInheritanceStack( ) method (see the section "Step 1: Calculating the inheritance stack"). The code for the method is shown in ***Figure 16***. The first parameter, className, is the name of the agent class; the second parameter, roleNames, is an array of role class names that must be fused with the agent at the current level. If the agent class name is null and roleNames is not (which means that the role chain is longer than the agent one), the agent class name to be pushed onto the internal stack (roleStack) is extracted from the role class at the current level, and so on, until the top of higher chain (java.lang.Object) is reached.

Duplicated members

During the copy of members from role classes to agent classes, there can be duplicated members: that

is, both agent and role can have variables of the same type and with the same name, or methods with the same signature. While multiple-inheritance languages provide a way to make member reference unambiguous (e.g., the scope operator of C++), Java does not take this case into consideration because it does not allow multiple inheritance. This implies that the Role Loader must avoid duplicated members. This problem can only occur in the second step when member declarations are copied: in fact, it may happen that one member of the role class clashes with a member of the agent class. When the Role Loader joins the classes, it must make a decision regarding how to proceed in order to avoid duplicated members.

Duplicated methods pose a difficult problem because currently the Role Loader is unable to copy a duplicated method, thus avoiding the generation of a ClassFormatError warning at agent instantiation. Warnings are issued because maintaining only the agent's methods could cause the role to be unusable. A warning is an information event that reports a role injection error; it can be thought of as a "light exception" because, unlike exceptions, a warning does not change the program execution flow. Warnings inform the agent, so that it can determine whether the chosen role is compatible with itself.

If the duplicated member is a variable, the Role Loader can successfully copy the duplicated role variable, keeping it separate from the agent variable, so that the agent and the role parts of the manipulated agent can each access their own variables. This is made possible because variables are not accessed in the bytecode by names but by offsets. Nevertheless, though the case of duplicated variables does not represent a real problem for the role injection process, the Role Loader notifies the manipulated agent of the duplication, again through warnings. If the Role Loader provides partial support to avoid duplicated members, why does a duplicated member situation happen? As shown in the next section, the agent chooses a role by means of a role descriptor (as described previously). This grants flexibility because different role implementations can be bound to the same descriptor, enforcing the locality of the role implementation. This also implies that an agent assumes a role only via semantic information, without knowledge about the syntactic structure of the role itself. In this situation, it may happen that a role's structure clashes with that of the agent, so that it must be notified about the problem. Analyzing warnings, the agent can understand what went wrong during the extension process and then decide to continue or to release the role.

Code example

To show how agents can exploit the capabilities of RoleX, this section provides a simple code example of an agent that assumes and uses the role whose role descriptor is shown in Figure 6.

The code is shown in ***Figure 17***, where BookerAgent is defined as a subclass of RoleSupport. The latter class defines a particular base-agent class, which embeds the invocation translator described in the section "RoleX at work," allowing the agent to use role operations in a simple way. Note that the use of RoleSupport, and therefore the use of the invocation translator, is not mandatory, but, as already detailed, it represents a good choice to keep the agent code simple and clean.

The code shown in Figure 17 is driven by the value of the boolean flag hasRole, which is used to quickly check if the agent has already been manipulated. Initially the flag has the value false, and thus the first code branch is executed, where the agent contacts the role descriptor repository, searching for a role that matches the requested one. After the agent finds the role, the role infrastructure is contacted through the special class RoleX, and the agent asks to inject the role into itself. In this way, the agent proxy stops the agent thread, contacting the role infrastructure and asking for a new instance of the Role Loader, which is in charge of injecting the selected role into the agent owned by the proxy. Note that some useful variables, such as the hasRole flag and the op array, are set before the request of role injection so that the original state of the agent copied after bytecode manipulation is the one the agent has when it asks to assume the role. If everything goes right, a new agent is created, and its execution restarts; that is, the run( ) method is executed again. This time the hasRole flag has a true value, so the second branch of code is executed.

In this part, the agent searches among the stored operation descriptors for the one that matches the flight booking operation, and then invokes it by using the special method act( ), inherited by RoleSupport. The act method accepts the descriptor related to the operation (as either a scalar or an array value) to run and the parameters to use, and then runs it. If an array of operation descriptors is

```
public class BookerAgent extends RoleSupport
{

        // flag to easily know if the agent has already a role
        boolean hasRole=false;
        // store the operation descriptors for quick operation execution
        OperationDescriptor op[]=null;
        . . .
        // execution method
        public void run(){
                int r, a;
                if(this.hasRole==false){
                  // build a set of keyword
                  String [] keys = . . .;
                  // get all the role descriptors stored into repository that
                  // match the keywords
                  // (suppose that keywords identify a single role descriptor,
                  // otherwise other discriminating data is required)

                  RoleDescriptor roleDesc[]=RoleRepository.getRoleDescriptor(keys);
                  If( roleDesc.length == 1 ){ chosenRole = roleDesc; }

                  // the descriptor at index r is the found descriptor,
                  // assume the corresponding role
                  // and store the role action descriptors in an array
                  try{
                      this.hasRole=true;
                      this.op=roleDesc[r].getOperationDescriptors();
                      RoleX.addRoleToMyself(chosenRole);
                  }
                  catch(RoleLoaderException e)
                  {
                      System.err.println("Exception during role loading!");
                      this.hasRole=false; this.op=null;
                  }
                  ...
                }
                else
                if(this.hasRole==true)
                {
                  // the agent has the role
                  // check for warnings
                  if(RoleX.getWarningCount()>0){
                  // get warnings and decide what to do (release the role,
                  // continue,etc.)
                  Warnings warns[]=RoleX.getWarnings();
                  }

                  // search for the descriptor that represents the action
                  for(int k=0;k<this.op.length;k++)
                  {
                      // check descriptors values like keywords, version, date
                      // actions and so on
                      ...
                      if(op[k].getAim().equals("book a flight") && ...){
                            a = k; break;
                      }
                  }
                // invoke the act method of the RoleSupport interface
                Object ret = this.act(op[a]);
                ...
                }
            }
}
```

Figure 17
Fragment of code of BookerAgent that assumes the *flight_booker* role

passed, the act( ) method executes all of them, returning an array of objects to the agent, where each element represents a single return value. After that, the agent can either release the role or run another operation.

It is important to note that the agent can check for warnings, and this check must be done after the agent is sure the role has been injected (in the second branch in the code). Warnings can still be retrieved using the RoleX class, which provides access to the Role Loader instance that has injected the role into the agent.

COMPARISON WITH OTHER APPROACHES

Interactions between agents have been studied extensively. For a complete comparison of existing role approaches, see References 15 and 19. General information about roles can be found in Reference 25.

One approach which offers extensive adaptability is Kendall's AOP (aspect-oriented programming) approach,[17] though it does not offer external visibility of roles. In fact, we know of no other role infrastructure that grants both adaptability and external visibility as RoleX does. Kendall's approach is an important work, which exploits AOP in order to make roles available to agents, with runtime adaptability and a new role concept. Though it was not designed in connection with roles, AOP provides interesting mechanisms to support the management of roles for agents.[17,26] Since an *aspect* is a property that cannot be an active stand-alone entity, but rather affects the behavior of components, its similarity to a role is evident.

AOP has a few drawbacks that our approach tries to overcome. First of all, AOP requires developers to learn a new paradigm and new tools, while our approach requires developers simply to write Java classes and a few XML documents. Furthermore, AOP cannot add external visibility to agents, but simply new capabilities (such as methods). Moreover, AOP is not intended to be applied to active entities like agents but rather to wrap around behaviors of classes (passive entities). Finally, AOP focuses on software development rather than addressing the issues of dynamic and open environments, such as those considered in the BRAIN project, and this makes AOP inadequate for the development of agents and roles.

Two other popular mechanisms for managing agent interactions are ACLs and KQML. ACL, the Agent Communication Language, has been proposed by the Foundation for Intelligent Physical Agents[27] (FIPA) as a way to deal with interactions based on speech acts.[28] The main aim of Knowledge Query and Manipulation Language (KQML) is to state, in a way that does not depend on context or ontology, what the intentions of an agent are (see, for example, Reference 29). Both FIPA's ACL and KQML can be used as protocols for agent communications and knowledge-sharing mechanisms, allowing agents to interact and deal with common tasks. The use of roles differs from these approaches because the latter allow developers to deal with interactions between agents, without helping them very much in dealing with interaction contexts. Roles can be more useful to design, develop, and even maintain complex applications, where there are many interactions and interacting agents.

Though ACLs and KQML are not role-specific approaches, they can be used and embedded into roles. For example, a RoleX event can be shipped within an ACL speech act. This is another important aspect of our approach: RoleX tries to grant maximum flexibility to role developers so that they are free to decide which interaction protocol to use. Of course, because RoleX has been designed to exploit events, developers will have facilities to deal with event operations; however, a role can be developed that partially or completely excludes events and uses only KQML or ACLs. Moreover, FIPA has a modeling working plan (see Reference 30) for the use of the Agent UML** (AUML)[31] that recognizes the part played by roles in the design and development of agent applications, and this means that probably there will be some standardization for role-based techniques that is not currently available.

CONCLUSIONS

In this paper, we have described a BRAIN implementation called RoleX. The main component of RoleX is the Role Loader, which, by modifying the bytecode of agents at runtime, allows them to merge into a single entity with a role when it is assumed. RoleX exhibits all the advantages derived from roles, such as separation of concerns, reuse of solutions, and locality in interactions. In addition, the RoleX implementation provides other advantages. It enables agents to dynamically assume and release roles at runtime, granting flexibility and adaptability.

Roles are not simply given to the agents, but rather agents are modified at the bytecode level to embody all the features of the assumed roles. Descriptors are used to uncouple role assumption from agents, to improve security, and to enable role composition (the construction of roles from smaller elements).

RoleX is currently implemented on the top of the IBM Aglets* platform,[32] though its design allows it to be easily ported to other systems. We have already exploited the RoleX infrastructure to implement some role-based agent applications, such as conferencing support[15] and an automatic e-mail account configurator.[33] Currently we are exploring the area of e-democracy, with an application that enables users' mobile devices to dynamically play appropriate roles to enable "attending" a convention and voting for a candidate.

During the development of these applications, we found RoleX to be robust, though its development is not complete yet. Because role methods are injected into the agent, developers must understand that the execution of a role method is resolved at runtime by a call to an added method of the agent itself. This can cause some confusion initially, but thanks to the RoleX model's simple and powerful API, developers can easily adapt to the RoleX model after a short learning period.

RoleX performance measurements (see Reference 13) show that the assumption/release time depends directly on the role's total bytecode size. We are currently working to improve RoleX's performance by exploiting the same Role Loader to inject different roles in different agents. Though our approach comes from a specific requirement (adding roles to agents), it can be exploited in other situations as well, where two or more Java classes need to be joined in a dynamic way, such as in the addition of dynamic services to components. Further examples can be found in the techniques proposed to grant a transparent Java thread migration to implement strong mobility (see References 33 and 34) and in reflective systems such as the 2K operating system.[36]

Additional documentation on RoleX, including examples and source code, is freely available at the BRAIN Web site.[14]

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc. or Object Management Group, Inc.

CITED REFERENCES AND NOTE

1. M. Luck, P. McBurney, and C. Preist, *Agent Technology: Enabling Next Generation Computing—A Roadmap for Agent Based Computing*, AgentLink—European Coordination Action for Agent-Based Computing (2003), http://www.agentlink.org/roadmap.
2. N. R. Jennings, "An Agent-Based Approach for Building Complex Software Systems," *Communications of the ACM* **44**, No. 4, 35–41 (2001).
3. G. Cabri, L. Leonardi, and F. Zambonelli, "BRAIN: A Framework for Flexible Role-Based Interactions in Multiagent Systems," *Proceedings of the 2003 Conference on Cooperative Information Systems (CoopIS)*, Catania, Italy (2003), pp. 145–161.
4. G. Cabri, "Role-Based Infrastructures for Agents," *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS 2001)*, Bologna, Italy (2001), pp. 210–214.
5. G. Cabri, L. Leonardi, and F. Zambonelli, "Separation of Concerns in Agent Applications by Roles," *Proceedings of the 2nd International Workshop on Aspect Oriented Programming for Distributed Computing Systems (AOPDCS 2002)*, Vienna, Austria (2002), pp. 430–438.
6. Y. Aridor and D. Lange, "Agent Design Patterns: Elements of Agent Application Design," *Proceedings of the Second International Conference on Autonomous Agents*, ACM Press (1998), pp. 108–115.
7. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *IEEE Computer* **20**, No. 2, 38–47 (1996).
8. A. Tripathi, T. Ahmed, R. Kumar, and S. Jaman, "Design of a Policy-Driven Middleware for Secure Distributed Collaboration," *Proceedings of the 22nd International Conference on Distributed Computing System (ICDCS)*, Vienna, Austria (2002), pp. 393–400.
9. M. Fowler, "Dealing with Roles", Working draft (1997) http://martinfowler.com/apsupp/roles.pdf.
10. B. B. Kristensen and K. Østerbye, "Roles: Conceptual Abstraction Theory and Practical Language Issues," *Theory and Practice of Object Systems (TAPOS)* **2**, No. 3, special issue on subjectivity in object-oriented systems, 143–160 (1996).
11. B. Demsky and M. Rinard, "Role-Based Exploration of Object-Oriented Programs," *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida (2002), pp. 313–324.
12. D. Baumer, D. Riehle, W. Siberski, and M. Wulf, "Role Object," in *Pattern Languages of Program Design 4*, N. Harrison, B. Foote, and H. Rohnert, Editors, Addison Wesley Publishing Company, Reading, MA (1999), pp. 15–32.
13. G. Cabri, L. Ferrari, and L. Leonardi, "Exploiting Runtime Bytecode Manipulation to Add Roles to Java Agents," *Science of Computer Programming*, **54**, No. 1, 73–98 (2004).
14. The BRAIN project, http://polaris.ing.unimo.it/MOON/BRAIN/index.html.
15. G. Cabri, L. Ferrari, and L. Leonardi, "Enabling Mobile Agents to Dynamically Assume Roles," *Proceedings of*

The 2003 ACM International Symposium on Applied Computing (SAC), Melbourne, Florida (2003), pp. 56–60.

16. N. Ubayashi and T. Tamai, "RoleEP: Role-based Evolutionary Programming for Cooperative Mobile Agent Applications," *Proceedings of the International Symposium on Principles of Software Evolution (ISPSE2000)*, Kanazawa, Japan (2000), pp. 243–251.
17. E. A. Kendall, "Role Modeling for Agent Systems Analysis, Design, and Implementation," *IEEE Concurrency* 8, No. 2, 34–41, (2000).
18. M. Fischetti, "Tireless Travel Agent Special Report: The Rise of E-Business/Wheeling and Dealing," http://domino.research.ibm.com/comm/wwwr_thinkresearch.nsf/pages/travel199.html.
19. G. Cabri, L. Ferrari, and L. Leonardi, "The Role Agent Pattern: a Developer's Guideline," *Proceedings of the 2003 IEEE International Conference on System, Man and Cybernetics*, Washington D.C. (2003), pp. 4114–4119.
20. G. McCluskey, *Using Java Reflection*, SUN Technical Articles (1998), http://java.sun.com/developer/technicalArticles/ALT/Reflection/.
21. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano, "A Bytecode Translator for Distributed Execution of 'Legacy' Java Software," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001)*, LNCS 2072, Springer Verlag (2001), pp. 236–255.
22. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Second Edition (1999), http://java.sun.com/docs/books/vmspec/.
23. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, Finland (1997), pp. 220–242.
24. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA (1995).
25. G. Cabri, L. Ferrari, and L. Leonardi, "Agent Role-Based Collaboration and Coordination: A Survey About Existing Approaches," To be presented at the *International Conference on Systems, Man and Cybernetics (SMC)*, The Hague, Netherlands, (2004).
26. *Communications of the ACM* **33**, No. 10, special issue on aspect-oriented programming (2001).
27. The Foundation for Intelligent Physical Agents (FIPA), http://www.fipa.org.
28. A "speech act" is a communicative action emulating human verbal communication.
29. T. Finin, Y. Labrou, and J. Mayeld, "KQML as an Agent Communication Language," *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*, Gaithersburg, Maryland, USA (1994), pp. 456–463.
30. J. Odell and M. Nodine, *Modeling Working Plan*, Foundation for Intelligent Physical Agents (2003), http://www.fipa.org/docs/wps/f-wp-00022/f-wp-00022.html.
31. The FIPA Agent UML Web Site, http://www.auml.org/.
32. D. B. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, Reading, MA (1998).
33. G. Cabri, L. Ferrari, and L. Leonardi, "Mailconfigurator: Automatic Configuration of E-Mail Accounts through Java Mobile Agents," *Proceedings of the Third International Conference on Principles and Practice of Programming Java (PPPJ)*, Las Vegas, Nevada (2004) pp. 141–142.
34. T. Sakamoto, T. Sekiguchi, and A. Yonezowa, "Bytecode Transformation for Portable Thread Migration in Java," *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA 2000)*, Zürich, Switzerland (2000), pp. 16–28.
35. E. Truyen, B. Robben, B. Vanhaute, T. Canny, W. Joosen, and P. Verbaeten, "Portable Support for Transparent Thread Migration in Java," *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA 2000)*, Zürich, Switzerland (2000). pp. 29–43.
36. *2K: A Component-Based Network-Centric Operating System for the Next Millennium*, http://choices.cs.uiuc.edu/2k/.

Accepted for publication September 1, 2004
Internet publication January 13, 2005.

Giacomo Cabri
University of Modena and Reggio Emilia, Dipartimento di Ing. dell'Informazione, via Vignolese 905, 41100 Modena, Italy (cabri.giacomo@unimo.it). Mr. Cabri has been a research associate in computer science at the University of Modena and Reggio Emilia since 2001. He received a Laurea degree in electronic engineering from the University of Bologna in 1995 and a Ph.D. degree in computer science from the University of Modena and Reggio Emilia in 2000. He is affiliated with 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.

Luca Ferrari
University of Modena and Reggio Emilia, Dipartimento di Ing. dell'Informazione, via Vignolese 905, 41100 Modena, Italy (ferrari.luca@unimo.it). Mr. Ferrari is a Ph.D. degree student in computer science at the University of Modena and Reggio Emilia, where he is currently affiliated with the Department of Information Engineering. He received a Laurea degree in computer science engineering from the University of Modena and Reggio Emilia in 2002. His research activity covers the study of models, methodologies, and environments for mobile agent systems, object-oriented programming, and Java-based technologies.

Letizia Leonardi
University of Modena and Reggio Emilia, Dipartimento di Ing. dell'Informazione, via Vignolese 905, 41100 Modena, Italy (leonardi.letitzia@unimo.it). Dr. Leonardi has been a full professor of computer science at the University of Modena and Reggio Emilia since 2001, where she teaches basic and advanced computer science courses. She received a Laurea degree in electronic engineering in 1982 and a Ph.D. degree in computer science in 1989, both from the University of Bologna. She is affiliated with 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. ■