Paper Title: Developing Role-Based Open Multi-Agent Software Systems

Authors: Dr. Haiping Xu, Assistant Professor
Computer and Information Science Department
University of Massachusetts Dartmouth
Email: hxu@umassd.edu

Dr. Xiaoqin Zhang, Assistant Professor
Computer and Information Science Department
University of Massachusetts Dartmouth
Email: x2zhang@umassd.edu

Rinkesh J. Patel, Graduate Student
Computer and Information Science Department
University of Massachusetts Dartmouth
Email: g_rpatel@umassd.edu

Corresponding Author:

Dr. Haiping Xu, Assistant Professor
Computer and Information Science Department
University of Massachusetts Dartmouth
285 Old Westport Rd.
North Dartmouth, MA 02747

Phone: (508) 910-6427
Fax: (508) 999-9144
Email: hxu@umassd.edu

Date: April 8, 2006
Abstract

An open multi-agent system (MAS) is a dynamic system in which agents can not only join or leave an agent society at will, but also take or release roles at runtime. Traditional ways of developing multi-agent systems are not suitable for development of open multi-agent systems because they assume a fixed number of agents in a multi-agent system. The number of agents is determined during system analysis using role modeling. In this case, role models are only used at conceptual level with no realizations in the implemented system. In this paper, we first propose a formal framework for role-based modeling of open multi-agent software systems. We specify the role organization and the role space as containers of conceptual roles and role instances, respectively. We further define agents as members of an agent society, where agents can take or release role instances from a role space dynamically. Then we propose a three-layered design model for development of role-based open MAS. Our formal role-based agent model serves as the first layer of the development model that is independent of both the application and solution domains. Our approach provides a potential solution for automated MAS development, which is illustrated by a prototype of the Role-based Agent Development Environment (RADE). Finally, we use a case study to show how role-based open MAS can be developed using our approach.

Keywords: Role-based modeling, Open multi-agent systems, Object-Z formalism, Model-driven development, Role-based Agent Development Environment (RADE)
1. Introduction

Multi-agent systems (MAS) are rapidly emerging as a powerful paradigm for modeling and developing complex software 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 multi-agent systems 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 occur 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, roles are atomic constructs and cannot be defined in terms of other roles [1]. This approach is feasible when designing small-scale, closed systems, especially when an agent only takes a single role. However, when an agent takes more than one role, or if an agent can take or release roles at runtime, such an approach becomes infeasible. This is because when role assignments are dynamic, the interaction relationships between agents become quite complicated, and they usually 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 design phase, and to find a way to deduce agent interaction relationships from agent-role mappings and role relationships dynamically. In this paper, we first propose a methodology for role-based modeling of open multi-agent systems. We define a role organization that provides the ontology for modeling roles and their relationships. As one of the most important role relationship types, inheritance is explicitly modeled using Object-Z formalism [2]. A role space is then defined as a container for role instances, and also as a server to provide services to agents for accessing role instances from the role space. Based on the concepts of role organization and role space, we define an agent society as a community that consists of a set of agents, which may join or leave the agent society at will, and 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 [3]. To support separation of concerns and reuse of various design models, we propose a three-layered development model, in which our formal role-based open MAS model serves as the first layer of the
development model that is independent of both the application and solution domains. Our approach provides a potential solution for automated MAS development, which is illustrated by a prototype of the Role-based Agent Development Environment (RADE). Finally, we use a case study to show how role-based open MAS can be developed using our approach.

The rest of this paper is organized as follows. In Section 2, we describe the related work and highlight the relationships to our research. In Section 3, we present a formal framework for role-based open MAS using Object-Z formalism. Our formal framework consists of three key concepts, namely role organization, role space and agent society. In Section 4, we propose a three-layered development model for role-based open MAS. In our approach, we use our formal role-based agent model as a generic model, which is independent of both the application and solution domain. Based on the generic model, we drive our application specific model by incorporating application logic, and we show how to set up agent interaction relationships using an A-R mapping mechanism. In the third layer, we further incorporate implementation technologies into the application specific model and build a platform specific model. In Section 5, we provide an example of organizing a conference to illustrate how role-based open MAS can be developed using our approach. Finally, In Section 6, we provide conclusions and our future work.

2. Related Work

There are three main strands of work to which our research is related, i.e., work on formal modeling of agent-based systems, work on role-based agent development methodologies, and work on model-driven 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. They proposed a four-tiered hierarchy comprising entities, objects, agents and autonomous agents [4]. The basic idea of their approach is that all components of the world are entities with attributes. Of these entities, objects are entities with capabilities of actions, agents are objects with goals, and autonomous agents are agents with motivations. Fisher’ work on Concurrent
METATEM used temporal logic to represent dynamic agent behaviors [5]. Such a temporal logic is more powerful than the corresponding classic logic and is useful for the description of dynamic behaviors in reactive systems. Fisher took the view that a multi-agent system is simply a system consisting of concurrently executing objects. Xu and Shatz used agent-oriented G-nets, which is a high-level formalism of Petri net, to model and verify behavioral properties of multi-agent systems [6]. Based on the agent-oriented G-net model, certain properties of a multi-agent system, e.g., concurrency and deadlock freeness, can be verified by using existing Petri net tools [7]. Hilaire and his colleagues proposed a mechanism for dynamic role playing that is specified with a formalism called OZS [8]. The OZS formalism combines Object-Z and statecharts, and can be used to specify multi-agent systems based upon role, interactions and organizations [9]. In their approach, Object-Z is used to specify the transformational aspects, and statecharts are used to specify the reactive aspects of a multi-agent system. The specification approach is based on an organizational meta-model called RIO (Role, Interaction and Organization), and allows validation and verification of MAS.

In summary, formal methods are typically used for specification of agent systems and agent behaviors. Existing work in this direction either do not directly use role modeling for agent design, or use role modeling simply as conceptual guidelines for agent development during the analysis phase. Furthermore, as stated by Cabri and his colleagues, a common limitation of those approaches is the lack of support for development of multi-agent systems during all phases [10]. In this paper, we propose our formal role-based open multi-agent system framework based on three key notions, namely, role organization, role space and agent society, where role classes can be defined in a role organization, instantiated in a role space, and role instances can be taken or released by agents form an agent society dynamically. We overcome the limitation of current role-based modeling approaches by illustrating how our formal model can be used in a three-layered agent development framework.

A second strand of related work is to propose role-based methodologies for the development of multi-agent systems. Typical examples of such efforts include the Gaia methodology [11] and Multiagent
Systems Engineering (MaSE) methodology [12]. The Gaia methodology models both the macro (social) aspect and the micro (agent internals) aspect of a 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 a detailed design phase using conventional object-oriented methodology.

Similarly, the MaSE methodology is a specialization of more traditional software engineering methodologies [12]. 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 and design of roles are tightly coupled. Different from the above methodologies, in our approach, the components of role instances and agent instances are loosely coupled, where agents can take or release role instances at runtime without knowing the internal structure of role instances. As a consequence, role classes and agent classes can be designed and implemented independently. Thus, the specification and design of agent classes can be significantly simplified.

Previous efforts on model-driven development of multi-agent systems can be summarized as follows. Bernon and his colleagues attempted to unify three existing methodologies – ADELFE, Gaia, and PASSI – by studying their meta-models and concepts related to them. The unification would be useful to build tools using the MDA approach to automatically transform a meta-model into a model depending on a target platform [13]. Gracanin and his colleagues proposed a model-driven architecture framework in extending the Cognitive Agent Architecture (COUGAAR), which is an open source, distributed agent architecture [14]. The proposed framework consists of two main parts: General COUGAAR Application Model (GCAM) and General Domain Application Model (GDAM). The GCAM provides model representation of the COUGAAR basic constructs; while the GDAM, which is built upon the foundation
of GCAM, defines the requirements and the detailed design. In Amor, Fuentes and Vallecillo’s work, the authors showed how to use the MDA approach to drive agent implementation from agent-oriented design, which is independent of both the methodology used and the concrete agent platform selected [15]. The transformation process can be partially automated by using a platform-neutral agent model, called Malaca. More recently, De Maria, Silva and Lucena proposed an MDA-based approach to developing MAS [16]. They first used MAS-ML, which was an MAS modeling language, to model MAS by creating the platform independent models (PIM). Then the MAS-ML models were transformed into UML models using the ASF framework, which defined a set of object-oriented models that implemented the MAS entities modeled in MAS-ML. Finally, the UML models were transformed into code. Most of the previous efforts emphasized on automatically transforming a PIM into platform-specific models (PSM). However, as some researcher suggested, it could be distinctly nontrivial, and even impossible, to support and evolve semantically correct PSM for complex platform such as J2EE or .Net [17]. In contrast to the above approaches, our approach emphasizes on developing three levels of models, namely AIPIM (Application Independent Platform Independent Model), ASPIM (Application Specific Platform Independent Model) and ASPSM (Application Specific Platform Specific Model), which can be viewed as steps in a refinement process. In each level of the development model, role components and agent components are always separated and designed independently. Role instances and agent instances interact with each other only at runtime through the A-R mapping mechanism. Therefore, our approach follows the principle of component-based software engineering (CBSE), where role entities and agent entities can actually be developed by different groups of people.

3. A Framework for Role-Based Multi-Agent System

3.1 An Organizational Approach

In most of the existing MAS development process models, an agent cannot change its roles at runtime; however, in reality, when we use a software agent to represent a human user, it should be allowed to change its roles dynamically. For example, when we use agent technology to simulate a startup software company, the agent representing the company builder (the CEO role) first creates a number of positions,
such as team leader roles and programmer roles. During the process of hiring, a newly hired employee is only allowed to take the roles defined, but she/he can take one or more than one role at the same time. While the company is running, existing employees may leave the company, and drop the roles that were previously taken; meanwhile, new employees can be hired to take the available position roles. In addition, it is possible that, at runtime, an employee is assigned to take another role under certain conditions. For example, an employee who previously takes a programmer role can be promoted to take a team leader role based on his/her excellent performance. To model this kind of dynamic and open system, conventional agent development methodologies become not as appropriate. Therefore, we propose our role-based methodology for open MAS to separate the concepts of role and role instance, where a role is defined as a conceptual role; while a role instance is a concrete implementation of a conceptual role. We further define a role organization that contains conceptual roles, in which the relationships of roles can be defined as a subset of the following four types of relationships, namely inheritance, aggregation, association and incompatibility. We then propose a concept called role space that consists of role instances. Instead of simply using role concepts during the analysis phase, we explicitly create role instances at runtime; thus, agents can take or release role instances from a role space dynamically. A generic model of role-based open multi-agent systems is illustrated as in Figure 1.

Figure 1. A generic model of role-based open multi-agent systems (adapted from [3])
As shown in Figure 1, a role organization defines a set of conceptual roles (or role classes) 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. In other words, role_C views role_D’s responsibilities and capabilities as part of its own. Role_D and role_E have an association relationship, for example, role_D is responsible to provide certain information to role_E if there is such a request. 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.

A role space is defined upon a role organization, and it consists of a set 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 defined in the role organization. Since the relationships between role instances can be easily derived from their class relationships, it is not necessary to explicitly show their relationships at this layer.

An agent society is also defined upon a role organization, and it contains a set 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 runtime, but can also release them if the role instances are not needed any more for achieving its goals. The relationships between agents depend 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 passed down as agent relationships.

Since agents take roles only at runtime, the definition of an agent does not depend on that of roles. In other words, the development of agents can be independent of the development of roles, so we can
simplify the development of agents by concentrating on issues such as selection of goals, reasoning mechanisms, and communication mechanisms. 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 group of people, for example, two different companies.

3.2 Role-Based Agent Model

For specifying the proposed role-based model of open multi-agent systems, we build a framework using Object-Z [2]. Our framework is composed of a set of classes which specifies the basic constructs in the role-based open MAS model. We now provide some key definitions of the basic constructs, such as Role, RoleOrganization, RoleSpace, Agent and AgentSociety, in our formal model as adapted from [3].

**Definition 3.1 Role Class**

A role class, 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 entity.

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

```plaintext
Class Role
  attributes : P Attribute
  goals : P Goal
  plans : P Plan
  actions : P Action
  permissions : P Permission
  protocols : P Protocol
  beTaken : B

Init
  permissions = ∅
  protocols = ∅
  beTaken = false

setPermission
  ∆permissions
  perm? : Permission
  permissions' = permissions ∪ {perm?}

addProtocol
  ∆protocols
  prot? : Protocol
  protocols' = protocols ∪ {prot?}
```
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 describes the characteristic properties of a role, including a role name and a role identification. A *Role* is defined to have a set of goals and a set of plan trees, 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 plan trees of type *Plan*. A plan tree describes how to achieve a goal or subgoal by executing several actions in a specified order. Each plan tree is associated with a goal or a subgoal; however, a goal or subgoal may associate with more than one plan tree, and the most suitable one will be selected to achieve that goal or subgoal. To carry out a certain plan, a role needs the capability of performing certain associated actions. The state variable *actions* refer to a set of actions of type *Action*, and a certain action will be triggered to execute when an associated plan tree is selected to carry out. 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 in order to achieve a goal or subgoal. The permissions are accessing rights of a role for information related resources. 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 of how a role may interact with other roles. An example of such protocol is the contract net protocol [18]. 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; therefore, 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, plan trees, 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.
The behaviors of a role can be modified at runtime, for example, the permissions of a role can be modified and new protocols can be added to a role dynamically. This is achieved by providing the operations of setPermission and addProtocol as operation schemas as shown in the Role class schema.

**Definition 3.2 Role Organization**

A role organization is defined as 2-tuple \( RO = (SR, REL) \), where \( SR \) is a set of conceptual roles, and \( REL \) is the relationship function maps two conceptual roles to a role relationship \( \mathcal{R} \in \{\text{inheritance, aggregation, association, incompatibility}\} \).

Before we can define the class schema \( RoleOrganization \), we must first define the type of \( RoleMetaClass \). 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. Based on the concept of \( RoleMetaClass \), we formally define the class schema \( RoleOrganization \) as follows:
By defining the state variable *roles* as a set of elements of type *RoleMetaClass* or its derivatives, *roles* refers to a set of subclasses of the *Role* class and the *Role* class itself. Accordingly, the function *relationship* is defined for relationships between classes (roles) instead of objects (role instances). Such relationships include *inheritance* relationship, *aggregation* relationship, *association* relationship and *incompatibility* relationship, which will be described in details in Section 4.2. The *Role* class is the root class of all its descendents, and it exists at the very beginning of 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 also be added. This is automatically achieved by updating the function *relationship* by adding a mapping of \{(*r*, *role*) \mapsto *inheritance*\}. Relationships other than the *inheritance* relationship between role classes must be set up by applying the operation *setRelationship*.

**Definition 3.3 Role Space**

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 more than one role space. Role instances can be added into or deleted from a role space dynamically. A role space provides services to access role instances in the role space according to role attributes.

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

As shown in the class schema `RoleSpace`, 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 provides services to agents for accessing the appropriate role instances according to role attributes to achieve their goals.

**Definition 3.4 Agent**

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

An agent has attributes such as the agent name, agent owner and agent identification. As shown in the above class scheme of Agent, 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 [4]. The sensor of an agent perceives related environment changes of type Environment and transforms the inputs 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 with certain attributes. The agent then searches the role space for any available role instances that satisfies the role properties and takes each needed available role instance from the role space to achieve its goals. Notice that the reasoningMechanism in reality is more complicated than what we have defined, e.g., it also includes a function to choose plan trees 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. In addition, the \textit{Agent} class schema defines two fundamental operations: \textit{takeRole} and \textit{releaseRole}. The \textit{takeRole} operation takes an available role instance from a role space, and set it as unavailable to other agents. On the other hand, the \textit{releaseRole} operation releases a role instance and set it to be available for other agents.

\textbf{Definition 3.5 Agent Society}

An \textit{agent society} is defined upon a role organization and consists of a set of agent instances that are of type \textit{Agent}. An agent society provides services to receive or remove agent instances such that agent instances can join or leave an agent society dynamically.

The structure of agent society is often determined by organizational design which is independent of the agents themselves [19]. The class scheme of \textit{AgentSociety} is formally defined as follows.

\begin{verbatim}
AgentSociety
| roleOrganization : RoleOrganization
| #roleOrganization.roles > 1

agentInstances : ℘ Agent
| interaction : Agent x Agent -> Message

\forall a \in agentInstances, \exists r \in a.rolesTaken \bullet r.getClass \in roleOrganization.roles
\forall a1, a2 \in agentInstances, a1 \neq a2,
\exists r1 \in a1.rolesTaken, \exists r2 \in a2.rolesTaken,
roleOrganization.relationship(r1.getClass, r2.getClass) = Association \bullet
(a1, a2) \in dom interaction

Init
agentInstances = ∅

createAgentInstance
ΔagentInstances
agent? : Agent
\forall a \in agentInstances \bullet a \neq agent?
agentInstances' = agentInstances \cup \{agent?\}

deleteAgentInstance
ΔagentInstances
agent? : Agent
agent? \in agentInstances
\forall r \in agent?.rolesTaken \bullet agent?.releaseRole(r)
agentInstances' = agentInstances \setminus \{agent?\}
\end{verbatim}
As shown in the formal AgentSociety class schema, the AgentSociety class is defined upon a roleOrganization. Since both role spaces and agent societies are built on role organizations, a correspondence exists 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 instance from a corresponding role space. Those agents who do not take any role instances from a corresponding role space shall leave the agent society eventually. Note that the correspondence between a role space and an agent society does not imply that an agent can take roles only from one role space. In contrast, an agent may join multiple agent societies and take roles from different role spaces.

As shown in the AgentSociety class scheme, 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 belonging to an agent society takes roles of type defined in the 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. Initially, an agent society only contains a leading agent instance who creates the agent society. During runtime, other agent instances can join the agent society to achieve their goals and leave the society when their goals have been achieved.

4. Model-Driven Development of Role-Based Open MAS

Inspired by OMG’s Model-Driven Architecture (MDA) [20], we propose a three-layered development model for developing role-based open MAS. Our approach supports separation of concerns such that the architecture domain, the application domain and the solution domain can be considered separately. Similar to the MDA approach, our approach provides a potential solution to the automated development of role-based open MAS. In other words, some portion of the program code can be automatically generated using development tools.
4.1 Three-Layered Development Model

Our three-layered development model consists of three relatively independent models, namely Application Independent Platform Independent Model (AIPIM), Application Specific Platform Independent Model (ASPIIM), and Application Specific Platform Specific Model (ASPSM). The purpose of this approach is to separate software architecture from application domain and to separate application logic from the underlying technologies to improve reusability and development process.

![Diagram of three-layered development model]

**Figure 2.** Three-layered development model for developing role-based open MAS

As shown in Figure 2, the three-layered development model is defined in three steps. The first step is to define the AIPIM model which is a generic model that matches our role-based development methodology for open MAS. The second step is to define the ASPI model that is based on the AIPIM model. The ASPI model involves knowledge from the application domain. In the third step, based on the ASPI model, we define the ASPS model that further incorporates information from the solution domain.

There is a one-to-one mapping between the classes defined in the role-based formal MAS model described in Section 3 and the classes defined in the AIPIM model for role-based open MAS. For example,
the Role class defined in Object-Z is defined as a Role class in the AIPI model, which is an instance of the RoleMetaClass.

Figure 3. The AIPI model of role-based open MAS in UML class diagram

Figure 3 shows a simplified AIPI model in UML class diagram, which illustrates the key classes in this model. From the figure, we can see that a RoleOrganization contains instances of ClassRelationship and instances of RoleMetaClass, which are Role classes. We will describe the ClassRelationship in more details in Section 4.2. A RoleSpace contains any number of Role instances, but a role instance can belong to only one role space. Notice that a Role class (including its subclasses) can be associated with more than one role organizations. An AgentSociety contains any number of agent instances, and an agent can join more than one agent society. For example, a company can have any number of employees, but any employee may be allowed to work for two different companies at the same time. In addition, an agent can take any number of role instances; however, any role instance can only be taken by one agent.
The ASPI model defines a high-level abstraction that is specific to a particular application; but the model is independent of any implementation technology. In other words, the ASPI model describes an open multi-agent software system that supports the application logic, but whether the system will be implemented on a main frame with J2EE or .NET platform plays no role in such a model. One advantage of using role-based agent development is to simplify the definition of the Agent class such that certain capabilities to achieve a goal can be encapsulated into a role component. Two key issues in defining the ASPI model are to define the role organization and to define the mapping from agent instances to role instances. We discuss these two issues in more details in Section 4.2 and Section 4.3, respectively.

The third model is called ASPS model, which defines the multi-agent system that is specific to a particular application as well as the implementation technologies. Based on the ASPI model, the ASPS model incorporates knowledge from the solution domain, and specifies the open MAS in terms of some specific implementation technologies. For example, the open MAS can be implemented using EJB and Java servlets on a J2EE platform, or it can be developed using Microsoft .Net techniques. Alternatively, we can use web services techniques, such as IBM WebSphere and Sun JWSDP [21] for agent communications. In Section 5, we will use a case study to show how to develop the ASPS model based on the communication mechanism that is supported by the ADK (Agent Development Kit) [22].

### 4.2 Class Relationships in a Role Organization

When designing an open MAS application using role-based modeling, we first need to design the Role classes and their relationships in a role organization. In a role organization, role hierarchy defines the relationships among different role classes. The relationship types between two role classes consist of the following: inheritance relationship, aggregation relationship, association relationship and incompatibility relationship. We now give the definitions to these four relationships as well as some related key concepts such as the leading role and the composite roles.
**Definition 4.1 Inheritance Relationship**

An *inheritance relationship* between two role classes represents the generalization or specialization relationship between two role classes, where one class is a specialized version of another. 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 [23].

**Definition 4.2 Leading Role**

A *leading role* is responsible for hiring other roles in achieving its goal [3]. For example, a company CEO is a leading role, who is responsible for hiring new employees. A leading role is defined by the LeadingRole class that 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 role instances it needs to achieve its goals. This functionality can be defined by an operation called updateHiringNumber, which updates the needed number of role instances for a certain type of roles.

**Definition 4.3 Composite Role**

A *composite role* is defined by the CompositeRole class, which is a subclass of the Role class. In the CompositeRole class, the state variable of subRoles describes a set of role instances of type Role or its derivatives. Subroles can be added into or deleted from the subRoles set by applying the operation addSubRole or deleteSubRole.

**Definition 4.4 Aggregation Relationship**

In an *aggregation relationship* between two role classes, one of the classes must have been defined as a subclass of the CompositeRole class. The aggregation relationship between roles is most suitable for defining the hierarchy of a role organization. For instance, we can use a composite role to represent a team, a group or even a role organization.
**Definition 4.5 Association Relationship**

The association relationship is one of the most common relationships between classes [24]. Associations may have an association name, role names and multiplicity. The association name indicates an action that an instance of one role may perform on an instance of another role. The multiplicity of an association denotes the number of instances of the role classes that can participate in their relationship. To describe such a relationship in a more precise manner, we add a condition \([\text{cond}]\) in front of the association name. The association relationship only exists between instances of role classes when the \(\text{cond}\) is evaluated to true.

**Definition 4.6 Incompatibility Relationship**

Under certain condition, when two roles cannot be taken by an agent at the same time, we say these two roles have an incompatibility relationship. An example of such a relationship between a BankerRole and a LoanBorrowerRole (denoted as a dotted arc with a small circle) is illustrated in Figure 4. In this example, a banker who works for a bank is not allowed to borrow loan from the same bank.

![Figure 4. An example of incompatibility relationship](image)

**4.3 A-R Mapping Mechanism**

Multi-agent systems have been proposed as one of the most promising approaches to creating open systems due to their capabilities of dynamically reorganizing themselves as the system goals and constituent agents change [19]. In our approach, the openness of a multi-agent system means the openness of both the role space and the agent society. The openness of a role space refers to a space where role instances can be added into or deleted from a role space dynamically; while the openness of an 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 from a role space at runtime. 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 4.7 A-R Mapping**

An A-R mapping is a process for agents from an agent society \( \Theta \) defined upon role organization \( \Phi \) to take or release role instances in a role space \( \Gamma \). Both \( \Theta \) and \( \Gamma \) are defined upon the role organization \( \Phi \). Formally, the A-R mapping is defined by the following function:

\[
A-R \text{ mapping} \triangleq f : Agent \rightarrow \mathbb{P} \downarrow \text{Role}
\]

where \( f \) is a partial function that maps from 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:** A user creates a leading agent \( \alpha \) in agent society \( \Theta \). The leading agent \( \alpha \) is responsible for initializing and managing the agent society \( \Theta \). Ordinary agents representing different users may join the agent society \( \Theta \), and are ready to take role instances from the role space \( \Gamma \).

2. **Creating role instances:** The leading agent \( \alpha \) makes a request to the role space \( \Gamma \) to instantiate the major leading role class that is defined as a subclass of the \textit{LeadingRole} class in the role organization \( \Phi \). The leading agent \( \alpha \) takes the major leading role instance as soon as it is available. The leading agent \( \alpha \) further makes requests to the role space \( \Gamma \) to create all the role instances that are needed to achieve its goal.

3. **Role assignment:** The role space \( \Gamma \) waits for requests from some ordinary agent \( \beta \) in agent society \( \Theta \), and do the following:

   3.1 If the request is to query about a role instance, then

   a. Search the role space \( \Gamma \) for any available role instances with the requested role attributes.

   b. If there is a match, reserve the role instance and notify agent \( \beta \) to take that role instance.
Else notify agent $\beta$ that there is no available role instances, go to Stage 3.

Else if the request is to take a role instance, then

Assign the requested role instance to agent $\beta$, and check its role incompatibility as follows:

For any role instances $r_1, r_2 \in \beta.rolesTaken$,

If $\Phi.\text{relationship}(r_1.\text{getClass}, r_2.\text{getClass}) = \text{incompatibility}$, and the condition for that relationship is $\text{true}$, then

Suspend any activities of role instances $r_1, r_2$ until the conflict is resolved.

Else if the request is to release a role instance, then

Release the role instance from agent $\beta$.

3.2 Setting up agent interaction relationships: The role space $\Gamma$ notifies the agent society about the updated role assignment and updates the interaction relationships between agent $\beta$ and other agents from agent society $\Theta$ as follows: for any agent instance $\gamma \in \Theta.agentInstances$, where $\beta \neq \gamma$, if $\exists r_1 \in \beta.rolesTaken, r_2 \in \gamma.rolesTaken$ such that $\Phi.\text{relationship}(r_1.\text{getClass}, r_2.\text{getClass}) = \text{association}$, then $(\beta, \gamma) \in \text{dom } \Theta.\text{interaction}$.

3.3 Goto stage 3.

As shown from the above algorithm, the condition for role incompatibility of an agent $\beta$ is checked at runtime. Whenever the condition is satisfied, agent $\beta$ must negotiate with other agents to resolve the conflicts. In case that the condition cannot be turned into $\text{false}$, one of the role instances in conflict must be released by agent $\beta$.

4.4 Tool Support for Design of ASPI Model

To facilitate rapid development of the ASPI model, we developed a Role-based Agent Development Environment (RADE) prototype. The major tasks of the current version of the RADE system are to provide tool supports for design of role organization, visualization of role space and agent society, and automated role assignment using $A-R$ mapping. The toolkit for design of role-organization is similar to the Rational Rose toolkit [25], but it is specific to support design of role classes and their relationships.
Figure 5 shows the user interface of the RADE prototype for design of role organizations. When the high-level design of the role organization is finished, the system will prompt the user to fill out the attributes and operations defined in the *Role* class (i.e., the root class) according to the class schema defined in Section 3. Then the system prompts the user to define additional attributes and operations for each role classes in the role organization. Finally, the code for the *RoleOrgnization* package can be automatically generated by clicking on the “CodeGen” menu on the top of the window.

![Figure 5. User Interface of the RADE prototype for design of role organization](image)

The RADE prototype also supports generation of code for the role space and agent society. The role space works as a server that receives requests for querying about the availability of role instances, taking role instances and releasing role instances. In the RADE prototype, the system can graphically show the available role instances as well as related objects in the role space. Similarly, an agent society also works as a server that receives requests from agents to join or leave the agent society. Notice that the agent
society contains a proxy of each registered agent; while the real agent can run on a remote machine. The RADE prototype can dynamically show currently registered agents and keep track of each agent’s behaviors.

When various agents joins the agent society and starts to request roles instances, the automatic A-R mapping mechanism is invoked. According to the algorithm for the A-R mapping mechanism as shown in Section 4.3, the role space works reactively to take requests from ordinary agents to query, take or release role instances from the role space. In this process, possible conflicts of role instances taken by the agent are marked on that agent. If the condition for such conflicts does exist and cannot be resolved, the agent must make a request to the role space to release any role instances in conflict in order to resume the activities of other roles.

4.5 Design of the ASPS Model

A role-based open multi-agent system is defined as a distributed system, in which each agent runs on a different machine. Therefore, an agent society is essentially a virtual society that contains only the proxy of each registered agent running on a remote machine. In addition, the role space server and the agent society server do not have to be residing on the same host. When an agent running on a remote machine wants to use the role instances to achieve its goals, it should be able to invoke methods defined on the role instances from the role space on a different machine. An agent in an agent society should also be able to find the other agents in the same society, and communicate with them asynchronously. This facilitation can be supported by using a middleware associated with the agent society server. Figure 6 shows the ASPS model architecture of role-based open MAS.

Since each agent works on behalf of a human being, the system provides the interface to human users to provide the initial instructions and any information to agents. Notice that an agent running on a remote machine, so the ordinary agents and the leading agent shown in Figure 6 are all proxies of real agents. The leading agent in an agent society manages the agent society. All other agents called ordinary agents
represent ordinary users who can join or leave agent society freely, and can also take or release roles from the role space at runtime. Notice that both role space and agent society are associated with a database, where the database can record information about role instances and role assignments, and information about agents currently in the society, respectively.

![Diagram of ASPS model architecture of role-based open MAS](image)

**Figure 6.** The ASPS model architecture of role-based open MAS

5. **A Case Study: Organizing a Conference**

Consider an example of organizing a conference. There are different roles in running a conference: program committee (PC) chair, program committee member, primary PC member and author. Program committee chair is responsible for assigning papers to program committee members for reviewing. Each paper has \( n \) reviewers. A reviewer cannot review his/her own papers. For each paper, there is a primary PC member assigned by the program committee chair, who is responsible for reading the reviewers’ comments, solving conflicts among different reviewers and making decisions on whether to accept or reject the paper. The ASPI model of the agent-based conference organizer application can be illustrated in Figure 7.
As shown in Figure 7, the \textit{PCChairRole} class is defined as a subclass of the \textit{LeadingRole} class; while the \textit{AuthorRole} and \textit{PCMemberRole} classes are defined as subclasses of the \textit{Role} class. The \textit{PrimaryPCMemberRole} is a special \textit{PCMemberRole} that makes decisions on paper acceptance; therefore, it is defined as a subclass of the \textit{PCMemberRole} class. A \textit{PCChairRole} is responsible for assigning papers to a \textit{PCMemberRole}, thus there is an “assign papers” association relationship between these two classes. A \textit{PrimaryPCMemberRole} makes decisions on the acceptance of a paper; therefore, it has an association relationship with the paper’s author for notification of the result. In addition, the \textit{AuthorRole} has an incompatibility relationship with both the \textit{PCMemberRole} and the \textit{PrimaryPCMemberRole}. This implies that at any time a \textit{PCMemberRole} or a \textit{PrimaryPCMemberRole} cannot review his/her own paper.

![Diagram](image)

\textbf{Figure 7.} The ASPI model of the agent-based conference organizer application

When we design the ASPS model, we use Sun Jini as a middleware for agents to communicate with agent society and role space as well as for agents to communicate with each other. The Jini architecture is
intended to resolve the problem of network administration by providing an interface where different components of the network can join or leave the network at any time [26][27]. The heart of the Jini system is a trio of protocols called *discovery*, *join*, and *lookup*. *Discovery* occurs when a service is searching for a lookup service with which to register. *Join* occurs when a service has located a lookup service and wishes to join it. And *lookup* occurs when a client or user needs to locate and invoke a service described by its interface type and possibly, other attributes. Our ASPS model for the agent-based conference organizer application is supported by the ADK (Agent Development Kit) toolkit we have developed previously [22]. More specifically, both role space and agent society registered the services they provide with the Jini community, so agents can look up a certain service and invoke it as needed. Meanwhile, each agent also registers itself as a proxy in the Jini community, so agents can find each other and communicate with each other asynchronously. For a detailed description of this approach, refer to previous work [22] for how agents can communicate with each other asynchronously.

The open multi-agent system developed based on the ASPS model provides a user interface for a user to submit a paper and/or apply for a PC member role. An agent represents an author who can take an author role from the role space; while a user who wants to be a PC member may take a PC member role. During the process, role assignment is automatically done by the role space server. When the submission deadline is reached, the paper assignment process starts. The PC chair agent matches the area of interests of each agent who takes a PC member role with the keywords of each paper, and generates an initial paper assignment table. A simulation result for such a table is illustrated in Figure 8.

As we can see from this table, the initial paper assignment is not balanced: some paper has been assigned to as many as 7 reviewers (for example Paper_4); while some paper only has one reviewer (for example Paper_12). To balance the number of reviewers for each paper, the PC chair needs to find additional reviewers for those papers that do not have enough reviewers, and may drop some reviewers for those papers that have too many reviewers. It is possible that a reviewer who is requested to review a new paper is not willing to do so. This requires that the PC chair can negotiate with the requested PC member to
achieve its goal. A simplified interaction protocol for such negotiation is shown in Figure 9 (a). As the figure shows, the PC chair first makes a request to a PC member for reviewing a paper. The PC member has the choice either to accept or reject the request. If the request is accepted, the PC chair should notify the PC member about the due date. If the PC member’s reply is negative, the conversation ends; otherwise, the PC chair confirms with the PC member for the new paper assignment.

![Simulation Result: Reviewer](image)

**Figure 8.** Simulation results for paper assignment (initial result)

![Interaction Protocols](image)

**Figure 9.** Examples of interaction protocols between a PC chair and a PC member
Similarly, for each paper, the PC chair needs to appoint a primary PC member to be in charge of that paper. A simplified interaction protocol for such communication is illustrated in Figure 9 (b).

The user interface of the PC chair agent is shown in Figure 10. From the figure, we can see that the PC chair communicates with two agents, i.e., Agent_4 and Agent_6, and finally appoints Agent_6 as the primary PC member for Paper_8.

![Figure 10. User interface of the PC chair agent](image)

After the paper assignment is balanced and each paper has been assigned to a primary PC member, the system generates the final paper assignment table. Figure 11 shows the simulation results for the final paper assignment.
6. Conclusions and Future Work

This paper proposes a role-based methodology for development of open multi-agent software systems. The proposed concept of role organization, role space and agent society separates the design of roles and agents, which simplifies the agent development process. A three-layered development model for developing open MAS is presented and illustrated by a case study of organizing a conference. The simulation result shows that our approach is feasible and effective for developing open MAS. In addition, our approach supports rapid development of open MAS using our RADE prototype. For future work, we will formalize the design process of the ASPI model and ASPS model, and based on the formal definitions of these models, we will partially automate the model transformation process from AIPI model to ASPI model, ASPI model to ASPS model, and ASPS model to Java code. In future versions of this project, we will incorporate these transformation tools into our RADE system.

References


