RiBAC: Role interaction Based Access Control Model for Community Computing

Youna Jung[1], Amirreza Masoumzadeh[1], James B.D Joshi[1], Minkoo Kim[2]

[1]School of Information Sciences, University of Pittsburgh
{yjung, amirreza, jjoshi}@sis.pitt.edu
[2]College of Information Technology, Ajou University, Korea
minkoo@ajou.ac.kr

Abstract. Community computing is an agent-based development paradigm for ubiquitous computing systems. In a community computing system, ubiquitous services are provided by cooperation among agents. While agents cooperate, they interact with each other continuously to access data of other agents and/or to execute other agent's actions. However, in cases of security-critical ubiquitous services such as medical or military services, an access control mechanism is necessary to prevent unauthorized access to critical data or action. In this paper, we propose a family of *Role interaction Based Access Control* (RiBAC) models for Community Computing, by extending the existing RBAC model to consider role interactions. As a basic model, we propose the core RiBAC model. For the convenience of management and to provide more fine-grained access control, we propose Hierarchical RiBAC (H-RiBAC), Constrained RiBAC (C-RiBAC), and Constrained Hierarchical RiBAC (CH-RiBAC) models. Finally, we extend the existing community computing framework to accommodate the specification and enforcement of RiBAC policies.

Keywords: Cooperation, Community computing, Role interaction, Role-based Access Control, Multiagent system

1 Introduction and Motivation

The capacity and intelligence of newly developed computing elements are growing day by day. For highly complex problems requiring diverse capabilities, an approach based on cooperation among elements can be an efficient solution [1]. Many researchers have tried to fulfill application requirements using cooperation among individual computing elements. For example, ubiquitous computing systems are often developed using cooperation among computing elements because such systems require, in general, many different capabilities of various computing elements. Because such a cooperation based approach involves continuous and rich interactions, multiagent technology is frequently used to design and develop cooperation based ubiquitous computing systems. In addition, agents' characteristics such as intelligence and autonomy are suitable for developing intelligent ubiquitous computing systems that can adapt to dynamically changing situations.

Jung *et al.* [2] propose Community Computing (CC) as an agent-based development paradigm for ubiquitous computing systems. The objective of CC framework is to provide ubiquitous services through dynamic cooperation among agents. The CC approach focuses more on cooperation compared to the other multiagent methodologies. As part of the CC approach, Jung *et al.* have proposed a cooperation model and two different CC models. However, security of such a CC based multiagent system has not been addressed in the literature.

Ubiquitous services are currently being expanded to various applications such as u-healthcare, u-government, u-city, etc. Security and performance issues are some key challenges to the deployment of such emerging ubiquitous systems, and hence a CC system for ubiquitous applications should incorporate efficient security mechanisms. In order to guarantee a secure CC system, first of all, the system should authenticate agents. During cooperation, agents interact with other agents to get information or request execution of other agents' actions, which may be critical. To ensure security of such critical actions or data, we need a proper access control mechanism to ensure that agents are engaged in only authorized activities.

In this paper, we propose a family of role interaction based access control (RiBAC) models that extend the standard RBAC models by incorporating authorized role-based interactions among agents. We define two types of interaction permissions to capture authorized interactions among agents. Moreover, we extend the CC specification framework to include the RiBAC policy specification and enforcement capabilities.

The remainder of this paper is organized as follows. In Section 2, we present the background on the CC model. In section 3, we propose the family of RiBAC models. In Section 4, we present the extended CC framework that includes the core RiBAC policy specification for communities. In Section 5, we discuss related work and in Section 6, we present our conclusions and discuss future work.

2 Community Computing

In this section, we briefly introduce the CC approach used for developing ubiquitous systems, where cooperation among agents is a basic issue. In order to design and develop a CC system, we have earlier proposed two CC models: the *simple community computing* (SCC) model [2] and *community situation based static* CC model [3]. In this paper, we focus on an extension of the SCC model to incorporate access control requirements.

2.1 Related Cooperation Based Approaches

Many cooperation based approaches have been proposed in the literature with the goal of solving emerging large and complex problems. Several groupwares to support CSCW (Computer Supported Cooperative Work) have been proposed in the literature that effectively perform common tasks through information sharing among all users [4, 5].

Multiagent based approaches have been frequently used to develop complex and intelligent systems. Agents in multiagent systems have features such as *flexibility* and *autonomous* problem solving behavior, and the richness of *interactions* that are useful for solving complex problems. In a typical multiagent system, agents interact with each other in order to achieve their common goals. Zambonelli *et al.* [6] propose Gaia methodology in which a multiagent system is regarded as a collection of computational organizations consisting of various interacting roles, and the cooperation among agents playing different roles aimed towards fulfilling the requirements of the system. PICO (Pervasive Information Community Organization) is a middleware framework for dynamically creating mission-oriented communities of autonomous and ubiquitous software objects, called *delegents*, that offer ubiquitous services [7].

In [8], Ishida *et al.* introduce the notion of community computing to support the process of organizing diverse and amorphous people who are willing to share knowledge and experiences. The objective of their approach is to make a city-scale supporting system to assist a human's everyday life – by creating a community that represents a real human community. Their work supports the process of sharing member's preferences and knowledge so that they can reach consensus.

In [10], Blau emphasizes community computing as an essential emerging technological environment where users share each other's computing capabilities and their identities are spread all over various devices, and points out the need for significant research in this area.

2.2 Community Computing Model

As a cooperative approach to provide ubiquitous services, we have earlier developed an agent based approach called the Community Computing (CC) model [2, 3]. The model helps to realize ubiquitous services by utilizing cooperation among intelligent agents in a ubiquitous environment. In CC approach, services are provided by communities of agents having a common goal. This approach helps to intuitively design ubiquitous services based on agent cooperation. A community consists of agents cooperating with other agents in order to achieve the community's goals, and the problems of ubiquitous computing systems are solved by such communities. We introduce the essential concepts of community computing below.

- *Community* - it is a metaphor to abstract a proactive organization that comprises members cooperating with each others to achieve a particular set of goals. A community has goals, necessary roles, cooperation, and role-member binding information. In the CC model, different types of communities are represented as different community templates. At the execution time, a community instance is dynamically created according to the corresponding community template.
- *Role* - it is a well-defined position in a community, with an associated set of expected capabilities. A role represents a particular responsibility necessary to achieve a community's goal. The capability of a role is represented by actions.
- *Cooperation* - it is a set of cooperative interactions among members assuming the roles defined for a community in order to achieve community's goal(s).
- *Member* - it is a metaphor that abstracts an individual agent involved in a CC system. We can consider a human user as a member by using the agent of his/her

personal device. An agent can play different roles in different communities simultaneously.

- *Role-member binding* - in order to create a community instance, we have to find most appropriate members for each role. We refer to this process as role-member binding.
- *Society* - it is a metaphor to abstract a CC system.

In the SCC model, a community has a set of roles, one goal, and mapping information between roles and member agents' types. Each role has attributes, contexts, actions, and the condition for membership assignment. A context of a role is implicitly defined by attributes of the role. The role-membertype mapping indicates which agent types can take which roles defined for a community. The goal description part indicates the initiator role and participant roles of cooperation, and the cooperation itself. To describe a cooperation, the SCC framework uses constructs of OCCAM, a parallel computing language, such as SEQ, PAR, ALT, IF, etc.
As a running example, we explain a part of a community description in a simple CC model (see Fig. 1). The example community is based on an emergency service scenario as follows. While an old man is walking in a street, he suddenly falls down. In order to provide an emergency service to him, an instance of 'EmergencyService' community is created. This community type consists of five roles; patient, ambulance, paramedic, medical doctor, and hospital manager. For each role, agents are selected by the casting condition and the role-membertype mapping condition described in the SCC model. After the creation of a community instance, all member agents cooperate to provide the first-aid service to the patient while the patient is transferred to a close by hospital. When the patient falls down on a street, the patient agent interacts with ambulance agent and medical doctor's agent. The patient agent calls the nearest ambulance and requests help for a doctor. At this time, the patient agent should grant the access to patient's information to doctor and ambulance. After obtaining the patient's location, the ambulance moves to where the patient is located. At the same time, a doctor makes a prescription for the emergency patient using patient's health information, and sends it to the paramedic and the hospital. When ambulance arrives, the paramedic brings the patient into the ambulance and then provides first-aid treatment according to the doctor's prescription. Finally, the patient is transferred to the hospital, and the goal of 'EmergencyService' community instance is achieved.

```
Platform Independent Community Implementation Description {
  Community EmergencyService {
    Role PATIENT {
      Attribute: LOCATION; BLOOD_PRESSURE; PULSE; BODY_TEMPERATURE;
      Context: EMERGENCY;
      Cast: EMERGENCY; }
    Role AMBULANCE {
      Attribute: AVAILABILITY; DRIVER; LOCATION; PATIENT_LOCATION;HOSPITAL_LOCATION;
      Context: ARRIVE_ON_PATIENT; ARRIVE_ON_HOSPITAL;
      Action: transfer_patient_to_hospital; adjust_temperature; adjust_ambulance_speed;
      Cast: AVAILABILITY=AVAILABLE; LOCATION= nearest(PATIENT.LOCATION);}
    Role MEDICAL_DOCTOR {
      Attribute: AVAILABILITY; MAJOR; FIRSTAID_TREATMENT;
      Action: remote_examine; make_prescripton;
      Cast: AVAILABILITY=AVAILABLE; MAJOR=EMERGENCY; }
```

```
Role PARAMEDIC {
  Attribute: AVAILABILITY; LOCATION;
  Action: save_firstaid_treatment; give_firstaid; bring_patient_to_ambulance; bring_patient_to_hospital;
  Cast: AVAILABILITY=AVAILABLE; LOCATION= nearest(AMBULANCE.LOCATION);}
Role HOSPITAL_MANAGER {
  Attribute: EMERGENCY_ACCEPTABILITY; LOCATION;
  Action: ready_for_emergency_patient;
  Cast: EMERGENCY_ACCEPTABILITY=ACCEPTABLE; LOCATION= nearest(PATIENT.LOCATION);}
Role-MemberType Mapping {
  PATIENT:Personal_agent;AMBULANCE:Ambulance_agent; MEDICAL_DOCTOR:Personal_doctor_agent;
  PARAMEDIC:Personal_paramedic_agent; HOSPITAL_MANAGER:Hospital_agent; }
Goal Providing_emergency_service(initiator:PATIENT; participant:AMBULANCE,MEDICAL_DOCTOR,
  PARAMEDIC,HOSPITAL) {
  PATIENT{
    PAR{SEND(MsgType="request", ToWhom=AMBULANCE, certificate(Location));
        SEND(MsgType="request", ToWhom=MEDICAL_DOCTOR, certificate(Healthinfo); ) }
  AMBULANCE{
    IF(RECEIVE(MsgType="request", ToWhom=AMBULANCE, certificate(Location)))
      transfer_patient_to_hospital; }
  MEDICAL_DOCTOR{
    IF(RECEIVE(MsgType="request", ToWhom=MEDICAL_DOCTOR, certificate(Healthinfo)))
      SEQ{
        remote_examine;make_prescripton;
        PAR{
          SEND(MsgType="request", ToWhom=PARAMEDIC, certificate(firstaid_treatment));
          SEND(MsgType="request", ToWhom=HOSPITAL_MANAGER, certificate(firstaid_treatment));}}}
  PARAMEDIC{
    IF(RECEIVE(MsgType="request", ToWhom=PARAMEDIC, certificate(firstaid_treatment)))
      save_firstaid_treatment;
    IF(AMBULANCE.ARRIVE_ON_PATIENT){
      bring_patient_to_ambulance;
      give_firstaid; }
    IF(AMBULANCE.ARRIVE_ON_HOSPITAL;)
      bring_patient_to_hospital;
      IF(PATIENT.LOCATION = HOSPITAL.LOCATION) { SUCCESS; }
  HOSPITAL_MANAGER{
    IF(RECEIVE(MsgType="request", ToWhom=HOSPITAL, certificate(firstaid_treatment))
      ready_for_emergency_patient; } }    }
```

Fig. 1. A part of description for 'EmergencyService' community in a simple community computing model

3 Role Interaction-based Access Control Model

In this section, we propose role interaction based access control (RiBAC) models for the SCC model. Note that agent interaction is a key issue in a CC model. Furthermore, interactions authorized for agents are basically defined by what roles within the community the interacting agents are playing. Such interactions can hence be cast as accesses authorized for agents playing specific roles. For fine-grained role-based policy specification, we categorize agent interactions within a community into two types, as depicted in Fig. 2.

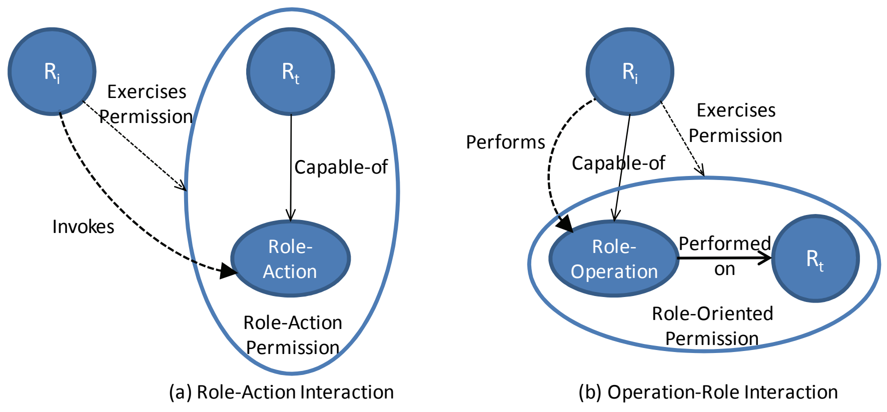

Fig. 2. Two types of Interaction Permissions in Role-based Agent Interaction

Role-action interaction, shown in Fig.2.a, involves an initiator role (r_i) interacting with a target role (r_t) to indicate that the target role should perform some action it is capable of – in other words, we model this as the initiator role authorized to invoke the target role's action. The pair *role* and its *action invocable* by other roles can be considered as a *role-action permission*.

In *Operation-role interaction*, depicted in Fig.2.b, an initiator role can interact with a target role by performing some operation on the target role itself. In this paper, the pair *operation* and a *target role* is termed as a *role-oriented permission*; we use the term *object-oriented permission* to describe traditional RBAC permission that represents an *operation* over an *object*.

It is important to note that in a typical scenario there could exist interdependencies among different types of interactions and *object-oriented* permissions. For instance, a particular *role-action* permission may include several *object-oriented* permissions needed to complete the defined action. If such permission interdependency details could be provided by the underlying environment model, it can be used for access control policy analysis.

In the following subsections, we define the core RiBAC model that extends traditional RBAC with the notion of interaction permissions. We also provide a hierarchical version of the model to leverage hierarchical structures for permission inheritance. It is followed by a constrained RiBAC model.

3.1 Core RiBAC Model

Fig. 3 illustrates the core RiBAC model. Instead of users in standard RBAC model, agents (*AGENTS*) are the entities that can request for access in a MAS environment. Agents are assigned to roles (*ROLES*) and can exercise the permissions assigned to the roles by activating them in a session (*SESSIONS*).

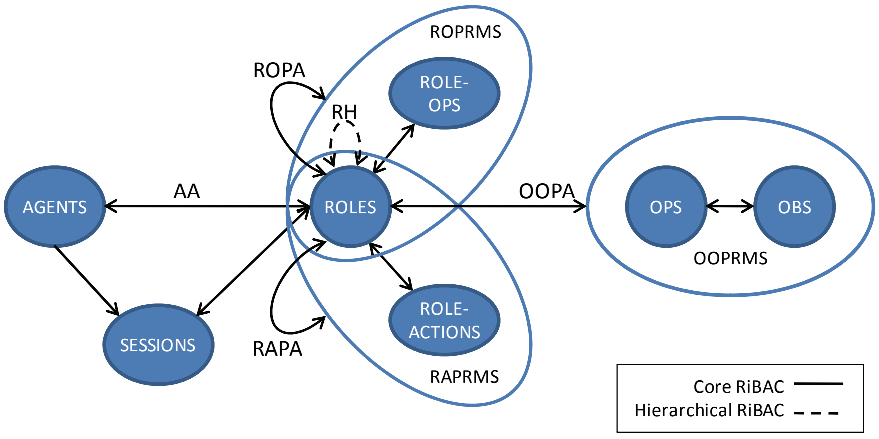

Fig. 3. RiBAC Model

Depending on the application, various objects could exist in the environment which needs to be accessed by agents. The valid pairs objects (*OBS*) and operations on them (*OPS*) form the *object-oriented* permissions (*OOPRMS*). Roles are authorized for object-oriented permissions that are assigned to them through the object-oriented permission assignment relation (*OOPA*).

Interaction permissions include role-action and role-oriented permissions. The valid pairs of roles and their actions (*ROLE-ACTIONS*) invocable by other roles form the role-action permissions (*RAPRMS*). Role-action permissions are assigned to initiator roles according to the policy through role-action permission assignment relation (*RAPA*). An agent that has activated a role is authorized to exercise the assigned role-action permissions (to its role) on any agent that is assuming the target role in the permission. The valid pairs of an operation (*Role-OPS*) and a target role that the operation can be performed on form role-oriented permissions (*ROPRMS*). Role-oriented permissions are assigned to authorized initiators using the role-oriented permission assignment relation (*ROPA*).

Note an interaction permission related to a role can also be assigned to the same role; this will allow agents with the same role in the community to interact with each other. For instance, a guarding agent in a patrol community should be able to ask for help from other guarding agents.

The formal definition of the core RiBAC model follows. It consists of the following basic sets:

- *AGENTS*: the set of all participating agents in a community
- *ROLES*: the set of all roles available in a community
- *SESSIONS*: the set of all sessions created for agents in a community
- *OBS*: the set of all objects in the environment
- *OPS*: the set of all applicable operations on objects in the environment
- *OOPRMS* $\subseteq$ *OPS* $\times$ *OBS*, the set of all object-oriented permissions
- *ROLE-ACTIONS*: the set of all actions that are defined for community roles and can be invoked through interactions
- *RAPRMS* $\subseteq$ *ROLES* $\times$ *ROLE-ACTIONS*, the set of all role-action permissions

- *ROLE-OPS*: the set of all operations that are performable on roles through interactions
- $ROPRMS \subseteq ROLE\text{-}OPS \times ROLES$, the set of all role-oriented permissions

The following relations define the access policy in RiBAC:

- $AA \subseteq AGENTS \times ROLES$, the agent to role assignment
- $OOPA \subseteq OOPRMS \times ROLES$, the object-oriented permission to role assignment
- $RAPA \subseteq RAPRMS \times ROLES$, the role-action permission to role assignment
- $ROPA \subseteq ROPRMS \times ROLES$, the role-oriented permission to role assignment

The following relations capture the runtime state of access control through sessions:

- $SessionAgent(s\text{: } SESSIONS) \rightarrow AGENTS$, the mapping of session *s* to its corresponding agent
- $SessionRoles(s\text{: } SESSIONS) \rightarrow 2^{ROLES}$, the mapping of session *s* to the set of active roles in it

The following functions retrieve the authorization information according to the policy:

- $authorized\_roles(a\text{: } AGENTS) \rightarrow 2^{ROLES}$, the mapping of agent *a* to the set of its authorized roles that it can activate
- $authorized\_ooprms(r\text{: } ROLES) \rightarrow 2^{OOPRMS}$, the mapping of role *r* to the set of its authorized object-oriented permissions
- $authorized\_raprms(r\text{: } ROLES) \rightarrow 2^{RAPRMS}$, the mapping of role *r* to the set of its authorized role-action permissions
- $authorized\_roprms(r\text{: } ROLES) \rightarrow 2^{ROPRMS}$, the mapping of role *r* to the set of its authorized role-oriented permissions
- $authorized\_prms(r\text{: } ROLES) \rightarrow 2^{OOPRMS \cup RAPRMS \cup ROPRMS}$, the mapping of role r to the set of its authorized object-oriented and interaction permissions. Formally: $authorized\_prms(r) = authorized\_ooprms(r) \cup authorized\_raprms(r) \cup authorized\_roprms(r)$

In order to demonstrate the usage of core RiBAC model, we revisit the the 'EmergencyService' community explained in Section 2 (Fig. 1) in Fig. 4. Fig. 5 illustrates the same example policy using graphical notations.

```
ROLES = {Patient, Doctor, Paramedic, Hospital, Ambulance}
OBS = {hospital_medical_equipment, termometer, ambulance_medical_equipment,
    ambulance_vehicle}
OPS = {operate, read}
OOPRMS = {OOP1=(operate,hospital_medical_equipment), OOP2=(read,termometer),
    OOP3=(operate,ambulance_medical_equipment), OOP4=(operate,ambulance_vehicle)}
ROLE-ACTIONS={give_health_status, give_location, remote_examine, give_prescription,
    provide_firstaid, prepare_for_patient, transfer_patient}
RAPRMS={RAP1=(Patient,give_health_status), RAP2=(Patient,give_location),
    RAP3=(Doctor,remote_examine), RAP4=(Doctor,give_prescription),
    RAP5=(Paramedic,provide_firstaid), RAP6=(Hospital,prepare_for_patient),
    RAP7=(Ambulance,transfer_patient)}
ROLE-OPS={bring_into_ambulance, provide_firstaid}
ROPRMS={ROP1=(bring_into_ambulance,Patient), ROP2=(provide_firstaid,Patient)}
```

OOPA={(OOP1,Doctor), (OOP2,Doctor), (OOP3,Paramedic), (OOP4,Ambulance)}
RAPA={(RAP1,Doctor), (RAP1,Paramedic), (RAP1,Ambulance), (RAP2,Ambulance), (RAP3,Patient), (RAP4,Paramedic), (RAP4,Hospital), (RAP5,Doctor), (RAP6,Doctor), (RAP7,Patient)}
ROPA={(ROP1,Paramedic), (ROP2,Paramedic)}

Fig. 4. An example core RiBAC policy specification for 'EmergencyService' community

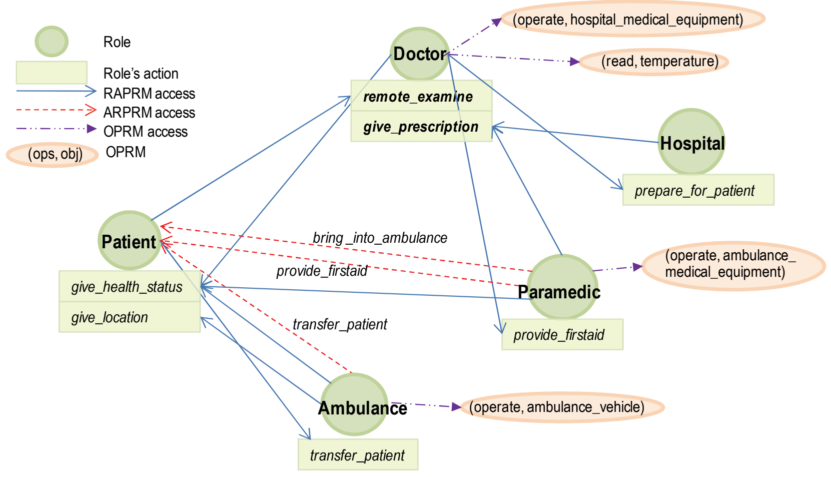

Fig. 5. Graphical representation of the example core RiBAC policy for 'EmergencyService' community

3.2 Hierarchical RiBAC Model (H-RiBAC)

In this section, we propose the hierarchical RiBAC model. One advantage of RBAC model is its ability to leverage hierarchical structure of roles for better permission management. Analogous to standard RBAC, permissions in RiBAC (including object-oriented and interaction permissions) can be inherited through a role hierarchy. We define the role hierarchy *RH* and override the authorization functions in core RiBAC to cope with it as follows:

- $RH \subseteq ROLES \times ROLES$ is a partial order relation on *ROLES*, denoted as $\geq$, where $r \geq r'$ only if all permissions of r' are inherited by r and agents assigned to r can also activate r'. Formally: $r \geq r' \Rightarrow authorized\_prms(r') \subseteq authorized\_prms(r) \wedge [r' \subseteq authorized\_roles(a);\ (a,r) \in AA]$
- $authorized\_roles(a: AGENTS) \rightarrow 2^{ROLES}$, the mapping of agent a to the set of its authorized roles that it can activate in presence of role hierarchy. Formally: $authorized\_roles(a: AGENTS) = \{\, r \in ROLES \mid (a,r') \in AA,\ r' \geq r\}$
- $authorized\_ooprms(r: ROLES) \rightarrow 2^{OOPRMS}$, the mapping of role r to the set of its authorized object-oriented permissions in presence of role hierarchy. Formally: $authorized\_ooprms(r) = \{p \in OOPRMS \mid r \geq r',\ (r',p) \in OOPA\}$
- $authorized\_raprms(r: ROLES) \rightarrow 2^{RAPRMS}$, the mapping of role r to the set of its authorized role-action permissions in presence of role hierarchy. Formally: $authorized\_raprms(r) = \{p \in RAPRMS \mid r \geq r',\ (r',p) \in RAPA\}$

- *authorized_roprms(r: ROLES)* $\to 2^{ROPRMS}$, the mapping of role *r* to the set of its authorized role-oriented permissions in presence of role hierarchy. Formally: *authorized_roprms(r) = {p* $\in$ *ROPRMS | r* $\geq$ *r', (r',p)* $\in$ *ROPA}*

We modify our example to form a role hierarchy among paramedic, doctor, and ambulance, also introducing two new roles. Fig. 6 illustrates a graphical presentation of the hierarchy relation among roles and their assigned permissions. In the hierarchy, the role 'Basic_Medical_Service' and the role 'Medical_Staff' are intermediate roles that are not assigned directly to agents. According to the role hierarchy, 'Paramedic' and 'Doctor' have permissions of 'Medical_Staff' and 'Basic_Medical_Service'. 'Ambulance' also inherits the permission of basic medical service to get the patient health status. Using such patient's health information, an ambulance adjusts the temperature and speed of the vehicle in order to minimize risks to the patient's health. The formal specification of the example policy is shown in Fig 7.

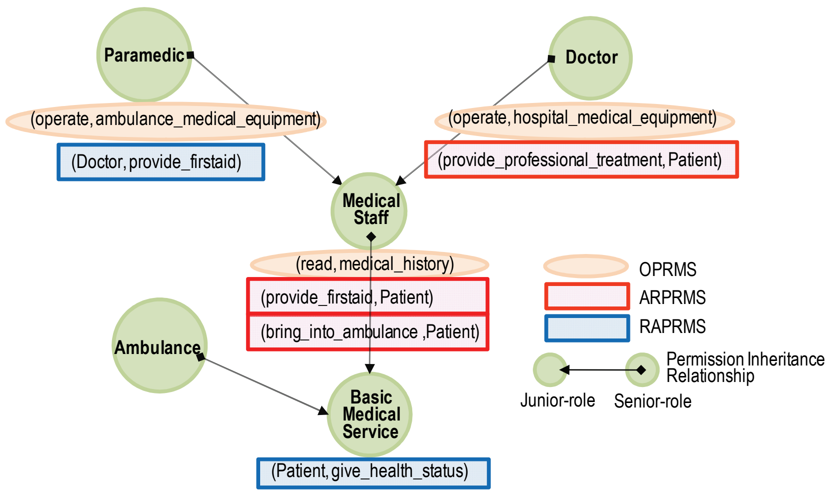
Paramedic
Doctor
(operate, ambulance_medical_equipment)
(operate, hospital_medical_equipment)
(Doctor, provide_firstaid)
(provide_professional_treatment, Patient)
Medical Staff
(read, medical_history)
(provide_firstaid, Patient)
(bring_into_ambulance ,Patient)
Ambulance
Basic Medical Service
(Patient, give_health_status)
OPRMS
ARPRMS
RAPRMS
Permission Inheritance Relationship
Junior-role
Senior-role

Fig. 6. A role hierarchy example for the 'EmergencyService' community

ROLES = {Patient, Doctor, Paramedic, Hospital, Ambulance, Medical_Staff, Basic_Medical_Service}
RH={(Medical-Staff,Basic-Medical-Service), (Ambulance,Basic-Medical-Service),
(Doctor,Medical-Staff), (Paramedic,Medical-Staff)}
OBS = {hospital_medical_equipment, ambulance_medical_equipment, medical_history}
OPS = {operate, read}
OOPRMS={OOP1=(operate,hospital_medical_equipment),
OOP2=(operate,ambulance_medical_equipment), OOP3=(read,medical_history)}
ROLE-ACTIONS={give_health_status, provide_firstaid}
RAPRMS= { RAP1=(Patient,give_health_status), RAP2=(Paramedic,provide_firstaid)}
ROLE-OPS={bring_into_ambulance, provide_firstaid, provide_professional_treatement}
ROPRMS={ROP1=(bring_into_ambulance,Patient), ROP2=(provide_firstaid,Patient),
ROP3=(provide_professional_treatement,Patient)}
OOPA={(OOP1,Doctor), (OOP2,Paramedic), (OOP3,Medical-Staff)}
RAPA={(RAP1,Basic-Medical-Service), (RAP2,Paramedic)}
ROPA={(ROP1,Medical-Staff), (ROP2, Medical-Staff), (ROP3, Doctor)}

Fig. 7. An example policy of H-RiBAC for 'EmergencyService' community

3.3 Constrained RiBAC Model (C-RiBAC)

Constrained RiBAC (C-RiBAC) adds separation of duty and cardinality constraints to the core RiBAC model. Separation of duty (SoD) constraints have been discussed in the RBAC literature as a mechanism to minimize the likelihood of fraud and major errors through simultaneous access of users to key organizational tasks or deliberate collusion of users. Community computing environments have similar vulnerabilities as organizations. We propose static and dynamic SoD constraints for RiBAC. In static SoD, no agent can be assigned to a specific number or more of roles in a role set. The *SSoD* relation is defined as follows:

- $SSoD \subseteq 2^{ROLES} \times N$, a collection of pairs *(rs,n)* that defines static SoDs, where for each *(rs,n)* no agent should be assigned to *n* or more roles from the set *rs*. Formally: $(rs,n) \in SSoD \Rightarrow \nexists a \in AGENTS,\ |authorized\_roles(a) \cap rs| \geq n$.

In contrast to static SoD, dynamic SoD enforces the SoD constraint on role activations instead of agent-role assignments. As a consequence an agent cannot activate certain roles together in one session. The *DSoD* relation is defined as follows:

- $DSoD \subseteq 2^{ROLES} \times N$, a collection of pairs *(rs,n)* that defines dynamic SoDs, where for each *(rs,n)* no agent can activate *n* or more roles from the set *rs* together in one session. Formally:
  $(rs,n) \in DSoD \Rightarrow \nexists s \in SESSIONS,\ |\{r \in SessionRoles(s) | r \in rs\}| \geq n$.

In addition to SoD constraints, an access control mechanism can enforce cardinality constraints. For instance, a community can require a minimum/maximum number of agents to play some particular role in the community; otherwise the community may fail to achieve its goal. Cardinality constraints can be static or dynamic. Static cardinality constraints are applicable on agent-role assignment relation, while dynamic cardinality constraints are enforced on active roles in agents' sessions. We define four different cardinality constraints as follows:

- $SMinCardinality \subseteq ROLES \times N$, a collection of pairs *(r,n)* that defines static minimum cardinality for roles, where for each *(r,n)* at least *n* agents should be assigned to the role *r*. Formally:
  $(r,n) \in SMinCardinality \Rightarrow |\{a \in AGENTS | r \in authorized\_roles(a)\}| \geq n$.
- $SMaxCardinality \subseteq ROLES \times N$, a collection of pairs *(r,n)* that defines static maximum cardinality for roles, where for each *(r,n)* at most n agents should be assigned to the role *r*. Formally:
  $(r,n) \in SMaxCardinality \Rightarrow |\{a \in AGENTS | r \in authorized\_roles(a)\}| \leq n$.
- $DMinCardinality \subseteq ROLES \times N$, a collection of pairs *(r,n)* that defines dynamic minimum cardinality for roles, where for each *(r,n)* at least *n* agents should have activated the role *r* at a particular time. Formally: $(r,n) \in DMinCardinality \Rightarrow |\{s \in SESSIONS | r \in SessionRoles(s)\}| \geq n$.
- $DMaxCardinality \subseteq ROLES \times N$, a collection of pairs *(r,n)* that defines dynamic maximum cardinality for roles, where for each *(r,n)* at most *n* agents should be allowed to activate the role *r* at a particular time. Formally:
  $(r,n) \in DMaxCardinality \Rightarrow |\{s \in SESSIONS | r \in SessionRoles(s)\}| \leq n$.

In the presence of various constraints, it is important to ensure that a RiBAC policy is consistent. A static minimum cardinality of *m* and a static maximum cardinality of

n ($n<m$) for the same role are impossible to be enforced at the same time. Respecting the following rule by the model prevents such a conflict:

- $\forall r \in ROLES\ \forall m,n \in N,\ (r,m) \in SMinCardinality\ \wedge\ (r,n) \in SMaxCardinality \Rightarrow m \leq n$

If we assume the same situation above however with dynamic constraints instead, role r cannot be activated at all. Although, in the latter case the role r becomes useless, but there is no consistency issue for policy enforcement.

The two types of static cardinality and the dynamic maximum cardinality are easily enforceable by keeping a track of assigned or activated roles in a community and avoiding the violation of them. However, the dynamic minimum cardinality is a little tricky to enforce depending on the environment. We assume that there is a proper enforcement mechanism employed in the community to force agents to keep the minimum active roles according to the dynamic minimum cardinality. For instance upon creation of the community, the system can force some agents to activate their roles (even without their discretion), and otherwise can fail the creation.

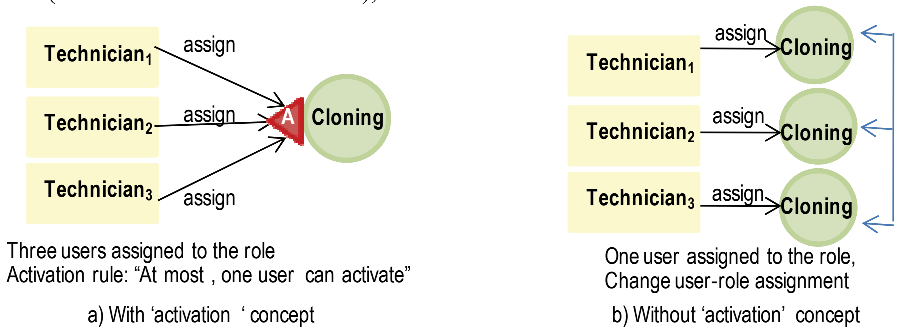
Technician$_1$
assign
Technician$_2$
assign
A
Cloning
Technician$_3$
assign
Three users assigned to the role
Activation rule: "At most , one user can activate"
a) With 'activation ' concept
Technician$_1$
assign
Cloning
Technician$_2$
assign
Cloning
Technician$_3$
assign
Cloning
One user assigned to the role,
Change user-role assignment
b) Without 'activation' concept

Fig. 8. Need for the 'activation' concept in community

In fact, the SCC model does not include explicit notion of activation since it assumes that the assigned roles are activated as soon as the agents take the roles. We believe that such an assumption is not adequate enough and need to be removed to support scenarios where explicit notion of activation is required. As an example, consider a biotechnology project community in which there is a role for cloning body tissues and three technicians are able to take the role as shown in Fig. 8. In this case, three technicians can be assigned to the 'cloning' role. However, this job should be performed by a totally isolated technician because it is a very delicate job. If one technician does perform cloning, then we should prevent accesses to cloning task from another technician. In order to enforce that, we can specify a policy that allows at most one user to activate the 'cloning' role at a time (dynamic maximum cardinality constraint). Although an alternative way is to change the role assignments every time a user wants to access the 'cloning' role according to the community's situation as shown in Fig. 8, such an approach would be very cumbersome due to frequent changes in the policy.

3.4 Constrained Hierarchical RiBAC Model (CH-RiBAC)

A comprehensive RiBAC model is formed by combination of hierarchical and constrained RiBAC models. However, the implications of such combination should be precisely captured. For instance, consider role r_1 has dynamic maximum cardinality constraint of 3, and there exist role r_2 which is senior to r_1 ($r_2 \geq r_1$). In such a configuration, if more than 3 agents activate role r_2 it can be interpreted as violation of the cardinality constraint because agents assigned to r_2 can also assume r_1 through the role hierarchy. However, agents acting as role r_2 may not necessarily act as role r_1 all the time (only sometimes require r_1's permissions), which makes the mentioned interpretation too rigid.

In order to provide more flexibility and truly capture the behavior of constraints in the presence of role hierarchy, we adopt the notion of hybrid hierarchy that is originally defined in the context of Generalized Temporal RBAC (GTRBAC) [11]. A hybrid hierarchy differentiates between permission usage and role activation capability in a hierarchy, by taking into account three possible relations: permission inheritance (I), activation (A), and inheritance-activation (IA). If role r_1 is I-senior to role r_2 ($r_1 \geq_I r_2$), it inherits all the permissions r_2 has. If role r_1 is A-senior to role r_2 ($r_1 \geq_A r_2$), then a user assigned to r_1 can activate r_2 but the role r_1 does not inherit r_2's permissions. Finally, r_1 is IA-senior to r_2 if and only if r_1 is both I-senior and A-Senior to r_2 ($r_1 \geq_{IA} r_2$). Formal definitions for semantics of hybrid hierarchy in RiBAC involve minor changes to the overridden functions in Section 3.2. The hierarchy relation ($\geq$) in the definition of function *authorized_roles* should be replaced with activation relation ($\geq_A$), and the hierarchy relation ($\geq$) in the definition of other authorization functions should be replaced with permission inheritance relation ($\geq_I$).

By leveraging the activation and permission inheritance relationships, we achieve more flexibility in policy specification. For instance, to resolve the problem mentioned in the above example we can specify r_2 A-senior to r_1. Therefore, whenever an agent activates the role r_2, the cardinality constraint is respected, and an agent can also activate the role r_1 when it needs but according to the cardinality constraint.

The definitions for dynamic constraints in presence of hybrid hierarchy are overridden as follows (static constraint definitions remain valid):

- $DSoD \subseteq 2^{ROLES} \times N$, a collection of pairs (rs,n) that defines dynamic SoDs in presence of hybrid hierarchy, where for each (rs,n) no user can activate or use permissions of n or more roles from the set rs together in one session. Formally:
  $(rs,n) \in DSoD \Rightarrow \nexists s \in SESSIONS, |\{r | r' \geq_I r, r' \in rs, r' \in SessionRoles(s)\}| \geq n$.
- $DMinCardinality \subseteq ROLES \times N$, a collection of pairs (r,n) that defines dynamic minimum cardinality for roles in presence of hybrid hierarchy, where for each (r,n) at least n agents should have activated the role r or its I-senior at a particular time. Formally:
  $(r,n) \in DMinCardinality \Rightarrow |\{s \in SESSIONS | r' \geq_I r, r' \in SessionRoles(s)\}| \geq n$.

- *DMaxCardinality* $\subseteq$ *ROLES* $\times$ *N*, a collection of pairs *(r,n)* that defines dynamic maximum cardinality for roles in presence of hybrid hierarchy, where for each *(r,n)* at most *n* agents should be allowed to activate the role *r* at a particular time. Formally:
  $(r,n) \in DMaxCardinality \Rightarrow |\{s \in SESSIONS| \; r' \geq_I r, \; r' \in SessionRoles(s)\}| \leq n.$

4 Extended Simple Community Computing Model

In this section, we extend the SCC specification framework to allow specifying core RiBAC policies as shown in Fig. 8. We refer the readers to [2] for the complete details of SCC specification language. Based on the formal definition described in Fig. 9, we represent an example of SCC model involving core RiBAC policies for the emergency service scenario in Fig. 10.

```
<RiBAC_policy_description>:= RiBAC Policy { <Role_Policy>* }
<Role_Policy>:= <Role_Name> { <Role_OOPRMSs>*, <Role_ROPRMSs>*, <Role_RAPRMSs>* }
<Role_OOPRMSs>:= OOPRMSs = { <OOPRM>+ },
<OOPRM>:=(<OPS>,<OBS>) , <OPS>:=<String>, <OBS>:=<String>
<Role_ROPRMSs>:=ROPRMS = { <ROPRMS>+ }, <ROPRMS>:= (<Action_Name>,<Role_Name> )
<Role_RAPRMSs>:=RAPRMS = { <RAPRMS>+ }, <RAPRMS>:= (<Role_Name>,<Action_Name>)
```

Fig. 9. BNF definition for describing core RiBAC Policy in the SCC model

```
Platform Independent Community Implementation Description {
  Community EmergencyService {
    Role PATIENT { ...}
    ..........
    Role-MemberType Mapping { .... }
    Goal Providing_emergency_service( ..... }
    RiBAC Policy {
      DOCTOR {
        OOPRMSs={(operate,hospital_medical_equipment), (read, temperature)},
        RAPRMSs={(PATIENT,give_health_status),(PARAMEDIC,provide_firstaid),
          (HOSPITAL,prepare_for_patient)} }
      PATIENT {
        RAPRMSs={(DOCTOR,remote_examine),(AMBULANCE,transfer_patient)} }
      AMBULANCE {
        OOPRMSs={(operate,ambulance_vehicle) },
        ROPRMSs={(transfer_patient,PATIENT)}
        RAPRMSs={(PATIENT,give_health_status),(PATIENT,give_location)} }
      PARAMEDIC {
        OOPRMSs={(operate,ambulance_medical_equipment) },
        ROPRMSs={(bring_into_ambulance,PATIENT), (provide_firstaid,PATIENT)}
        RAPRMSs={(PATIENT,give_health_status),(DOCTOR,give_prescription)} }
      HOSPITAL_MANAGER {
        RAPRMSs={(DOCTOR,give_prescription)} }
    }}
```

Fig. 10. An example of the simple community computing model employing core RiBAC

Note that the access control policies for agent interactions are derived from the cooperation definition of communities. Therefore changes in cooperation results in change of access control policies. For the current extension, based on the underlying assumptions in SCC, we consider only predefined cooperation and therefore predefined access control policy. As one of our future works, we leave room for developing more advanced extensions in which policies can be dynamically reconfigured based on changes in cooperation.

In order to enforce RiBAC policies in a CC system, we propose an extension to our existing computation model [2]. In the extended model, policies regarding object-oriented permissions are enforced in a centralized way by the society manager. For policies related to agent interactions, we enforce them in a distributed way. Agents receive the interaction permission specifications in which they are interaction targets from community manager. Based on such specifications, target agents can enforce control over interactions targeted to them. Also note that agents may receive specification about all the permissions they have from community manager, in order to be able to plan based on their accesses. Fig. 11 shows the extended computational model of a community computing system to enforce RiBAC policies.

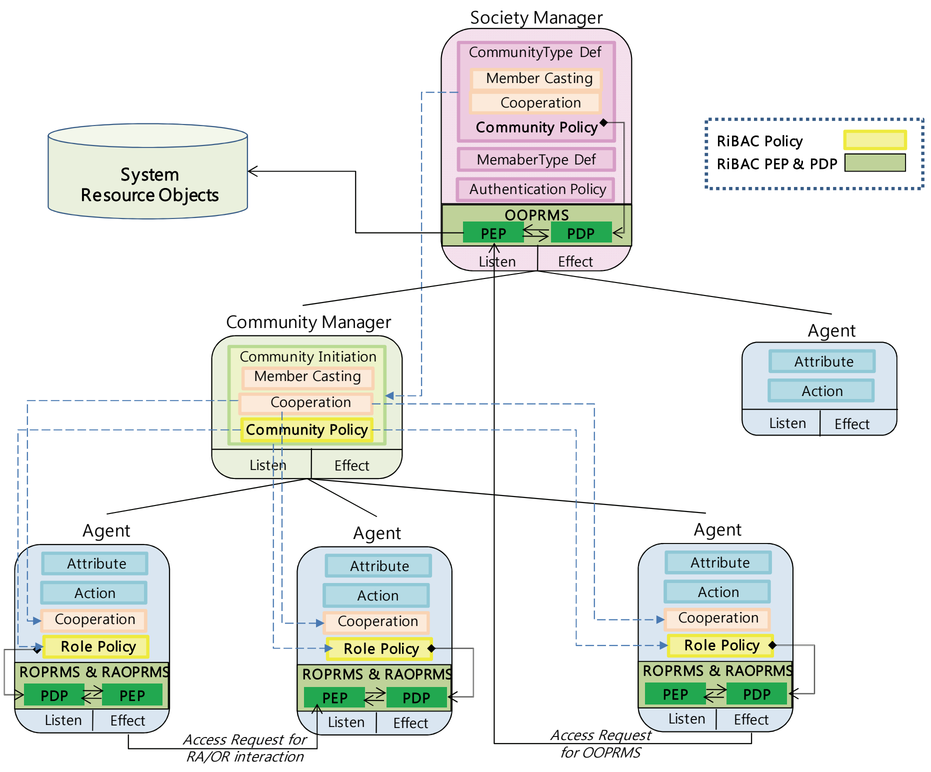

Fig. 11. Enforcement Architecture for core RiBAC

5 Related Works

Many researchers have investigated the security requirements and challenges in multi-agent systems, and pointed out the need for access control in these environments [12, 13]. However, most solutions proposed for access control in MAS are mainly concerned about distributing authorization information using trust management frameworks [14, 15, 16], and less about the access control model itself. These proposals usually adopt SPKI/SDSI (simple public key infrastructure/simple distributed security infrastructure), which is able to manage authorization in a distributed manner using authorization certificates. For instance in [16], Wen *et al.* propose semi-distributed authorization scheme, where agents acquire authorization certificates from an authorization server based on the role certificates their corresponding human users provide.

The closest work to the theme of this paper has been done by Omicini *et al.* in the context of an infrastructure for coordination support in agent-based systems, called TuCSoN [17]. In [18], the authors integrate simple access matrix model (based on agent identity) in a decentralized fashion to authorize exchange of communication tuples among agents. As mentioned, only simple access control lists are allowed by this scheme with an added dimension for controlling tree-structured agents. Later, Omcini *et al.* explore the integration of RBAC into the TuCSoN infrastructure [19]. In order to control the coordination protocol, the authors define a prolog-like role policy definition language. The policies can specify the authorized actions considering the current state of the role and conditions, while determining the next state. The states are managed as part of an alternative for RBAC session. While their approach seems flexible and powerful, the definition of a state-based policy can be very impractical. Also their approach does not include explicit semantics for authorized role interactions, which has been emphasized in this paper, and provides no formal semantics for SoDs and role hierarchy.

Gaia methodology [6] involves role concept and an interaction model among agents. In Gaia, some access control concepts are discussed such as role permissions (on objects), or organizational safety rules that could act as separation of duty constraints. However, we have a more specific approach to specify authorized interaction compared to the interaction notion in Gaia. Our interaction modeling approach is more practical to enable specification and control over interactions in detail. In addition, we provide hierarchical relations among roles to enable more manageable access control policies.

6 Conclusion and Future Work

In order to control accesses to critical data or actions of other agents, , we have proposed a family of RiBAC (Role interaction Based Access Control) models including core RiBAC, H-RiBAC that incorporates role hierarchy, C-RiBAC that incorporates SoD and cardinality constraints, and CH-RiBAC that incorporates constraints and hybrid hierarchy. These are extensions of the standard RBAC models and cover the role interaction as one of the important aspects of MAS. RiBAC models are useful for securing ubiquitous systems characterized by significant agent interactions. We have extended the earlier proposed simple community computing modeling framework to incorporate the proposed RiBAC models.

As future work, we plan to extend the proposed work to cope with context-aware ubiquitous environments by integrating it with time and location based RBAC (LoTRBAC) model [20]. We are currently implementing a working prototype of the proposed work. Moreover, we will investigate models that could support administration and delegation of role interaction permissions in the context of community computing. We also plan to explore security analysis and policy verification method, as well as efficient enforcement techniques for RiBAC policies.

Acknowledgments. This research is supported by Foundation of ubiquitous computing and networking project (UCN) Project, the Ministry of Knowledge Economy (MKE) 21st Century Frontier R&D Program in Korea and a result of subproject UCN 08B3-S2-10M, and by the US National Science Foundation award IIS-0545912.

References

1. Wooldridge, M., Jennings, N.R.: The Cooperative Problem-Solving Process. Journal of Logic Computation 9 (4), pp. 563--592, Oxford University Press (1999)
2. Jung, Y., Lee, J., Kim, M.: Multi-agent based Community Computing System Development with the Model Driven Architecture. In Proc. of 5th International Joint conference on Autonomous Agents and Multiagent Systems (AAMAS'06), pp. 1329--1331 (2006)
3. Jung, Y., Lee, J. Kim, M.: Community Computing Model supporting Community Situation based Cooperation and Conflict Resolution, LNCS, vol. 4761, pp. 47--56, Springer-Verlag Berlin Heidelberg (2007)
4. Wilson, P., et. al.: Computer Supported Cooperative Work. Oxford, Intellect Books, UK (1991)
5. Borghoff, U.M., Schlichter, J.H.: Computer-Supported Cooperative Work: Introduction to Distributed Applications. Springer-Verlag, Berlin (2000)
6. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing Multiagent Systems: The Gaia Methodology. ACM Transactions on Software Engineering and Methodology 12 (3), pp. 317--370 (2003)
7. Kumar, M., Shirazi, B., Das, S.K., Singhal, M., Sung, B., Levine, D.: Pervasive Information Communities Organization PICO: A Middleware Framework for Pervasive Computing. IEEE Pervasive Computing 2 (3), pp.72--79 (2003)
8. Ishida T. Ed.: Community Computing and Support Systems. LNCS, vol. 1519, Springer-Verlag (1998)
9. Van den Besselaar, P., Tanabe, M., Ishida, T.: Introduction: Digital Cities Research and Open Issues. LNCS, vol. 2362, pp. 1--9, Springer-Verlag (2002)
10. Blau, J.: Microsoft: Community Computing is On the Way. InfoWorld Magazine, http://www.infoworld.com/article/05/11/22/HNcommunitycomputing_1.html
11. Joshi, J.B.D., Bertino, E., Latif, U., Ghafoor, A.: A Generalized Temporal Role-Based Access Control Model. IEEE Transactions on Knowledge and Data Engineering 17(1), pp. 4--23, 2005.
12. Beydoun, G., Low, G., Mouratidis, H., Henderson, B.: Modelling MAS-Specific Security Features. IEEE 2nd Symposium on Multi-Agent Security and Survivability, pp.75--84 (2005)

13. Mouratidis, H., Giorgini, P., Manson, G.: Modeling Secure Multiagent Systems. In Proc. of AAMAS 2003, pp. 859--866 (2003)
14. Hu, Y., Tang, C.: Agent-Oriented Public Key Infrastructure for Multi-agent E-service. In Proc. of the 7th Int'l Conference on Knowledge-Based Intelligent Information and Engineering Systems, pp. 1215--1221 (2003)
15. Poggi, A., Tomaiuolo, M., Vitaglione, G.: A Security Infrastructure for Trust Management in Multi-agent Systems. In Proc. of the Conference on Trusting Agents for Trusting Electronic Societies, pp. 162--179 (2004)
16. Wen, W., Mizoguchi, F.: An Authorization-based Trust Model for Multiagent Systems. Applied Artificial Intelligence 14(9), pp. 909--925 (2000)
17. Omicini, A., Zambonelli, F.: Coordination for Internet Application Development. Autonomous Agents and Multi-Agent Systems 2(3), pp. 251--269 (1999)
18. Cremonini, M., Omicini, A., Zambonelli, F.: Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach. In Proc. of the 4th International Conference on Coordination Languages and Models, pp. 99--114 (2000)
19. Omicini, A., Ricci, A., Viroli., M.: RBAC for Organisation and Security in an Agent Coordination Infrastructure. In Proc. of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems, pp. 65--85 (2004)
20. Chandran, S.M., Joshi, J.B.D.: LoT-RBAC: A Location and Time-based RBAC Model. In Proc. of the 6th International Conference on Web Information Systems Engineering, pp. 361--375 (2005)