An Integrated Role-Based Approach for Modeling, Designing and Implementing Multi-Agent Systems*

Xiaoqin Zhang, Haiping Xu & Bhavesh Shrestha

Computer and Information Science Department
University of Massachusetts at Dartmouth
North Dartmouth, MA 02747 -U.S.A.
{x2zhang | hxu | g_bshrestha }@umassd.edu

Abstract

To facilitate the development of multi-agent systems and improve the reusability, robustness and feasibility of these systems, we have developed a role-based agent development framework (RADE). In this paper, we present an integrated approach for modeling, designing and implementing multi-agent systems using RADE. We describe the design of agents and motivations within such framework. We introduce a practical approach for modeling agent's motivation and specifying agent's goals, where a role-agent mapping mechanism is developed based on this design. Dynamic task allocation is achieved through the creation of role instances and the mapping from role instances to agents. We also introduce the RTÆMS language based on the extension of TÆMS to model the plan tree for each goal. This representation enables the reuse of general planning/scheduling and collaboration/cooperation mechanisms developed in multi-agent system research community. We have developed an automatic agent generation interface and also implemented a simple demo system in health care domain.

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

* This material is based upon the research work supported by the College of Engineering, UMass Dartmouth.

1. Introduction

Multi-Agent System (MAS) is a suitable programming paradigm for distributed information systems and applications, where resources, data, control and services are widely distributed. However, the application of multi-agent system has been limited by the difficulty to develop such systems. Considerable amount of time and highly-experienced programmers are required to develop a multi-agent system. After such system is built, it is also difficult to test and maintain the system because of its complexity. The reusability of such system is low, it is unlikely to use an existing system for another application domain with little or minor change.

A number of approaches for defining and developing autonomous agents and multi-agent system from different directions have been studied by many researchers. Luck and d'Inverno presents a formal definition of agent including goal and motivation [11]. [21] describes a new BDI agent framework - the SRI Procedural Agent Realization Kit (SPARK) to develop agent systems that can scale to real world applications. [25] presents how to use a Java-based platform to implement BDI agents. Some researchers use UML and its extension to model agents and the interactions among agents. [13] presents an intermediate language UML-AT for translation between models in different language. [23] describes a meta-encoding schemas for compiling non-monotonic logic theories into Verilog Hardware Description Language descriptions. [22] introduces MAS-ML for modeling multi-agent systems. [6] proposes use of UML activity dia-

grams to model agent plans and actions. [10] demonstrates that a variety of adaption of business process can be handled through business protocol composition. [1] proposes a process to specify an agent-oriented information system with successive refinements using extended UML and AUML diagrams and notation. [3] proposes an ontology based on the language metamodel as a formal specification of MAS design models. The group of work most related to our work is the role-based methodology for developing of multi-agent systems. Typical examples of such efforts include the Gaia methodology [29] , its extension [15] and Multi-agent Systems Engineering (MaSE) methodology [9]. The Gaia methodology models both the macro (social) aspect and the micro (agent internals) aspect of the multi-agent system. The methodology covers the analysis phase and the design phase. Similarly, the MaSE methodology is a specialization of more traditional software engineering methodologies. During the analysis phase of the MaSE methodology, a set of roles are produced, which describes entities that perform some functions.

We have been working on a set of technologies and mechanisms to ease and formalize the development of MAS, and to increase its reliability and reusability too. We aim to cover the analysis and modeling, design and implementation phases. The first goal is to **separate concerns**. There are multiple issues in a multi-agent system, 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 (AIPI), the application specific, platform independent model (ASPI), and the application specific and platform specific model (ASPS) are developed in the three consecutive phases respectively [31]. Another approach for separating concern is to separate the domain knowledge and the intelligent problem-solving capabilities. We adapt a role-based modeling approach. In this approach, conceptual roles are defined with the domain related knowledge, such as goals, permissions, organizational relationship, and interaction protocols, etc; where agent is a concrete entity equipped with motivations, resources and problem-solving capabilities. However, our role-based approach is different from other proposed role-based approaches [17, 14, 5]. 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 second goal is to **automate the agent generation process**, while utilizing the existing tools and mechanisms as much as possible. We propose to create agents using a drag-and-drop mechanism where the user can select components to plug in the agent depending on the application requirement. Rather than a practical reasoning agent architecture such as BDI, we adopt a utility-driven agent architecture with quantitative reasoning capabilities. Our high-level design is based on roles, however, the mapping from role instances to agents in our work is different from other role assignment mechanisms [7] . Besides the logical reasoning on the matching of motivations and the conflicts among different roles, we adapt a quantitative model of motivation named MQ framework [27]. Based on this MQ framework, the agent can perform a quantitative reasoning on how important a role instance is given its preference, its utility function and its current achievement. In the definition of a role, we introduce a RTÆMS language (Role-Based Task Analyzing, environment Modeling, and Simulation) to represent the domain knowledge about how to achieve a goal. RTÆMS language is an extension of TÆMS language [8] - a hierarchical task network representation language with task inter-relationships and quantitative descriptions of different alternatives to achieve a goal. When an agent takes a role instance, it has access to this RTÆMS representation of the goal. As a result, the existing planning/scheduling [28] and coordination [18] mechanisms based on TÆMS language can easily be exploited by the agent.

The main contribution of this work include proposing an integrated approach for modeling, designing and implementing multi-agent systems, and the development of a prototype system to support such approach. This approach bridges the formalized role-based MAS models and the utility-driven agent architecture that are suitable for dealing with complex tasks and sophisticated organizational context. The uniqueness of this work includes the following. First, role instance is used not only as a design concept but also a real entity in the system runtime. By dynamically creating, taking and releasing role instances, dynamic task allocation is accomplished. Second, agents are able to perform quantitative reasoning on choosing role instances based on a quantitative modeling of motivation. Third, the RTÆMS description of complex tasks supports the easy plug-in reuse of existing domain-independent planning/scheduling and coordination mechanisms.

The role-based design approach and the agent architecture are presented in [30]. In this paper we focus on the definition and implementation of agents, the dynamic role-agent mapping mechanisms, the automatic agent generation process and a case study of applying this approach to a health care domain. This paper is organized as the follows. We first present an overview of the RADE approach in Section 2. The detailed description of agent

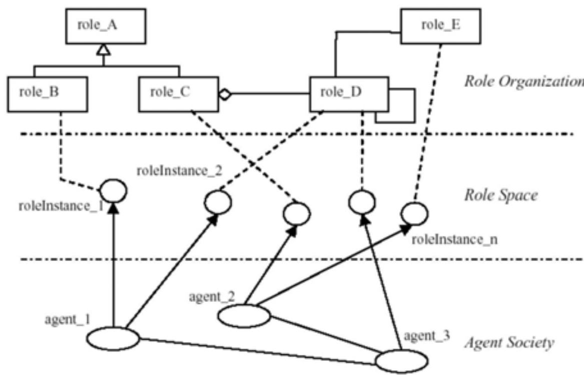

Figure 1. A generic model of role-based open multi-agent systems

is presented in Section 3. The definition and more details about role are described in Section 4. Section 5 describes operation details of multi-agent systems including the role-agent mapping mechanisms, planning, scheduling, collaboration and cooperation mechanisms among agents. The automated agent generation process is presented in Section 6. The case study of a health care application is described in Section 7. Lastly, the conclusion and discussion of the future work is presented in Section 8. Related work are discussed in various places.

2. OVERVIEW OF RADE APPROACH

The basic idea of the role-based agent development environment (RADE) is illustrated in Figure 1. The top level is the *role organization*, including the conceptual roles and their relationships such as inheritance, aggregation, association and incompatibility. The second level is the *role space*, which consists of multiple role instances, each role instance is instantiated from a conceptual role dynamically. The bottom level is the *agent society*, which consists of multiple agent entities. Agent can take and release role dynamically, the mapping from role instances to agents is called **R-A mapping**.

In order to separate software architecture from application domain and to separate application logic from the underlying technologies to improve reusability and development process, we have proposed a three-layered development model in [31]. This development model is defined in three steps. The first step is to define the Application Independent Platform Independent Model (AIPI model), which is a generic model that corresponds to the role-based development methodology for open MAS. The AIPI model includes the definition of *Role, Role Space, Role Organization, Agent* and *Agent Society*. The second step is to define the Application Specific Platform Independent Model (ASPI model) that is based on the AIPI model. The ASPI model involves knowledge from the application, including the definition of specific role classes, role organization classes, agent classes, etc. In the third step, based on the ASPI model, it defines the Application Specific Platform Specific Model (ASPS model) that further incorporates information on software platform, middleware and communication mechanisms.

Agent

$attributes : \mathbb{P}\, Attribute$
$motivations : \mathbb{P}\, Motivation$
$utilityFunction : MQState \rightarrow utility$
$sensor : Environment \nrightarrow SensorData$
$reasoningMechanisms :$
$\mathbb{P}\, SensorData \times \mathbb{P}\, Motivation \rightarrow \mathbb{P} \downarrow Role$
$\mathbb{P}\, SensorData \times \mathbb{P}\, Motivation \times \mathbb{P} \downarrow Role$
$\rightarrow \mathbb{P}\, CurrentGoal$
$\mathbb{P}\, SensorData \times \mathbb{P}\, Motivation \times \mathbb{P}\, CurrentGoal$
$\rightarrow \mathbb{P}\, CurrentSchedule$
$executionMechanisms :$
$\mathbb{P}\, SensorData \times \mathbb{P}\, CurrentPlan \rightarrow newEnvironment$
$rolesTaken : \mathbb{P} \downarrow Role$

Figure 2. Definition of agent class

In the actual software system, agent instances are automatically generated based on the definition of agent classes. Each agent instance is a software entity that performs specific functions and also coordinates and communicates with other agent instances. On the contrast, role classes are defined to incorporate domain knowledge and organizational relationship. Each role class is associated with specific goals and detailed descriptions of how to achieve such goals. The relationships among different role classes also depict the organizational relationships among the real-world entities represented by these roles. Such information is expected to be provided by domain experts rather than software engineers. In the system runtime, role instances are created dynamically either by a human user or by agents to represent that there are certain goals needed to be realized. Those role instances are mainly to carry domain knowledge and they do not actually perform any actions like agents. When a role instance is taken by an agent, the agent will use the knowledge incorporated in this role instance to achieve the goals defined in this role instances.

3. AGENT DEFINITION

Agent is an entity with attributes, motivations, sensors and a set of reasoning mechanisms. Figure 2 shows the formal definition of agent class in Object-Z [12] . Agent

attributes include agent 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 [19], *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.

3.1. Agent Motivation

We adopt a quantitative view of motivation in our practice. *Motivation* is defined as a set of *motivation quantities* (*MQ*s) [27] 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:

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

A concrete goal (task), e.g., schedule a meeting with the family doctor, contributes to 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: $U_i = U_{f_i}(MQ_i)$, 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.

3.2. Extended MQ Definition to Support Automatic Agent Generation and Dynamic Organizations

The original MQ framework is intended to support agent control in soft real-time environment, where agents are handling multiple tasks and each task has specified temporal constrains. It is assumed that all MQ types are designed by the user when the agent is created, and the MQ types 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 to automate 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 agent 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:

```
┌─ MQ ──────────────────────────
│  name : String
│  subject : ℙ entity
└───────────────────────────────
```

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

[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.

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.

Such extension makes it possible to support dynamic organization structure. For example, agent x has a motivation $MQ_{serveGroup}(\{G \mid x \in G\})$ 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, it also forms a group B with other agents. According to the motivation to serve the groups it belongs to, agent x is willing to work on goals that serve both group A and B.

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

Definition 3.1 *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. $subject(MQ_i) \supseteq subject(MQ_j)$ **and** $subject(MQ_i) \subseteq subject(MQ_j)$.

Definition 3.2 *MQ type MQ_i is a* **special case** *of MQ_j if and only if:*

1. $name(MQ_i) == name(MQ_j)$ **and**

2. $subject(MQ_i) \subseteq subject(MQ_j)$.

Dynamic organization structure is very important for multi-agent systems to function efficiently, as other researchers also recognized. [24] propose a framework for modeling agent organizations called OMNI, which allows both the representation of the global organizational requirements and the autonomy of individual agents. Our framework has this same virtue, though we adopt a quite different approach that combines role-based modeling and quantatitive reasoning.

3.3. 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.

3.4. Reasoning Mechanisms

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

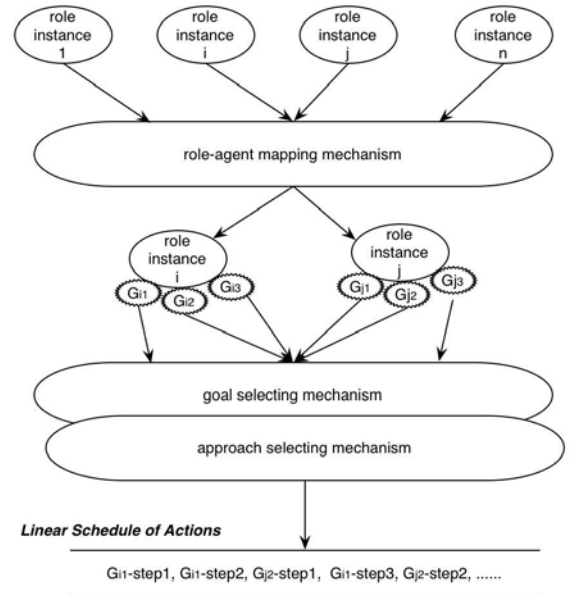

Figure 3. Agent's reasoning mechanisms

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 contribute to its motivations, how each goal could be achieved given 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. The user can select appropriate components from such toolkits and add them to the agent, the user can also customize these general mechanisms/toolkits by setting some parameters. These general mechanisms/toolkits are reusable for agents in different application domains.

Figure 3 shows an agent's reasoning mechanisms. In general, agents decide what to do using the reasoning mechanisms. The decisions are 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

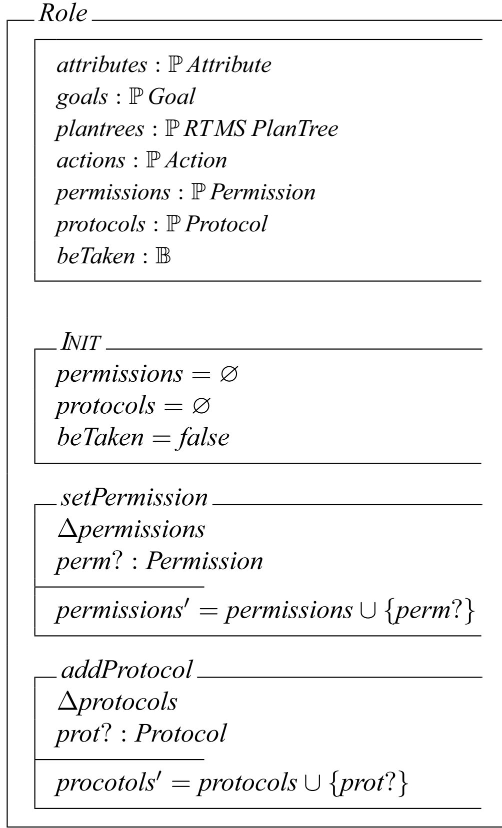

Figure 4. Definition of role

two issues are inter-related, which are solved by planning-scheduling mechanisms. More details of these two types of reasoning mechanisms are described in Section 5 after the detailed description of role is presented.

3.5. 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 environment state. Some of these execution mechanisms are domain-dependent. For example, the personal assistant agent is built 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 change the agent's internal state, and have no effect on the outside environment directly, while the execution mechanisms change the outside environment directly.

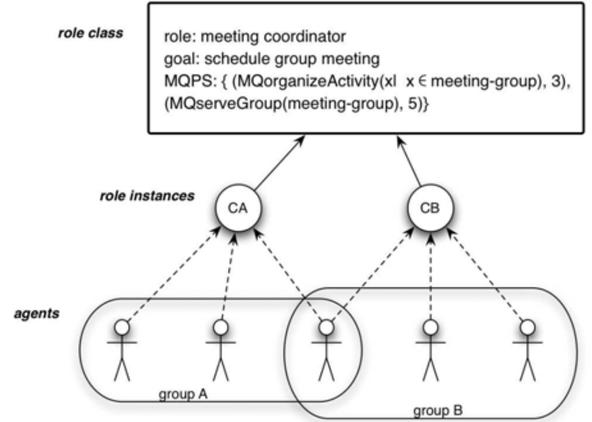

Figure 5. Meeting coordinator role example

4. Role Definition

Figure 4 shows the definition of role class. 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.

4.1. Goal Definition

The definition of a goal contains the name of the goal 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 built in agents' motivations. For example, there is a *meeting coordinator* role, which has a goal defined as:

goal name: *schedule group meeting*

$$MQPS : \{(MQ_{organizeActivity(x|x \in meeting_group)}, 3), \\ (MQ_{serveGroup(meeting_group)}, 5)\}$$

This goal generates two types of MQs, meaning that the achievement of this goal contributes to two abstract goals: organize activity (for any member that belongs to this meeting group) and serve this meeting group. The degrees of the contributions are represented by the units of the MQs, 3 and 5 respectively in this example. It should be noticed that the *meeting group* is an abstract concept

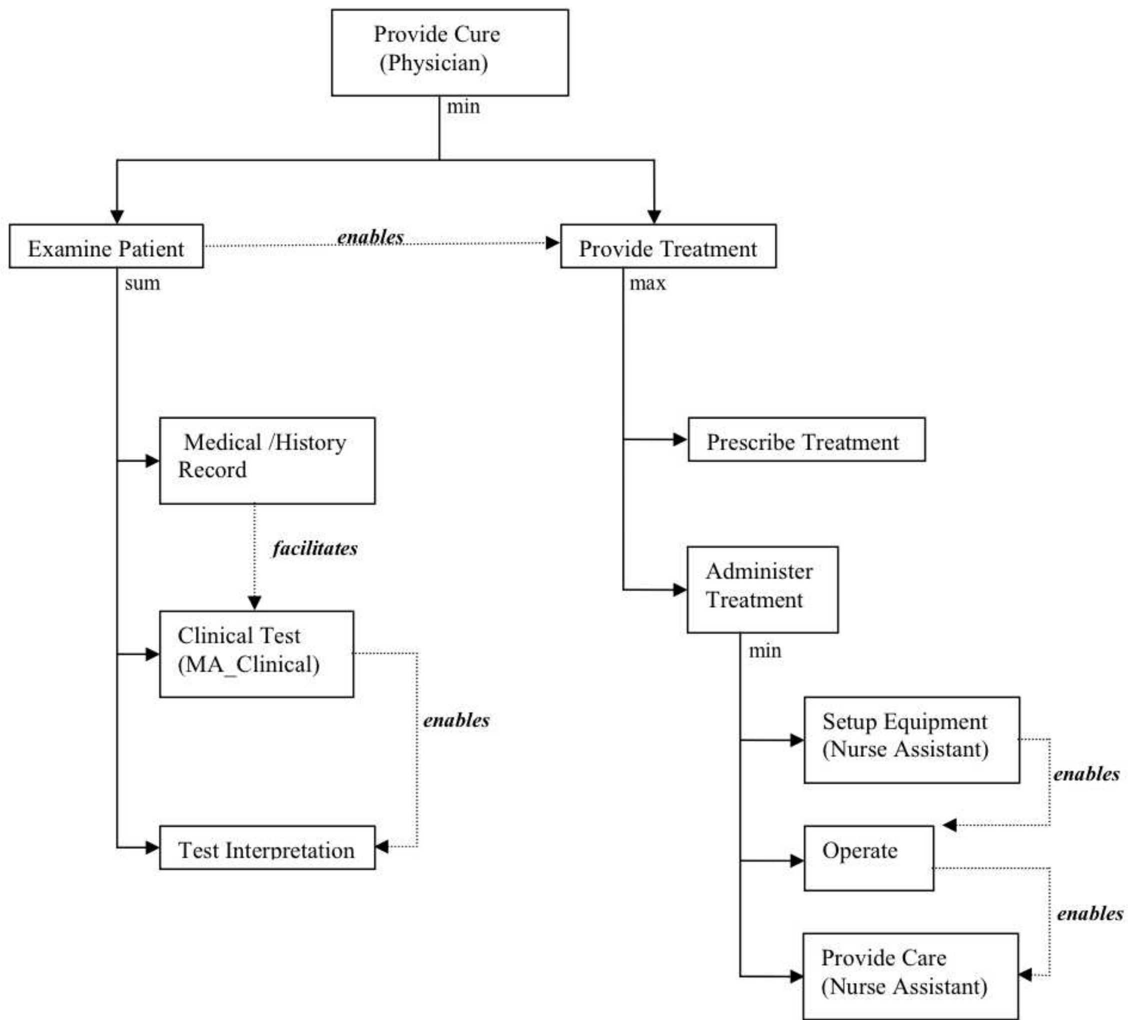

Figure 6. Plan tree for goal *Provide Cure* in RTÆMS representation

when this role is defined as a role class, this concept represents any group who needs to have meetings. When an role instance is instantiated from this class, this abstract concept is instantiated as a concrete group. 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, as shown in 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.

4.2. 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 what roles are needed to carry this goal or subgoal.

For example, Figure 6 shows the plan tree for the goal *Provide Cure*, which belongs to the role *Physician*. The goal *Provide Cure* consists of two subgoals: *Examine Patient* and *Provide Treatment*. The **min** quality accumulative function (**qaf**) associated with the goal *Provide Cure* specifies the following relationship:

$$Quality(ProvideCure) = min(Quality(ExaminePatient), Quality(ProvideTreatment))$$

In other words, the **min** quality function associated with a goal describes that the success of this goal depends on the success of all of its sub-goals. On the other hand, the use of **max** quality function represents that there are several alternatives to achieve the goal. For example, to *Provide Treatment* for the patient, the *Physician* can choose either *Prescribe Treatment* or *Administer Treatment*. Other available quality accumulation functions include **sum** and **seq_sum,** etc.

Each subgoal can further be decomposed into smaller goals, i.e. *Examine Patient* includes three subgoals: *(Read) Medical History Record*, *Clinical Test* and *Test Interpretation*. For some **non-local** goals - the tasks need to be performed by other roles, the specification of the other role is included in the plan tree description. For example, *Clinical Test* should be performed by a *Clinical Medical Assistant* (*MA_Clinical*), and *Setup Equipment* and *Provide Care* are goals belonging to the *Nurse Assistant* role.

The dash lines represent the interrelationship between goals/sub-goals. For example, *Clinical Test* **enables** *Test Interpretation* describes the fact that the first goal *Clinical Test* needs to be achieved successfully before it is possible to implement the second goal *Test Interpretation*. In addition, *(Read) Medical History Record* **facilitates** the *Clinical Test* process because it may provide some useful information about the patient. Other types of interrelationships defined in TÆMS include **disables** and **hinders**.

The primitive goal (lowest-level goal) in the RTÆMS representation can be specified with more details in another plan tree that is associated with another role. For example, the plan tree for the subgoal *Provide Care* is described in Figure 10, this information belongs to the role *Nurse Assistant*.

The RTÆMS shows all possibilities 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 5.

5. Operation of the Multi-Agent Systems

In this Section, we will discuss more details on how a multi-agent system will be developed and operated based on the RADE framework that we have presented in [30] and earlier in this paper.

5.1. Dynamic Mapping Process Between Role Instances and Agents

In RADE framework, agents can dynamically choose the role instances, and role instances can be created dynamically too. In the development phases, roles and agents are designed separately. In the implementing phases, agents are created by users. In addition, there is a *role space* component built in the system with the following functionalities:

1. Keep record of all role instances that have been created and their current status: whether this role instance has been taken and the creator of this role instance.
2. Reply messages from agents for querying the current available role instances.
3. Create new role instances according to the requests from agents or users.
4. Delete obsoleted role instances according to the requests from agents or users.
5. Monitoring the role-agent mapping processes by verifying the qualification of agents and checking the constraints on role interrelationships, to ensure the new role instance is compatible with other role instances that have already been taken by the same agent.

When the system execution starts, one or more role instances are created by a human user. Those agents who are interested in taking a particular role instance send messages to the role space. The role space then checks the qualification of the agents. The verification process is based on two criteria:

1. Whether the agent (A) has the capability to take this role instance (R). The following conditions are checked:
   $Actions(R) \subseteq ExecutionMechanism(A)$ or $Certification(R) \subseteq Qualification(A)$, where **Certification** and **Qualification** are attributes that belong to Role and Agent classes respectively.

2. Whether this role instance is consistent with other role instances that the agent currently takes. This condition is checked based on the incompatibility relationships defined in the role organization.

After this process, a list of qualified agents is sent to the creator of this role instance (in this case, the creator is the human user, it can be an agent too). The creator then selects one agent from this list to take the role instance. This selection is totally based on the creator's preference, the user can define different criteria for the selection, such as based on the profile of the candidate agent, or the experience of previous interaction with the candidate agent.

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 or to achieve some necessary preconditions. 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.

An agent decides whether it is interested in a role instance by checking if there is a goal that belongs to the role instance matches 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 is a **special case** of MQ_y.
According to the above definition, there may be multiple role instances an agent is interested at the same time. How much the agent is interested in a particular role instance depends on the following:

- The type and number of units of MQ associated with the goal that belongs to this role instance.
- The agent's preference on different *MQ*s given its current MQ accumulations.
- The agent's resource and capability.

An heuristic search algorithm has been presented in [27], which is used to select the most appropriated tasks based on agent's MQ preference, MQ states and resource limitations. Similar mechanisms can be adopted here for agent to select the appropriated role instances.

Since each goal defined in a role instance essentially represents a task to be accomplished, so the role-agent mapping process is a task allocation process. In this process, the agent decides which task it would like to take depending on the user-defined preference functions, its previous experience on accomplishment of such tasks and its resource limitation. On the other hand, which agent is chosen to perform this task also depends on the qualification requirement, the organizational rules (represented as the incompatibility relationship) and other dynamic issues such as the agent's previous performance.

Kamboj and Decker has proposed an organizational self-design approach in semi-dynamic environment [16]. It uses TÆMS language as the underlying representation for problems. Agents can be dynamically created or merged together depending on the needs of the system at runtime. It also uses role-assignment to assign a task to an agent. However, in that work, a role is defined as a TÆMS subtree rooted at a particular node, which is different from our work, where a role is a position in an organization associated with organizational rules and interaction rules. Additionally, our work proposes an integrated approach for designing and implementing MAS. In our approach, a lot of domain knowledge can be represented in the definition of roles. We also adapt a motivational quantitative measure for agents to evaluate what tasks are interesting. These make our work quite different from theirs.

5.2. 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 [28]. Such toolkits can be modified and used for RTÆMS plan trees. 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.

5.3. 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 collaboration and cooperation mechanisms in multi-agent systems. Our intention is to develop a set of domain-independent mechanisms for collaboration and cooperation, so that they can be reused in different applications. This need is also recognized by other researchers [4]. In ROPE project [2], cooperation process is build as separated component from the concrete agents, the ROPE engine provides the 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 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 that 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 [18]. we propose to develop (or reuse some of GPGP) similar mechanisms in RADE framework based on RTÆMS language. Agents collaborate and cooperate

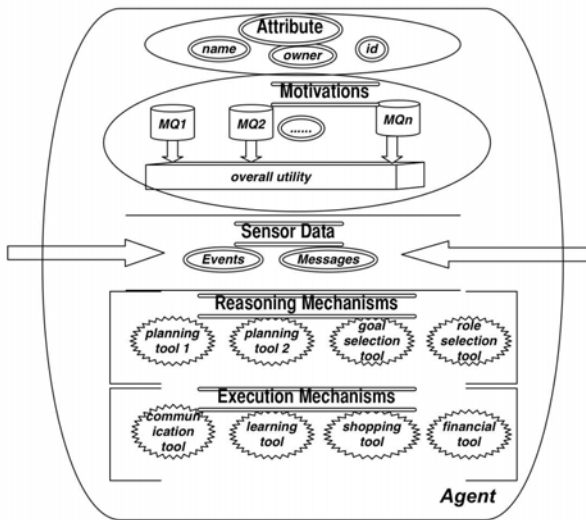

Figure 7. An general agent architecture

with each other using this set of mechanisms and also according to the protocols defined in the role, which specify how the interaction between different roles should be proceeded.

6. Automatic Agent Generation Process

The automatic agent generation process is based on a component-based agent architecture. The user can select which components to be included in this agent, and the user can also specify a set of attributes of the agent.

Figure 7 shows a general agent architecture. Each agent has a set of attributes. Its motivation is a set of *MQ*s it accumulates and tracks, which are mapped into its overall utility through specific utility functions. An agent 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 tools 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 plugged into the agent seamlessly.

Based on this general agent architecture, we developed a tool to support the automatic agent generation process. This tool is created by extending the current JAF framework [26] developed by MAS lab at UMass Amherst. This tool includes a graphic user interface, which can be used to create new agents, modify existing agents, run agents and delete agents. A screen shot of the graphicial user interface is shown in Figure 8.

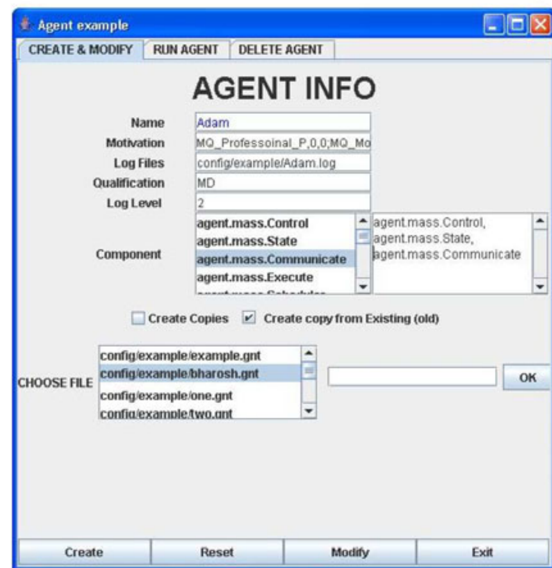

Figure 8. Automatic agent generation interface

Agent class is defined by a set of attributes, motivations, utility function, and a set of reasoning mechanisms and execution mechanisms. Individual users can create their own agent classes through this interface. The user can define a variety of attributes including name, qualification, and other parameters for recording information during the agent execution process such as log file name and log level. **Qualification** is an attribute that describes a particular capability this agent class owns, which is used in the role-agent mapping process to decide whether an agent is qualified for a particular role.

The user also defines the agent's motivation by specifying a set of *motivational quantities* (*MQ*s) [27] that the agent tracks and accumulates. The user also defines the agent's reasoning and execution mechanisms by selecting a number of ready-to-plug-in components such as: planning, scheduling, communication, etc. Currently all available components are created in JAF project, new components can be created and added to this selection list at any time in the future.

After an agent class is defined, one or multiple agent instances (the executable programs) can be created from this class definition. Each agent instance is an independent program and the agent is named after its class with a unique number ID. For example, when the user creates an agent class X and three agent instances of this class, the three agents are named as X_1, X_2 and X_3 respectively.

The user can run agents from this interface by clicking the "RUN AGENT" menu box on the top, and selecting a number of agents to run from a list of agents that have already been created. Multiple agents can be created and run on difference machines. The user can also choose

to delete existing agents by clicking on the "DELETE AGENT" menu box.

7. Case Study: Health Care Application Domain

We have implemented a prototype system including a role-definition component and an agent definition and creation component. Using this system, we implement a simple health care application as an example to demonstrate this integrated role-based approach for modeling, designing and implementing multi-agent systems, including the definition of role and agent classes, automatic agent generation process and the dynamic mapping process between agents and role instances. The purpose of this health care application system is to assist health care providers and patients to schedule and coordinate their activities so as to provide feasible and efficient health care services for patients.

7.1. Define Roles

One advantage of this role-based multi-agent system approach is the support of the **separation of concerns** principle. We believe that a complicated information system should be developed collectively by both the domain experts and the software experts. For example, in this health care application domain, there are a lot of domain knowledge that is not familiar to the software engineers. Health care domain experts are the best candidates to engineer such knowledge in the system. Hence we developed a role-definition tool with graphical user interface for the domain experts to represent those domain knowledge through role definition.

In this demo example, we pretend ourselves to be domain experts by reading some books [20] and articles in medical application domain. We created a simplified system just to verify the feasibility of this approach. In this process, we recognized the difficulty and inefficiency for a software engineer to grasp the vast amount of domain knowledge in a short period of time, which enhance our belief of the **separation of concerns** principle.

In this simplified system, we define the following role classes:

1. *Patient*: who seeks for health care.
2. *Physician*: who determines whether diagnostics are to be undertaken, provides prescriptions, performs medical and surgical interventions, has the ability to direct patient care and advance a patient to the next step of care.
3. *Medical Assistant*: a health care professional who performs a variety of clinical, clerical and administrative duties within a health care setting. There are two roles defined as subclasses of this role class:
    - *Administrative Medical Assistant* (*MA_Admin*): Medical assistant who performs the administrative job.
    - *Clinical Medical Assistant* (*MA_Clinical*): Medical assistant who performs the clinical job.
4. *Nurse*: there are two roles defined as subclasses of this role class:
    - *Nurse Assistant* a nurse who assesses the patient's medical problem, provides care and helps setup laboratory specimen and medical instruments.
    - *Nurse Practitioner*: a registered nurse who has completed an advanced training program in primary health care delivery, and may provide primary care for non-emergency patients, usually in an outpatient setting.

Figure 9 shows the RADE interface for user to create role classes and define the interrelationships among role classes. In this example, the interrelationships include *inheritance, association* and *incompatibility*. An inheritance relationship describes the generalization/specification relationship between two role class. For example, both *MA_Admin* and *MA_Clinical* inherit the *Medical Assistant* role class since they are specified medical assistants. Association is a very common relationship between role classes, it indicates an instance of one role class may perform an action on an instance of another role class. Association relationships exist between *Physician* and *Nurse*, *Physician* and *Patient*, etc. Incompatibility relationship describes the constraints that the role instances of the two role classes cannot be taken by the same agent for the same interaction scenario. For example, an agent cannot take a *Physician* role instance for treating a *Patient* role instance if the agent is taking this *Patient* role instance right now, however the agent can take another *Physician* role instance for treating another *Patient* role instance that is not taken by this agent. The definition of such relationships depends on the domain knowledge, so we feel the domain experts are the best candidates to use this interface to define the role classes and their interrelationships.

Each role is defined with a goal, a plan tree, a motivational quantity production set (MQPS), a certificate and other attributes. A goal is a task that this role needs to accomplish, and the plan tree specifies the domain knowledge of how to accomplish this goal in terms of decomposing it as sub-goals.

For example, *Physician* role is defined with a goal to *Provide Cure*. The **plan tree** shown in Figure 6 provides

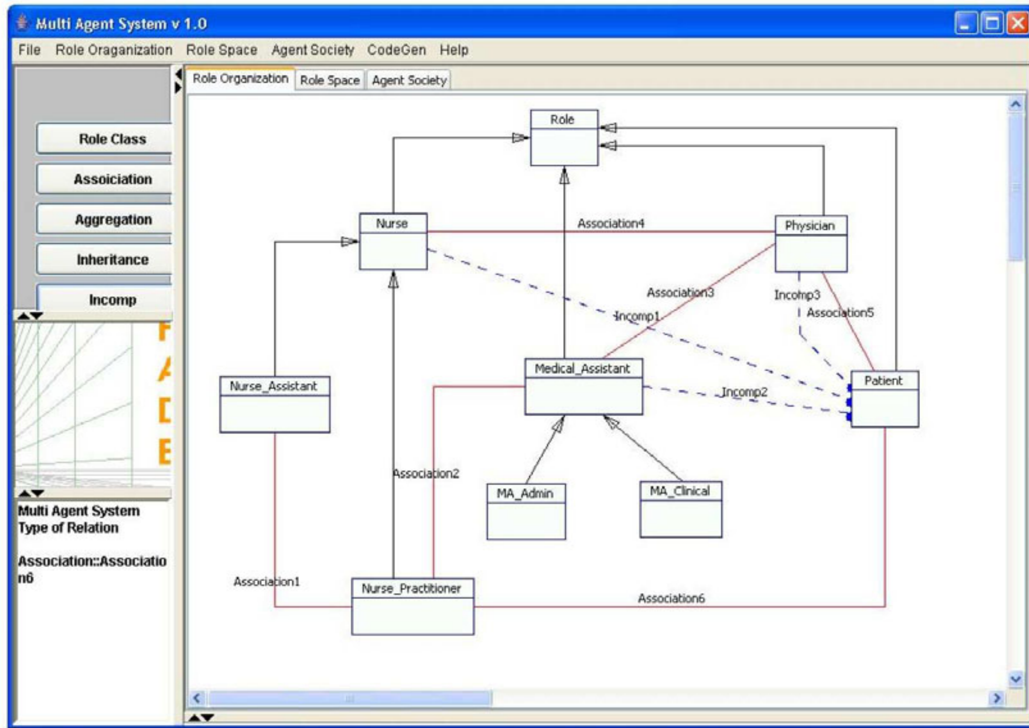

Figure 9. RADE interface for creating roles

ROLE: Physician
GOAL: Provide Cure
MQPS: $(MQ_{professional_P}, p1), (MQ_{moral_P}, p2), (MQ_{experience_P}, p3)$
CERTIFICATE: MD (Doctor of Medicine)

domain knowledge of how to accomplish this goal. Detail explanation of the plan tree is in Section 4.2. Figure 10 shows the plan trees for those goals *Get Cure*, *Assist Patient*, *Provide Cure*, and *Provide Care*, which belongs the role *Patient*, *Administrative Medical Assistant*, *Physician*, and *Nurse Assistant* respectively.

The **MQPS** specifies the type and the number of units of motivational quantities that can be collected by the agent after it accomplishes the goal defined in the role. For the agent who is taking the *Physician* role, it collects $p1$ units of $MQ_{professional_P}$, $p2$ units of MQ_{moral_P} and $p3$ units of $MQ_{experience_P}$. The MQPS specification in the role definition and the agent's motivation are used by the agent to determine whether it is interested in a role instance, and how interested it is.

The **Certificate** defined in the role describes the qualification requirement for this role. This role can only be taken by an agent who has this specified certificate. For example, *Physician* role is defined with a certificate of MD (Medical Doctor).

7.2. Define Agents

Agents are the real programming entities running in the system. In this example, each agent represents a personal assistant for a human user in the real world. The agent is responsible for scheduling the user's daily tasks according to the user's preference and constraints. The agent is also responsible for coordinating with other agents when coordination is needed between its own user and other users.

As Figure 8 shows, a user creates an assistant agent named Adam. The user specifies his preference on choosing tasks by defining the motivation[2] of this agent as:

Motivation : $\{MQ_{professional_P}, 0, 0; MQ_{moral_P}, 1, 1; MQ_{experience_P}, 2, 2\}$

This specifies three long-term goals this user has: professional achievement, moral achievement and experience achievement, as a physician, which are represented by three types of MQs shown in Table 1 . The function index specifies a utility function that maps a certain number of units of MQ of this type into the agent's local utility. Since the function can be a non-linear function and is also context sensitive, the initial amount of this type MQ is also important. The user also provides this agent with his qualification MD so that this agent can be qualified for a

[2]In this simple demo system we implemented, there is only one agent instance created from each agent class, also there is no dynamic organization in this demo either. So in the following description, we ignore $MQ_{subject}$ and use only the MQ_{name} to represent a specified MQ type.

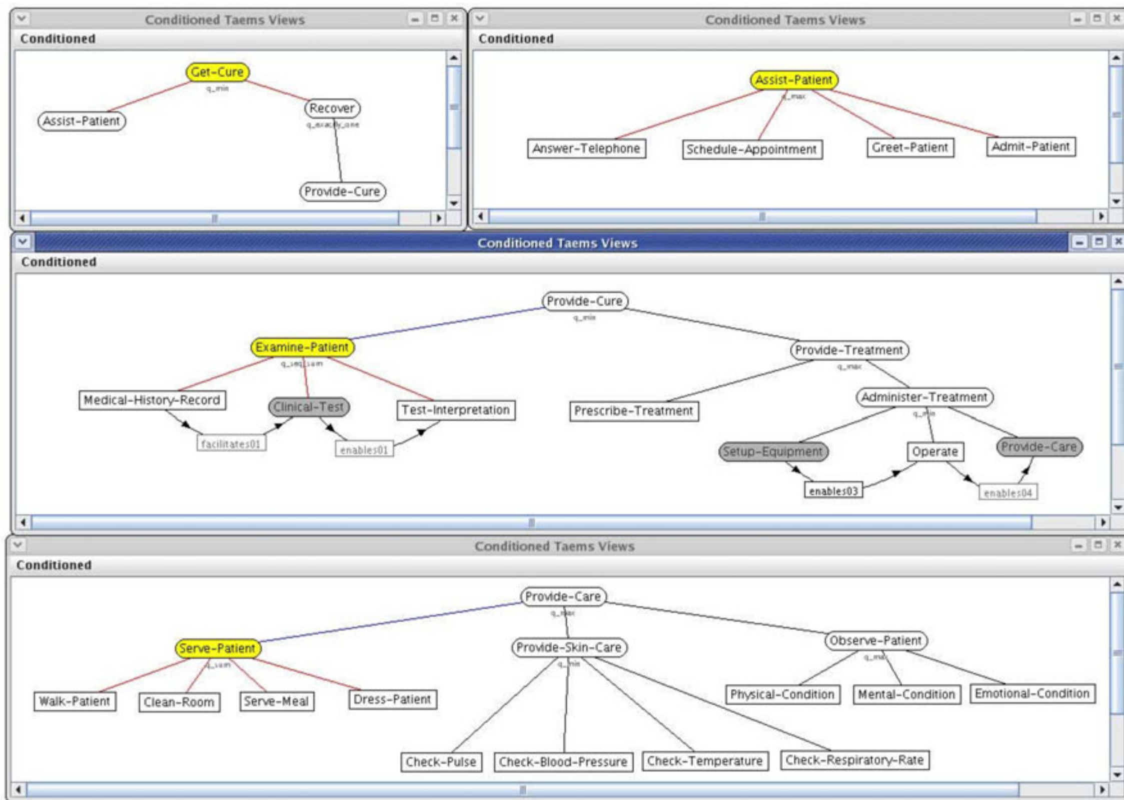

Figure 10. Plan tree definitions for multiple goals

Table 1. Agent's motivation - represent user's objectives

MQ Type	Function Index	Initial Amount
$MQ_{professional_P}$	0	0
MQ_{moral_P}	1	1
$MQ_{experience_P}$	2	2

Physician role.

7.3. Runtime Scenario

Next we present a runtime scenario to describe how the system works including how the dynamic task allocation is accomplished through the role-agent mapping mechanism.

This system is modeling a hospital organization. A special role space agent is created. This agent is not taking any active role in the system, rather, it is mainly responsible for maintaining and managing the role instances in the system, as we described in Section 5.1. Due to the limited implementation effort and time available, we make the following simplifications in this implementation, compared to the process described in Section 5.1:

- The role space checks the plan tree of a role instance when this role instance is taken by an agent and recognizes the needs to create new role instances, rather than the agent sends a request to the role space to create new role instances. This simplification is valid when the goals and plan trees are simple and there is no need for the agent to choose from different plans.
- The role space selects the appropriate agent for the role instance after verifying the qualification and consistency of the candidates, rather than sends a short list to the creator of this role instance and let the agent who creates this role instance to make selection according to the criteria defined by its user. This simplification is valid when there is only a few candidates.

When the system is initialized, the system administrator creates several *Patient* role instances to express the expected service requirements from patients. The number of *Patient* role instances depends on the capability of this hospital. These patient role instances are posted on the role space and are not active until they are taken by agents. When a (real) patient Bryan enters in this hospital for service, a personal assistant agent named Bryan is created for this patient, and this agent takes one *Patient* role instance.

When agent Bryan takes the *Patient* role instance, it has one goal to achieve: *Get Cure*. The plan tree of this goal describes that two subgoals *Assist Patient* and *Pro-*

vide Cure must be achieved so that the goal *Get Cure* can succeed. The goal *Assist-Patient* belongs to a *MA_Admin* (*Administrative Medical Assistant*) role and the goal *Provide Cure* belongs to a *Physician* role. Based on this information, a *Physician* role instance and a *MA_Admin* role instance are created by the role space.

Three other agents, Adam, Cathy and David that represent three medical professionals have already been created and are active in the system. They have been idle and sent requests to the role space for available role instances. When the *MA_Admin* and *Physician* role instances are created in the role space, all three agents who are interested in taking any additional role instances receive a message for this update.

After receiving this message, the agent looks at the goal associated with this role instance, especially the MQPS and to see if it matches its own motivation. If the MQPS contains the same type of MQ the agent has in its motivation, the agent is interested in taking this role instance. For example, this *Physician* role instance has MQPS as: $(MQ_{professional_P}, p1)$, $(MQ_{moral_P}, p2)$, $(MQ_{experience_P}, p3)$, all these three types MQs belong to agent Adam's motivation. So Adam is interested in this role instance. How interested Adam is for this role instances depends on the actual values of p1, p2 and p3, the exact structures of the mapping functions with index 0, 1, and 2, and the current accumulation of these MQs for agent Adam.

If there are multiple available role instances interested to agent Adam, it will compare the degree of interest it has towards these role instances and select the most interested ones, and send requests to the role space. It is also possible that the role space would receive requests from multiple agents for the same role instance. The role space verifies the qualification of each agent by matching the agent's qualification to the certificate requirement defined in the role class that this role instance belongs to. For example, agent Adam is qualified for this role instance because it has a MD qualification that matches the certificate requirement of the *Physician* role class. The role space also checks if this role instance is compatible with other role instances the agent is taking right now. For instance, suppose agent Bryan has a MD qualification and is also interested in this *Physician* role instance; however, according to the incompatibility between the *Physician* role and the *Patient* role, agent Bryan cannot take this role instance because it takes the *Patient* role instance related to this *Physician* role instance.

After verifying the qualification and checking the consistency, the role space then selects an appropriate agent (Agent Cathy) for the *MA_Admin* role instance, whose goal is to Assist-Patient. The plan tree for the goal *Assist Patient* consists of four subgoals: *Greet Patient*, *Schedule Appointment*, *Admit Patient*, and *Answer Telephone*. All of these subgoals can be performed by the same agent who takes the *MA_Admin* role instance, so no new role instance needs to be created. After assigning the *MA_Admin* role instance to an agent, the role space then assigns the *Physician* role instance to another appropriate agent (Agent Adam) based on its qualification. The goal of the *Physician* role is to *Provide Cure*, the role space reads the plan tree associated with the goal and finds that to accomplish this goal, three subgoals *Clinical Test*, *Setup Equipment* and *Provide Care* must be accomplished by other roles. In response to this need, new role instances *Nurse Assistant* and *MA_Clinical* are created. The role space then selects appropriate agents to take these roles. This process will continue until no more new role instance is needed and all role instances have been taken.

After a goal defined in a role instance is accomplished, the agent will collect the MQs as defined in the MQPS of this role instance. The agent will release this role instance, and this role space will delete this role instance. In the system runtime, new role instance is created according to the need to accomplish a certain goal. Agent is mapped to the role instance according to the matching of the motivation, the qualification and the compatibility. Since each role instance is associated with a goal, the mapping process is also a task allocation process. In this process, the agent is reasoning on its local utility achievement, described as its motivation and MQ mapping functions. The domain-related constrains such as qualification and compatibility are defined in the role and monitored by the role space. This implementation realizes the **separation of concerns** principle.

8. Conclusion and Future Work

In this paper we presented a prototype of automated agent generation system in connection with a previously developed role-based agent modeling and designing system. This integrated framework supports the role-based designing of multi-agent system and the implementation of utility-driven agents utilizing a variety of existing agent reasoning and coordination mechanisms. We also presented a case study of the development of a multi-agent system for health care domain. We described how the roles are defined, how agents are created, and how the role instances are mapped to agents. We also described a runtime scenario that shows the dynamic task allocation is accomplished through the creating, taking and releasing of role instances.

The future work includes further development of the system from the current prototype. Especially we are interested in implementing the quantitative description in RTÆMS and incorporating the scheduling/planning and coordination mechanisms in agents. We are also

interested in providing support for users to define interaction protocols in role classes, and then integrating those domain-dependent protocols with the domain-independent communication mechanisms in agents.

Acknowledgments. We thank Mr. Michael McGuire for providing health care domain knowledge. We also thank Prof. Lesser, Dr. Horling, Dr. Wagner and Prof. Decker for a variety of software and mechanisms developed in UMass Multi-Agent Systems lab, including JAF, TÆMS , MQ and GPGP.

REFERENCES

[1] Ricardo Melo Bastos and Marcelo Blois Ribeiro. MASUP: An Agent-Oriented Modeling Process for Information Systems. In Ricardo Choren, Alessandro Garcia, Carlos Lucena, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications Series.* 2005.

[2] 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.

[3] Anarosa A. F. Brandao, Viviane Torres da Silva, and Carlos J. P. de Lucena. A knowledge-based approach to the specification and verification of MAS design. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1373–1373, New York, NY, USA, 2005. ACM Press.

[4] 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.

[5] 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.

[6] Viviane Torres da Silva, Ricardo Choren Noya, and Carlos J. P. de Lucena. Using the UML 2.0 activity diagram to model agent plans and actions. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 594–600, New York, NY, USA, 2005. ACM Press.

[7] Mehdi Dastani, Virginia Dignum, and Frank Dignum. Role-assignment in open agent societies. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 489–496, New York, NY, USA, 2003. ACM Press.

[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] Scott A. DeLoach, Mark F. Wood, and Clint H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3), 2001.

[10] Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. An overview of business process adaptations via protocols. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1326–1328, New York, NY, USA, 2006. ACM Press.

[11] Mark D'Inverno and Michael Luck. *Understanding Agent Systems*. SpringerVerlag, 2004.

[12] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards & Interfaces*, 17(5–6):511–533, 1995.

[13] RubŐn Fuentes, Jorge J. GŮmez-Sanz, and Juan PavŮn. Integrating agent-oriented methodologies with UML-AT. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1303–1310, New York, NY, USA, 2006. ACM Press.

[14] Vincent Hilaire, Abder Koukam, Pablo Gruer, and Jean-Pierre Muller. 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.

[15] Thomas Juan, Adrian R. Pearce, and Leon Sterling. ROADMAP: extending the Gaia methodology for complex open systems. In *AAMAS*, pages 3–10. ACM, 2002.

[16] Sachin Kamboj and Keith S. Decker. Organizational self-design in semi-dynamic environments. In *AAMAS '06: Proceedings of the fifth international*

joint conference on Autonomous agents and multiagent systems, pages 335–337, New York, NY, USA, 2006. ACM Press.

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

[18] 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.

[19] 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.

[20] Michael R. McGuire. *Steps Toward a Universal Patient Medical Record - A Project Plan to Develop One*. Universal Publishers, 2004.

[21] David Morley and Karen Myers. The SPARK Agent Framework. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 714–721, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Viviane Torres Da Silva and Carlos J. P. De Lucena. From a conceptual framework for agents and objects to a multi-agent system modeling language. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):145–189, 2004.

[23] Insu Song and Guido Governatori. Designing agent chips. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1311–1313, New York, NY, USA, 2006. ACM Press.

[24] Javier Vázquez-Salceda, Virginia Dignum, and Frank Dignum. Organizing multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 11(3):307–360, 2005.

[25] R. Vieira, ç. F. Moreira, R. H. Bordini, and J. H§bner. BDI agent programming in agentspeak using Jason. In *Proceedings of the Sixth International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VI)*, pages 143–164, 2005.

[26] Regis Vincent, Bryan Horling, and Victor Lesser. An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator. *Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems.*, 1887, January 2001.

[27] 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.

[28] 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.

[29] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[30] 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.

[31] Haiping Xu, Xiaoqin Zhang, and Rinkesh J. Patel. Developing role-based open multi-agent software systems. *International Journal of Computational Intelligence Theory and Practice (IJCITP)*, 2(1): 39-56, June 2007.