Exploiting runtime bytecode manipulation to add roles to Java agents

Giacomo Cabri*, Luca Ferrari, Letizia Leonardi

Dipartimento di Ingegneria dell’Informazione, Università di Modena e Reggio Emilia, Via Vignolese 905, 41100 Modena, Italy

Received 16 January 2004; received in revised form 5 May 2004; accepted 20 May 2004
Available online 21 July 2004

Abstract

Thanks to their sociality, agents can interact with other agents in a cooperative or competitive way. Such interactions must be carefully taken into consideration in the development of agent-based applications. A good paradigm for modeling such interactions is the one based on the concept of roles, which is fully exploited in the BRAIN framework. The use of roles achieves several advantages, from separation of concerns between the algorithmic issues and the interaction ones, to the reuse of solutions and experiences in different applications. In this paper, we propose an interaction infrastructure for enabling Java agents to dynamically assume roles at runtime and then to use them. Our approach is based on the capability of modifying the bytecode of Java agents at runtime in order to add the members of role classes. An application example and a comparison with other approaches show the effectiveness of our approach.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Java agents; Dynamic bytecode modification; Roles; Interactions

1. Introduction

Agents are autonomous entities able to perform their task(s) requiring as little as possible user involvement. Thanks to their autonomy, agents can act on behalf of their user, leading to a new way of developing applications. The agent-oriented paradigm
is emerging as a feasible approach to the development of today’s complex software systems [14]. In fact, agents, thanks to their capability of both executing in a proactive way and reacting to environment changes, can naturally deal with dynamism, heterogeneity and unpredictability. Moreover agents can be mobile, which means that they can move to different sites during their execution. The mobility grants agents the capability of searching for, knowing, exploring and running on different environments. Of course, since an agent could execute in mutable and dynamic environments, it is very important to take into consideration the interactions between agents and the surrounding environment. During its life, in fact, an agent could interact with other agents or environments, in a cooperative or competitive way, and these interactions could be decisive for the tasks to be performed. Furthermore, in Multi-Agent Systems (MAS) agents routinely use rich social interactions to complete the set of tasks assigned to them.

Roles represent a good paradigm for modeling interactions among agents. A role can be thought of as an interface for interactions, providing a set of common instruments for dealing with and allowing interactions among entities. Furthermore, roles help the modularization and the organization of a MAS, separating responsibilities and rights among entities involved [28]. Roles are used not only in the agent scenario; in fact the concept of role is adopted in different areas of information technology, in particular to obtain uncoupling at different levels. Some examples of these areas are security—in particular, we can recall the Role Based Access Control (RBAC) [22] that allows uncoupling between users and permissions—and Computer Supported Cooperative Work (CSCW) [25], where roles ensure dynamism and separation of duties. Also in the area of software development, we can find approaches based on roles, especially in Object-Oriented programming [10,18] and in design patterns [12] such as the Role Object Pattern [3].

There are different advantages in modeling interactions by roles and, consequently, in exploiting derived infrastructures. First, this approach enables a separation of concerns between the algorithmic issues and the interaction ones in developing agent-based applications [7]—in fact, the algorithmic issues are developed in the agents, while the interaction ones are developed in the roles. Second, it permits the reuse of solutions and experiences. Therefore, since roles are related to an application scenario, designers can exploit previously defined roles for similar applications or situations. As an example, [4] reports on how roles can be exploited to easily build agent-oriented interfaces of Internet sites. Moreover roles can be seen as design patterns [1]: a set of roles along with their interaction relationships can be considered as a solution to a well-defined problem, and reused in different similar situations. Finally, the use of roles promotes locality in interactions. In fact, each local interaction context (i.e., environment) can define its allowed roles and rule the interactions among them.

Starting from the above comments it should be clear how roles can be useful in the analysis and design phase, easing the application development, but this does not suffice: roles must grant to agents the capability to adapt to the current execution environment, which in today’s applications is very dynamic and mutable. This leads to the need of a dynamic way to assume roles at runtime, which reduces the coupling between the assuming agents and the roles. With dynamic support for role assumption, agents can be free to exploit role capabilities only when they are required, keeping themselves simple, since they are not required to embed the above capabilities from the beginning.
The BRAIN (Behavioural Roles for Agent INteractions) framework aims at covering the agent-based application development in different phases, proposing an approach to agent interactions based on the concept of role [7]. Each role is a set of behaviors and capabilities that agents can exploit to perform their task(s) in a local context. In this paper, we discuss the implementation issues of an interaction infrastructure for the BRAIN framework, called RoleX (Role eXtension), thanks to which Java mobile agents can dynamically assume roles. The focus is in particular on the related mechanisms which allow us, using bytecode manipulation, to add role members (both method and variable) to Java agents. We consider Java as the language adopted to build agent systems for several reasons. First of all, thanks to its portability, network-orientedness and modularity, Java is the most exploited language for building mobile agent platforms and agent systems in general. Furthermore, Java relies on a strong security mechanism which makes it appropriate for agent systems where security is a must. Finally, since Java relies on an intermediate compiled format, the bytecode, we can manipulate it, according to security constraints and without recompilations, changing the program behavior.

The paper is organized as follows. The second section sketches the motivations of our approach. The third section introduces the BRAIN framework. The fourth section explains how in RoleX the role features are added to agents and, using an application example, how they can be exploited. The fifth section presents performance results, while the sixth section presents a comparison with other approaches. Finally, the last section reports on the conclusions and future work.

2. Motivations of our approach

Our approach is inspired by real life: since agents can act on behalf of a real user, their roles must be “played” as in real life. This mainly requires two features that other role-based approaches generally do not provide: dynamism and external visibility [6].

The dynamic role assumption allows an agent to assume at runtime a set of behaviors, capabilities and knowledge needed to perform its tasks. The agent external visibility, coherent with the assumed role, implies that the way other agents see it must be changed accordingly; this allows other agents to recognize the above one as playing the assumed role. At the language level the external visibility means that it is possible to apply casting operators (such as instanceof) to the above agent, obtaining its role.

Thanks to the granted external visibility, different operations can be performed in a simple way; for instance, as shown in Fig. 1, a server security manager\(^1\) can simply perform a role-based check and allow/deny some administrative operations on the server after recognizing the role of the agent. In fact, the agent recognized as playing the user role is not allowed to modify the database (i.e., to perform a write operation), while the agent playing the administrator role can do it.

Our role approach is more oriented to agents than other approaches that exploit the Object-Oriented paradigm for enabling agents to assume, use and release one or more roles [16,27]. In those cases, roles are conceived as entities separated from agents and

\(^1\) Not related to the Java SecurityManager.
directly used by them. Instead, our approach is centered on roles, fundamental for the agent interactions, which are dealt with in a new way: each role is conceived as a first-class entity, but it becomes a single thing with the agent that has assumed it by extending its code. This is a new way to conceive roles, more similar to the human real life. In fact in the real world, when a person assumes a “role”, for example the employee role, she/he acts as an employee that means she/he can perform employee actions thanks to her/his role. But in this case the role is not an external entity that the person uses; it becomes a part of her/his extended knowledge. In other words the person does not use an external entity to perform an employee action, but simply performs it since all employee capabilities are part of the person’s behavior. This also leads to another aspect: in the real life a person can be recognized also via her/his role. For example a person can be recognized as an employee, since she/he has all the employee capabilities. So in the real world a “role” ensures an intrinsic set of capabilities/behavior and an external visibility/behavior. Now consider what happens when the work day finishes: the person releases the employee role and assumes a new role, for example a swimmer role if she/he goes swimming after work. Here the dynamism required is emphasized: the person assumes/releases roles depending on what she/he wants to do. The same must happen in the agents’ world: the agents must feel free to assume/release roles in a dynamic way.

A way to obtain a single entity, which embeds common features from two (or more) entities, keeping again an Object-Oriented point of view is exploiting inheritance. If one agent inherits from the role that it is going to assume, it will own all role features. This implies the use of multiple inheritance. In fact, each agent class has to be the subclass of a specific platform-dependent base class, as shown in Fig. 2(a) where the agent class must inherit from the base Aglet class to execute on the IBM Aglets Platform [19]. As shown in Fig. 2(b), the assumption of a role should force a multiple inheritance on the agent chain.

Since an approach based on inheritance requires the capability to exploit multiple inheritance, which Java does not have, there is a need to find an alternative way to obtain such behavior dealing with Java agents. It is important to note that, even if the language allows multiple inheritance, a multiple-inheritance-based approach will not lead to a good dynamism, since the agent and role classes have to be defined statically at the implementation phase. Moreover, the use of static multiple inheritance should force all derived instances to play the same role. For this reason our approach does not introduce a
multiple-inheritance mechanism—even if the result can look as if it has—but uses a more complex dynamic assumption mechanism.

A way to ensure the external visibility of the role is by the use of Java interfaces, which can be thought as a way to simulate the multiple inheritance. If an agent class is forced to implement a particular interface, all the casting operators (such as the already mentioned instanceof) will recognize the interface, and so the interface can be used to represent the external aspect of the role. Nevertheless, Java interfaces cannot define a behavior, which means they cannot include method definitions and/or mutable variables. For this purpose a class is needed; such a class is in charge of implementing the role behavior.

Starting from the above considerations, we propose an approach where a role is composed of two components: a role class, which defines the role behavior, and a role interface, which defines the role external visibility. As detailed below, the role class should implement the corresponding interface.

Giving the definition of a role as a couple of class and interface does not suffice for obtaining the dynamism and the external visibility mentioned above. In fact, the agent is to be forced to implement the role interface while assuming the role, and to discard it when it releases the role. To achieve this, our approach performs a manipulation of the agent class bytecode, with the aim of obtaining a new agent class extended with the role embedded and the appropriate interface implemented. In other words, the agent base structure is changed at runtime without any source code alteration or decompile/recompile sequences, and then 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 an extension process because the role features extend the agent ones.

As readers will see, our approach introduces some features not provided by Sun in the Java language. We are not trying to modify the Java language as it is, but simply to overcome some limitations of the language in order to make available to developers more capabilities. We stress how the choice of Java is not arbitrary, since the need of integration with existing systems and all the good features present in the language itself cannot be found in other languages.

3. The BRAIN framework

The BRAIN (Behavioural Roles for Agent INteractions) framework [24] is based on the concept of role and aims at covering the agent-based application development at.

---

Fig. 2. The agent inheritance: (a) without a role; (b) inheriting from a role.
different phases. For this purpose, it provides for a model of interactions that is based on roles, an XML-based notation to describe the roles and interaction infrastructures supporting agents in the management of roles. Such infrastructures are based on the adopted model and rely on the XML notation defined (see Fig. 3).

There are several definitions of the concept of role, which in BRAIN is defined as: a set of capabilities (including knowledge) and an expected behavior. An interaction infrastructure can, for example, implement the role model using an action–event mechanism, where the actions represent the capabilities that the role grants to the agent, and events must be managed by the agent to show the expected behavior (see Fig. 4). This is the implementation adopted by the RoleX infrastructure described in this paper.

The interaction system can control interactions and enforce local policies, such as allowing or denying interactions between agents playing given roles. Note that all policies applied to roles must be considered as additions to the programming language policy management, such as the Java Security Manager. This means that all policies applied inside the BRAIN infrastructure can be applied in a separate way from the Java policy system, which could be kept enabled. Fig. 4 shows how an interaction between two agents occurs. This model of interactions is very simple and very general, and well suited to the main features of the agents: the actions can be seen as the concrete representation of agent proactiveness (i.e., the capability of carrying out their goals), while the events reify the agent reactivity (i.e., the capability of reacting to environment changes).
The notation proposed by BRAIN, called XRole [8], enables the definition of roles by means of XML documents; this ensures interoperability and allows different representations tailored on the needs of the different phases of the application development. 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, ensuring continuity during the entire development. For instance, during the analysis phase, the analysts create XRole documents following the XML Schema shown in Fig. 5, 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 use. In the design phase, the same XRole documents can be translated into more detailed HTML documents to suggest functionalities of the entities involved. Finally, at the implementation phase, again the same XRole documents can be exploited to obtain Java classes that implement the role properties.
4. The RoleX system

4.1. An application example

In the rest of this paper we will refer to a particular application in order to better explain our approach. The application that we have chosen is an automatic network management through mobile agents. As introduced in Section 2 (see Fig. 1), the application is composed by mobile agents which explore the network, exploiting different server’s services. Agents that want to use services must assume the user role. Sometimes there is the need for performing some maintenance operations, and for this reason we designed also the administrator role. An agent playing the latter is able to execute particular tasks on the server itself.

The use of roles can simplify the application design. In fact, each server can maintain its specific set of user and administrative roles, according to the services provided. In this way, an agent who is going to play the user role will be able to exploit the server services without knowing specific details (since they are embedded in the locally available role) and without the need of knowing all services available on other servers (since the role is tied to a specific server). Similarly the administrator role, since it is tied to a specific server, embeds details related to the service management. For example a server can decide to allow an SMB service reboot only after having mounted an NFS directory. The easiest way to achieve this is to build a specific administrator role that performs mount and the reboot in the right order.

Now we detail why role assumption/release should be as dynamic as our role system allows. Imagine that there is an agent that is moving among hosts exploiting their services (i.e., playing in each server the user role). At a given server the agent playing the user role recognizes that a particular service is not available, and cannot be used. To resolve the problem the agent assumes at runtime the administrator role, performs one (or more) maintenance tasks in order to make the services available, and then releases the administrator role and returns to playing the user role.

Of course only particular agents must be allowed to assume the administrator role, since granting administrative capabilities to all agents can be very dangerous. Nevertheless, this is a problem related to security issues, and we are not going to detail them here due to limitations of space. It suffices that our role system enables security capabilities, so users can specify which agents are enabled to assume a particular role and which are denied from doing this. Furthermore, our system exploits base capabilities provided by the Java Security Manager, so, even if an agent assumes the administrator role, it must own Java permissions in order to exploit it.

4.2. Role, action and event descriptors

To ensure a high level of abstraction in the role design process, our BRAIN approach proposes the use of descriptors. Descriptors are an abstraction layer of roles, and thanks to this they can be useful also in the assumption–exploitation process carried out by the agents. To improve descriptor modularity we propose a three-level descriptor model, where a role descriptor describes a whole role, action descriptors describe actions that the role allows and, finally, event descriptors describe which events the role uses in performing
its services. Descriptors are nested starting from the role one, which embeds one or more action descriptors, while the latter embed one or more event descriptors. As their name suggests, descriptors are entities that describe a role, an action or an event, for example by means of information such as keywords, a context, an aim, a version, a creation date, etc. In other words descriptors enable developers to work with role metadata, which can represent a role, or one of its components, by its semantic meaning, rather than its syntactic (i.e., implementation) meaning. Thanks to this structure the role metadata can be accessed with different granularities according to the descriptor nesting level.

Since descriptors represent an abstraction of roles, using them the agent programmers need to know not which is the physical class that implements a role, but only the descriptor of the role to search for. For example, if the agent must assume the administrator role, the programmer could write code that searches not for an administrator role but for a role with an administrator description (see Fig. 6). The agent can further verify the retrieved descriptor to be sure that the role is the right one for its purposes.

Descriptors are derived from XML documents written exploiting the XRole notation of BRAIN. As already noted, Fig. 5 reports the XML Schema to which XRole documents must conform. In order to enable agents to easily access descriptors, our approach automatically translates XML descriptors into a set of Java objects each time a new role is installed\(^2\) in the system. In particular, we have defined a class for each descriptor level, so that the system is in charge of creating instances of these classes. In this way, an agent can directly access the descriptors without needing an XML-parser. Anyway we still use XML notation to ensure information portability across the development phases.

Note that the descriptors are needed also for other reasons, not only for relieving programmers of the knowledge of role implementation. In particular, a role descriptor hides from the agent the physical location of the role implementation. Imagine what could happen if the agent could find the physical implementation of a role: it could use it without assuming the role, for example by a normal object reference, escaping from the role system control. This could lead to situations in which the policy applied to roles could not be checked, since the role system has no control on the assumption/use/release process. In our implementation, this risk is avoided by means of an armored role repository.

\(^2\) The term “installed” here means that a new role is available in the system.
which provides the role classes only to the RoleLoader, a component of the infrastructure explained in the following.

Another reason for using role descriptors is role composition: two or more roles could be mapped into the same descriptor to indicate that a role is actually made up of multiple implementations and the agent must assume all of them. This is useful for keeping role modularity allowing, at the same time, the composition of roles. For example, with regard to Fig. 1, an agent can require both the administrator and the user roles to perform administrative and common operations at the same time. Since both the roles are available, the agent can assume both in a double assumption. Nevertheless, thanks to role composition, the agent can make a single assumption that grants both roles. Composition allows also a development similar to the EJB one [11], since roles developed by third parties can be composed providing a single descriptor, as detailed above.

A further reason is achieving granularity: it is possible to define small roles (and their small descriptors) and compose them to obtain more complex roles, in a similar way to the Object-Oriented building blocks case. In this way an agent can assume the smallest part of a role that grants it the needed capabilities, wasting fewer resources, while another agent can assume easily all the composed roles with a single assumption process.

Finally, descriptors are very useful in the case of code refactoring: a role implementation could change (for example because a new version is released) but the descriptor can remain the same. This allows a high degree of flexibility. The descriptors allow the agent programmers to disregard the work of role programmers, and vice versa, because the role behavior is described in a separate way. A part of an example of role descriptor, with regard to the administrator role, is reported in Fig. 7.

The part of role descriptor of Fig. 7 reports a single administrative operation (Reboot SMB), which allows the agent playing the administrative role to reboot an SMB server. A Java prototype of such operation should be, according to the above XML descriptor, the following:

```java
void SMB_Reboot (int time);
```

where the parameter time is the number of seconds before the effective reboot. From the XML descriptor of Fig. 7, we can extract another characteristic of the Reboot SMB operation, which is the RebootEvent. This is an event sent when the operation is performed. Its aim is to inform all agents connected to an SMB server that the administrator agent is going to reboot the server.

4.3. Adding roles to agents

As already mentioned, our approach defines that the Java implementation of a role has to be composed of two parts: a Java interface (role interface) and a Java class (role implementation); the interface provides external visibility while the class provides the effective role behavior.

A role assumption means that the RoleX system (i) adds each role class member (both methods and fields) to the agent class, in order to add the set of capabilities of the role and, at the same time, (ii) 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.
<?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/RoleDescriptionSchm">
  <GenericRoleDescription>
    <description>Server Administrator</description>
    <roleName>administrator</roleName>
    <keyword>administrator</keyword>
    <keyword>server maintainer</keyword>
    ...
    <version>1</version>
    <OperationDescription>
      <name>Reboot SMB</name>
      <aim>reboot the SMB server</aim>
      <keyword>restart SAMBA</keyword>
      <version>1.0</version>
      <methodName>SMB_Reboot</methodName>
      <returnType>
        <className>java.lang.Void</className>
      </returnType>
      <parameter>
        <className>java.lang.Integer</className>
      </parameter>
      <sendingEvent>
        <name>RebootEvent</name>
        <eventClassName>examples.role.RebootEvent</eventClassName>
        <isResponse>false</isResponse>
      </sendingEvent>
    </OperationDescription>
  </GenericRoleDescription>
</role>

Fig. 7. A part of an example of an XML administrator role descriptor.

Since the above mechanism must result in the definition of a new class for the agent instance that wants to assume a role, our approach exploits a special class loader, called RoleLoader, that can change the agent behavior and the external appearance. After the RoleLoader has successfully carried out the role assumption process (i.e., the addition of the members and the interface), it can reload the agent, restarting it. The assumption process can be briefly described as in Fig. 8, where the agent searches the role repository for an appropriate role descriptor and then asks the RoleLoader to reload itself with the chosen role. If everything goes well, the RoleLoader sends the new agent an event (called reload event) to indicate that the agent has been reloaded. After the reload event, the agent can resume its execution. The RoleLoader can be unable to load the role into the agent for several reasons (e.g., bad role format, permissions); in these cases it throws up an exception that the original agent can catch. Analyzing this exception the agent can understand why the assumption failed and can decide what to do (for example, retry or choose another role). Releasing a role is similar to the above process, but this time the RoleLoader removes each role member and the role interface reloading the agent without them.
To perform the role assumption/release, the RoleLoader performs runtime bytecode manipulation; this manipulation is completely made in memory without needing source code recompilation. The bytecode alteration is needed to work with and to modify class definitions. In fact, to obtain an agent extended with a role we need to create a new manipulated agent class from which a new agent instance is obtained. As already stated, the RoleLoader is a particular kind of Java class loader, and in particular it is a subclass of SecureClassLoader. The fact that our approach is performed through a class loader allows us to work in compliance with the Java Security Manager.

Our implementation of the RoleLoader is based on the Javassist bytecode manipulation engine [23], even if the simple use of such engine alone is not enough to completely satisfy our goals. In fact, our approach takes into consideration role code reusability and separation of concerns. For this reason the assumption mechanism is performed through several steps (suppose that the agent has already contacted the RoleLoader):

1. the RoleLoader calculates the inheritance stack for the role class and the agent class (i.e., it calculates all superclasses of both);
2. for each level of the inheritance stack, the RoleLoader copies all the members (both methods and fields) from the role implementation to the agent one; then the loader adds the role interface to the implemented interface list of the agent class;
3. a new agent instance is created from the manipulated class;
4. each field value is copied into the agent instance so that it does not lose its current state.

Through these steps, even if one agent and the assumed role have been developed separately, they dynamically become a single entity with the correct external visibility. Each step will be detailed in the following subsections. It is important to note the fact that above steps apply only to the agent that is asking to be fused with the role. In other words, even if the RoleLoader manipulates a class, not all instances of that are modified.
too, which means that not all agents, instances of the same original class, are modified if one of them assumes a role. This emphasizes agent autonomy, since similar instances can act and adapt in different ways (i.e., using different roles), and are performed through the capability of class loader name spaces.

Role release is similar to the role assumption, but this time the RoleLoader removes role members from the agent class, still keeping the agent state intact. Thanks to the fact that a role assumption never destroys an agent class member, the assumption process is always reversible; thus an agent is really free to assume and release any role.

Step 1: calculating the inheritance stack

The first step is needed to grant role-inherited properties to the agent, promoting role code reusability. In fact, the role implementation could be not a single class but the bottom of an inheritance chain. For example, the administrator role could inherit properties from the user 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, a role implementation (sub)class expects to find some capabilities in its superclasses, so we must ensure that this condition will remain true. To better explain this concept, we refer to Fig. 9. The figure shows, via a class diagram, the inheritance chains of an agent and a role that the agent is going to assume. Both the role and the agent classes are represented by the bottoms of their respective chains. This means that the bottom classes must be joined. At the superclass level the same must occur; that is, the superclasses must be joined also. This must be done for each chain level. In this way, our system 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 except calculating the inheritance stack—that is, how a role class and an agent class must be joined and at what level.
Table 1
The inheritance stack calculated by the RoleLoader

<table>
<thead>
<tr>
<th>Agent's chain</th>
<th>Role's chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Object</td>
<td>none</td>
</tr>
<tr>
<td>...</td>
<td>none</td>
</tr>
<tr>
<td>agent_level2</td>
<td>none</td>
</tr>
<tr>
<td>agent_level1</td>
<td>role_level1</td>
</tr>
<tr>
<td>original_agent</td>
<td>role_implementation</td>
</tr>
</tbody>
</table>

Referring to Fig. 9, the computed inheritance stack is reported in Table 1; note that the root class java.lang.Object is kept in only one chain. Every row in the stack indicates which classes will be joined into one, and will be used in the second step to provide the knowledge of from which class the members will be copied in the agent chain and in the fourth step to provide the knowledge of which members values must be copied.

**Step 2: copying members’ declarations**

This step performs the member declaration copy 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. Only this step uses the bytecode manipulation that allows the system to modify the class definitions. Note that no members are removed from the original agent class. In our implementation, only an adding mechanism is provided and this ensures a correct execution of the agent in every situation.

In this step, the bytecode manipulation is also used to force the agent class to implement the role interface. Since every class contains a list of the implemented interfaces [20], this is done simply by adding the role interface to that list in the agent class.

**Step 3: creating a new agent object**

After the above two steps, our system makes a new agent class available, to which the role has been joined. To obtain a new agent, the system must create a new instance of the manipulated agent class. This step does it, creating an agent instance from the manipulated class, which is linked (directly or indirectly) to all the manipulated classes.

**Step 4: copying members’ values**

In this step, which is the last step, every attribute value is copied from the original agent to the newly created agent. This step ensures that the agent state will not be lost during the reloading process. Note that the first two steps involved classes, while the other two steps involve objects, requiring that a reference to the original agent object is maintained until all values are copied. At the end of the last step the original agent is disposed of and the new one starts executing.

To avoid other agents still maintaining a direct reference to the original (i.e., not manipulated) agent, we adopt a protection mechanism based on the concept of proxy [13]. This means that it is not possible to obtain a direct reference to an agent, and that the only way to refer an agent is through its proxy, which masquerades as the agent itself. Only the role system can deal with the direct reference to agents, so all other running entities will not perceive any difference between a role assumption and release.
4.4. Class loading order

The previous section has described the role assumption detailing the main steps. This section covers the RoleLoader internal behavior, and in particular the fact that the presence of a cache imposes a specific order of the loading of the class.

As already stressed, the RoleLoader is a particular kind of Java class loader, and this means that, like all other class loaders, it has a private class cache. This cache maintains the definition of each loaded class, so the class loading is faster if an already loaded class is required.

While the class loader cache is an important component, which can speed up the class loader response time, it can result in a disadvantage for our approach. In fact, the above cache is a private member of the class java.lang.ClassLoader, and this means that a subclass (such as the RoleLoader) cannot access it. In other words, a class loader cannot remove or add any entry in the cache. But for what reason should the cache management produce an execution error in our RoleX system? Each Java class loader has a unique namespace, which means that each class is uniquely identified by the class loader into its namespace. As readers can easily understand, it is not possible to have more than one class definition, since this would mean that two classes with the same name but with different definitions would exist in the class loader name space. Applied to our role system, this implies that the RoleLoader must load the new agent class starting from the top of the chain. The member copy is made starting from the base classes, going down the inheritance chain until the agent class (i.e., the last class) is reached, as shown in Fig. 9. In fact, when the original agent class is loaded, all of its base classes should be into the loader cache, since a class loading 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 would be made from the bottom of the inheritance chain to the top: the first class manipulated would be original_agent, which is joined with role_implementation. In this case, when the class original_agent is loaded by the RoleLoader (before the manipulation process is started), it should be linked with its superclass, agent_level1; this class should be then linked with its superclass in turn, agent_level2, and so on until the java.lang.Object class is reached. As detailed in the Java Language and Virtual Machine specifications, 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 RoleLoader, to the class agent_level1 already present in the cache. Since the loader already has the class into its cache, it would not reload it and the manipulated agent class will be linked with a not manipulated superclass. Recalling that 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, would produce an exception because the class namespace of the RoleLoader would have two (or more) classes with the same name but different bytecode definitions. In other words, two or more classes would clash in the class loader namespace throwing up a LinkageError for duplicated class definition. 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 the above concepts, take a look to what happens, step by step, during a role assumption like that in Fig. 9. In such a situation, when the original_agent is loaded, our
RoleLoader 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 that 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 an URL), and then manipulate it. But the manipulation, if made after a linkage operation, will cause an error, because two classes with the same name (agentLevel1) 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 separated 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.

Another important aspect of the RoleLoader is that it cannot use the delegation model like all other Java class loaders. Introduced from the API 1.2, the delegation model imposes that a class loader should ask its parent class loader to define a class before it can do this on its own. This ensures efficiency and code portability, since a class is defined only by a class loader (for example system classes, like those in the java.lang package, are defined only by the first class loader). But, as specified above, the RoleLoader cannot use the delegation model because, if it does, different class definitions would be present at runtime, producing namespace clashes. In fact, working with the delegation model, the RoleLoader should ask its parent to load and define an agent class. This leads to the definition of an unmodified agent class, since the parent class loader has not performed bytecode manipulation on it. In fact, working with the delegation model, the original agent class definition is always found, since each class loader will ask its parent for that class.

For the above reasons the RoleLoader starts the bytecode manipulation from the top (reverse class loading), loading all the classes by itself (discarding the delegation model). During this process the java.lang.Object is skipped (i.e., not loaded or manipulated) in order to ensure code portability and to avoid execution faults in the Virtual Machine. As explained in the previous subsection, each manipulation level is calculated in the first step, so that the RoleLoader, when it starts the bytecode manipulation, knows exactly which classes must be joined together.

4.5. Particular cases

During the member addition from the role classes to the agent ones, two particular cases can arise and must be treated in a special way. The former case is related to the inheritance chain length, while the latter is related to the chance of having duplicated members. Both these cases are detailed in the following sections.

Inheritance chain length

The problem of the inheritance chain length arises when the role inherits from a number of classes larger than that of the agent. In this case, unlike the situation shown in Fig. 9, there are a few exuberant classes in the role inheritance chain that cannot be copied at a corresponding level of the agent chain (see Fig. 10(a)).

In this situation the computed inheritance stack is calculated in a different way, in order to find a solution to this problem. In particular, the RoleLoader, during the stack computation, removes the java.lang.Object class from the agent chain, and replaces it
Fig. 10. (a) A role inheritance chain longer than the agent one; (b) the solution.
with the first exuberant class of the role chain. This solution avoids the exuberant classes of the role chain being lost.

This solution implies that, unlike in the case detailed by Fig. 9, the agent inheritance chain changes, so that its top is “attached” to the bottom of the exuberant part of the role chain, as shown in Fig. 10(b). This solution implies that the final agent inheritance chain is composed by classes of different chains, which are the darker ones in Fig. 10(b). Of course, while doing this, the RoleLoader removes the java.lang.Object class at the top of the agent chain, since there could not be anything over that. This implies also, as already stated, that no modifications are made to the java.lang.Object class, in both the chains, in order to ensure code portability and to avoid JVM faults.

**Duplicated members**

Since our approach simulates multiple inheritance, this section details a common problem in multiple-inheritance languages: the possible presence of duplicated members. While multiple-inheritance languages are likely to take this problem into consideration, since they can provide a way to make the member reference unambiguous, Java does not allow multiple inheritance and thus there is no mechanism for referring unambiguously to a duplicated member. This lack implies that in Java there cannot be real duplicated members, so the RoleLoader must avoid them.

The only step in which this problem can happen is step 2, during the declaration member copying: in fact, it may happen that one member of the role class clashes with one of the agent class. For example, both agent and role could have one variable of the same type and with the same name, or one method with the same signature. When the RoleLoader joins the classes, it must take a decision on how to proceed in order to avoid the duplicate members.

If the duplicated member is a variable, the RoleLoader can successfully copy the two occurrences, keeping them separated. In fact, since variables are stored in the .class file as members of an array (i.e. they are accessed by an offset), they are copied in different positions, so that the agent and the role can access their own variables.

The case of a duplicated method is harder, since currently if the RoleLoader proceeds with the copying, a ClassFormatError will be thrown up. To avoid this problem the RoleLoader copies only methods that are not duplicated, giving priority to agent’s methods.

Maintaining only agent’s methods could cause role non-usability, so the RoleLoader notifies the manipulated agent about the problems that occurred via warnings. A warning is an information event that reports an error that has occurred; it can be thought of as a “light exception”, since, unlike exceptions, a warning does not change the program execution flow. Of course, warnings are not a solution to the problem of duplicate members, only an ‘advertisement’ to the agent, provided so that it can understand whether the role chosen is compatible with itself at all.

While the RoleLoader can provide partial support for avoiding duplicated members, it is important to understand why they can arise. As detailed also in the next section, the

---

3 An example of this is the scope resolution operator (::) in C++.
4 The cause of this error is: duplicate field/signature exception.
agent chooses a role via a role descriptor (described in the previous paragraphs). This ensures flexibility, since different roles can be bound to the same descriptor, enforcing the locality of the role implementation. This also implies that the 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 structure clashes with the agent’s one, so it must be notified about the problem. It is for this reason that duplicated member can arise, and for this reason we provide a warning mechanism to notify the agent of one or more problems. Analyzing warnings, the agent can understand what was wrong during the extension process, and then decides whether to continue with or to release the role.

4.6. How agents can use roles

This section details how agents can use dynamically assumed roles, without knowing their methods—only knowing their descriptors. This section shows, from a code point of view, how an agent can use added role members at runtime.

It is important to note that the agent cannot refer directly to a role member, since this will produce a compile-time error due to the absence of the member itself at compilation time. In fact, since the role members are added only at runtime, it is impossible for the agent to keep a reference to a role member “statically”.

The solution to the above problem is the use of reflection: thanks to this the agent can find and use added role members. Nevertheless, the use of reflection to discover new members can be complex and can result in a waste of time and energy for the agent developer. In order to make the use of our RoleX system easier, we provide an invocation translator, an entity that each agent embeds and which helps it to access added role members. For example, in the case of an action, the invocation translator searches for the related method and then invokes it using reflection.

The invocation translator constructs the binding between operation descriptors and role members (see Fig. 11): the agent expresses its will to do an operation, passing an operation descriptor to the invocation translator (step 1); the latter translates the descriptor into a valid Java method, searching for that method using introspection, and then invokes it using reflection (step 2). When the invoked method returns, the invocation translator takes back the result values (if any) and forwards them to the agent (step 3). Thanks to reflection, the execution of a role action looks like a normal method invocation.
public class ServerAdminAgent extends Aglet implements RoleSupport{
    // execution method
    public void run(){
        int r, a;

        if(this.hasRole==false){
            // prepare the keywords to search for a role
            String[] keys = ...;
            // suppose the given keywords point
            // to a single role descriptor, otherwise
            // the agent needs to use other discriminating data
            RoleDescriptor descs[], roleDesc;
            descs=RoleRepository.getDescriptorsByKeywords(keys);
            if(descs.length==1){ roleDesc=descs[0]; }

            // assume the corresponding role
            // and store the role action descriptors in an array
            try {
                this.hasRole=true;
                RoleLoader.addRoleToMyself(roleDesc);
            }
            catch(RoleLoaderException e){
                System.err.println("Exception during role loading!");
                this.hasRole=false;
            }
        }
    }
}

Fig. 12. A fragment of code of ServerAdminAgent that assumes the administrator role.

Fig. 12 shows a code example of role use, where the ServerAdminAgent searches for the appropriate role descriptor. The code shown in Fig. 12 has two main branches,
depending on the value of the Boolean flag hasRole. The meaning of this flag is to indicate whether the agent has been reloaded with the role or not. If the flag is false the agent searches for a role and asks the RoleLoader to load the role found. Note that this step is made within a try–catch block and, if it fails, the flag value is reset. After the manipulation, the agent restarts its execution from the run() method, but this time the flag hasRole is set so the agent searches for an action descriptor and executes a role action.

The use of the flag hasRole allows the agent to readily know how to proceed without introspection on itself (to know whether the role has been added), and furnishes a simple solution to programmers.

5. Performance evaluation

In order to evaluate the applicability of our approach we have done some performance tests, whose results are described in this section.

In our tests, we have built a single agent whose main aim is to continually assume and release a defined role, so as to place the system under pressure. Our current RoleX implementation contains some predefined hooks that allow event logging, so it is possible to know what happens during the application run. The RoleLoader uses these hooks to register instants when it starts manipulating the agent class, allowing us to know how much time is required to perform such an operation. Time instants are reported by RoleX in a table which reports the agent identity, the time (UTF) and the event that the time value is related to (see Fig. 13).

Our evaluation aimed at identifying what influences the role assumption/release time. To do this, different trials were done, changing the role inheritance chain length. We split the tests into four groups, depending on the role chain length: inheritance chains of one class (only the role), ten classes, fifty classes and one hundred classes. To make the test correspond to reality more closely, we distributed the members of each class evenly on the inheritance chain, such that the total number of members was always one hundred (equally distributed between methods and fields). Table 2 reports the result values for a set of trials on an Intel PC running Red Hat Linux 7.3 (kernel 2.4.18) with the JDK 1.4.1,01. All values have been calculated as averages over more than twenty executions.
As Table 2 shows, the main result is that the assumption time is directly dependent on the compiled bytecode size of the role. In fact, disregarding the case of a single class, the assumption ratio is always constant and its value can be placed near to ten bytes per ms, as shown in the third column. The important thing to note is that the ratio is independent of the inheritance chain length, but, as already mentioned, it depends only on the bytecode size of all role classes. With regard to the first cell in the fifth column, one can note how the ratio is lower, about 2.5 bytes per ms. This is due to the fact that there is a fixed time needed for the system initialization. In fact a new RoleLoader must be created for each assumption, due to classloader’s namespace collisions, and the role class repository (i.e. where the role classes are stored) must be contacted. The above actions influence the assumption time when the role class is quite small and this is why the ratio is lower than in other cases. We are currently working to improve performance, reducing the threshold time (see Section 7).

Another important thing to note is that the release time, shown in the third column, is always constant, and does not depend on the role. In fact, since the application used as a test is made by a single role, when the agent releases the role it is reloaded without it; that means it is reloaded starting from the original agent class chain. Since the original agent chain is always the same, the time needed to reload it is also always the same.

Other tests made on different machines have reported similar results and the same dependencies between the assumption time and the bytecode size of the role used.

In addition to a set of performance analyses, we have already exploited the RoleX infrastructure to implement some role-based agent applications, such as a conference support [9], and an automatic email account configuration system [5]. Currently we are exploring the area of e-democracy, with an application that enables users to exploit mobile devices and run mobile agents to dynamically play appropriate roles such as attending a convention and voting for a candidate. These applications have demonstrated that our approach can be successfully applied to role-based agent applications, even if it could be improved in several ways, such as by increasing the performance of the implementation, and further simplifying the developer API.

6. Comparison with other role approaches

This section briefly shows other role approaches and their differences with respect to ours.

A first approach that can be exploited to implement roles is the traditional Object-Oriented one. In such approach a role is defined as a class, whose methods define the
behavior (i.e., the services) of the role itself. An example of this approach can be found in Fig. 14.

While the OO approach is the simplest one, its main limitation is that the specific features concerning the role played by one agent are not separated from the general features, for instance from the mobility or the planning features. It is important to note that this is a static approach: the agent needs to refer to the class that implements the role. Even if it can be made more dynamic, for example by using object creation services (e.g., factories), the agent always needs to maintain a reference to the role object. This leads to a coupling at compile time while at runtime the agent and the role are separated entities.

The need for an explicit reference to the role object implies that the agent cannot refer to specific site-dependent implementations unless they are built as a subclass of the agent’s reference type. This forces the role developer to build roles as subclasses of a common accepted base class, reducing the expressivity of the role itself. Apart from the dynamism introduced by our approach, since we do not fix the role bounds, everything can become a role, depending on the application needs and semantics.

An approach that partially solves the OO problems is the one based on the Aspect-Oriented Programming (AOP). While it has not been designed in connection with roles, AOP seems to provide interesting mechanisms for supporting the management of roles for agents [2, 16]. AOP starts from the consideration that there are behaviors and functionalities that are orthogonal to the algorithmic parts of the objects [17]. So, it proposes the separate definition of components and _aspects_, to be joined together by an appropriate compiler (the _Aspect Weaver_), which produces the final program. The separation of concerns introduced by AOP permits one to distinguish the algorithmic issues from the behavioral issues. Since an aspect is a property that cannot be encapsulated in a stand-alone entity, but rather affects the behavior of components, the similarity with a role is evident. Kendall exploits the AOP to concretely implement the concept of role in agent applications [16].

While the AOP approach is similar to ours, it has some limitations that our approach overcomes.

First of all there is a strong coupling between the aspect and the agent itself: the aspect must be built taking into account the agent structure (i.e., the agent class), so that the aspect weaver can join them correctly. This implies that each role/aspect must have a different implementation depending on the agent that is going to use it. This does not help role developers, since it does not allow strong code reusability. AOP focuses on software development rather than addressing the issues of dynamic and wide-open environments, such as the ones considered in the BRAIN project, and this made AOP inadequate for the agent/role development.
As a last note, not tied to the implementation, we can say that AOP does not provide a support for the designer as effective as the one provided by XRole, our XML-based notation that allows a richer and more comprehensible description of the roles and the possibility of formatting appropriate presentations of roles in an easy way.

7. Conclusions and future work

In this paper we have detailed the implementation issues of a BRAIN implementation, called RoleX, which allows Java agents to assume, use and release roles at runtime. While there are other role approaches which allow agents to dynamically assume roles (e.g. [27]), we decided to develop RoleX since its point of view is different: the agent and the role do not remain separated, bound in a certain way, but become a single entity. Furthermore, this happens with a high degree of transparency.

The main component of RoleX is the RoleLoader, which enables the agent to dynamically assume roles. This is achieved by modifying the bytecode of the agents, adding the features of the role(s) that they want to play. RoleX adopts the action–event interaction model and relies on the XRole notation to describe roles.

RoleX exhibits all the advantages derived from roles, such as separation of concerns, reuse of solutions and locality in interactions. In addition to these advantages, the specific ones of our system can be summarized as follows:

- It enables agents to dynamically assume roles at runtime, ensuring flexibility and adaptability. Roles are not simply given to the agents: agents are modified at code level to embody all the features of the roles. The use of descriptors decouples the role assumption, improves security and enables role composition.
- It ensures a high degree of role reusability, because it deals not only with the classes of agents and roles, but also with their whole inheritance chains.
- It allows agent and role programmers to work separately, since they can compile their classes (representing agents and roles) separately because the binding between them is dealt with completely at runtime.

While our approach comes from a specific requirement (adding roles to agents), it can be exploited also in other situations where two or more Java classes need to be joined in a dynamic way, such as the addition of dynamic services to components. Further examples can be found in the techniques constructed to ensure a transparent Java-thread migration to implement strong mobility (see [21,26]) and in reflective systems such as the 2K operating system [15].

RoleX, like the BRAIN framework, is publicly available for download at the BRAIN Web site [24].

Acknowledgements

This work was 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”.

96

References


