A METHODOLOGY FOR ROLE-BASED MODELING OF OPEN MULTI-AGENT SOFTWARE SYSTEMS

Haiping Xu and Xiaoqin Zhang
Computer and Information Science Department
University of Massachusetts Dartmouth
North Dartmouth, MA 02747
Email: {hxu, x2zhang}@umassd.edu

Keywords: Role-based modeling, Open multi-agent software systems, Object-Z formalism, A-R mapping.

Abstract: Multi-agent systems (MAS) are rapidly emerging as a powerful paradigm for modeling and developing distributed information systems. In an open multi-agent system, agents can not only join or leave an agent society at will, but also take or release roles dynamically. Most of existing work on MAS uses role modeling for system analysis; however, role models are only used at conceptual level with no realizations in the implemented system. In this paper, we propose a methodology for role-based modeling of open multi-agent software systems. We specify role organization and role space as containers of conceptual roles and role instances, respectively. Agents in an agent society can take or release roles from a role space dynamically. The relationships between agents are deduced through a mechanism called A-R mapping. As a potential solution for automated MAS development, we summarize the procedures to generate a role-based design of open multi-agent software systems.

1 INTRODUCTION

Multi-agent systems (MAS) are rapidly emerging as a powerful paradigm for modeling and developing distributed information systems. However, to specify and design multi-agent systems is not an easy task. Methodologies for developing multi-agent systems are therefore proposed to provide software engineers guidelines to develop MAS in a systematic manner. Among them, role-based analysis and design is one of the most effective methodologies for agent-based system analysis and design. Most of the existing work defines roles as conceptual units that only happen in the analysis phase. The roles abstracted from use cases are abstract constructs used to conceptualize and understand the system. They have no realizations in the implemented system after the analysis stage. In most of the cases, all roles are atomic constructs and cannot be defined in terms of other roles (Juan et al., 2002). This approach is feasible when designing small-scale, closed system, especially when an agent only takes a single role. However, in an open multi-agent system, agents can not only join or leave an agent society at will, but also take or release roles dynamically. When an agent takes more than one role, and further more, if an agent takes or releases roles at run time, this approach becomes inappropriate. This is because when role assignments are dynamic, the interaction relationships between agents become quite complicated, and usually they cannot be determined at design time. To develop an open and dynamic multi-agent system, it becomes vital for us to introduce the concept of role instance (a concrete implementation of a conceptual role) into the development process, and to design algorithms to deduce agent interaction relationships from role assignments and role relationships. In this paper, we propose a methodology for role-based modeling of open multi-agent systems. In our approach, we define a role organization that provides the ontology for modeling roles and their relationships. A role space is then defined as a container of role instances, as well as a middleware for agents to find the appropriate role instances from the role space. Based on the concepts of role organization and role space, we propose that agent society consists of a set of agents, which may take or release roles from a corresponding role space dynamically. The relationship between agents in an agent society can be deduced through a mechanism called *A-R mapping*. To support automated software development, we also propose a development process to design role-based open MAS.

There are two main strands of work to which our research is related, i.e., work on formal modeling of agent systems, and work on role-based methodology for development of multi-agent systems. Previous work on formal modeling of agent systems has been based on formalisms, such as Z, temporal logic, and Petri nets, to specify agent systems or agent behaviors. Luck and d'Inverno tried to use the formal language Z to provide a framework for describing the agent architecture at different levels of abstraction (Luck and d'Inverno, 1995). They proposed a four-tiered hierarchy comprising entities, objects, agents and autonomous agents. Fisher's work on Concurrent METATEM used temporal logic to represent dynamic agent behavior (Fisher, 1995). Xu and Shatz proposed the agent-oriented G-nets, which is a high-level formalism of Petri net, to model and verify multi-agent behaviors by using existing Petri net tools (Xu and Shatz, 2003). More recently, a formalism called OZS, which is a combination of Object-Z and statecharts, is used to specify multi-agent systems (Hilaire et al., 2004). With this approach, Object-Z is used to specify the transformational aspects, and statecharts are used to specify the reactive aspects of an MAS.

In summary, formal methods are typically used for specification of agent systems and agent behaviors. Existing work in this direction either does not directly use role modeling for agent design, or uses role modeling simply as conceptual guidelines for agent development during the analysis phase. In contrast, we propose our formal role-based open multi-agent system framework, where role classes can be explicitly instantiated, and role instances can be taken or released by agents at run time.

A second strand of related work is to propose role-based methodologies for development of multi-agent systems. Typical examples of such efforts include the Gaia methodology (Wooldridge et al., 2000) and Multiagent Systems Engineering (MaSE) methodology (DeLoach et al., 2001). The Gaia methodology models both the macro (social) aspect and the micro (agent internals) aspect of the multi-agent system. The methodology covers the analysis phase and the design phase. Specifically, in the analysis phase, the role model and interaction model are constructed. Based on the analysis models, in the design phase, three models, i.e., the agent model, service model and acquaintance model, are constructed during the initial design of the system, and then are refined during the detailed design phase using conventional object-oriented methodology. Similarly, the MaSE methodology is a specialization of more traditional software engineering methodologies. During the analysis phase of the MaSE methodology, a set of roles are produced, which describes entities that perform some function within the system. In MaSE, each role is responsible for achieving, or helping to achieve specific system goals and subgoals. During the design phase, agent classes are created from the roles defined in the analysis phase. In other words, roles are the foundation upon which agent classes are designed, and thus, the design of agent classes depends on role specifications. In our proposed approach, agent components and role components are loosely coupled, where agents take or release roles at run time without having the knowledge of the internal structure of role instances. Consequently, role classes and agent classes can be designed at the same time. Thus, the specification and design of agent classes can be significantly simplified.

The rest of this paper is organized as follows: Section 2 presents a role-based MAS specification using Object-Z formalism. It describes a three layered system model for development of open MAS. The three layers are *role organization*, *role space* and *agent society*. Section 3 first presents the design of role organization in terms of role relationships, then it summarizes a development process for role-based open MAS, and discusses how agents from an agent society take or release roles in a corresponding role space, and how to build up agent interaction relationships using the *A-R mapping* mechanism. Finally, in Section 4, we provide conclusions and our future work.

2 ROLE-BASED SPECIFICATION

2.1 An Organizational Approach

To facilitate the design of open MAS, we explicitly separate the concepts of *role organization* and *role space* that consist of *conceptual roles* and *role instances*, respectively. A role organization is defined at a conceptual level, in which roles have relationships such as *inheritance*, *aggregation*, *association* and *incompatibility*. On the other hand, a role space consists of role instances, which are concrete implementations of conceptual roles, and can be taken or released by agents at run time. A three-layered general model of role-based open multi-agent systems is illustrated as in Figure 1.

As shown in Figure 1, the role organization defines a set of conceptual roles and their relationships. For example, *role_B* and *role_C* are defined as subclasses of *role_A*. *Role_D* is defined as a part of *role_C*, which implies that *role_C* views *role_D*'s responsibilities and capabilities as part of its own. *Role_D* and *role_E* have an association relationship, where *role_D* and *role_E* may be responsible for providing certain information to each other when there are such requests. In addition, *role_D* has a reflective association relationship to itself, for example, when *role_D* represents a type for team members, team members are required to discuss on certain topics.

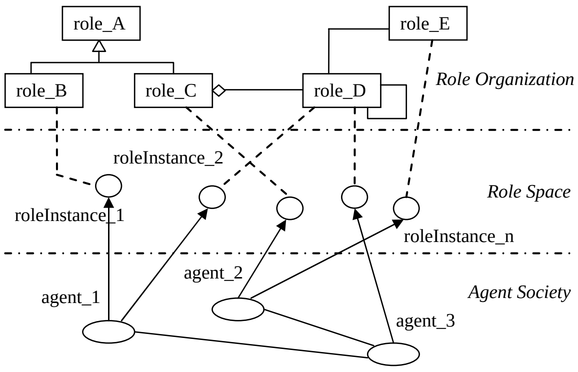

Figure 1: A general model of role-based open MAS

At the second layer, we define a role space that consists of role instances. Each role instance must be of a role type defined in its corresponding role organization. For example, *roleInstance_2* is of type *role_D*. The relationships between role instances can be easily derived from their class relationships. Therefore, it is not necessary to explicitly show their relationships at this layer.

At the third layer, we define an agent society that consists of agent instances. Agents are free to join or leave the agent society, and they can take one or more than one role instances from the role space. For example, *agent_1* takes two role instances, i.e., *roleInstance_1* and *roleInstance_2*, which are of type *role_B* and *role_D*, respectively. An agent can not only take roles at run time, but also release them if they are not needed any more. The relationships between agents are based on the relationships between roles that are taken. For example, *agent_1* and *agent_3* have an interaction relationship because *role_D* has a reflective association relationship with itself; *agent_2* and *agent_3* have an interaction relationship because *role_D* and *role_E* have an association relationship. Notice that relationships of *inheritance* and *aggregation* between roles are not carried down to agent relationships.

Since agents take roles dynamically, agents are not designed based on role modeling. In other words, the development of agents can be totally independent of the development of roles. Thus, as one of the major advantages of our approach, the components of agents and roles are loosely coupled, and practically, they can be developed independently by different groups of people, for example, two different companies.

2.2 A Formal Model for Open MAS

To specify our proposed role-based model of open multi-agent systems, we build a framework using Object-Z formalism (Duke et al., 1995), which is an extension to Z formal specification language for modular design of complex systems. The framework is composed of a set of classes that specify the basic concepts including *Role*, *RoleOrgnaization*, *RoleSpace*, *Agent* and *AgentSociety*. We now provide a few key definitions giving the formal structure of our role-based open MAS models.

Definition: A *role*, or a *conceptual role*, is defined as a template of role instances that has *attributes*, *goals*, *plans*, *actions*, *permissions* and *protocols*. A *role instance* is a fully instantiated role.

The class schema *Role* can be formally defined based on its state schemas and operation schemas as follows:

$Role$

$$\begin{array}{l} attributes : \mathbb{P}\ Attribute \\ goals : \mathbb{P}\ Goal \\ plans : \mathbb{P}\ Plan \\ actions : \mathbb{P}\ Action \\ permissions : \mathbb{P}\ Permission \\ protocols : \mathbb{P}\ Protocol \\ beTaken : \mathbb{B} \end{array}$$

$\textsc{Init}$

$$\begin{array}{l} permissions = \varnothing \\ protocols = \varnothing \\ beTaken = false \end{array}$$

$setPermission$

$$\begin{array}{l} \Delta permissions \\ perm? : Permission \\ \hline permissions' = permissions \oplus \{perm?\} \end{array}$$

$addProtocol$

$$\begin{array}{l} \Delta protocols \\ prot? : Protocol \\ \hline procotols' = protocols \cup \{prot?\} \end{array}$$

The *Role* class consists of a state variable *attributes* that represents a set of role attributes, whose elements are of type *Attribute*. The attributes of a role describe the characteristic properties of a role, including role name and role identification. A *Role* is defined to have a set of goals and a set of plans, as well as a set of actions. The state variable *goals* describe a set of goals of type *Goal*, which consists of a goal set that specifies the goal domain and goal states of a role. The state variable *plans* represent a set of plans of type *Plan*. A plan describes how to achieve a goal or subgoal by executing several actions in a specified order. Each plan is associated with a goal or a subgoal; however, a goal or subgoal may associate with more than one plan, and the most suitable one will be selected to achieve that goal or subgoal by the agent who takes this role according to the run-time circumstance. To carry out a certain plan, a role has the capability of performing some actions. The state variable *actions* refer to a set of actions of type *Action*, which this role is capable to execute. A role has a set of permissions when realizing goals or subgoals. The state variable *permissions,* whose elements are of type *Permission,* describe the resources that are available to that role and the accessing rights of that role for information when achieving a goal or subgoal. For example, a role may have the right to read a particular piece of information, to modify it, or even to generate new information. The state variable *protocols* define a set of protocols of type *Protocol*, which describes the way how a role may interact with other roles. An example of such protocol is the contract net protocol (Smith, 1980). Finally, the Boolean state variable *beTaken* defines if a role instance has already been associated with an agent. A *true* value indicates that a role instance has already been taken by an agent, and thus, it is not available for other agents.

The concept of *role instance*, i.e., an instantiated role, is similar to the concept of *object*, which is an instantiated entity of a *class*. Notice that, although a role instance has certain goals, plans and actions, it does not have the responsibility to choose the most appropriate plan and the corresponding actions to achieve a certain goal or subgoal. Instead, such activities are the responsibility of agents.

To support role behavior changing at run time, *permissions* can be modified and new *protocols* can be added to a role at run time. This is achieved by providing the operations of *setPermission* and *addProtocol* in the *Role* class schema.

Definition A *role organization* defines a set of conceptual roles and the relationships between these conceptual roles.

Before we can define the class schema *RoleOrganization*, we must first define the types of *RoleMetaClass* and *Relationship*. A *metaclass* is a class whose instances are classes. Every class has a metaclass, of which it is the sole instance. The *RoleMetaClass* specifies the *Role* class in terms of its attributes and behaviors. Therefore, an instance of type *RoleMetaClass* is the *Role* class. The *Relationship* type is defined as [*inheritance* | *aggregation* | *association* | *incompatibility*].

As shown in the class schema *RoleOrganization* below, the state variable *roles* are defined as a set of elements of type *RoleMetaClass* or its derivatives, thus *roles* refers to a set of subclasses of the *Role* class and the *Role* class itself. Accordingly, the function *relationship* is defined as relationships between classes (roles) instead of objects (role instances). Such relationships include *inheritance* relationship, *aggregation* relationship, *association* relationship and *incompatibility* relationship. We will describe them in details in Section 3.1. The *Role* class is the root class of all its descendents, and it exists at the very beginning when creating the role organization. New role classes can be added into the role organization. When a new class *role?* is added, the *inheritance* relationship between *role?* and its superclass *r* must be added too. This is automatically achieved by updating the function *relationship* by adding a mapping of {*(role?, r)* ↦ *inheritance*}. Relationships other than the *inheritance* relationship between role classes must be set up by applying the operation *setRelationship.*

RoleOrganization

$roles : \mathbb{P} \downarrow RoleMetaClass$
$relationship :$
$\quad \downarrow RoleMetaClass \times \downarrow RoleMetaClass \nrightarrow Relationship$

$\forall r1, r2 \in roles, r1 \neq r2 \bullet (r1, r2) \in dom\ relationship$

INIT

$roles = \{Role\}$

addRole

$\Delta roles, relationship$
$role? : \downarrow RoleMetaClass$

$role? \notin roles \ \wedge \ roles' = roles \cup \{role?\}$
$\exists r \in roles \bullet relationship' =$
$\qquad relationship \cup \{(role?, r) \mapsto inheritance\}$

setRelationship

$\Delta relationship$
$r1?, r2? : \downarrow Role$
$rela? : Relationship$

$r1?, r2? \in roles \wedge rela? \neq inheritance \ \wedge$
$\quad relationship' = relationships \oplus \{(r1?, r2?) \mapsto rela?\}$

Definition A *role space* is a container of a set of role instances that are of types defined in a *role organization*. Each role space corresponds to a single role organization; however, a role organization can be mapped to a set of similar role spaces. Role instances can be added into or deleted from a role space dynamically. A role space provides services to create or delete role instances as well as to find a certain role instance according to role attributes.

A *role space* is defined upon a *role organization*. As shown in the class schema *RoleSpace* below, we define *roleOrganization* as a global instance of type *RoleOrganization*, in which the number of role classes must be more than one. If the *roleOrganization* is modified, the *role space* must be updated accordingly in order to be consistent with the conceptual roles and role relationships defined in the *roleOrganization*. For example, when a certain conceptual role *cr* is deleted from the *roleOrganization* (for simplicity, the operation schema *deleteRole* is not defined in the class schema *RoleOrganization*), any role instances of type *cr* must also be deleted. This is important because later on, we will see that an *agent society* is also defined based on the role types and class relationships from a *role organization*. Therefore, this ensures that the types of role instances in a role space are always consistent with that of role instances an agent may take.

RoleSpace

$\mathit{roleOrganization} : \mathit{RoleOrganization}$

$\#\mathit{roleOrganization.roles} > 1$

$\mathit{roleInstances} : \mathbb{P} \downarrow \mathit{Role}$

$\forall\, \mathit{ri} \in \mathit{roleInstances} \bullet \mathit{ri.getClass} \in \mathit{roleOrganization.roles}$

$\textsc{Init}$

$\mathit{roleInstances} = \varnothing$

createRoleInstance

$\Delta \mathit{roleInstances}$
$\mathit{ri}? : \downarrow \mathit{Role}$

$\mathit{ri}?.\mathit{getClass} \in \mathit{roleOrganization.roles}$
$\mathit{ri}? \notin \mathit{roleInstances} \wedge \mathit{roleInstances}' = \mathit{roleInstances} \cup \{\mathit{ri}?\}$

deleteRoleInstance

$\Delta \mathit{roleInstances}$
$\mathit{ri}? : \downarrow \mathit{Role}$

$\mathit{ri}? \in \mathit{roleInstances} \wedge \mathit{roleInstances}' = \mathit{roleInstances} - \{\mathit{ri}?\}$

findRoleInstance

$\Xi \mathit{roleInstances}$
$\mathit{ra}? : \mathit{Role.Attribute}$
$\mathit{ri}! : \downarrow \mathit{Role}$

$(\mathit{NotFound} \wedge \mathit{ri}! = \mathit{null}) \vee$
$\quad \exists\, \mathit{ri} \in \mathit{roleInstances} \bullet \mathit{ri.attributes} = \mathit{ra}? \ \wedge\ \mathit{ri}! = \mathit{ri}$

The state variable *roleInstances* refers to a set of role instances of type *Role* or its derivatives, which must have already been defined in the *roleOrganization*. Initially, the *role space* contains zero role instances. Role instances can be added into or deleted from a role space dynamically. In addition, a role space also serves as a middleware for agents to find appropriate role instances according to role attributes to fulfill their motivations.

Definition An *agent* or an *agent class* is defined as a template of agent instances that has *attributes*, *motivations*, *sensor*, *reasoningMechanism*, and a reference *rolesTaken* to a set of role instances. An *agent instance* is a fully instantiated agent.

As shown in the following class scheme *Agent*, an agent has attributes such as agent name, agent owner and agent identification. An agent also has motivations of type *Motivation*. A motivation is defined as any desire or preference that can lead to the generation and adoption of goals and affect the outcome of the reasoning or behavioral task intended to satisfy those goals (Luck and d'Inverno, 1995). The *sensor* of an agent perceives related environment changes of type *Environment* and transforms them into a set of sensor data. The *reasoningMechanism* is defined as a composite function that takes a set of sensor data and a set of motivations as arguments and maps them to a set of goals and subgoals. Based on the goals and subgoals, the function further derives a set of needed roles. The agent then takes each needed role from a role space to fulfill its motivations. Notice that the function *reasoningMechanism* in reality is more complicated than what we have defined, e.g., it also provides the functionality to choose plans according to a set of sensor data and a set of goals and subgoals. A more sophisticated definition of the *reasoningMechanism* is beyond the scope of this paper. The state variable *rolesTaken* refers to a set of roles that are taken by the agent. The *Agent* class schema defines two key operations: *takeRole* and *releaseRole*. The *takeRole* operation takes an available role instance from a role space, and set it to be unavailable to other agents. On the other hand, the *releaseRole* operation releases a role instance and set it to be available to other agents.

Agent

$\mathit{attributes} : \mathbb{P}\, \mathit{Attribute}$
$\mathit{motivations} : \mathbb{P}\, \mathit{Motivation}$
$\mathit{sensor} : \mathit{Environment} \rightarrowtail \mathit{SensorData}$
$\mathit{reasoningMechanism} :$
$\quad \mathbb{P}\, \mathit{SensorData} \times \mathbb{P}\, \mathit{Motivation} \rightarrow \mathbb{P}\, \mathit{Goal} \rightarrow \mathbb{P} \downarrow \mathit{Role}$
$\mathit{rolesTaken} : \mathbb{P} \downarrow \mathit{Role}$

$\textsc{Init}$

$\mathit{rolesTaken} = \varnothing$

takeRole

$\Delta \mathit{rolesTaken}$
$\mathit{ri}? : \downarrow \mathit{Role}$

$\mathit{ri}?.\mathit{beTaken} = \mathit{false} \wedge \mathit{ri}?.\mathit{beTaken}' = \mathit{true}$
$\mathit{ri}? \notin \mathit{rolesTaken} \wedge \mathit{rolesTaken}' = \mathit{rolesTaken} \cup \{\mathit{ri}?\}$

releaseRole

$\Delta \mathit{roleTaken}$
$\mathit{ri}? : \downarrow \mathit{Role}$

$\mathit{ri}?.\mathit{beTaken} = \mathit{true} \wedge \mathit{ri}?.\mathit{beTaken}' = \mathit{false}$
$\mathit{ri}? \in \mathit{rolesTaken} \wedge \mathit{rolesTaken}' = \mathit{rolesTaken} - \{\mathit{ri}?\}$

Definition An *agent society* is defined upon a role organization and consists of a set of agent instances that are of type *Agent*. An agent society provides services to create or delete agent instances such that agent instances can be added into or deleted from an agent society dynamically.

The structure of agent society is often determined by organizational design which is independent of the agents themselves (Dastani et al., 2003). As shown in the *AgentSociety* class schema below, the *AgentSociety* class is defined upon a *roleOrganization* of type *RoleOrganization*. Since both role spaces and agent societies are built on role organizations, there is a one to one mapping between a role space and an agent society when they share the same role organization. This implies that role instances created in one role space can only be taken by agents that belong to its corresponding agent society; meanwhile, any agent belongs to an agent society must take at least one role from a corresponding role space; otherwise, it shall leave the agent society eventually. Note that this does not mean an agent can take roles only from one role space. In contrast, an agent may join more than one agent societies and take roles from different role spaces.

$$
\begin{array}{l}
\textit{AgentSociety} \\
\hline
\quad \textit{roleOrganization} : \textit{RoleOrganization} \\
\quad \hline
\quad \#\textit{roleOrganization.roles} > 1 \\
\\
\quad \hline
\quad \textit{agentInstances} : \mathbb{P}\, \textit{Agent} \\
\quad \textit{interaction} : \textit{Agent} \times \textit{Agent} \rightarrowtail\!\!\!\!\!\!\!+\; \textit{Message} \\
\quad \hline
\quad \forall\, a \in \textit{agentInstances}, \exists\, r \in a.\textit{rolesTaken} \bullet \\
\quad\quad r.\textit{getClass} \in \textit{roleOrganization.roles} \\
\quad \forall\, a1, a2 \in \textit{agentInstances}, a1 \neq a2, \\
\quad\quad \exists\, r1 \in a1.\textit{rolesTaken}, \exists\, r2 \in a2.\textit{rolesTaken}, \\
\quad\quad \textit{roleOrganization.relationship}(r1.\textit{getClass}, r2.\textit{getClass}) \\
\quad\quad\quad = \textit{association} \bullet (a1, a2) \in \textit{dom interaction} \\
\quad \hline
\\
\quad \textsc{Init} \\
\quad \hline
\quad \textit{agentInstances} = \varnothing \\
\quad \hline
\\
\quad \textit{createAgentInstance} \\
\quad \hline
\quad \Delta \textit{agentInstances} \\
\quad \textit{agent}? : \textit{Agent} \\
\quad \hline
\quad \textit{agent}? \notin \textit{agentInstances} \wedge \\
\quad\quad \textit{agentInstances}' = \textit{agentInstances} \cup \{\textit{agent}?\} \\
\quad \hline
\\
\quad \textit{deleteAgentInstance} \\
\quad \hline
\quad \Delta \textit{agentInstances} \\
\quad \textit{agent}? : \textit{Agent} \\
\quad \hline
\quad \forall\, r \in \textit{agent}?.\textit{rolesTaken} \bullet \textit{agent}?.\textit{releaseRole}(r) \\
\quad \textit{agent}? \in \textit{agentInstances} \wedge \\
\quad\quad \textit{agentInstances}' = \textit{agentInstances} - \{\textit{agent}?\} \\
\quad \hline
\\
\hline
\end{array}
$$

An *agent society* contains a set of agent instances of type *Agent*, referred to by the state variable *agentInstances*. The variable *interaction* is defined as a function which, when applies to a source agent and a destination agent, may generate a message of type *Message*. An agent instance belongs to an agent society takes roles of type defined in its corresponding role organization, upon which the agent society is defined. When two agents have an *association* relationship between their role instances, they may have interactions by sending messages to each other. Similar to a role space, the *agent society* contains zero agent instances initially. Agent instances can be added into or deleted from an agent society dynamically.

3 ROLE-BASED MAS DESIGN

3.1 Class Relationships in a Role Organization

The first step to design a multi-agent system is to design *Role* classes and their relationships. Role hierarchy defines the relationships among different roles in a role organization. In addition to the *aggregation* relationships and *association* relationships between classes, *inheritance* is a mechanism for incremental specification and design, whereby new classes may be derived from one or more existing classes. Inheritance therefore is particularly significant in the effective reuse of existing specifications (Stepney et al., 1992).

As a simple example of the *inheritance* relationship, consider a role type called *LeadingRole*, which is responsible for hiring other roles in fulfilling its goal.

$$
\begin{array}{l}
\textit{LeadingRole} \\
\hline
\quad \textit{Role} \\
\\
\quad \hline
\quad \textit{hiringNumber} : \downarrow \textit{RoleMetaClass} \rightarrowtail\!\!\!\!\!\!\!+\; \mathbb{N} \\
\quad \hline
\\
\quad \textsc{Init} \\
\quad \hline
\quad \textit{Role}.\textsc{Init} \\
\quad \textit{hiringNumber} = \{\textit{Self} \mapsto 1\} \\
\quad \hline
\\
\quad \textit{updateHiringNumber} \\
\quad \hline
\quad \Delta \textit{hiringNumber} \\
\quad \textit{rc}? : \downarrow \textit{RoleMetaClass}; \quad \textit{num}? : \mathbb{N} \\
\quad \hline
\quad \textit{hiringNumber}' = \textit{hiringNumber} \oplus \{\textit{rc}? \mapsto \textit{num}?\} \\
\quad \hline
\\
\hline
\end{array}
$$

As shown in the above class schema, the *LeadingRole* class is defined as a subclass of the *Role* class. Therefore, a leading role inherits all the data fields, e.g., *attributes*, *goals* and *plans*, as well as all operations defined in the *Role* class. In addition, a leading role records how many group members it needs. This functionality is defined by the operation *updateHiringNumber*, where *hiringNumber* is defined as a function to map a role type to the number of role instances needed. Note that in each role organization, there exists a *major* leading role, which is responsible for hiring other roles in that organization.

One more role class relationship in a role organization is called an *incompatibility* relationship, which exists when two roles cannot be taken by an agent at the same time under certain conditions. An example of such relationship (denoted as a dotted arc with a small circle) between a *BankerRole* and a *LoanBorrowerRole* is illustrated in Figure 2. In this example, a banker belonging to a bank is disallowed to borrow loan from the same bank.

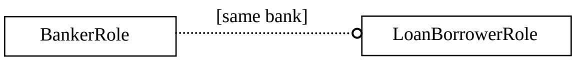

Figure 2: Example of incompatibility relationship

3.2 A Design Process for MAS

The purpose of our proposed approach is to ease software engineer's effort in developing open multi-agent systems. As we mentioned before, when agents take more than one roles, agent relationships in an agent society become very complicated. To make the design of intelligent agents simple, we avoid explicitly defining relationships between agent instances in an agent society. Instead, we deduce the interaction relationships between agent instances in an agent society from role assignments and role relationships in its corresponding role organization.

As a summary, we briefly describe the 5-step generic procedure to design open MAS as follows:

1. Design the set of *Role* classes Ω and their relationship Π_1: Ω X $\Omega \rightarrow$ [IH | AG], where IH and AG represent the relationship types of *inheritance* and *aggregation*, respectively.
2. Design the role organization Φ according to the class schema *RoleOrganization*, and define any *association* relationships and *incompatibility* relationships between classes, i.e., Π_2: Ω X $\Omega \rightarrow$ [AS | IC], where AS and IC represent the relationship types of *association* and *incompatibility*, respectively.
3. Design the role space Γ according to the class schema *RoleSpace*. The role space Γ should support creating, advertising and searching for role instances. It may use existing middleware, e.g., Sun Jini, for its purpose.
4. Refine the *Agent* class with a set of sensors and a set of appropriate reasoning mechanisms. This step may be overlapped with Step 1-3.
5. Design agent society Θ according to the class schema *AgentSociety*. The agent society Θ contains a set of agent instances of type *Agent*, and it corresponds to the role organization Φ with the same design purpose for agent organization or society.

Since we design multi-agent systems as open systems, agents may join or leave agent societies freely, and they can take roles at run time according to their motivations. Thus, the relationships between agents cannot be determined at design time; instead, they must be set up dynamically. In Section 3.3, we discuss how to deduce agent interaction relationships and check role incompatibility for agents.

3.3 Open Role Space and Open Agent Society

Multi-agent systems are one of the most promising approaches to creating open systems because of their capabilities to dynamically reorganize themselves as the system goals and constituent agents change (Dastani et al., 2003). In our approach, the openness of a multi-agent system refers to two things: open role space and open agent society. Open role space refers to a space where role instances can be added into or deleted from dynamically; while open agent society implies that agents can not only join or leave the system at will, but more importantly, they can take or release role instances in a role space dynamically. The procedure of taking or releasing role instances in a role space is a mapping process from agents in an agent society to role instances in a role space. We call this mapping process the *A-R mapping*.

Definition *A-R mapping* is a process for agents from an agent society Θ to take or release role instances in a role space Γ. Both Θ and Γ are defined upon the same role organization Φ. Formally, the state of *A-R mapping* is defined by the following function:

$$A\text{-}R\ mapping\ state \mathrel{\widehat=} f : Agent \nrightarrow \mathbb{P} \downarrow Role$$

where f is a partial function which maps each agent instance to a set of role instances.

The process of *A-R mapping* is a dynamic process of role assignment, which involves the following steps:

1. *Initialization:* The agent society Θ makes a request to the role space Γ to instantiate the major *LeadingRole* class defined in the role organization Φ, and create a role instance for it.
2. *Role assignment*: for each agent α in the agent society Θ, do the following:
   a. When agent α receives any sensor data from its environment, it may decide to generate some

new goals or subgoals based on the sensor data and agent α's motivations.

b. With its reasoning mechanisms, agent α further deduce a set Ω of needed roles of types defined in the role organization Φ. If none of the roles in set Ω is of type *LeadingRole*, go to step 2.d.

c. If any role in role set Ω is a leading role of type *LeadingRole*, agent α takes the corresponding role instance from the role space Γ, if available, updates the hiring number of other roles as needed, and makes requests to the role space Γ to create role instances for those roles under hiring.

d. Repeat the following for a period of time T: Search the role space Γ for any role instances that match roles in role set Ω. If there is a match, agent α takes that role instance. If all roles in role set Ω have been matched with some role instances in Γ, go to Step 3.

e. If any role in the role set Ω cannot be matched with a role instance in the role space Γ, agent α may decide to release all role instances or keep its current occupations.

3. *Marking role incompatibility:* for each agent α, mark its role incompatibility as the following: for any role instances r_1, $r_2 \in \alpha$.*rolesTaken*, if Φ.*relationship*(r_1.*getClass*, r_2.*getClass*) == *incompatibility*, mark agent α as potential *role incompatibility* with a self-loop.
4. *Setting up interaction relationships:* for each agent α, set up the interaction relationships between agent α and other agents from the same agent society Θ as the following : for any agent instance $\beta \in \Theta$.*agentInstances*, where $\alpha \neq \beta$, if $\exists$ $r_1 \in \alpha$.*rolesTaken*, $r_2 \in \beta$.*rolesTaken* such that Φ.*relationship*(r_1.*getClass*, r_2.*getClass*) == *association*, then $(\alpha, \beta) \in$ dom Θ.*interaction*.

The condition for role incompatibility of an agent α is checked at run time. Whenever the condition is satisfied, agent α must communicate with other agents to resolve the conflicts. In case the condition cannot be turned into *false*, one of the role instances in *conflict* must be released by agent α.

4 CONCLUSIONS AND FUTURE WORK

This paper describes a role based methodology for development of open multi-agent systems. The proposed concept of role space separates the design of roles and agents, which simplifies agent development. Inheritance relationships between roles support reuse of role design. For our future work, we will refine the *association* relationship into more specific relationships, e.g., *subordination* relationship and *collaboration* relationship. Based on the refined relationships, more specific role organizations can be designed. Social norms or rules can be specified in a role organization, and then checked for consistency in its corresponding agent society. This verification process may be automated due to our formal approach and the *A-R mapping* mechanism.

REFERENCES

Dastani, M., Dignum, V., and Dignum, F. 2003. Role-assignment in open agent societies. *Proceedings of the 2nd International Joint Conference on Autonomous Agents & Multiagent Systems*, Australia, pp. 489-496.

DeLoach, S. A., Wood, M. F., and Sparkman, C. H. 2001. Multiagent systems engineering. *The International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 3, pp. 231-258.

Duke, R., Rose, G., and Smith, G. 1995. Object-Z: a specification language advocated for the description of standards. *Computer Standards and Interfaces*, vol. 17, North-Holland, pp. 511-533.

Fisher, M. 1995. Representing and executing agent-based systems. *Proceedings of the International Workshop on Agent Theories, Architectures, and Languages*, M. Wooldridge and N. Jennings, eds., LNCS, vol. 890, Springer-Verlag, pp. 307-323.

Hilaire, V., Simonin, O., Koukam, A., and Ferber, J. 2004. A formal approach to design and reuse of agent and multiagent models. *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering (AOSE-2004)*, New York.

Juan, T., Pearce, A., and Sterling, L. 2002. ROADMAP: extending the Gaia methodology for complex open systems. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, Bolognia, Italy, pp. 3-10.

Luck, M. and d'Inverno, M. 1995. A formal framework for agency and autonomy. *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, AAAI Press / MIT Press, pp. 254-260.

Smith, R. G. 1980. The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Transactions on Computer*, vol. C-29, pp. 1104-1113.

Stepney, S., Barden, R., and Cooper, D. (eds) 1992. *Object orientation in Z.* Workshops in Computing, pp. 59-77.

Wooldridge, M., Jennings, N. R., and Kinny, D. 2000. The Gaia methodology for agent-oriented analysis and design. *International Journal of Autonomous Agents and Multi-Agent Systems*, vol. 3, no.3, pp. 285-312.

Xu, H. and Shatz, S. M. 2003. A framework for model-based design of agent-oriented software. *IEEE Transactions on Software Engineering (IEEE TSE)*, vol. 29, no. 1, pp. 15-30.