PAPER MANUSCRIPT SUBMITTED TO INTERNATIONAL JOURNAL OF COMPUTATIONAL INTELLIGENCE THEORY AND PRACTICE

(Final Version)

Paper Title: **Developing Role-Based Open Multi-Agent Software Systems**[1]

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

[1] Manuscript submitted 8 April 2006; Revised 20 March 2007; Accepted 9 May 2007.

Developing Role-Based Open Multi-Agent Software Systems

Haiping Xu, Xiaoqin Zhang and Rinkesh J. Patel

Computer and Information Science Department

University of Massachusetts Dartmouth

North Dartmouth, MA 02747

Email: {hxu, x2zhang, g_rpatel}@umassd.edu

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 multi-agent system development methodologies are not suitable for developing open multi-agent systems because they assume a fixed number of agents that are specified during the system analysis phase. In this paper, we propose a formal role-based modeling framework for open multi-agent software systems. We specify role organizations and role spaces as containers of conceptual roles and role instances, respectively, where role instances can be dynamically taken or released by agents from agent societies. To support rapid development of role-based open multi-agent systems, we introduce a three-layered design model of open MAS, and developed a prototype Role-based Agent Development Environment (RADE). Finally, we present a case study to demonstrate how an open MAS application can be efficiently built on RADE.

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 guidelines for software engineers 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 system analysis phase. The roles abstracted from use cases are high-level constructs used to conceptualize and understand the system. There are no realizations of agent roles in the implemented system beyond the analysis stage. In most of the cases, roles are atomic constructs and cannot be defined in terms of other roles [1]. Such approaches are feasible when developing small-scale and closed multi-agent systems, especially when an agent only takes a single role. However, in an open multi-agent system, when an agent is allowed to take more than one role, and also take or release roles at runtime, these approaches become not suitable. 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 is vital for us to introduce the concept of role instance (a concrete implementation of a conceptual role) into the design phase, and 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 for software agents to access role instances from the role space. Based on the concepts of role organization and role space, we define an agent society as an agent community where agents 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 the software engineering principle of "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 Role-based Agent

Development Environment (RADE). Finally, we use a case study to show how a role-based open MAS application can be developed efficient on RADE.

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 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, and describe our RADE prototype. In Section 5, we provide an example of organizing a conference to illustrate how role-based open MAS applications 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 focused on designing formal specification languages or using existing formalisms, such as Z, temporal logic, and Petri nets, to specify agent systems or agent behaviors. Brazier and his colleagues developed a high-level modeling framework called DESIRE (framework for DEsign and Specification of Interacting REasoning components), which enables specification of a system's conceptual design [4]. The DESIRE framework can explicitly support modeling the knowledge, interaction, and coordination of complex tasks and reasoning capabilities in agent systems. Shapiroa and Lespérance proposed the Cognitive Agents Specification Language (CASL) that can be used to model the negotiation process of personal agents in a multi-agent system. The CASL also supports modeling an agent's preferences with mental states (e.g., knowledge and goals) at an abstract level [5]. Bharadwaj presented an integrated formal framework for the specification and analysis of multi-agent systems [6]. In his proposed approach, agents are specified in a language called Secure Operations Language (SOL) that supports modular development of secure agents. Luck and d'Inverno used the Z formalism to provide a framework for describing agent

architectures at different levels of abstraction. They proposed a four-tiered hierarchy comprising entities, objects, agents and autonomous agents [7]. 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's work on Concurrent METATEM used temporal logic to represent dynamic agent behaviors [8]. 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 proposed a high-level Petri net, called agent-oriented G-nets, to model and verify behavioral properties of multi-agent systems [9]. Based on the agent-oriented G-net model, certain properties of a multi-agent system, e.g., concurrency and deadlock freeness, can be verified using existing Petri net tools [10]. Hilaire and his colleagues proposed a mechanism for dynamic role playing specified in OZS [11]. The OZS formalism combines Object-Z and statecharts, and can be used to specify multi-agent systems based upon role, interactions and organizations [12]. In their approach, Object-Z is used to specify the transformational aspects; while statecharts are used to specify the reactive aspects of a multi-agent system. Finally, a formal model of agency that is closely related to our proposed work is the Belief-Desire-Intention (BDI) agent model [13]. The BDI agent architecture has come to be one of the well-known and best studied models of practical reasoning agents, which provides an explicit representation for agent mental states, namely beliefs, desires and intentions. Our specification for an agent class follows the BDI agent model; however, we separate domain knowledge (belief), domain goals (desires), and domain plans (intentions) from the agent class, and modularize them into corresponding role classes. Thus, our specified agent class is application independent, and as a consequence, our proposed approach supports separate of concerns, which can significantly simplify the MAS development process due to our modular development methodology.

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 only as conceptual guidelines for agent development during the system 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 [14]. 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 are defined in a role organization and instantiated in a role space; while role instances can be taken or released by agents form an agent society dynamically. We overcome the limitations of current role-based modeling approaches by demonstrating 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 development of multi-agent systems. Typical examples of such efforts include the Gaia methodology and Multiagent Systems Engineering (MaSE) methodology [15, 16]. The Gaia methodology models both the macro (social) aspect and the micro (agent internals) aspect of a multi-agent system [15]. 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 traditional software engineering methodologies [16]. 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. More recently, Hameurlain and Sibertin-Blanc proposed a formal specification model of roles for complex interactions in multi-agent systems [17]. Their approach is based on the RICO (Role-based Interaction COmponents) specification model, which is a role-based interactions abstract model for specification of role components in agent-based applications. Due to their formal specification of role components in Petri nets, their approach supports verification of certain safety properties such as mutual exclusion and concurrency.

Different from the above methodologies, in our proposed approach, we explicitly model conceptual roles and role instances for role-based open MAS. The components of role instances and agent instances are loosely coupled, where agents can take or release role instances at runtime. Furthermore, in our three-layered MAS development model, role classes and agent classes can be designed and implemented independently, which simplifies the development of open role-based MAS.

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 (i.e., 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 (Model-Driven Architecture) approach to automatically transform a meta-model into a model depending on a target platform [18]. 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 [19]. 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 and his colleagues' 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 [20]. The transformation process can be partially automated by using a platform-neutral agent model, called Malaca. More recently, Maria and her colleagues proposed an MDA-based approach to developing MAS [21]. They used MAS-ML, which was an MAS modeling language, to model MAS by creating the platform independent models (PIM), and then tried to transform the MAS-ML models into UML models. 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 [22]. In contrast to the above approaches, our approach emphasizes on developing three levels of models, namely AIPI (Application Independent Platform Independent) model, ASPI (Application Specific

Platform Independent) model, and ASPS (Application Specific Platform Specific) model. The development of the three different levels of models 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 software development teams.

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

3.1 An Organizational Approach

Most of the existing MAS development process models do not support dynamic role assignment for agents; however, many agent-based applications require that agents should be allowed to change their roles at runtime. For example, when we use agent technology to simulate a startup software company, the agent representing the CEO of the company 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 predefined, but the employee may take 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 replace the previous ones by taking the available position roles. It is also possible that, under certain conditions, an employee needs to change a role at runtime. For example, an employee who previously takes a programmer role can be promoted to take a team leader role based on the employee's excellent performance. To model this kind of dynamic and open system, conventional agent development methodologies become quite inappropriate. 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 define a role organization that contains conceptual roles with one of the following relationships among each other, namely *inheritance*, *aggregation*, *association* and *incompatibility*. We also introduce a

concept called *role space* that consists of role instances. Instead of simply using conceptual roles during the system 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 in Figure 1.

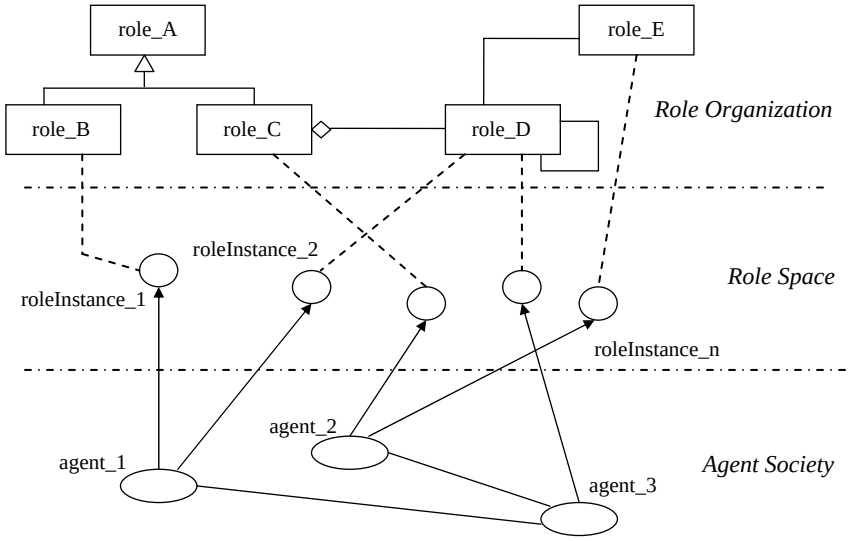


Figure 1. A generic model of role-based open multi-agent systems (adapted from [3])

As shown in Figure 1, a role organization contains a set of conceptual roles (or role classes) with 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 team member role, team members are required to discuss on certain topics. A role space containing a set of role instances is defined based on a role organization. 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 contains a set of agent instances, where agents are

free to join or leave the agent society, and take 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. Note that relationships of *inheritance* and *aggregation* between roles are not passed down as agent relationships.

3.2 Role-Based Agent Model

To formally specify our proposed role-based model of open multi-agent systems, we use Object-Z formalism, which is an extension to the Z formal specification language for modular design of complex systems [2]. Our framework is composed of a set of classes that define the basic constructs in the role-based open MAS model. We now provide some key definitions of the basic constructs in our formal role-based agent 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*, *domain knowledge*, *domain goals*, *domain plans*, *domain actions*, *permissions* and *protocols*. A *role instance* is a fully instantiated role entity.

The class schema *Role* can be formally specified in Object-Z based on its state and operation schemas as shown in Figure 2. The *Role* class consists of a state variable *attributes*, which represents a set of role attributes that describe the characteristic properties of a role, including a role name and a role identification. A *Role* is defined to have a set of domain knowledge, domain goals, domain plans, and domain actions. The state variable *domainKnowledge* specifies a set of domain knowledge that a role must possess to achieve its domain goals. The state variable *domainGoals* describes the current goal states

and a set of domain goals that a role may achieve. The state variable *domainPlans* represents a set of plan trees that are used 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 to perform certain associated actions. The state variable *domainActions* refer to a set of actions that will be trigged to execute when an associated plan tree is selected to carry out. The state variable *permissions* describes 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* defines the way how role instances may interact with each other, e.g., the contract net protocol [23]. Finally, the Boolean state variable *beTaken* defines if a role instance has already been taken by an agent. A *true* value indicates that a role instance has already been taken, thus it is not available for other agents.

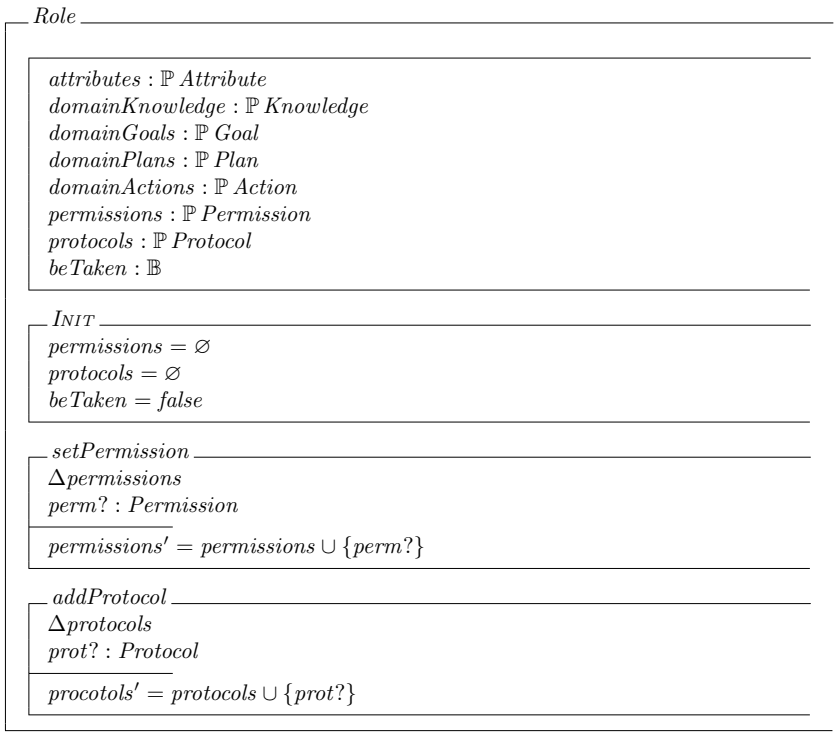


Figure 2. Formal specification of the Role class in Object-Z

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*. Note that, although a role instance has certain goals, plan trees, and actions, it cannot start to execute until it is taken by an agent. It is an agent's responsibility to choose the most appropriate plan and the corresponding actions to achieve a certain goal or a subgoal. Furthermore, we may modify or update role permissions and add new communication protocols to a role instance. This is achieved by providing the operations of *setPermission* and *addProtocol* defined as operation schemas 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 $\lambda \in \{$*inheritance*, *aggregation*, *association*, *incompatibility*$\}$.

In order to define the class schema *RoleOrganization*, we need to define the type of *RoleMetaClass* first. 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 shown in Figure 3. By defining the state variable *roles* as a set of elements of type *RoleMetaClass* or its derivatives, *roles* refers to a set of classes including 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 Section 4.2. The *Role* class is the root class of all its descendents, which exists initially when a role organization is created. 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 specified. This can be done automatically by updating the function *relationship*

with a new mapping of {(r, role?) $\mapsto$ *inheritance*}. However, other relationships between role classes must be set up manually by applying the operation *setRelationship*.

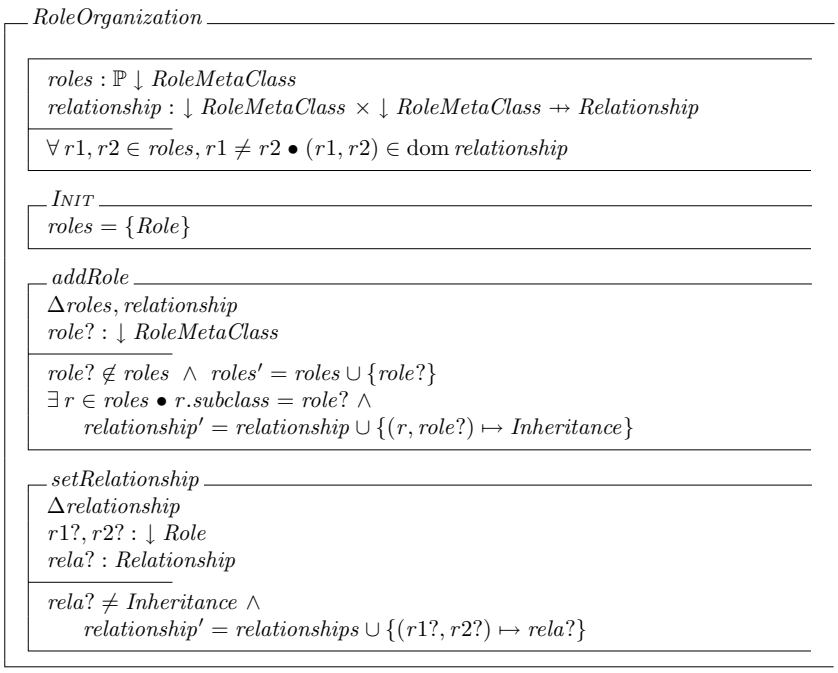


Figure 3. Formal specification of the RoleOrganization class in Object-Z

Definition 3.3 *Role Space*

A *role space* is a container of a set of role instances 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 for software agents to access role instances created in the role space.

The class schema *RoleSpace* is formally defined in Object-Z as shown in Figure 4. 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 it ensures that the types of role instances in a role space are always consistent with that of role instances an agent may take.

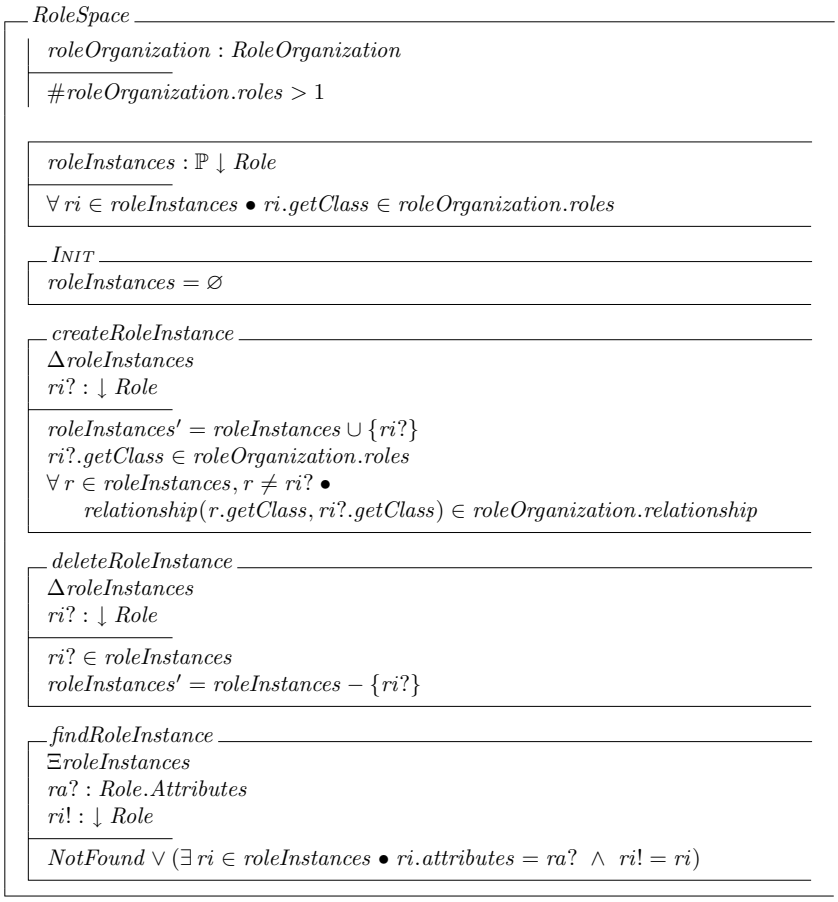


Figure 4. Formal specification of the RoleSpace class in Object-Z

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, which are specified by the operation schema *createRoleInstance* and *deleteRoleInstance*, respectively. In addition, a role space should also provide services for software agents to search for

appropriate role instances according to certain criteria. An example of such services is defined by the operation schema *findRoleInstance* for retrieving a role instance by role attributes.

Definition 3.4 *Agent*

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

The class schema *Agent* can be formally specified based on its state schemas and operation schemas as shown in Figure 5. An agent is identified by its attributes such as the agent name, agent owner and agent identification. As shown in the *Agent* class scheme, an agent has motivations, which is defined as any desire or preference that can lead to the generation and adoption of goals, and also affect the outcome of the reasoning or behavioral task intended to satisfy those goals [7]. The *sensor* of an agent perceives related environment changes and transforms the inputs into a set of sensor data. The *reasoningMechanism* is defined as a 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 *roleMatchingMechanism* 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. To realize an agent's goal, a committed plan is derived according to the role instances and the knowledge possessed by the agent, which includes the agent knowledge and the domain knowledge of each role instance taken by the agent. This mechanism is defined as a function *committedPlan* in the *Agent* class. The state variable *rolesTaken* refers to a set of roles that are currently taken by the agent. The *Agent* class schema also defines two fundamental operations: *takeRole* and *releaseRole*. The *takeRole* operation takes an available role instance from a role space, and set it as unavailable to other agents. On the other hand, the *releaseRole* operation releases a role instance and set it to be available for other agents.

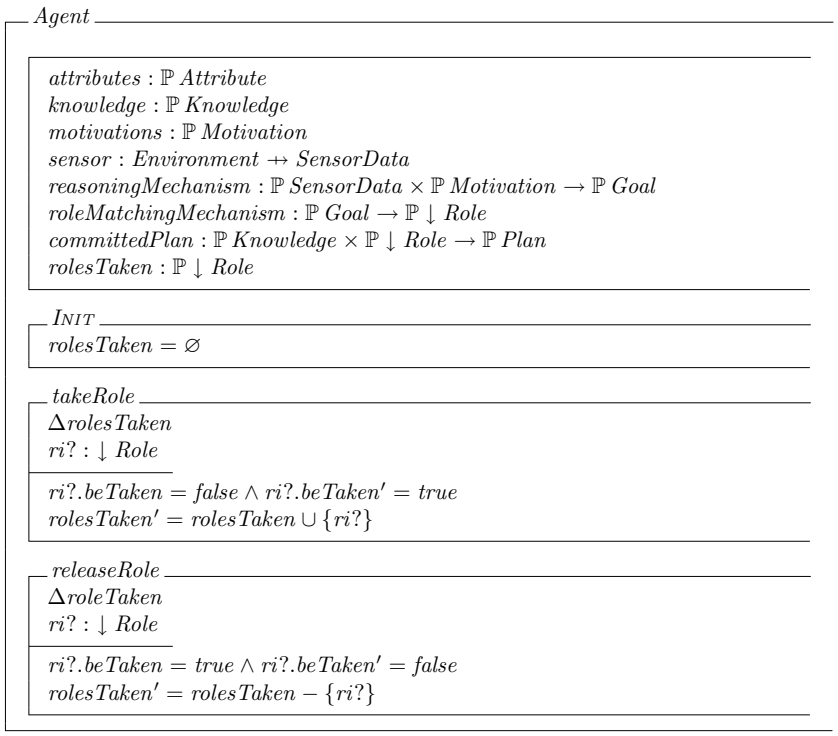


Figure 5. Formal specification of the Agent class in Object-Z

Definition 3.5 *Agent Society*

An *agent society* defined upon a role organization consists of a set of agent instances of type *Agent*. An agent society provides services for agent instances to join or leave the agent society dynamically.

The structure of an agent society is often determined by organizational design that is independent of the agents themselves [24]. The class scheme of *AgentSociety* is formally defined in Object-Z as shown in Figure 6. Since both role spaces and agent societies are defined on role organizations, a correspondence exists between a role space and an agent society when they share the same role organization. This implies that a role instance created in a role space can only be taken by an agent from an agent society with the same role organization; meanwhile, any agent belongs to an agent society must take at least one role instance from a role space with the same role organization. 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 role instances from different role spaces.

Figure 6. Formal specification of the AgentSociety class in Object-Z

As shown in the *AgentSociety* class scheme, an *agent society* defines a state variable *agentInstances* that refers to a set of agent instances of type *Agent*. The state variable *interaction* is defined as a function, which applies to a source agent and a destination agent, and may generate a message. An agent instance belonging to an agent society takes role instances of types 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. In an agent society, agent instances can join or leave the agent society dynamically, which are specified by the operation schema *join* and *leave*, respectively.

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

Inspired by OMG's Model-Driven Architecture (MDA) [25], 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 automated development of role-based open MAS. In other words, it is possible to build a MAS development tool to automatically generate partial code for a role-based open MAS application.

4.1 Three-Layered Development Model

Our three-layered development model consists of three relatively independent models, namely Application Independent Platform Independent (AIPI) model, Application Specific Platform Independent (ASPI) model, and Application Specific Platform Specific (ASPS) model. The purpose of this approach is to separate software architecture from an application domain and to separate application logic from the underlying technologies to support reusability in an agent development process.

As shown in Figure 7, the three-layered development model is defined in three steps. The first step is to define the AIPI 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 AIPI model and 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 AIPI model for role-based open MAS. For example, the *Role* class and the *RoleSpace* class defined in Object-Z are also defined as a *Role* class and *RoleSpace* class in the AIPI model, which are illustrated in a simplified AIPI model in UML class diagram as shown in Figure 8.

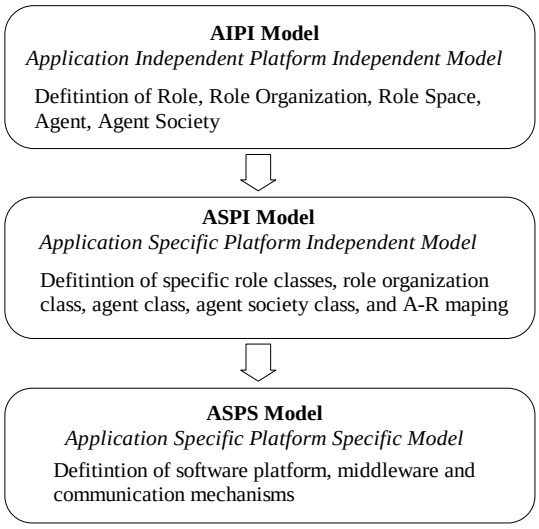


Figure 7. Three-layered development model for developing role-based open MAS

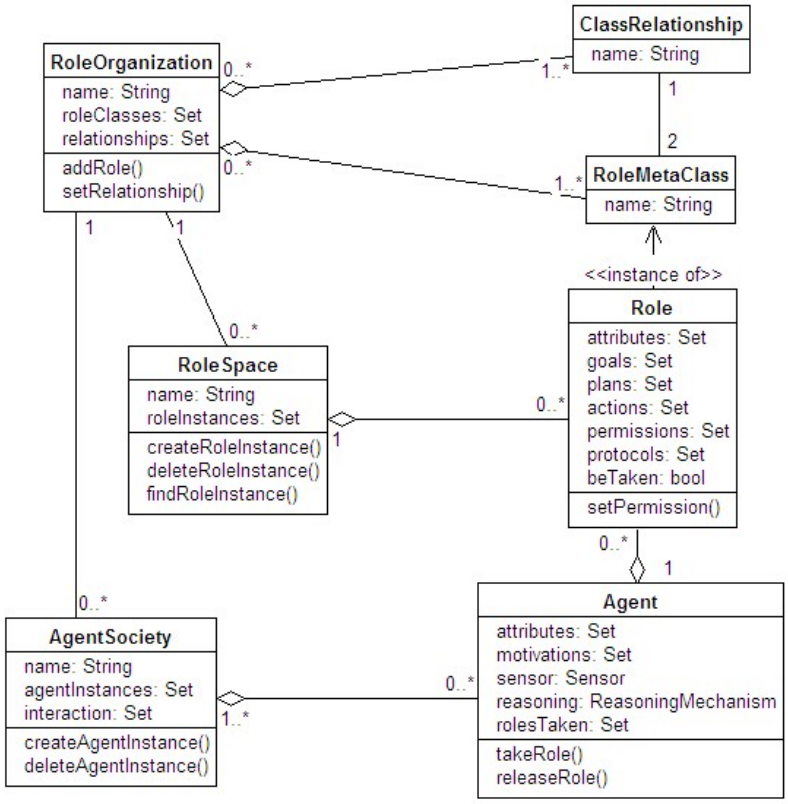


Figure 8. The AIPI model of role-based open MAS in UML class diagram

From Figure 8, we can see that a *RoleOrganization* contains instances of *ClassRelationship* and instances of *RoleMetaClass*, which are *Role* classes. We will describe *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. Note 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. As an example for such a scenario, a company may have any number of employees; while an employee may 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 Microsoft .NET platform is not considered in such a model. One advantage of using role-based agent development is to simplify the definition of an *Agent* class such that certain capabilities for achieving a goal, including domain knowledge and domain plans, 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 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 [26] for agent communications. In Section 5, we use a case study to show how to develop the ASPS model based on the communication mechanism that is supported by ADK (Agent Development Kit) [27].

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 definitions to these relationships as well as some related key concepts such as a *leading* role and a *composite* role.

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 [28].

Definition 4.2 *Leading Role*

A *leading role* is responsible for hiring other roles in achieving its goal. For example, a company CEO is a leading role, which is responsible for hiring new employees. The *LeadingRole* class is defined as a subclass of the *Role* class [3]. Therefore, a leading role inherits all the data fields, e.g., *attributes*, *domaingoals* and *domainplans*, as well as all operations defined in the *Role* class. In addition, a leading role records the number of role instances that are required 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 *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 role classes must be a subclass of the *CompositeRole* class. The *aggregation* relationship between role classes 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 [29]. 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 `[cond]` in front of the association name. The association relationship only exists between instances of role classes when `cond` is true.

Definition 4.6 *Incompatibility Relationship*

Under certain conditions, 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 9. In this example, a banker who works for a bank is not allowed to borrow loan from the same bank.

Figure 9. An example of incompatibility relationship

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 [24]. In our approach, the openness of a multi-agent system is specified by 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 Θ defined upon role organization Φ 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 *A-R mapping* is defined by the following function:

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

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 α in agent society Θ. The leading agent α is responsible for initializing and managing the agent society Θ. Ordinary agents representing different users may join the agent society Θ, and are ready to take role instances from the role space Γ.
2. *Creating role instances:* The leading agent α makes a request to the role space Γ to instantiate the major leading role class that is defined as a subclass of the *LeadingRole* class in the role organization Φ. The leading agent α takes the major leading role instance as soon as it is available. The leading agent α further makes requests to the role space Γ to create all the role instances that are needed to achieve its goal.

3. *Role assignment*: The role space Γ waits for requests from an ordinary agent β in agent society Θ, and do the following:

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

      a. Search the role space Γ for any available role instances with the requested role attributes.

      b. If there is a match, reserve the role instance and notify agent β to take that role instance.

         Else notify agent β 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 β, and check its role incompatibility as follows:

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

            If $\Phi.relationship(r_1.getClass, r_2.getClass) == incompatibility$, and the condition for that relationship is *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 β.

   3.2 *Setting up agent interaction relationships:* The role space Γ notifies the agent society about the updated role assignment and updates the interaction relationships between agent β and other agents from agent society Θ 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 $\Phi.relationship(r_1.getClass, r_2.getClass) == association$, then $(\beta, \gamma) \in \text{dom}\ \Theta.interaction$.

   3.3 Goto stage 3.

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

4.4 Tool Support for Design of ASPI Model

To facilitate rapid development of the ASPI model, we developed a prototype Role-based Agent Development Environment (RADE). 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 enable automatic role assignment using *A-R mapping*. The toolkit for design of a role-organization is similar to the Rational Rose toolkit [30], but it is specific to support design of role classes and their relationships. Figure 10 shows the user interface of the RADE prototype for design of a role organization. When the high-level design of the role organization is complete, 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 *RoleOrganization* package can be automatically generated by clicking on the "CodeGen" menu on the top of the window.

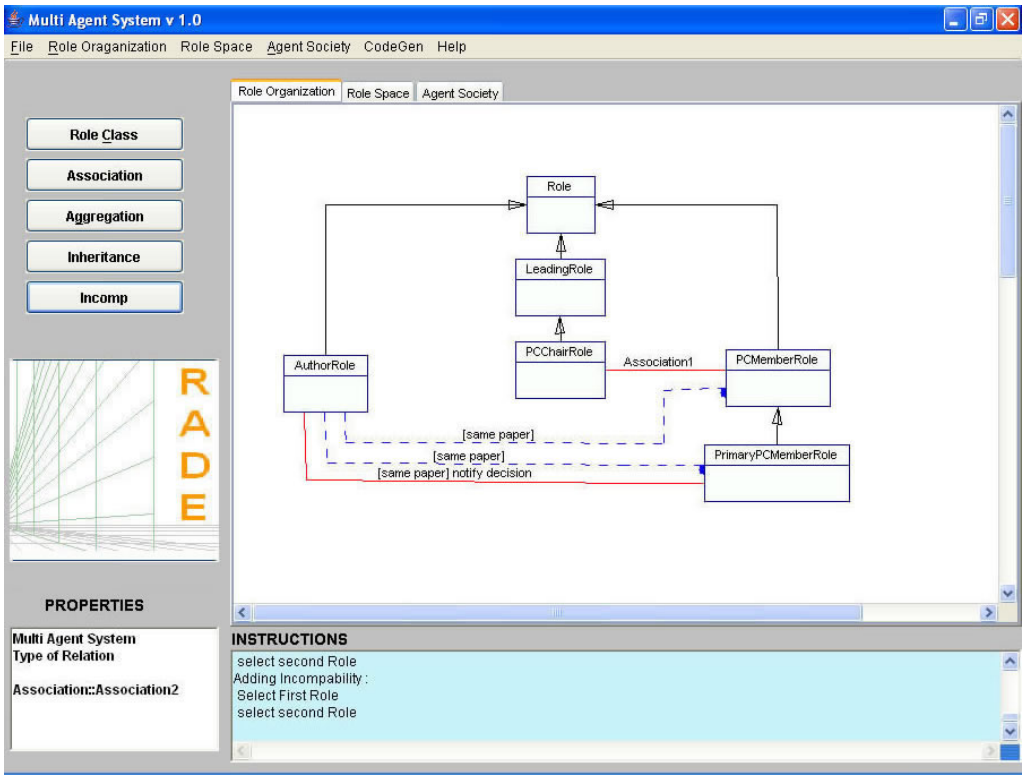


Figure 10. User Interface of the RADE prototype for design of role organization

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. Note 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 process requests from ordinary agents for querying, taking or releasing role instances from the role space. During the *A-R mapping* process, possible conflicts of role instances taken by an agent are marked on that agent. If the condition for such a conflict is true and the conflict cannot be resolved, the agent must make a request to the role space to release any role instances *in conflict*, and adjust its goals or motivations accordingly before making new requests to take role instances from the role space.

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. An agent society is essentially a virtual society that contains only the proxy of each registered agent running on a remote machine. Furthermore, 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 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 a middleware associated with the agent society server. Figure 11 shows the ASPS model architecture of a role-based open MAS.

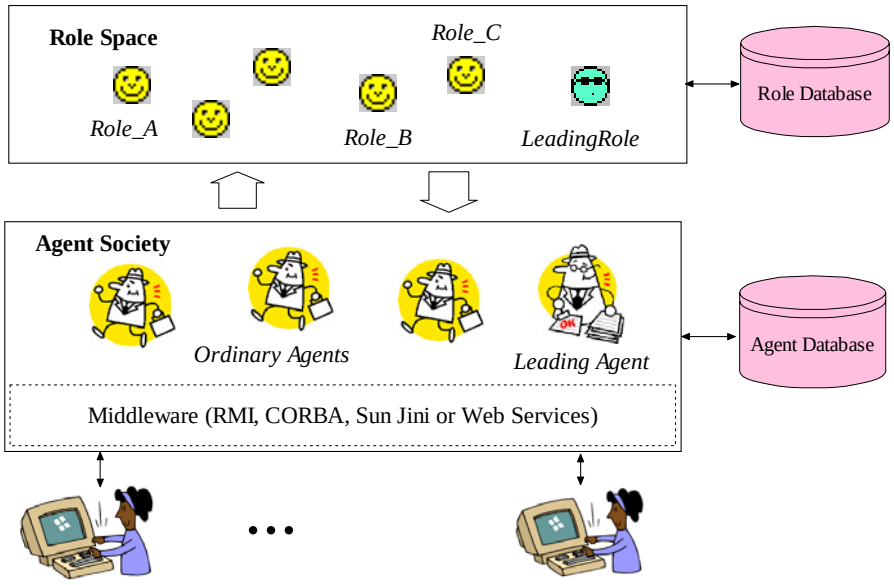


Figure 11. The ASPS model architecture of role-based open MAS

Since each agent works on behalf of a human user, the system provides a user interface for initial instructions to agents. Note that an agent running on a remote machine, so the ordinary agents and the leading agent shown in Figure 11 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. Both the role space and the agent society are associated with a database, namely the *Role Database* and the *Agent Database*, which record information about role instances and role assignments, and information about agents currently in the society, respectively. For security purpose, both the agent society and role space should ensure that only trustable agents and roles can be recorded in the agent database and the role database, respectively. Furthermore, the agent society should also be responsible for agents to take appropriate role instances from the role space by exerting certain amount of control over role assignment and enforcing related security policies. Detailed descriptions about security related issues in role-based open MAS is beyond the scope of this paper; however, some preliminary work on agent security can be found in our previous work [31].

5. A Case Study: Organizing a Conference

Consider an example of organizing a conference, which requires different roles such as 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 will be reviewed by at least *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 is illustrated in Figure 12. As shown in the figure, the *PCChairRole* class is defined as a subclass of the *LeadingRole* class; while the *AuthorRole* and *PCMemberRole* classes are defined as subclasses of the *Role* class. The *PrimaryPCMemberRole* is a special *PCMemberRole* that makes decisions on paper acceptance; therefore, it is defined as a subclass of the *PCMemberRole* class. A *PCChairRole* is responsible for assigning papers to a *PCMemberRole*, thus an "assign papers" association relationship is defined between these two classes. A *PrimaryPCMemberRole* makes decisions on accepting a paper; therefore, it has an association relationship with the paper's author for notification of the result. In addition, the *AuthorRole* has an incompatibility relationship with both the *PCMemberRole* and the *PrimaryPCMemberRole*. This implies that at any time a *PCMemberRole* or a *PrimaryPCMemberRole* cannot review his/her own paper.

When we design the ASPS model, we use Sun Jini as a middleware for agents to communicate with agent societies and role spaces, and also 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 [32]. 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 that we developed previously [27]. 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 using asynchronous message passing. For a detailed description of this approach, refer to previous work [27] for how agents can communicate with each other asynchronously.

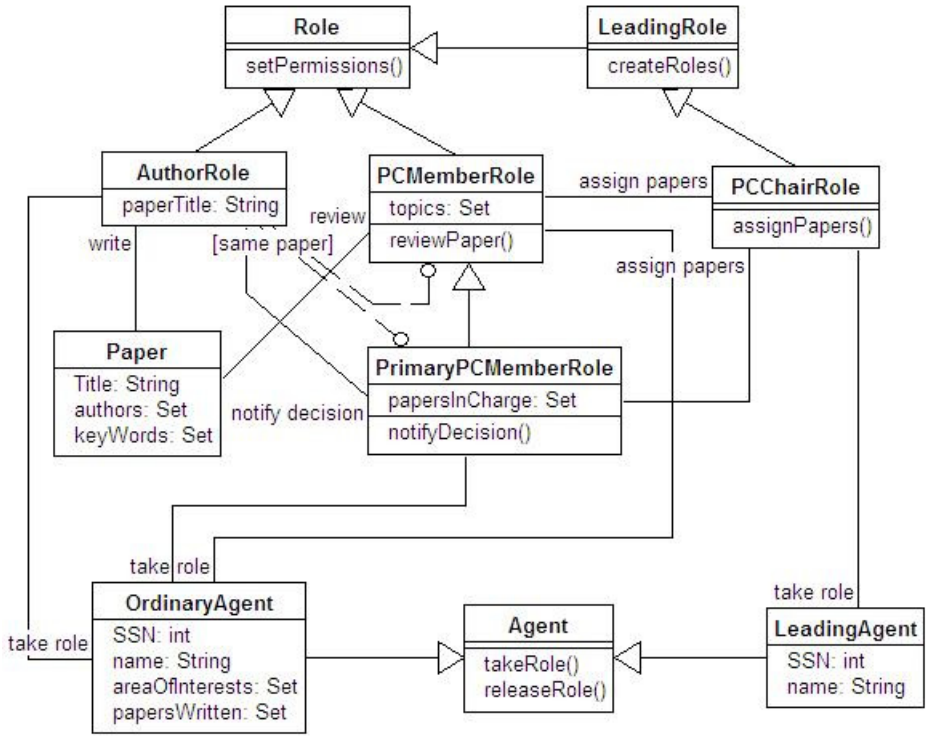
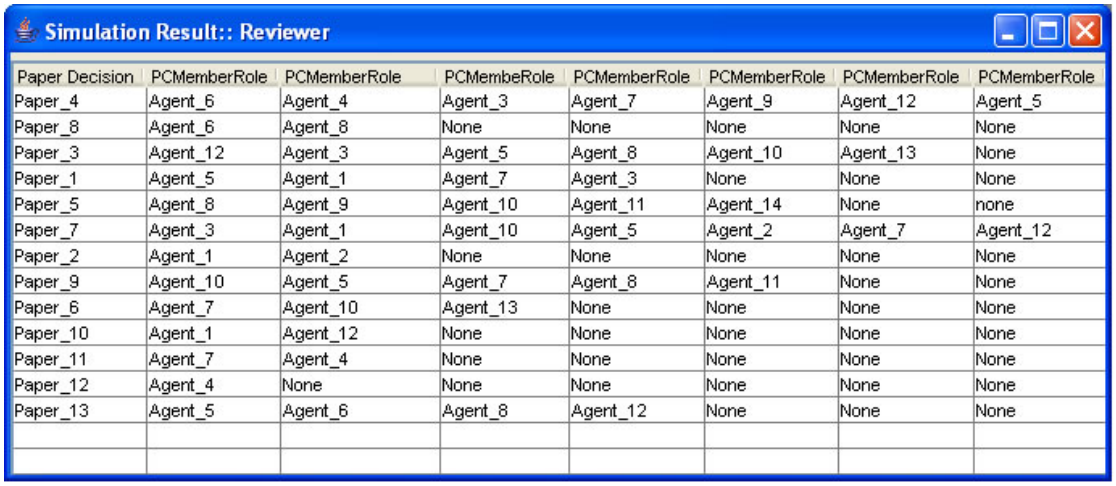


Figure 12. The ASPI model of the agent-based conference organizer application

The open multi-agent system application developed based on the ASPS model provides a user interface for a user to submit a paper 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 13.

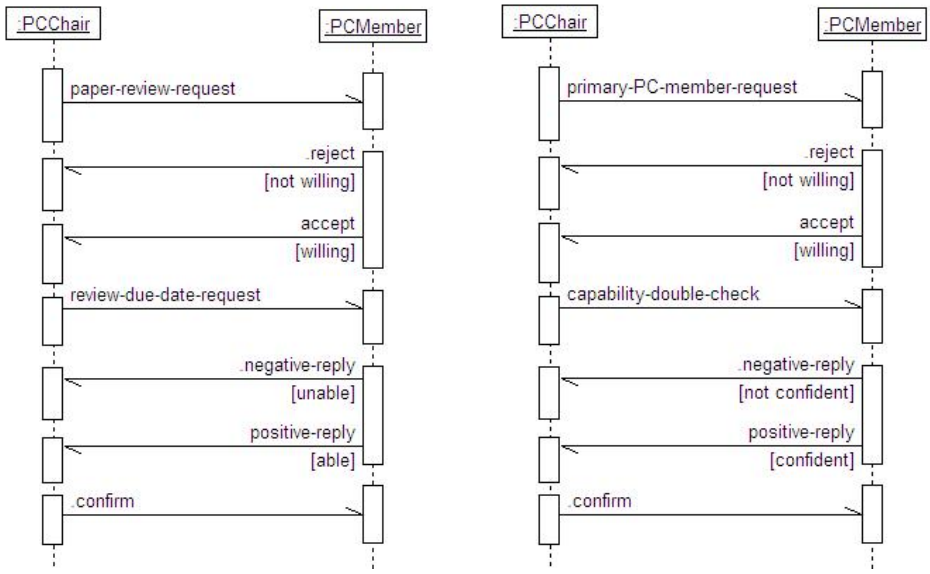


Paper Decision	PCMemberRole	PCMemberRole	PCMembeRole	PCMemberRole	PCMemberRole	PCMemberRole	PCMemberRole
Paper_4	Agent_6	Agent_4	Agent_3	Agent_7	Agent_9	Agent_12	Agent_5
Paper_8	Agent_6	Agent_8	None	None	None	None	None
Paper_3	Agent_12	Agent_3	Agent_5	Agent_8	Agent_10	Agent_13	None
Paper_1	Agent_5	Agent_1	Agent_7	Agent_3	None	None	None
Paper_5	Agent_8	Agent_9	Agent_10	Agent_11	Agent_14	None	none
Paper_7	Agent_3	Agent_1	Agent_10	Agent_5	Agent_2	Agent_7	Agent_12
Paper_2	Agent_1	Agent_2	None	None	None	None	None
Paper_9	Agent_10	Agent_5	Agent_7	Agent_8	Agent_11	None	None
Paper_6	Agent_7	Agent_10	Agent_13	None	None	None	None
Paper_10	Agent_1	Agent_12	None	None	None	None	None
Paper_11	Agent_7	Agent_4	None	None	None	None	None
Paper_12	Agent_4	None	None	None	None	None	None
Paper_13	Agent_5	Agent_6	Agent_8	Agent_12	None	None	None

Figure 13. Simulation results for paper assignment (initial result)

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 (e.g., Paper_4); while some paper only has one reviewer (e.g., 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 negotiate with the requested PC member to achieve its goal. A simplified interaction protocol for such negotiation is shown in Figure 14 (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. 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 communications is illustrated in Figure 14 (b).

The user interface of the PC chair agent is illustrated in Figure 15. From the screenshot, 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*.



(a) (b)

Figure 14. Examples of interaction protocols between a PC chair and a PC member

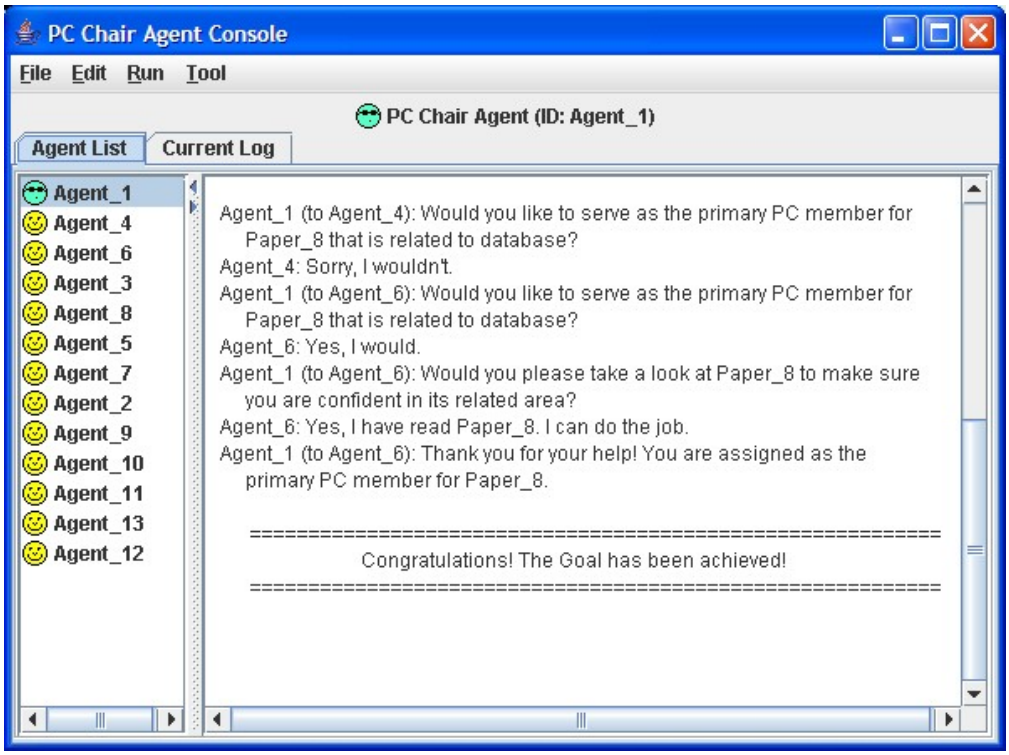
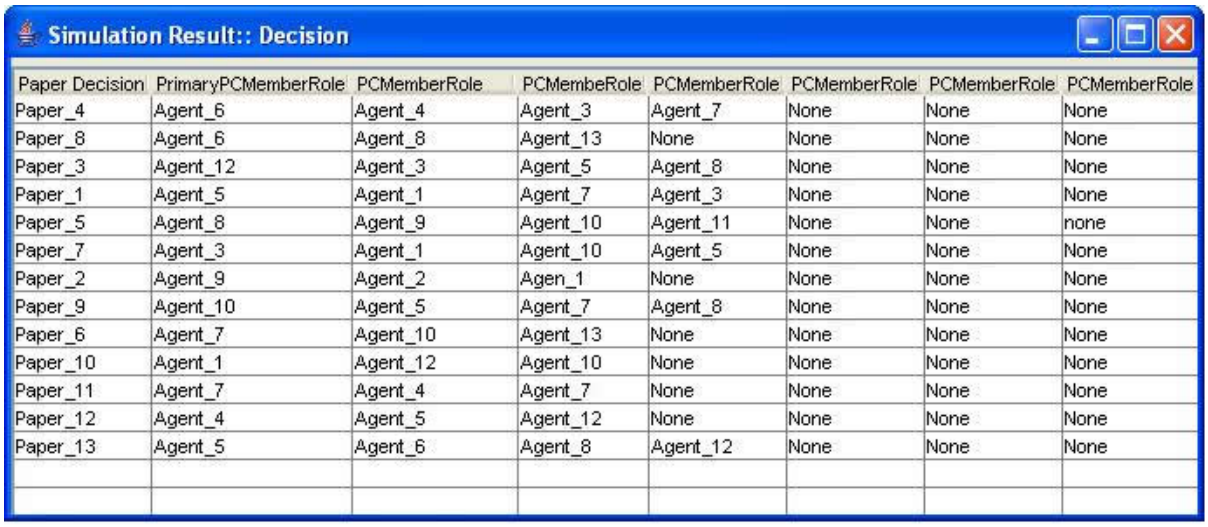


Figure 15. User interface of the PC chair agent

After the paper assignment become balanced and each paper has been assigned to a primary PC member, the system generates the final paper assignment table. Figure 16 shows the simulation results for the final paper assignment.



Paper Decision	PrimaryPCMemberRole	PCMemberRole	PCMembeRole	PCMemberRole	PCMemberRole	PCMemberRole	PCMemberRole
Paper_4	Agent_6	Agent_4	Agent_3	Agent_7	None	None	None
Paper_8	Agent_6	Agent_8	Agent_13	None	None	None	None
Paper_3	Agent_12	Agent_3	Agent_5	Agent_8	None	None	None
Paper_1	Agent_5	Agent_1	Agent_7	Agent_3	None	None	None
Paper_5	Agent_8	Agent_9	Agent_10	Agent_11	None	None	none
Paper_7	Agent_3	Agent_1	Agent_10	Agent_5	None	None	None
Paper_2	Agent_9	Agent_2	Agen_1	None	None	None	None
Paper_9	Agent_10	Agent_5	Agent_7	Agent_8	None	None	None
Paper_6	Agent_7	Agent_10	Agent_13	None	None	None	None
Paper_10	Agent_1	Agent_12	Agent_10	None	None	None	None
Paper_11	Agent_7	Agent_4	Agent_7	None	None	None	None
Paper_12	Agent_4	Agent_5	Agent_12	None	None	None	None
Paper_13	Agent_5	Agent_6	Agent_8	Agent_12	None	None	None

Figure 16. Simulation of paper assignment (final decision)

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. 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 application on 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 the RADE project, we will incorporate these transformation tools into RADE, and also define security mechanisms to ensure the trustworthiness of role-based open multi-agent systems.

Acknowledgments: This material is based upon work supported by the Research Seed Initiative Grant, College of Engineering, UMass Dartmouth. We thank Prof. Nabil Hameurlain and Dr. Vincent Hilaire for providing some valuable references related to the RADE project. We also thank all anonymous referees for the careful review of this paper and the many suggestions for improvements they provided.

References

[1] T. Juan, A. Pearce, and L. Sterling, "ROADMAP: Extending the Gaia Methodology for Complex Open Systems," In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS2002)*, Bolognia, Italy, 2002, pp. 3-10.

[2] R. Duke, G. Rose and G. Smith, "Object-Z: a Specification Language Advocated for the Description of Standards," *Computer Standards and Interfaces*, Vol. 17, Issues 5-6, 1995, pp. 511-533.

[3] H. Xu and X. Zhang, "A Methodology for Role-Based Modeling of Open Multi-Agent Software Systems," In *Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS 2005)*, May 24-28, 2005, Miami, Florida, USA, pp. 246-253.

[4] F. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur, "DESIRE: Modeling Multi-Agent Systems in a Compositional Formal Framework," *International. Journal of Cooperative Information Systems*, Vol. 6, No. 1, 1997, pp. 67-94.

[5] S. Shapiro and Y. Lespérance, "Modeling Multiagent Systems with CASL - A Feature Interaction Resolution Application," In Castelfranchi, C. and Lespérance, Y., editors, *Intelligent Agents Volume VII - Proceedings of the 2000 Workshop on Agent Theories, Architectures, and Languages (ATAL-2000)*, LNAI, vol. 1986, 244-259, Springer-Verlag, Berlin, 2001.

[6] R. Bharadwaj, "A Framework for the Formal Analysis of Multi-Agent Systems," In *Proceedings of the Conference on Formal Approaches to Multi-Agent Systems (FAMAS)*, affiliated with ETAPS 2003, April 12, 2003, Warsaw, Poland.

[7] M. Luck and M. d'Inverno, “A Formal Framework for Agency and Autonomy,” In *Proceedings of the First International Conference on Multi-Agent Systems* (ICMAS-95), AAAI Press / MIT Press, 1995, pp. 254-260.

[8] M. Fisher, “Representing and Executing Agent-Based Systems,” In *Proceedings of the International Workshop on Agent Theories, Architectures, and Languages*, M. Wooldridge and N. Jennings (eds.), Lecture Notes in Computer Science, vol. 890, Springer-Verlag, 1995, pp. 307-323.

[9] H. Xu and S. M. Shatz, “A Framework for Modeling Agent-Oriented Software,” In *Proceedings of the 21st International Conference on Distributed Computing Systems* (ICDCS), April 2001, Phoenix, Arizona, pp. 57-64.

[10] H. Xu and S. M. Shatz, “A Framework for Model-Based Design of Agent-Oriented Software,” *IEEE Transactions on Software Engineering* (IEEE TSE), January 2003, Vol. 29, No. 1, pp. 15-30.

[11] V. Hilaire, A. Koukam, and P. Gruer, “A Mechanism for Dynamic Role Playing,” In *Agent Technologies, Infrastructures, Tools and Applications for E-Services*, LNAI 2592, Springer Verlag, 2002.

[12] V. Hilaire, O. Simonin, A. Koukam, and J. Ferber, “A Formal Approach to Design and Reuse of Agent and Multiagent Models,” In *Proceeding of the Fifth International Workshop on Agent-Oriented Software Engineering* (AOSE-2004), AAMAS 2004, New York, July 2004.

[13] D. Kinny, M. Georgeff, and A. Rao, “A Methodology and Modeling Technique for Systems of BDI Agents,” In *Proceedings of the Seventh European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, W. Van de Velde and J. W. Perram, eds., LNAI Vol. 1038, Springer-Verlag: Berlin, Germany, 1996, pp. 56-71.

[14] G. Cabri, L. Ferrari, and L. Leonardi, “Agent Role-Based Collaboration and Coordination: a Survey about Existing Approaches,” In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Oct. 2004, pp. 5473-5478.

[15] M. Wooldridge, N. R. Jennings, and D. Kinny, “The Gaia Methodology for Agent-Oriented Analysis and Design,” *International Journal of Autonomous Agents and Multi-Agent Systems*, Vol. 3, No.3, 2000, pp. 285-312.

[16] S. A. DeLoach, M. F. Wood and C. H. Sparkman, “Multiagent Systems Engineering,” *International Journal of Software Engineering and Knowledge Engineering*, Vol. 11, No. 3, June 2001.

[17] N. Hameurlain and C. Sibertin-Blanc, “Specification of Role-based Interactions Components in Multi-Agent Systems,” *Software Engineering for Multi-Agents System III: Research Issues and Practical Applications*, Lecture Notes in Computer Science, LNAI/LNCS, pp 180-197, Vol. 3390, Springer-Verlag, 2005.

[18] C. Bernon, M. Cossentino, M. Gleizes, P. Turci, and F. Zambonelli, “A Study of Some Multi-Agent Meta-Models,” In *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering* (AOSE-2004), The Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004), New York, USA, July 2004.

[19] D. Gracanin, S. A. Bohner, and M. Hinchey, “Towards a Model-Driven Architecture for Autonomic Systems,” In *Proceedings of 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems* (ECBS’04), 2004, pp. 500-505.

[20] M. Amor, L. Fuentes, A. Vallecillo, “Bridging the Gap Between Agent-Oriented Design and Implementation using MDA,” In *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering* (AOSE 2004), LNCS 3382, pp.93-108, New York, 2004.

[21] B. A. De Maria, V. T. Silva, and C. J. P. Lucena, “Developing Multi-Agent Systems Based on MDA,” In *Proceedings of the 17th Conference on Advanced Information Systems Engineering* (CAiSE’05), Porto, Portugal, June 13-17, 2005.

[22] D. Thomas, “MDA: Revenge of the Modelers or UML Utopia?” *IEEE Software*, Vol. 21, No. 3, May/June, 2004, pp. 15-17.

[23] R. G. Smith, “The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver,” *IEEE Transactions on Computer*, Vol. C-29, 1980, pp. 1104-1113.

[24] M. Dastani, V. Dignum and F. Dignum, “Role-Assignment in Open Agent Societies,” In *Proceedings of the Second International Joint Conference on Autonomous Agents & Multiagent Systems* (AAMAS 2003), Melbourne, Australia, ACM Press, 2003, pp. 489-496.

[25] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.

[26] R. Nagappan, R. Skoczylas, R. Sriganesh, *Developing Java Web Services: Architecting and Developing Secure Web Services Using Java*, Wiley, 2002.

[27] H. Xu and S. M. Shatz, “ADK: An Agent Development Kit Based on a Formal Model for Multi-Agent Systems,” *Journal of Automated Software Engineering* (AUSE), October 2003, Vol. 10, No. 4, pp. 337-365.

[28] S. Stepney, R. Barden, D. Cooper, editors, *Object Orientation in Z*. Workshops in Computing. Springer, 1992, pp. 59-77.

[29] J. Arlow, I. Neustadt, *UML and the Unified Process: Practical Object-Oriented Analysis and Design*, Addison-Wesley, 2002, pp.142-169.

[30] T. Quatrani, *Visual Modeling with Rational Rose 2002 and UML*, 3rd Edition, Addison-Wesley Professional, 2002.

[31] H. Xu, Z. Zhang, and S M. Shatz, “A Security Based Model for Mobile Agent Software Systems,” *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, August 2005, Vol. 15, No. 4, pp. 719-746.

[32] K. Arnold, B. O’Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath, *The Jini Specification*, Addison-Wesley, 1999.