How to Dynamically Add Roles to Agents

Giacomo Cabri, Luca Ferrari, Letizia Leonardi

Abstract— **Interactions are an important issue to be faced in the development of agent-based applications. In fact, their sociality allows decomposing large applications into collaborating agents, while open environments, such as the Internet, require agents belonging to different applications to compete to gain resources. In the BRAIN framework, interactions among agents are fruitfully modeled and implemented on the basis of roles; this approach achieves several advantages, from separation of concerns between the algorithmic issues and the interaction issues, to the reuse of solutions and experiences in different applications. In this paper we propose a mechanism to enable Java agents to dynamically assume roles at runtime. Our approach is based on the capability of modifying the bytecode of Java agents, to implement an appropriate interface and to add the related methods. An application example and the comparison with other approaches show the effectiveness of our approach.**

Index Terms— **Agents, Roles, Interactions, Dynamic assumption, Java.**

I. INTRODUCTION

In agent-based applications a global task is usually decomposed into smaller tasks that are carried out by the single agents; they must interact to achieve the goal, and such interactions are an important issue to be taken into consideration during the development of applications. The BRAIN (Behavioural Roles for Agent INteractions) framework [3] proposes an approach to agent interactions based on the concept of *role* [11]. There are different advantages in modeling interactions by roles and, consequently, in exploiting derived infrastructures. First, it enables a separation of concerns between the algorithmic issues and the interaction issues in developing agent-based applications [6]. Second, it permits the reuse of solutions and experiences; in fact, roles are related to an application scenario, and designers can exploit roles previously defined for similar applications; for instance, roles can be exploited to easily build agent-oriented interfaces of Internet sites [4]. Third, roles can also be seen as a sort of design patterns [1]: a set of related roles along with the definition of the way they interact can be considered as a solution to a well-defined problem, and reused in different similar situations. Finally, it promotes locality in interactions, since each local interaction context can define the allowed roles and rule the interactions among them. The concept of role is adopted in different areas of the computing systems, 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) [13] that allows uncoupling between users and permissions, and *Computer Supported Cooperative Work* (CSCW) [15], where roles grant dynamism and separation of duties. Also in the area of software development we can find approaches based on roles, especially in the *object-oriented programming* [8] and in *design patterns* [9], which remarks the advantages of a role-based approach.

Exploiting the role advantages, the BRAIN framework aims at covering the agent-based application development in different phases, and provide for (i) a *model of interactions* based on roles, (ii) an XML-based *notation* to describe the roles, and (iii) interaction *infrastructures* based on the previous model and notation, which enable agents to assume roles.

In this paper, we propose an implementation of interaction infrastructure for the BRAIN framework, thanks which mobile agents can dynamically assume roles. In particular, we focus on the mechanisms that enable such dynamic assumption of roles by mobile agents. In open and dynamic environments, for instance the Internet or the pervasive computing based ones [16], the agent capability of dynamically assuming a role at runtime can grant a high degree of adaptability to runtime situations and also permit to suit unexpected situations. We can find an example of dynamic assumption of a role in the film The Matrix, where the female character dynamically downloads the role of "helicopter driver" to escape[1]; in that case, the dynamic assumption of features is vital to run away from a dangerous situation. This is clearly fiction, but it gives an idea of what we want to do in the agent world, where the dynamic assumption of roles can also be useful; for instance, think at an agent that has to obtain a resource, and at runtime discovers that such resource is on sale by an auction: it can dynamically assume the role of bidder and interact with agents playing the role of seller or auctioneer. Of course, such dynamic assumption is not trivial, because agent developers can hardly deal with runtime situations, especially if they require a dynamic modification of the agents. Moreover, the dynamic assumption of roles involves several issues related to the used programming language, the intelligence of the agents, and the knowledge needed to assume a new role. To our purposes, we propose an implementation of an infrastructure where the code of the mobile agents is modified at runtime, adding the features related to the role they are going to assume. In addition, a mechanism to search for roles is exploited to further uncouple agents and roles. We take into

G. Cabri, L. Ferrari, and L. Leonardi are with the Dipartimento di Ingegneria dell'Informazione - Università di Modena e Reggio Emilia, 41100 Modena ITALY (corresponding author to provide phone: +39-059-2056190; fax: +39-059-2056126; e-mail: cabri.giacomo@ unimo.it).

[1] The scene is about at 1 hour and 47 minutes.

consideration agents implemented in Java, for two main reasons: (i) Java is the most used language to implement (mobile) agent platforms, thank to its portability, security, and network-orientedness; and (ii) the fact that Java relies on an intermediate bytecode allows us to modify it (respecting the security constrains) to add new functionalities.
The paper is organized as follows. Section 2 introduces the BRAIN framework, briefly sketching the interaction model and the XML notation. Section 3 shows the mechanisms we have implemented to support the dynamic assumption of a role by agents, providing an example to explain the expressed concepts. Section 4 concludes the paper.

II. The BRAIN Framework

The BRAIN (Behavioural Roles for Agent INteractions) framework [3] is based on the concept of *role* and aims at covering the agent-based application development at different phases. To this purpose, it provides for a *model of interactions* that is based on roles, an XML-based *notation* to describe the roles, and *infrastructures* based on the previous model and relying on the previous notation, which support agents in the management of roles (see Fig. 1).

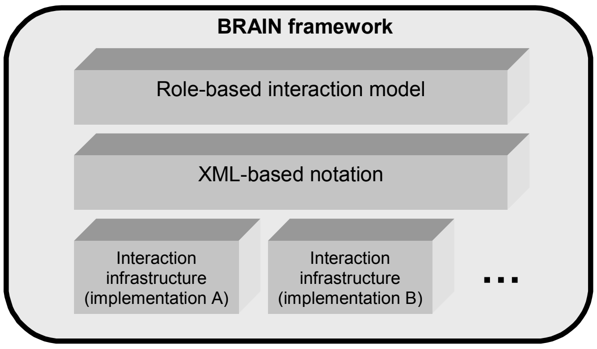

BRAIN framework
Role-based interaction model
XML-based notation
Interaction infrastructure (implementation A)
Interaction infrastructure (implementation B)
...

Fig. 1. The BRAIN framework

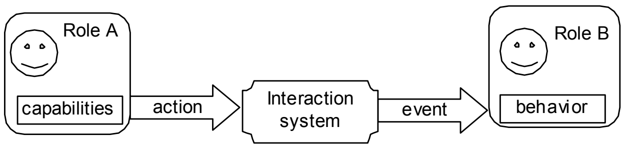


Fig. 2. The nteraction model in BRAIN

In BRAIN, a role is defined as a *set of capabilities* and an *expected behavior*. The former is a set of *actions* that an agent playing such role can perform to achieve its task. The latter is a set of *events* that an agent is expected to manage in order to "behave" as requested by the role it plays. Interactions among agents are then represented by couples (action, event), which are dealt with by the underlying interaction system, which is part of the BRAIN infrastructure; the interaction system can control interactions and enforce local policies, such as allowing or denying interactions between agents playing given roles. Fig. 2 shows how an interaction between two agents occurs. This model of interactions is very simple and very general, and well suits the main features of the agents: the actions can be seen as the concrete representation of *proactiveness* (i.e., the capability of carrying out their goals), while the events reify the *reactivity* (i.e., the capability of reacting to environment changes).
The notation proposed by BRAIN, called XRole [5], enables the definition of roles by means of XML documents; this grants 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 relies on the same information, granting continuity during the entire development. For instance, during the analysis phase, the analysts create XRole documents following the XML Schema proposed in the next section, which guides her in the definition of the role features. These XRole documents can be translated into HTML documents to provide high-level descriptions also for further uses. In the design phase, the same XRole documents can be translated into more detailed HTML documents to suggest functionalities of the involved entities. Finally, at the implementation phase, again the same XRole documents can be exploited to obtain Java classes that implement the role properties.
We have already implemented an infrastructure in BRAIN, called Rolesystem [7], which relies on abstract classes that represent roles. In that approach, an agent can register itself in the local interaction context with a particular role, and then can perform actions and manage events provided by the abstract class corresponding to the assumed role. Such infrastructure provides a degree of uncoupling between agents and roles. However, it does not enable agents to dynamically assume roles at runtime, and allows agents to directly access the role classes, introducing some security risks. This paper describes a new infrastructure that aims at overcoming the mentioned limitations.

III. Role Addition

To help us explaining the involved concepts, we propose as a concrete example the running of a conference. Let us suppose that two kinds of person attend the conference: the *listeners* and the *speakers*, which are the roles played by the attendees of the conference. For simplicity's sake, let us suppose that only one speaker could talk at a time, so that all the other persons are listeners. During the conference, as soon as the *speaker* has finished her speech, she let another person talk. This person, who a moment before was a *listener*, now becomes a *speaker*. This means that she has assumed the *speaker* role. Of course, with this role she can perform *speaker actions* like talking to microphone, showing slides, etc. By examining better the situation, we can point that (i) she has added to herself *speaker capabilities* and *behavior*, (ii) this addition has occurred *dynamically*, and (iii) she is recognized by other people as playing the speaker role.
Let us suppose that agents, which assume roles corresponding to that of their owners, support people attending the conference. The agent that assumes the *speaker* role (but it is the same for the *listener* role) must be provided with the appropriate capabilities and this should be made at runtime, when the capabilities are needed. In the following we explain how our approach enables the addition at runtime of members

(fields and methods) to Java classes that implement agents, thus allowing the dynamic assumption of roles.

A. Descriptors

Mobile agents can roam networks to carry out their tasks in the most appropriate site. When an agent is at a site and decides to play a role needed for its purpose, it asks the infrastructure which roles are available, chooses the one(s) that better suits its needs, and then assumes it (by the mechanism explained in the following). To grant a high level of abstraction in the decision process, we propose descriptors for roles, actions and events. A descriptor is an object that describes a role, an action or an event, for example with keywords, an aim, a version, a creation date and any further needed piece of information. A role descriptor describes what such role does but not how (with which operations) it is done. For the case of events, the descriptor describes the kind and the context of the occurred event, but not how to manage it (which is left to the agent). A role descriptor includes also the descriptors of the corresponding actions and events.

Using descriptors, the agent programmer does not need to know which is the physical class that implements a role, but only the descriptor of the role to search for. For example, if the agent must assume the *speaker role* the programmer could write code that search not for a speaker role but for a role with a speaker description (see Fig. 3). The agent can further verify the retrieved descriptor to be sure that the role is the right for it.

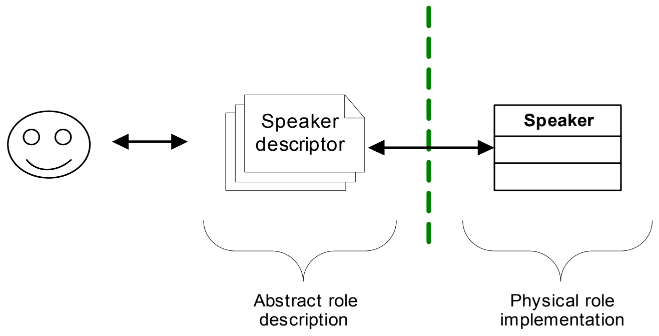


Fig. 3. An agent searching for the speaker role

Roles are described in XML exploiting the XRole notation of BRAIN. Fig. 4 reports the XML Schema to which XRole documents must be conform. From these XML documents, we derive the code of Java classes that concretely implement descriptors.

Note that the descriptors are needed also for other reasons, not only for relieving programmers of the knowledge of implementation details of the roles. In particular, a descriptor hides to the agent the physical location of the role implementation. Imagine what could happen if the agent can find the physical implementation of a role: it could use it without assuming it; in our implementation, this risk is avoid with an armored *role database*, which provides the role classes only to the role class loader, introduced in the following. Another reason is role composition: two or more different roles could be mapped into the same descriptor to indicate that a role is actually made of multiple implementations and the agent must assume all of them. This is very useful when a role is used to store information and not only actions. Finally, a further reason to use descriptors is that 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 about the work of role programmers, and viceversa, because the role behavior is described in a separate way. Moreover descriptors are a valid help for decision systems based on neural network [2] (not reported here due to page limitation).

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

<xsd:schema
xmlns:xsd="http://www.w3c.org/2001/XMLSchema"
  targetNameSpace="http://agentgroup.ing.unimo.it/rol
es/schema"
  xmlns:myns="http://agentgroup.ing.unimo.it/roles/sc
hema" >

<xsd:element name='role'>

  <xsd:complexType      name='GenericRoleDescription'
minOccurs='1'>

      <!-- the aim of the role -->
      <xsd:element    name='aim'    type='xsd:string'
maxOccurs='1' />

      <!-- English description of the role -->
      <xsd:element                  name='description'
type='xsd:string' maxOccurs='1' />

      <!-- the keywords of the role -->
      <xsd:element  name='keyword'  type='xsd:string'
minOccurs='0' />

      <!-- the name of the role -->
      <xsd:element  name='roleName'  type='xsd:string'
maxOccurs='1' />

      <!-- the version -->
      <xsd:element  name='version'  type='xsd:double'
maxOccurs='1' />

      <!-- the array of action descriptors -->
      <xsd:element             name='actionDescriptor'
type='myns:Actiondescription' minOccurs='0' />

      <!-- the array of event descriptors -->
      <xsd:element              name='eventDescriptor'
type='myns:Eventdescription' minOccurs='0' />

  </xsd:complexType>

  <!--action and event descriptors ... -->

</xsd:element>

</xsd:schema>
```

Fig. 4. The XML Schema of XML documents of the role descriptors

B. Adding Roles to Agents

While the descriptors provide a high-level description of the roles, we need also the corresponding code to be concretely added to the agents. In our approach the Java code of a role is composed of two parts: a Java interface (called *role interface*) and a Java class (called *role implementation*). The role implementations should implement the role interface but this is not mandatory. The role interface is used only for cast operators like instanceof.

The fact that an agent assumes a role means that the system adds each role implementation member (both methods and fields) to agent members, to adding the set of capabilities of the role; moreover, it forces the agent class to implement the role interface, to modify its expected behavior and to allow other agents to recognize it as playing that role (for instance, by means of the instanceof operator). Note that in Java there is no other way to add capabilities and to modify that expected behavior of an agent implemented by an already-defined class. Reconsider the conference example and see Fig. 5: it shows that an agent, to assume the role of *listener*, should inherit from two different classes because, often, the agent must inherit from a base agent class such as Aglet of the IBM Aglets platform [12]. Since Java does not allow multiple inheritance, we can only force the expected behavior by means of the interface implementation mechanism. This explains why we define also a Java interface.

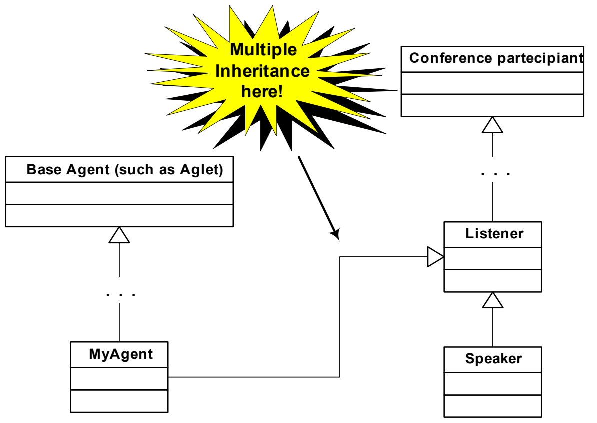
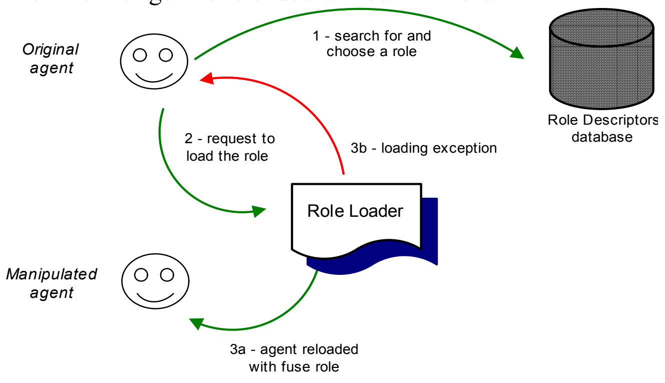


Fig. 5. Multiple inheritance needed to assume a role by agents

Our system is based on a special class loader, called "role loader", that could change agent behavior and external appearance. The idea is simple: an agent that wants to assume a new role asks our role loader to reload itself with the new role (see Fig. 6). If everything is right, the role loader sends the agent an event to indicate that the agent has been reloaded. After the reload event the agent can resume its execution. If the role loader is unable to load the role, it throws an exception that the original agent can catch. Analyzing this exception the agent can decide what to do (for example to retry or to choose another role). To release a role, the process is the same, but this time the agent is reloaded without that role.


Original agent
1 - search for and choose a role
Role Descriptors database
2 - request to load the role
3b - loading exception
Role Loader
Manipulated agent
3a - agent reloaded with fuse role
Fig. 6. The steps performed by an agent to assume a role

The role loader performs bytecode manipulation to add the role to the agent. This manipulation is completely made in memory without recompilation. The manipulation is needed to work with and to modify class definitions. Note that this manipulation is not dangerous for code portability and is compliant with the Java security manager.

Our implementation of role loader is based on *Javassist* bytecode manipulation engine [14]. The addition of the role's members to the agent class is performed in the following steps (suppose that the agent has yet contacted the role loader):

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

In the following we explain each of the above steps and their aim in detail.

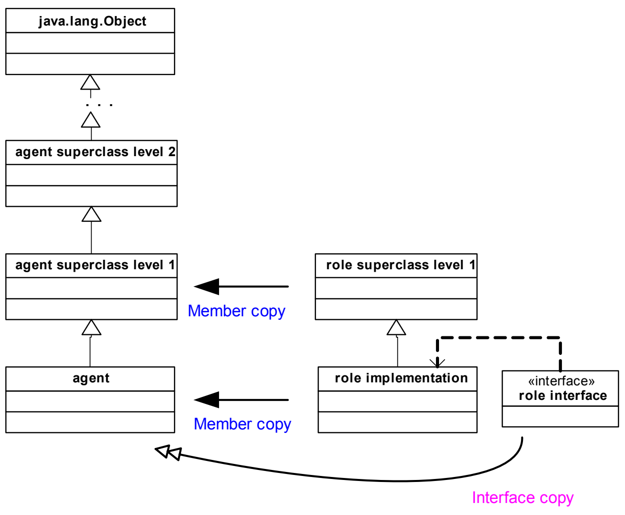


Fig. 7. Copy of the members from role classes to agent classes

The first step is needed to grant role-inherited properties. In fact, a role could be not a single class but the bottom (or the middle) of an inheritance chain. For example the role *listener* could inherit properties from the role *participant*. To grant that the role will work in the right way, every role superclass (that is every class at any level on the role inheritance chain) must be added to the agent classes at the corresponding level. In fact, a subclass role implementation expects to find some capabilities on its superclasses, so we must grant this condition will remain true. To better explain this concept, we refer to Fig. 7. The figure shows the inheritance chains of an agent and its role. Both the role and the agent are represented by the bottom of their respective chain. This means that the bottom classes must be fused. At the superclass level the same must occur, that is the superclasses must be fused also. This must be done for each chain level. Note that the role superclasses are less than the agent ones. In this way our system grants that both the role and the agent, after the fusion, will continue using inherited properties; in other words, the Java's super

operator will work well. This step does not do anything except calculate the inheritance stack, that is how a role class and an agent class must be fused and at what level.
Referring to Fig. 7, the computed 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 fused into one. Something similar is made to force the class to implement (in the Java sense) the role interface. The inheritance stack will be used in the second step to know from which class members will be copied (fuse) on the agent chain.

TABLE I.
THE CLASS STACK CALCULATED BY THE ROLE LOADER

Agent inheritance chain	Role inheritance chain
Java.lang.Object	None
…	…
agent superclass level 3	none
agent superclass level 2	role superclass level2
agent superclass level 1	role superclass level 1
Agent	role implementation

In fact, the second step does this copy consulting the inheritance stack and then copying every member from the role chain into the agent chain on 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 original agent. In our implementation only adding mechanism is provided and this grant a correct execution to the agent.

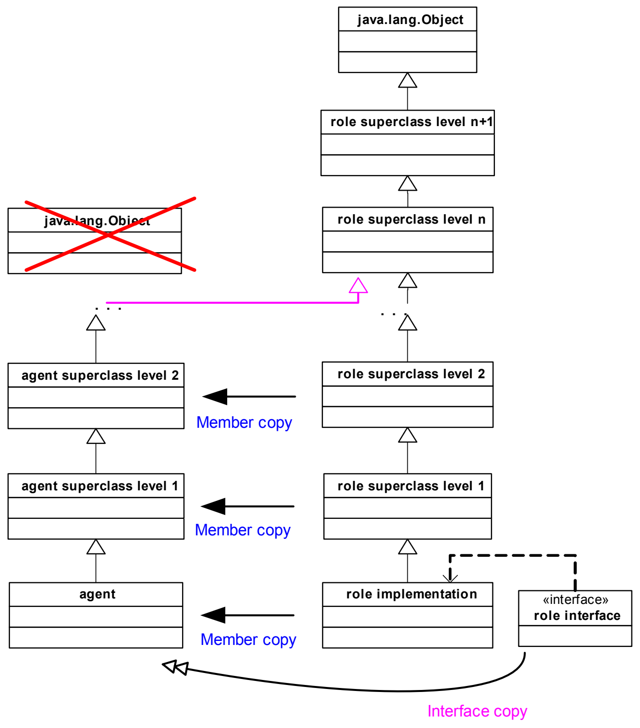


Fig. 8. Copy of the members from role classes to agent classes when the role chain is longer than the agent chain

At the end, in the last step, every value is copied from the original agent (and its superclasses) to the new created agent. This step grants that the agent state will not be lost during the reloading process.
Note that a special case could happen: the role inheritance chain can have more levels of the agent's one. This situation is different from the previous one, because the addition of methods is not symmetric, so it has a well-defined direction (from role to agent). How this situation is dealt with is reported in Fig. 8. In this case the computed stack removes the java.lang.Object class from the agent chain, and substitutes it with the first exuberant class of the role chain. Otherwise, the exuberant classes of the role chain would be lost. Remember that member copies occur at the same level, but over the java.lang.Object class there is (and could not be otherwise) nothing else and so the exuberant classes could not be copied on agent's chain levels. For this reason the agent's chain is changed, and is attached to the role's one. Note that no modification is made on the java.lang.Object class at top of the agent's chain, to grant code portability.
Note that during the fusing process there could be duplicate members (both variables and methods), i.e., which are in different classes but have the same signature. If the duplicate member is a variable, the role loader simply chooses one of them and discards the other one. For the methods, this duplication causes problems, because they can have different bodies, and a coexistence is not possible. This problem could be avoided by renaming the duplicated members during the agent and role fuse process. The rename operation is needed to grant that the JVM specification (see [10] chapter 4) is respected. Another way to avoid the duplicated member trouble is to inform the agent that the role assumption could be not correct (if no rename mechanism is provided) with a warning mechanism. In our implementation it is possible to not copy duplicated members and inform the agent by means of warnings (that are different from exceptions). By analyzing these warnings, the agent can understand what to do.

C. *Role Use by Agents*

In this subsection we explain how the agent can use a dynamically assumed role, without knowing its methods but only its descriptor. This problem concretely means: how can an agent invoke a method that before role assumption it did not have and after it has? Note that this issue is essential because we must grant to the programmers the capability of correctly compiling the agent code. In fact, the programmer does not know anything about the role implementation but know, indirectly, about which actions can be used, by the action descriptors, and about which events can occur, by the event descriptors. In the following, we focus on the action use, because the management of the events is similar and simpler.
The use of descriptors means that the programmer cannot write code that invokes methods corresponding to role actions in the usual way, because, since the agent has not those methods yet, a compile-time error would occur. Therefore, there must be an *invocation translator* that can do introspection on the agent to find which action (i.e., which method) must be call to response to an action description. If the agent takes with it an invocation translator, it can invoke role actions simply specifying the action description to its translator. The translator then performs introspection on the agent to find (if loaded) the

action (in terms of method) to invoke and then uses the Java reflection API to invoke it.

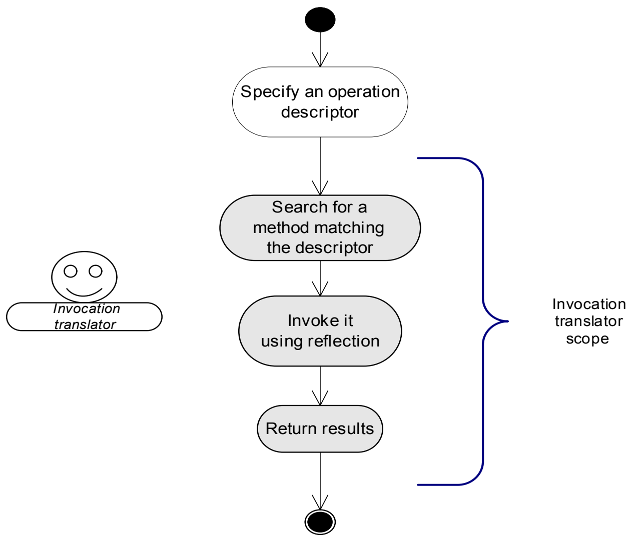

Specify an operation descriptor
Search for a method matching the descriptor
Invoke it using reflection
Return results
Invocation translator
Invocation translator scope

Fig. 9. State chart about a role-action invocation

Fig. 9 shows a simple state chart describing what happens when the agent invokes a role action. The agent specifies to the translator a descriptor of the action that wants to perform, the translator searches for a method that corresponds to the description and then invokes it. Note that the invocation translator is not a separated module of the agent but it is a component of the agent. This is not a trouble for mobile agents because the invocation translator is a very small module that does not affect agent weight for movements. Fig. 10 shows some code examples of role use, where the agent searches for appropriate role and action among the retrieved descriptors; our system makes available “filters” that allow partial descriptions and apply research rules.

IV. Conclusion

This paper has presented the implementation of mechanisms to enable the dynamic assumption of roles by Java agents at runtime. This is achieved by modifying the bytecode of the agents, adding the features of the role(s) they want to play.

In addition to the advantages deriving from a role-based approach, the specific ones of our system can be summarized as follows:

- It enables agents to dynamically assume roles at runtime, granting flexibility and adaptability. Roles are not simply given to the agents, but 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 grants 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 separation of concerns between agent issues and role issues, and also roles and agents can be implemented separately and joint at runtime, avoiding that agent programmers need to know role details and viceversa. Also, programmers do not need even the role libraries, because they can compile their agent and the binding with the role code is completely dealt with at runtime.

As a final note, we stress that the choice of a programming language that allows multiple inheritance, such as C++, would have simplified the effort to fuse two classes, but would not have gained the advantages of the Java language (which imposes single inheritance), in terms of portability and compliance with the existing mobile agent platforms.

```
public  class  MyAgent  extends  Aglet  implements
RoleSupport
{ // execution method
  public void run()
  { int r, a;
    //  get  all  the  role  descriptors  stored  into
database
    RoleDescriptor
roleDesc[]=RoleDatabase.getAllDescriptors();

  //  search  for  the  descriptor  that  represents  a
speaker role
  for(int k=0;k<roleDesc.length;k++)
  { //  check  descriptors  values  like  keywords,
version, date
    // actions and so on
    ...
    r = k;
    }
  //  the  descriptor  at  index  r  is  the  speaker
descriptor,
  // assume the corresponding role
  //  and  store  the  role  action  descriptors  in  an
array
  try
  {  RoleLoader.addRoleToMyself(roleDesc[r]);  }
  catch(RoleLoaderException e)
  {   System.err.println("Exception   during   role
loading!");
   }
  ...
    // load the action descriptors
    actDesc=roleDesc[r].getActionDescriptors();
    // try to invoke the "turn-on projector" action
    //  assume  that  the  array  "actDesc"  contains  the
action
    // descriptors
  //  search  for  the  descriptor  that  represents  the
action
  for(int k=0;k<actDesc.length;k++)
  { //  check  descriptors  values  like  keywords,
version, date
    // actions and so on
    ...
    a = k;
    }
    //  invoke  the  act  method  of  the  RoleSupport
interface
    Object ret = this.act(actDesc[a]);
    ...
    // release the role
    RoleLoader.releaserole(roleDesc[r]);
  }
}
```

Fig. 10. Fragment of code of an agent that assumes the speaker role

Acknowledgement

Work supported by the NOKIA Research Center of Boston and by the Italian MURST within the project “MUSIQUE - Multimedia Ubiquitous Service Infrastructure in a QoS Universal Environment”.

REFERENCES

[1] Y. Aridor, D. Lange, “Agent Design Pattern: Elements of Agent Application design”, in Proceedings of the International Conference on Autonomous Agents, ACM Press, 1998.

[2] J. P. Bigus, J. Bigus, “Constructing intelligent agents with Java – A Programmer’s guide to smarter applications”, John Wiley & Sons Inc., 1998.

[3] The BRAIN project, http://www.agentgroup.unimo.it/MOON/BRAIN/index.html

[4] G. Cabri, “Role-based Infrastructures for Agents”, in Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS 2001), Bologna (I), October 2001.

[5] G. Cabri, L. Leonardi, F. Zambonelli, “XRole: XML Roles for Agent Interaction”, in Proceedings of the 3rd International Symposium “From Agent Theory to Agent Implementation”, at the 16th European Meeting on Cybernetics and Systems Research (EMCSR 2002), Wien (A), April 2002.

[6] G. Cabri, L. Leonardi, F. Zambonelli, “Separation of Concerns in Agent Applications by Roles”, in Proceedings of the 2nd International Workshop on Aspect Oriented Programming for Distributed Computing Systems (AOPDCS 2002), at the International Conference on Distributed Computing Systems (ICDCS 2002), Wien (A), July 2002.

[7] G. Cabri, L. Leonardi, F. Zambonelli, “Implementing Role-based Interactions for Internet Agents”, in Proceedings of the 2003 International Symposium on Applications and the Internet (SAINT 2003), Orlando, Florida, USA, January 2003.

[8] B. Demsky, M. Rinard, “Role-Based Exploration of Object-Oriented Programs”, in Proceedings of the International Conference on Software Engineering 2002, Orlando, Florida, USA, May 19-25, 2002.

[9] M. Fowler, “Dealing with Roles”, http://martinfowler.com/apsupp/roles.pdf, 1997.

[10] T. Lindholm, F. Yellin, “The Java Virtual Machine Specification Second Edition”, http://java.sun.com/docs/books/vmspec/

[11] E. A. Kendall, “Role Modelling for Agent Systems Analysis, Design and Implementation”, IEEE Concurrency, Vol. 8, No. 2, pp. 34-41, April-June 2000.

[12] D. B. Lange, M. Oshima, “Programming and Deploying Java™ Mobile Agents with Aglets™”, Addison-Wesley, Reading (MA), August 1998.

[13] R. S. Sandhu, E. J. Coyne, H. L. FeinStein, C. E. Youman, “Role-based Access Control Models”, IEEE Computer, Vol. 20, No. 2, pp. 38-47, 1996.

[14] M. Tatsubori, T. Sasaki, S. Chiba, K. Itano, “A Bytecode Translator for Distributed Execution of "Legacy" Java Software”, in Proceedings of ECOOP 2001, LNCS 2072, Springer Verlag, 2001.

[15] A. Tripathi, T. Ahmed, R. Kumar, S. Jaman, “Design of a Plicy-Driven Middleware For Secure Distributed Collaboration”, in Proceedings of the 22nd International Conference on Distributed Computing System (ICDCS), Vienna (A), July 2002.

[16] M. Weiser, “Hot Topics: Ubiquitous Computing”, IEEE Computer, Vol. 26, No. 10, October 1993.