Towards Automated Development of Multi-Agent Systems Using RADE

Xiaoqin Zhang and Haiping Xu
Department of Computer and Information Science
University of Massachusetts at Dartmouth
North Dartmouth, MA 02740
Email: x2zhang, hxu@umassd.edu

***Abstract*— To facilitate the development of multi-agent systems and improve the reusability, robustness and feasibility of these systems, we proposed a role-based agent development framework (RADE). In this paper, we present more details on the design of agents and motivations within such framework. We introduce a practical approach to modeling agent's motivation and specifying agent's goals, where a role-mapping mechanism is developed based on this design. We also introduce the RTÆMS language based on the extension of TÆMS to model the plan tree for each goal. This representation is used to support the development of general planning/scheduling and collaboration/cooperation mechanisms.**

Keywords: Role-Based Agent Development, Multi-Agent Systems, Motivations, RTÆMS Language, Role-Agent Mapping

I. Introduction

Multi-Agent System (MAS) has become a suitable programming paradigm for distributed information systems and applications, where the resources, data, control and services are widely distributed. However, the implementation of MAS is not a trivial job, it takes considerable time and requires highly experienced programmers. It is also difficult to test and maintain the multi-agent system because of its complexity. The reusabilities of such systems are low, it is unlikely to use an existing system for another application domain with little or minor change.

We are working on a set of technologies and mechanisms to ease and formalize the development of MAS, and to increase its reliability and reuse-ability too. The basic idea is to separate concerns. There are multiple issues in a multi-agent systems, such as problem-solving issue, coordination issue, organization issue, communication issue, security issue, etc. Some of them are application-dependent, others are not. Some of them are platform-dependent and others are not. We have proposed a three-layered development process: the application-independent, platform-independent model, the application-specific, platform-independent model, and the application-specific and platform-specific model are developed in the three consecutive phases respectively. Another approach to separating concern is to separate the domain knowledge and the intelligent problem-solving capabilities. We adopt a role-based modeling approach, conceptual roles are defined with the domain related knowledge, such as goals, permissions, organizational relationship, and coordination protocols, etc; where agent is a concert entity equipped with motivations, resources and problem-solving capabilities. However, our role-based approach is different from other proposed role-based approaches ([1], [2], [3]). We introduce the concept of *role instance*, which is a concrete implementation of a conceptual role, and this approach provides a stronger support for system openness and dynamics. Our approach supports the dynamic creation of role instances, and agents can take a role instance and then create more role instances according to the needs to fulfill its goal. The basic idea of the role-based agent development environment (RADE) is illustrated in Figure 1.

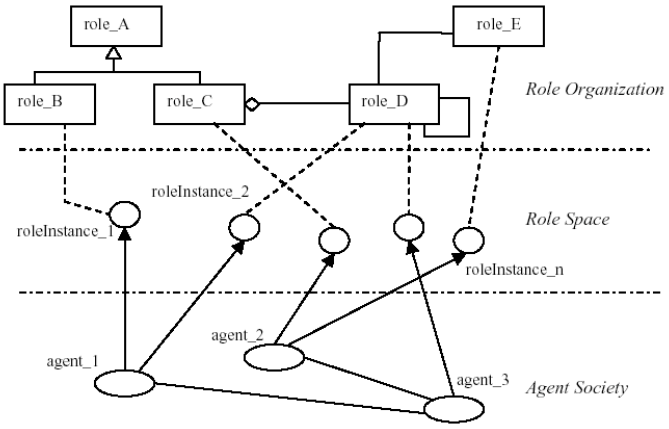


Fig. 1. A generic model of role-based open multi-agent systems (adapted from [4])

The three-layered development model and the role-based design approach are presented in [4] and [5], in this paper we focus on the design and implementation of agent, including the mapping from role instances to agents and the interaction among agents. This paper is organized as the follows. The detailed description of agent is presented in Section II. The definition and more details about role are described in Section III. Section IV describes operation details of multi-agent systems including the mapping mechanisms, the planning and scheduling, and the collaboration and the cooperation among agents. A case study is presented in Section V to illustrate the previous ideas. Lastly, we will present the conclusion and discuss the future work in Section VI.

II. Agent Definition

Agent is an entity with attributes, motivations, sensors and a set of reasoning mechanisms. Agent attributes include agent

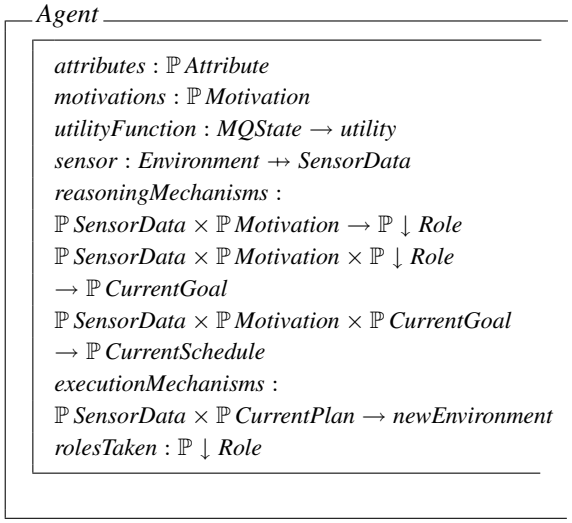


Fig. 2. Definition of Agent Class

names, user, identification and other descriptive characteristics, the values of these attributes are set when an agent instance is instantiated from the agent class. Different agent instances have different attribute values. According to [6], *motivation* is defined as "any desire or preference that can lead to the generation and adoption of goals and which affects the outcome of the reasoning or behavioral task intended to satisfy those goals". Motivation is the key for agent to decide which goals it should pursue and how to pursue a goal.

A. Agent Motivation

We adopt a quantitative view of motivation in our practice. *Motivation* is defined as a set of *motivation quantities* (*MQ*s) [7] that the agent tracks and accumulates. Each *MQ* is associated with a preference function[1] Each MQ represents progresses towards an abstract goal. An abstract goal is a long-term commitment to make progress toward certain direction but not a concrete task with a specified plan. For example, the designed purpose of a personal assistant agent is to serve its owner. With this purpose, the agent has motivation to manage the owner's address-book, organize daily appointment and purchase items desired by the owner. Therefore, this agent's motivation is represented as a set of three types of MQ:

$$\begin{aligned} &Motivation \quad of \quad Personal \quad Assistant \\ &= \{MQ_{manageAddressbook}, MQ_{organizeActivities}, MQ_{purchaseItems}\} \end{aligned} \quad (1)$$

A concrete goal (task), i.e., schedule a meeting with the family doctor, contributes the abstract goal *organize daily activity*, which is represented by the generation of a certain amount of $MQ_{organizeActivities}$. Agent is able to determine which role it should take by analyzing the (concrete) goal of the role and to find if the goal generates a certain type of *MQ* that this agent is interested in.

Each MQ_i is associated with a preference function U_{f_i}, which maps a specific amount of MQ_i into some quantity of utility U_i: $U_{f_i}(MQ_i \rightarrow U_i)$, where U_i is the utility associated with MQ_i and it is not inter-exchangeable with other type of utility. The overall utility of the agent U_{agent} depends on the accumulation of the different types of *MQ*s in its motivation: $\{MQ_i, MQ_j, MQ_k, ...\}$. The function: $U_{agent} = \gamma(U_i, U_j, U_k, ...)$ describes how different types of utilities are contributed to the agent's overall utility.

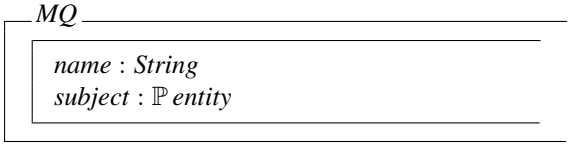
[1]The concept of *MQ* is originated from the work on soft real-time agent control by Wagner and Lesser. We extended the original *MQ* framework to make it more suitable for general agent design in RADE process.

B. Extended Definition of MQ to Support Automatic Development and Dynamic Organizations

The original MQ framework is intended to support soft real-time agent control, it is assumed that all MQ types are designed by the user when the agent is created, and the types of MQ are fixed in the runtime of the system. This assumption works fine for small-scale multi-agent systems when all agents are created by hand and the organization structure is fixed.

However, this original design does not fit the need of engineering the development of multi-agent system and support the dynamic organization structure. For example, it would be nice to automatically create two personal assistant agents for user A and user B, each agent has the motivations to manage the owner's address-book, organize daily appointment and purchase items desired by the owner. If we use the original definition as described in (1), confusion is unavoidable since the agents cannot distinguish their goals to serve different users. The confusion can be resolved by designing different types of *MQ*s with different names, such as: $MQ_{organizeActivitiesForUserA}$ and $MQ_{organizeActivitiesForUserB}$, however, this approach deviates from the intention to use an unified class design for all personal assistant agents. So, we extend the original MQ framework by introducing a parameter, namely *subject*, into the definition of *MQ*: every unique *MQ* type is defined by the *MQ* name and the *MQ* subject. The subject is the entity who is being served or benefited from the achievement of this *MQ*. For example, $MQ_{organizeActivities}(A)$ represent the motivation to organize activities for user A (assume "A" is the identification for this unique user). $MQ_{organizeActivities}(A)$ and $MQ_{organizeActivities}(B)$ are different *MQ*s and they are not inter-exchangeable. In the design phase, a unique pattern $MQ_{organizeActivities}(User)$ can be used for the personal assistant agent class, "User" refers to the agent's user, which is one of the attributes of the agent. When the two personal assistance agent instances are instantiated for user A and B, they have different values for their attributes such as name, user and identification.

The formal definition of *MQ* type is:



A brief representation is: $MQ_{name}(MQ_{subject})$. The subject of MQ is a set of entities, which can be defined in one of the following ways or a combination of them:

1) List the identification of the entities that belongs to this set, $\{id_1, id_2, ...id_n\}$, id_i is the identification of entity or a function that returns an entity identification, such as $Owner(id)$.

2) Specify the conditions for an entity to belong to this set, $\{x \mid condition(x)\}$. For example, $\{x \mid x \in group_A\}$ is a set of all members that belong to $group_A$, which is another entity.

With this extension, it becomes possible to support dynamic organization structure. For example, agent x has a motivation $MQ_{serveGroup}(\{y \mid x \in y\})$ to serve the groups it belongs to, this motivation is created for the agent class in the design phase, agent x is an instance of such agent class. In the system runtime, agent x joins a group A and also forms a group B with other agents, according to this motivation to serve the groups it belongs, agent x will work on goals that serve the benefit of group A or B.

Under this extended definition, we have the following definition on the relationships of MQs.

Definition 2.1: Two MQ types MQ_i and MQ_j are identical (inter-exchangeable) ($MQ_i == MQ_j$) if and only if:

1) $name(MQ_i) == name(MQ_j)$ **and**
2) $subject(MQ_i) \supset subject(MQ_j)$ **and** $subject(MQ_i) \subset subject(MQ_j)$.

Definition 2.2: MQ type MQ_i is a special case of MQ_j ($MQ_i \subset MQ_j$) if and only if:

1) $name(MQ_i) == name(MQ_j)$ **and**
2) $subject(MQ_i) \subset subject(MQ_j)$.

C. Sensor Data

Sensor data refers to the input for the agent. For robot agents, the sensor data is collected by different sensors, like camera, speedometer, etc. For software agents, sensor data refers to the messages and information the agent receives from the environment including other agents.

D. Reasoning Mechanisms

Each agent is equipped with a set of reasoning mechanisms, the reasoning mechanisms are used for the following purposes:

1) Decide what roles the agent should take or release at this moment, given the agent's motivation, current roles it is taking, the resource and time constraints.
2) Decide what goals the agent should pursue at this moment. The agent may take multiple roles and each role may have multiple goals, so the agent needs to decide which goals it need to focus on at this moment based on how the goals contributed to its motivations, how each goal could be achieved and the resource and time constraints.This issue is related to the next issue.
3) Decide how to achieve a goal given the available alternatives, resources and time constraints. Some planning and scheduling mechanisms are needed for this decision.

Given the formal definition of motivations, goals and the detailed description of alternatives to achieve a goal, it is possible to build some general, domain-independent reasoning mechanisms/toolkits, from which the user can select appropriate components and add them to the agents, the user can also customize these general mechanisms/toolkits by setting some parameters. These general mechanisms/toolkits are reusable for agents in different applications.

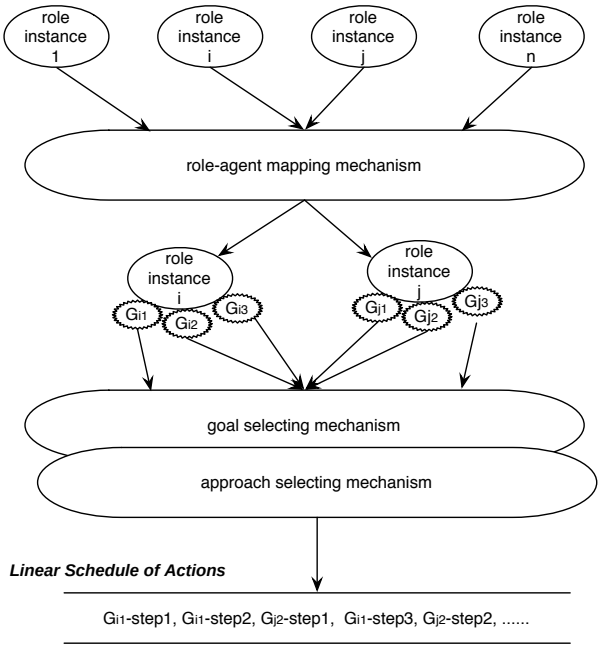


Fig. 3. Agent's Reasoning Mechanisms

Figure 3 shows the agent's reasoning mechanisms. In general, agents decide what to do using the reasoning mechanisms. The decision is to be made at different levels: selection of roles, selection of goals, and selection of the approach to fulfill the goals. The first issue is resolved by role-agent mapping mechanisms, and the later two issues are inter-related, which are solved by planning-scheduling mechanisms. More details of these two types of mechanisms are described in Section IV-A and IV-B after the detailed description of role is presented.

E. Execution Mechanisms

Execution mechanisms are used to generate the output, which changes the environment. For robot agents, their actors such as their motors, are the execution mechanisms, which are used to execute some actions to change the environment states. For software agents, the execution mechanisms are the primitive actions to change the outside environment state. Some of these execution mechanisms are domain-dependent. For example, the personal assistant agent is build with execution mechanism to perform an online purchase, which is not built in a mathematics theorem proven agent. Other execution mechanisms are application-independent but platform-dependent, such as sending a message. Some common execution mechanisms can be built as toolkits and reused for different applications.

The major difference between the reasoning mechanisms and execution mechanisms is: the reasoning mechanisms only changes the agent's inside state, and has no effect on the outside environment directly, while the execution mechanisms changes the outside environment directly.

In summary, Figure 4 shows the general architecture of an agent. Each agent has a set of attributes, and its motivation is a set of *MQ*s it accumulates and tracks, which are mapped into its overall utility through specified utility functions. An agent

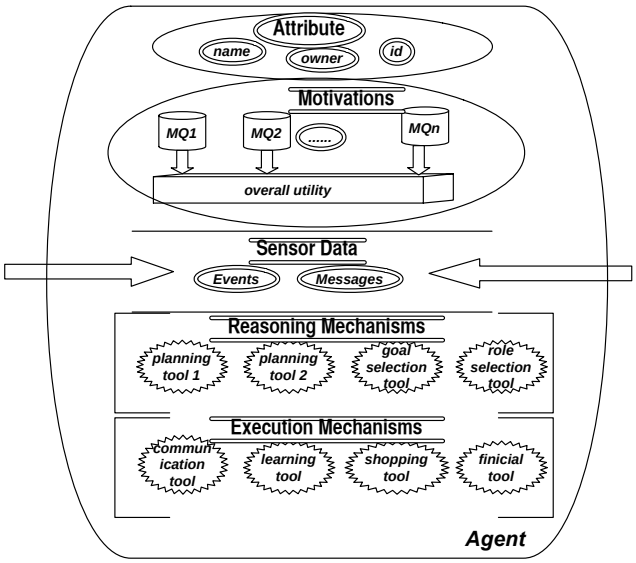


Fig. 4. An General Agent Architecture

also receives sensor data from outside environment including events and messages. An agent has a set of reasoning mechanisms including role/goal selection, and planning/scheduling mechanisms. The designer of the agent decides what reasoning tool should be built in for this agent, the designer also selects the appropriate execution tools for this agent according to the designed purpose of this agent. It is assumed there are a set of reasoning and execution mechanisms available as toolkit, which can be selected and plug into the agent seamlessly.

III. Role Definition

Same as agent, a role is defined with a set of attributes, such as role name and identification. A role is also defined with a set of goals, each goal is associated with a plan tree, which is a hierarchal description of the alternatives to accomplish a goal.

A. Goal Definition

The definition of a goal contains the name of the goal name and a MQ Production Set (*MQPS*):

$MQPS = \{(MQ_i, q_i), (MQ_j, q_j), (MQ_k, q_k)...\}$,

which represents the success accomplishment of this goal will generate q_i amount of MQ_i, q_j amount of MQ_j, q_k amount of MQ_k, etc. The *MQPS* describes how this goal contribute quantitatively to some higher-level goals (abstract goals), which are build in agents' motivations. For example, there is a *meeting coordinator* role, which has a goal defined as:

goal name*: schedule group meeting*

$$MQPS : \{MQ\_organizeActivity(x|x \in meeting\_group), \\ MQ\_serveGroup(meeting\_group)\}$$

This goal generates two type of MQs, meaning that the achievement of this goal contributes to two abstract goals: organize activity (for any member belongs to this meeting group) and serve this meeting group. It should be noticed that

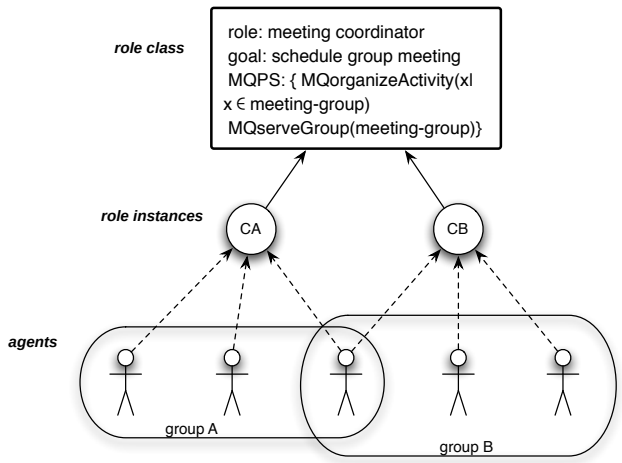


Fig. 5. Meeting Coordinator Role Example

the *meeting group* is an abstract concept when this role is defined as a role class, this concept can represent any group who would like to hold meetings. When an role instance is instantiated from this class, this abstract concept is instantiated as a concrete group too. Depending on the context when the *meeting coordinator* role instance is created, a specific group will replace this abstract *meeting group* in the goal definition. Assume that two *meeting coordinator* role instances *CA* and *CB* have been created (Figure 5), one for group *A*, and another for group *B*. Both of them have the goal of the same name but not the same *MQPS*. All agents who belong to group *A* are motivated to take the role *CA*, those agents who belong to group *B* are motivated to take the role *CB*, those agents belong to both groups are motivated to take both role instances.

B. Plan Tree Definition

For each goal associated with a role, there is a plan tree to describe the possible alternatives to achieve this goal. This plan tree is part of the domain knowledge and needed to be defined by the user. To represent this domain knowledge, we introduce RTÆMS (Role-Based Task Analyzing, environment Modeling, and Simulation) language based on the extension of the TÆMS language [8]. TÆMS is a hierarchical task representation language, which support the representation of the relationships among goals and subgoals, the quantitative description of the atomic approaches and uncertainties, and resources. We extend the TÆMS language by introducing a *role* attribute for task nodes that represent goals and subgoals. The attribute *role* specifies which roles are possible to carry this task.

For example, Figure 6 shows the plan tree for the goal *Organize Conference*. The goal *OrganizeConference* belongs to the role *ConferenceChair*, it consists of three sub-goals: *Program-Preparation*, , *BusinessPreparation* and *ConferenceExecution*. The **min** quality accumulative function (**qaf**) associated with the goal *OrganizeConference* specifies the following relationship:

$$Quality(OrganizeConference) = min(Quality(ProgramPreparation), \\ Quality(BusinesPreparation), Quality(ConferecenExecution))$$

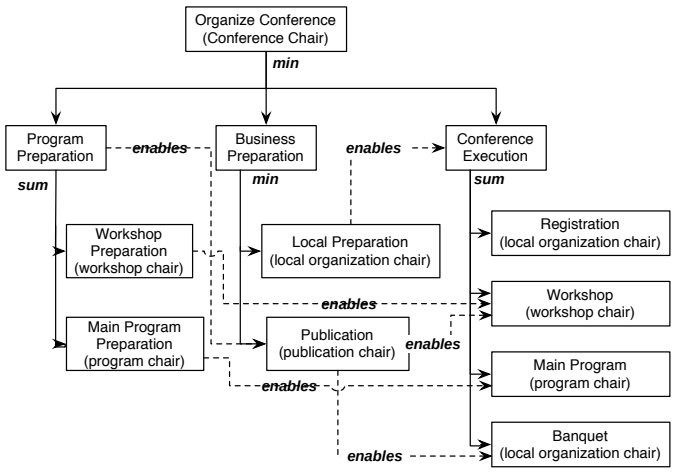


Fig. 6. Plan Tree for Goal *Organize Conference* in RTÆMS Representation

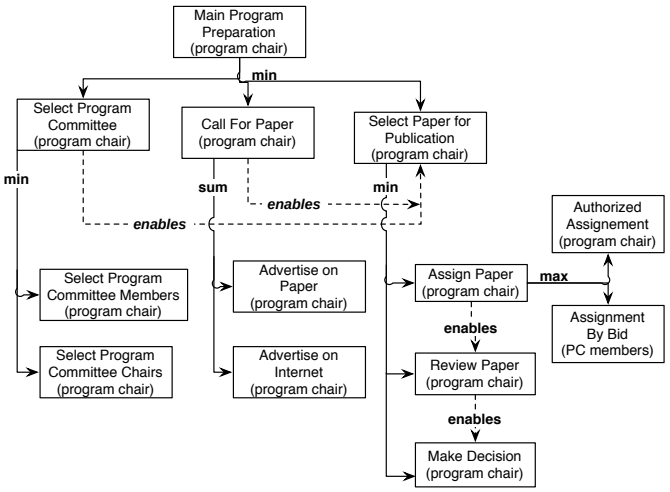


Fig. 7. Plan Tree for Goal *Main Program Preparation* in RTÆMS Representation

In other words, the success of this goal depends on the success of all of its sub-goals. Other available quality accumulation functions include **max, sum, seq_sum,** etc.

Each sub-goal can be divided into more detailed sub-goals. For instance, *ProgramPreparation* is divided into two sub-goals: *WorkshopPreparation* and *MainProgramPreparation*, which belongs to the role *WorkshopChair* and *ProgramChair* respectively.

The dash lines represent the interrelationship between goals/sub-goals. For example, *LocalPreparation* **enables** *ConferenceExecution* describes the fact that the first goal *LocalPreparation* has to be achieved successfully before it is possible to implement the second goal *ConferenceExecution*. Other types of interrelationships defined in TÆMS include: **facilitates, disables** and **hinders**.

The primitive goal (lowest-level goal) in the RTÆMS representation can be specified with more details in a plan tree associated with another role. For example, the plan tree for the goal *ProgramPreparation* is described in Figure 7, this information belongs to the role *ProgramChair*. In this example, there is an instance of the use of **max** quality function: there are two alternatives to achieve the goal *AssignPaper*, either the papers are assigned by *ProgramChair*s or the papers are assigned based on the biddings from *PCMember*s.

The RTÆMS shows all possibility to achieve a goal and the interrelationship among goals/subgoals. It provides fundamental knowledge for agents to plan and schedule its local activities, and it also supports the collaboration and cooperation among agents. More details are presented in Section IV.

IV. Operation of the Multi-Agent Systems

In this Section, we will discuss more details on how the multi-agent systems will be developed and operated based on the RADE framework we have presented in [4] and earlier in this paper.

A. Mapping From Role Instance to Agent

One important feature of the RADE framework is that the agent can dynamically choose the role instances. In the development phases, roles and agents are designed separately. When the system execution starts, one or more leading role instances are created by the human user. Those agents who are interested in taking a particular role instance send messages to the creator of this role instance (in this case, the creator is the human user). The creator then checks the qualification of the agents. If an agent is qualified for this role instance, the request will be granted. When an agent takes a role instance, it checks the goals that belong to this role instance and decides if more role instances need to be created to carry the subgoals. If this is the case, more role instances will be created and posted in the role spaces. The process described above is repeated until no more role instances are created.

Now we describe the answer to the following two questions:

1) **How does an agent select the role instances it wants to take?** An agent is interested in a role instance if some of the goals belong to the role instance match the agent's motivation. A goal G matches agent A's motivation if and only if:
$\exists MQ_x \in MQPS(G), \exists MQ_y \in Motivations(A)$, $MQ_x \subset MQ_y$ or $MQ_x == MQ_y$.
According to the above definition, there may be multiple role instances an agent is interested at the same time. The agent may send request for all of them or some of them. When more than one requests have been granted, the agent may confirm some of them or all of them depending on its resource and capability, and its preference on different MQs given its current MQ accumulations. [7] has presented a heuristic search algorithm to select the most appropriated tasks based on agent's MQ preference, MQ states and resource limitation. Similar mechanisms can be adopted here for agent to select the appropriated role instances.
2) **How to verify the qualification of an agent for a role instance?** The verification process is executed by the creator of the role instance (which could be the human user or another agent), this process is based on two criteria:
   a) Whether the agent (A) has the capability to take this role instance (R). The following condition is

checked: $Actions(R) \subset ExecutionMechanism(A)$

b) Whether this role instance is consist with other role instances the agent currently has. This condition is checked based on the incompatibility relationships defined in the role organization.

B. Planning and Scheduling

The planning and scheduling mechanisms are used to generate a linear schedule of activities for the agent to execute. The plan tree associated with each goal consists of all possible alternatives to achieve a goal, it is not a linear schedule. The agent needs to make decisions on how to achieve a goal based on this plan tree and the time/resource constraints. A general, domain-independent planner/scheduler for TÆMS task structure has been developed [9]. Similar toolkits can be developed for RTÆMS plan tree too. We propose to build multiple planning/scheduling toolkits using different technologies with varying complexities from heavy-duty contingency planner to quick and easy one-step-look-ahead planner. The agent builder can choose from them and the agent also can choose which one to use at that time if multiple planner/scheduler components are build in.

C. Collaboration and Cooperation

In an open agent society with distributed information, resources and tasks, agents need to collaborate and cooperate on their actions. Efficient collaboration and cooperation mechanisms are important to the performance of the system. Large amount of effort has been spend on the development of mechanisms for collaboration and cooperation in multi-agent systems. Our intention is to develop a set of domain-independent mechanisms for collaboration and cooperation, so they can be re-used in different applications. This need is also recognized by other researchers [10]. In ROPE project [11], cooperation process is build as separated component from the concrete agents, the ROPE engine provides execution of the cooperation process, which is described as a high-level petri-net class. However, the implementation of ROPE Engine is based on a shared memory, which is not always feasible for agents widely distributed on different machines. Additionally, the cooperation process in ROPE project is based on token and transition firing, which is not feasible enough to support more proactive cooperation and collaboration, i.e. agents are able to consider the cooperation and collaboration needs when they are planning their own activities.

The RTÆMS language supports collaborations and cooperation by specifying interrelationship among goals and subgoals, so agents know why they need collaboration and cooperation, when and with whom. A set of domain-independent general collaboration mechanisms (GPGP) based on TÆMS language has been developed [12]. we propose to develop (or reuse some of GPGP) similar mechanisms in RADE framework based on RTÆMS language. Agents collaborate and cooperate with each other using this set of mechanisms and also according to the protocols defined in the role, which specify how the interaction between roles should be proceeded.

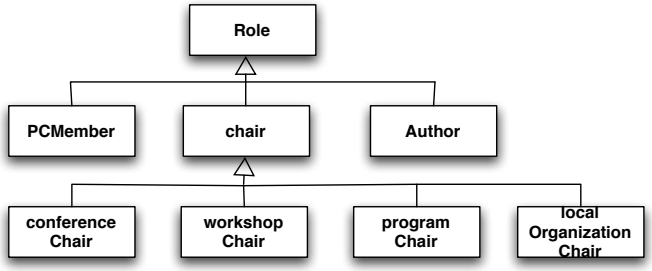


Fig. 8. Relationships among Role Classes

V. A Case Study

In this section, we use the conference organization application as an example to illustrate how our proposed approach works. In this application, we first define the following roles: *Chair, ConferenceChair, WorkshopChair, ProgramChair, LocalOrganizationChair, ProgramCommitteeMember*, and *Author*. All of these roles are subclasses of the *Role* class, and the *ConferenceChair, WorkshopChair, ProgramChair* and *LocalOrganizationChair* inherit the *Chair* role class. Figure 8 represents the inheritance relationships among these roles.

The *Chair* role is defined as a role that has permission to create new role instances according to its need, and it also equips with a protocol that specifies how to coordinate with others as a leader.

ChairRole

$permissions : \{createNewRoleInstances\}$
$protocols : \{coordinationAsLeader\}$

The *ConferenceChair* role inherits the permissions and protocols defined in the *Chair* role, and it has a goal *OrganizeConference*, which produces 10 units $MQ_{professionalService}$. There is a RTÆMS plan tree associated with this goal, which is shown in Figure 6. The *ProgramChair, WorkshopChiar* and *LocalOrganizationChair* are defined in a similar way.

ConferenceChairRole

$goals : \{organizeConference,$
$MQPS = \{(MQ_{professionalService}, 10)\}\}$
$planTrees : \{RTÆMS\ specification\}$

PCMemberRole

$goals : \{reviewPaper,$
$MQPS = \{(MQ_{professionalService}, 1)\}\}$
$planTrees : \{RTÆMS\ specification\}$

The *PCMember* has a goal to review paper, which generates 1 units $MQ_{professionalService}$. Since this role does not inherit the *Chair* role, so it does not have the permission to create new role instances, and does not perform as leader in coordination.

AuthorRole

$goals : \{publishPaper,$
$MQPS = \{(MQ_{researchAccomplishment}, 5)\}\}$
$planTrees : \{RT\text{Æ}MS\ specification\}$

The *Author* role is equipped with a goal to publish paper, the accomplishment of this goal will produce 5 units $MQ_{researchAccomplishment}$.

ProfessionalAgent

$name : String$
$motivations : \{MQ_{researchAccomplishment}, MQ_{professionalService}\}$
$rolesTaken : \mathbb{P} \downarrow Role$
$reasoningMechanisms : \{planning, scheduling, roleSelection\}$
$executionMechanisms : \{communication, coordination\}$

The *ProfessionalAgent* has motivations to make research accomplishment and contribute to professional services, which are represented by two special types of *MQ*s. Any goals that generated such *MQ*s would be attractive to the agent, hence the agent would be interested in taking any role defined above. However, the agent cannot take all of these roles given its limited capability and the constrains among those roles, so the agent's reasoning mechanisms and the consistence checking mechanisms will be used to determine which roles should be taken by the agent.

The system works as the following. First, all the roles and agent classes are defined by the user in AIPI model, the domain related knowledge is also represented as the plan trees and the protocols in ASPI model. Next, multiple professional agent instances are created, each represents a human user who is a professional. The user customizes his/her agent by specifying some personal constrains, availability and utility preferences. A role instance of *ConferenceChair* is created by a human user who is conference chair. This role instance is assigned to the professional agent who represents this chairperson. This professional agent then analyzes the plan tree of this *OrganizeConference* goal, and decides to create more role instances including the *WorkshopChair, ProgramChair*, and *LocalOrganizationChair*. Those agents who are interested in these role instances will send requests to the agent who is taking the *ConferenceChair* role, the agent decides the mapping of these role instances to the agents. After these role instances are taken by agents, each agent analyzes its plan trees and more role instances are posted in the role space, such as *PCMember* roles and *Author* roles. More agents take these roles and perform the tasks as defined in the role, the goal of the system to run a conference hence is achieved.

VI. Conclusion and Future Work

In this paper, we present a general design of agent architecture for RADE framework. We define the agent's motivation based on the extension of *MQ* framework, we also define the goal with MQ production set and develop RTÆMS language to represent the plan trees for goals. Based on these definitions, we describe the role-agent mapping mechanisms and criteria. we also discuss the ideas to develop general planning/scheduling and collaboration/cooperation toolkits.

Our future work include the implementation of an extened RADE framework including an interface for agent design, a set of plug-in toolkits for agent reasoning, execution and collaboration, and an demo of an automated generated multi-agent system and its operation on one application domain.

References

[1] Elizabeth A. Kendall. Role modeling for agent system analysis, design, and implementation. In *ASA/MA*, pages 204–218. IEEE Computer Society, 1999.

[2] Vincent Hilaire, Abder Koukam, Pablo Gruer, and Jean-Pierre Müller. Formal specification and prototyping of multi-agent systems. In *ESAW '00: Proceedings of the First International Workshop on Engineering Societies in the Agent World*, pages 114–127, London, UK, 2000. Springer-Verlag.

[3] Sen Cao, Richard A. Volz, Thomas R. Ioerger, and Yu Zhang. Role-based and agent-oriental teamwork modeling. In Hamid R. Arabnia and Youngsong Mun, editors, *IC-AI*, pages 1190–. CSREA Press, 2002.

[4] Haiping Xu and Xiaoqin Zhang. A methodology for role-based modeling of open multi-agent software systems. In Chin-Sheng Chen, Joaquim Filipe, Isabel Seruca, and José Cordeiro, editors, *ICEIS (3)*, pages 246–253, 2005.

[5] Haiping Xu, Xiaoqin Zhang, and Rinkesh J. Patel. Developing role-based open multi-agent software systems. Technical report, Computer and Information Science Department, University of Massachusetts Dartmouth, 2006.

[6] Michael Luck and Mark d'Inverno. A formal framework for agency and autonomy. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 254–260, San Francisco, CA, USA, 1995. AAAI Press.

[7] Thomas Wagner and Victor Lesser. Evolving real-time local agent control for large-scale mas. In J.J. Meyer and M. Tambe, editors, *Intelligent Agents VIII (Proceedings of ATAL-01)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2002.

[8] Keith Decker. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. In *Foundations of Distributed Artificial Intelligence, Chapter 16*, pages 429–448. G. O'Hare and N. Jennings (eds.), Wiley Inter-Science, January 1996.

[9] Thomas A. Wagner, Alan J. Garvey, and Victor R. Lesser. Criteria Directed Task Scheduling. *Journal for Approximate Reasoning (Special Scheduling Issue); a version is also available as UMass Computer Science Technical Report 1997-59*, 19:91–118, January 1998.

[10] Giacomo Cabri, Luca Ferrari, and Letizia Leonardi. Agent role-based collaboration and coordination: a survey about existing approaches. In *SMC (6)*, pages 5473–5478. IEEE, 2004.

[11] Michael Becht, T. Gurzki, Jurgen Klarmann, and Matthias Muscholl. ROPE: Role oriented programming environment for multiagent systems. In *Conference on Cooperative Information Systems*, pages 325–333, 1999.

[12] V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, and X.Q. Zhang. Evolution of t he GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143, July 2004.