Available online at www.sciencedirect.com
SciVerse ScienceDirect
journal homepage: www.elsevier.com/locate/cose
Computers & Security

CRiBAC: Community-centric role interaction based access control model

Youna Jung[a,*], James B.D. Joshi[b,1]

[a] ACIS Lab., Department of Electrical and Computer Engineering, University of Florida, P.O. Box 116200, 339E Larsen Hall, Gainesville, FL 32611-6200, USA
[b] LERSAIS, Department of Information Science, University of Pittsburgh, 410 IS building, 135 N. Bellefield Avenue, Pittsburgh, PA 15260, USA

ARTICLE INFO

Article history:
Received 2 August 2011
Received in revised form
3 January 2012
Accepted 1 February 2012

Keywords:
Secure interaction
Secure cooperation
Access control
RBAC
Community computing
Multi-agent system

ABSTRACT

As one of the most efficient solutions to complex and large-scale problems, multi-agent cooperation has been in the limelight for the past few decades. Recently, many research projects have focused on context-aware cooperation to dynamically provide complex services. As cooperation in the multi-agent systems (MASs) becomes more common, guaranteeing the security of such cooperation takes on even greater importance. However, existing security models do not reflect the agents' unique features, including cooperation and context-awareness. In this paper, we propose a Community-based Role interaction-based Access Control model (CRiBAC) to allow secure cooperation in MASs. To do this, we refine and extend our preliminary RiBAC model, which was proposed earlier to support secure interactions among agents, by introducing a new concept of interaction permission, and then extend it to CRiBAC to support community-based cooperation among agents. We analyze potential problems related to interaction permissions and propose two approaches to address them. We also propose an administration model to facilitate administration of CRiBAC policies. Finally, we present the implementation of a prototype system based on a sample scenario to assess the proposed work and show its feasibility.

1. Introduction

The rapid growth of networking technologies has significantly promoted the level of connectivity and interaction among distributed computing elements. Particularly, multi-agent systems (MASs) that feature rich interactions among agents have become a very active area of research. The rich interactions among agents promote seamless cooperation that has potential to address large and complicated problems which cannot be solved by an individual agent. Such benefits have been the key reasons that many researchers have been studying cooperation approaches for decades in many areas such as swarm intelligence, MAS, and Computer Supported Cooperative Work (CSCW). Several MASs that provide services through cooperation among agents have been proposed in the literature, including, Gaia (Zambonelli et al., 2003), Pervasive Information Community Organization (PICO) (Kumar et al., 2003), and Community Computing (Jung and Kim, 2010). Unlike cooperation in other areas, dynamic cooperation in MASs has been regarded as a critical issue due to the agent's autonomous and dynamic characteristics. Current MASs support the context-awareness necessary for dynamic cooperation to some degree (Zambonelli et al., 2003; Kumar et al., 2003; Jung and Kim, 2010); however, these systems pose significant adoption challenges because of security concerns.

* *Corresponding author*. Tel.: +1 352 392 1525; fax: +1 352 392 5040.
E-mail addresses: younajung@ufl.edu, younajung@gmail.com (Y. Jung), jjoshi@pitt.edu (J.B.D. Joshi).
[1] Tel.: +1 412 624 9982; fax: +1 412 624 2788.

0167-4048/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2012.02.002

In particular, access control is one of the critical security issues facing MASs. Several access control models have been proposed in the literature that may be adopted for MASs. Among them, the Role-based Access Control (RBAC) approach has generated significant interest because of its flexibility and potential benefits (Sandhu et al., 1996). All existing access control models – including RBAC – aim to protect objects/information against unauthorized accesses; however, they do not consider interactions among agents. In particular, in MASs, we need to carefully examine interactions as well as objects. During an interaction among agents, an agent might need to perform operations on a partnering agent or ask the partner to execute its task. However, serious security issues exist that are related to allowing such interactions or accesses between partnering agents without proper authorization verification (Jung et al., 2011). One partner may attempt to execute critical actions within another partner's system, such as, changing the partner's status or using the partner's functionality without appropriate authorization. Therefore, we need a suitable access control model to ensure that authorized agents can execute only authorized interactions. Beyond interaction among agents, MASs need to guarantee secure dynamic cooperation among agents in order to provide complex and diverse sets of services. To do so, we need an access control model that supports an agent's dynamic behavior and the its need for cooperation with other agents simultaneously.

To fulfill such requirements, we propose the Role-interaction based Access Control Model (RiBAC) that considers an interaction between roles as an entity that also needs to be protected. RiBAC extends the types of protection objects in traditional RBAC by incorporating authorized role-based interactions among agents. By employing RiBAC, the developers of MASs can enable agents to block misuse of its services by others. Previously, we have proposed an early version of RiBAC in Jung et al. (2009), which we refine and extend in this paper. Then, we propose the community-based RiBAC (CRiBAC) model as an extension of RiBAC by incorporating the concept of community that refers to a cooperative group of agents. This model aims to guarantee the security of dynamic cooperation among agents as well as the interactions among them. In particular, CRiBAC is based on the community computing paradigm proposed in Jung and Kim (2010), which is an agent-based computing paradigm in which services are provided through cooperation among agents. By importing its community concept, CRiBAC can deal with context-aware cooperation among agents. We also propose an administration model for CRiBAC, called ACRiBAC, to help administer CRiBAC policies in MAS-based applications. ACRiBAC includes a grant model, a revocation model, and administration functions for cooperation. We also present a technique to analyze the conflicts that may result due to the permitted interactions during cooperation. Finally, we present the implementation of a prototype system to demonstrate our work using an emergency scenario. In summary, the major contributions of our work are as follows:

- we propose the RiBAC model to support the security of agent interactions.
- we propose the CRiBAC model to deal with the context-aware secure cooperation among agents as well as the secure interactions within a community setting.
- we propose the ACRiBAC model to support the administration of systems employing CRiBAC.
- we develop an analysis technique to verify the CRiBAC policies.
- we implement the visual user interfaces to help specify and analyze CRiBAC policies.
- we implement a prototype based on an emergency scenario to demonstrate the practical feasibility of the proposed CRiBAC model.

The rest of this paper is organized as follows. In Section 2, we present a brief description of community computing and our motivation for the current work. Then, in Section 3, we propose the family of RiBAC models. We propose the CRiBAC model in Section 4. Next, we analyze a number of conflicting scenarios related to interaction permissions and then provide two solutions in Section 5. In Section 6, we propose the ACRiBAC model. In Section 7, we present the implementation of a prototype system and assess the feasibility and the usefulness of the proposed work. In Section 8, we discuss related work; finally, we present the conclusions and future work to be investigated in Section 9.

2. Preliminaries and motivation

In this section, we first introduce the community computing paradigm and examine the access control issues in such cooperative MASs. Then, we present our motivation for the proposed models.

2.1. *Community computing*

Community Computing (CC) is an agent-based computing paradigm where services are provided through dynamic cooperation among individual agents (Jung and Kim, 2010). Ordinarily, each agent performs its own task or set of tasks in a community computing system (CCS), known as a society. When a goal is identified, a community is formed by defining the necessary roles and the cooperation processes among roles; then, the community is dynamically created by recruiting the best available agents for each role. To achieve a goal, cooperation among community members begins as soon as all of the members are selected and the community is created. After the goal is achieved, the community is dissolved and members are released. When an agent participates in a community, it needs to perform tasks to play a community role(s). Therefore, agents should check their abilities as well as their willingness to participate before accepting a role as a community member. Once a community is established, the community's structure (such as its roles and cooperation processes), can be reused for another community having the same community goal. In Fig. 1, a general overview of CC is presented and its specification is shown in Table 1.

The basic concepts used in CC are as follows:

- *Community* – is a goal-oriented cooperative group of agents which take on one or more community roles. It is dynamically created and dissolved when the goals have been attained. All communities should belong to a society.

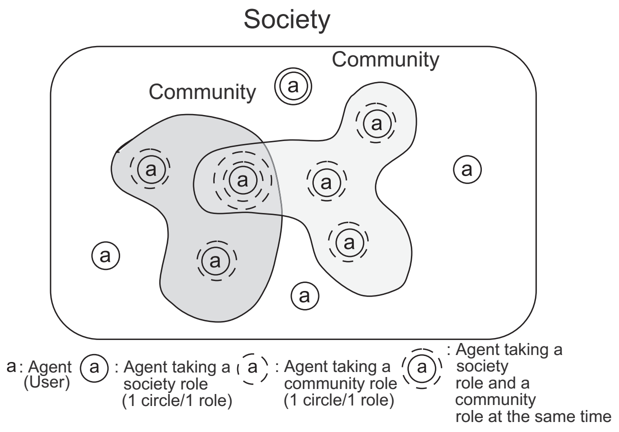

Society
Community
Community
a
a : Agent (User)
a : Agent taking a society role (1 circle/1 role)
a : Agent taking a community role (1 circle/1 role)
a : Agent taking a society role and a community role at the same time

Fig. 1 – Overview of community computing.

- *Society* – is the basic group of agents and communities. A society has several communities, each of which has several agents. All agents who belong to a society take on one or more society roles as long as they are part of the society.
- *Agent* – is a basic unit of societies and communities, having its own intelligence, tasks, and context information. While agents are doing their own work as a society member, they can also participate in one or more communities simultaneously.

2.2. Motivation

Ubiquitous services are currently being expanded to various applications such as u-healthcare, u-government, u-city, and so on. For practical adoption of such emerging services, security issues are key challenges. In order to provide secure services, CCSs for ubiquitous applications should incorporate efficient security mechanisms. There are many security issues related to ubiquitous computing systems such as authentication, privacy, and access control. In this paper, we concentrate on the access control issues.

In CCSs in which agents frequently interact with each other, it is important to control access to not only resources but also to agents' tasks or agents themselves. An agent can interact with its partner to execute the partner's tasks or to carry out its own tasks on the partner; such interactions can result in some critical security problems. It can be assumed that a smart home environment is one in which intelligent appliances are deployed and all family members have personal agents located on their own personal devices. A door lock agent can reset the password and perform such a task only for an authorized agent, e.g., the owner agent which represents the owner of the house. If an unauthorized entity (such as a neighbor's personal device) asks to reset the password via remote access, the door lock must reject the request for safety/security reasons. Similarly, only the owner should be able to open a safe. Otherwise, the safe should sound an alarm to prevent unauthorized access. It can be a significant security vulnerability to accept unauthorized accesses to agents or agents' tasks. Therefore, we need a proper access control mechanism to ensure that agents are engaged in only authorized activities. As a solution, we propose the Role Interaction based Access Control Model (RiBAC). In RiBAC, an interaction indicates an access to agents or agents' resources, such as information or a task; only acceptable interactions among agents based on roles are permitted.

Beyond the interactions among agents, it is important to consider the issue of cooperation among agents playing different roles. Over the last few decades, researchers have studied and developed cooperation systems because they can lead to effective ways to solve complex and large-scale problems. RiBAC can be used to ensure secure interactions; however, this is not enough for some cooperation systems. To secure cooperation, it is necessary that a guarantee that their cooperation does not lead to security threats exists. In order to do this, we propose CRiBAC, the Community based RiBAC model. CRiBAC employs the cooperation paradigm of the community computing approach. By using the community concept of community computing, we can control accesses in

Table 1 – The simplified community computing model (CCM) for CRiBAC.

Element (Abbr.)	Definition	Description
Context (*CONT*)		The set of all context information which represents agents' status in a society.
Task (*TSK*)		The set of all tasks performed by agents in a society.
Agent (*a*)	$<CONT_a, TSK_a>$	An agent has its own contexts and tasks, $a = <CONT_a, TSK_a>$. $CONT_a \in CONT$ is a set of contexts which represent the status of an agent *a*, and $TSK_a \in TSK$ is a set of tasks that an agent *a* can do. *A* is the set of all agents in a society.
Society Role (SR)		A set of all society roles in a society. Each agent takes one or more society roles when they belong to a society.
Community Role (CR)		A set of all community roles in a society. Each agent participating in a community has to take one or more community roles in the community, but community roles should be revoked from those agents when the community is terminated.
Role (R)	SR ∪ CR	The set of all roles in a society.
Role Member (A_r)	$A_r \in A$	A set of agents taking a role *r*.
Goal (G)		The set of goals of communities in a society.
Community (c)	$<g_c, CR_c, A_c>$	A community *c* has a goal (g_c), necessary community roles (CR_c) and community agents (A_c) who take on one or more community roles to achieve the goal g_c. $c = <g_c, CR_c, A_c>$, $g_c \in G$, $CR_c \in CR$, and $A_c \in A$. C is the set of all such communities in a society.
Society (S)	<C, A>	A community computing system (CCS) which provides cooperative services by dynamic and mission-oriented communities. A society consists of a set of communities C and a set of agents *A*.

a cooperation system more intuitively and conveniently. For example, to organize a community and to control cooperation among its agents, CRiBAC can be used to specify the community-related information such as the community's goal, roles, participants, and context within which it cooperates. When seeking suitable agents to create a community, such information might be quite useful to examine participating communities and confirm the community roles of a candidate agent as well as the candidate's tasks and context. Such information is also necessary for an agent to determine whether or not it wants to authorize other agents to interact with it. Furthermore, for efficient administration, it is much better to specify a cooperation group and corresponding relationships in the access control model. For these aforementioned reasons, in this paper, we propose CRiBAC by extending RiBAC.

3. RiBAC

RiBAC (Jung et al., 2009) is based on role-based agent interactions to protect not only resources in MASs, but also the agents' tasks and the agents themselves. In order to do so, an interaction between agents in RiBAC is regarded as an access to a partner's task or to the partner itself. In this section, we propose the interaction permissions in RiBAC and the RiBAC family. A general overview of RiBAC is shown in Fig. 2

3.1. *Interaction permissions*

RiBAC includes two types of role interactions: Role-Oriented (*RO*) interactions and Task-Oriented (*TO*) interactions (See Fig. 4). The *RO* interaction indicates that a subject role (R_s) initiating an interaction performs its operation on a targeted object role (R_o), while the *TO* interaction indicates that R_s commands its R_o to perform R_o's tasks. For example, a *paramedic* can transfer a *patient* to an ambulance or a hospital. Such interaction between a paramedic and a patient is a *RO* interaction. In this interaction, the *paramedic* is a subject role and the *patient* is an object role. In addition, a *doctor* who is in charge of the *patient* can order the *paramedic* to provide a first-aid instruction. This interaction between the *paramedic* role and the *doctor* role is a *TO* interaction. The *doctor* is a subject role and its object role is the *paramedic*. For a successful role interaction, a subject role must have corresponding interaction permissions and, in case of *TO* interaction, its object role should have all the permissions necessary for accomplishing the requested tasks.

Permissions in RiBAC include object-oriented permissions (*OPRMS*) and the two interaction permissions mentioned above. Depending on the application needs, various objects may exist in an environment which can be accessed by agents. A valid pair of an object and an operation on that object forms an *OPRMS*. A valid pair of an operation of a subject role and its target role forms a role-oriented permission (*RPRMS*). A *TPRMS* consists of a role and its task which can be invoked by other roles. Roles are authorized for permissions that are assigned to them through the permission assignment (*PA*).

3.2. *RiBAC family*

RiBAC is comprised of a family of four models: Basic RiBAC (RiBAC-B), Hierarchical RiBAC (RiBAC-H), Constrained RiBAC (RiBAC-C), and Constrained Hierarchical RiBAC (RiBAC-CH). RiBAC-B is the base model to control role interactions. RiBAC-H extends RiBAC-B with a role hierarchy for the convenient management of permissions. RiBAC-C supports RiBAC-B and also allows constraints to support more fine-grained access control requirements. RiBAC-CH supports both RiBAC-Hybrid Hierarchy (HH) and RiBAC-C. In Table 2, we provide the formal definitions for each model.

To specify the proposed models, in this paper, we propose the XML-based specification language. Due to its extensibility and interoperability, XML has been widely used as a policy language for enterprise-wide access control in the distributed environment in which highly heterogeneous entities collaborate (Bhatti et al., 2005). In an effort to use XML as a standard specification of access control policy, XML-based Access Control Language (XACML) has been proposed (XACML v.3.0, 2010). To support RBAC, XACML includes the RBAC profile XACML v.3 Core Hierarchical RBAC Profile v.1.0 (2010) but current profile does not support essential features of RBAC such as separation of duty (SoD) constraints and role hierarchy (Ferrini and Bertino, 2009). To overcome the shortcomings of XACML, Bhatti et al. proposed X-GTRBAC, an XML-based specification language based on the GTRBAC model (Bhatti et al., 2005). X-GTRBAC captures not only the semantics of XACML but also the semantics of RBAC's constraints and role hierarchy. In addition, it addresses the context-awareness, in particular temporal context, for dynamic fine-grained access control. In this paper, we propose a specification language for each proposed model based on X-GTRBAC.

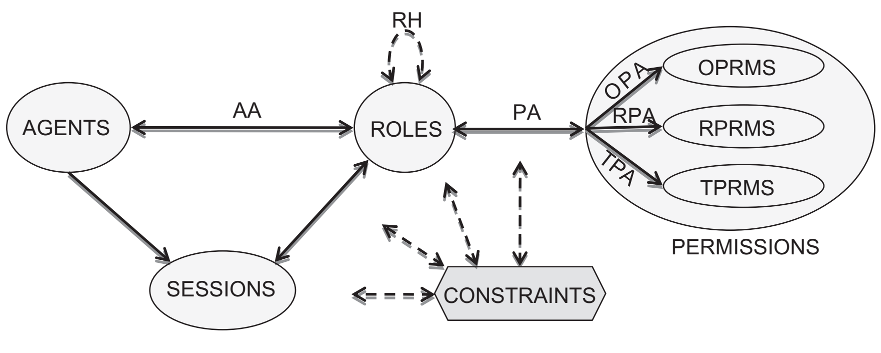


Fig. 2 – RiBAC overview.

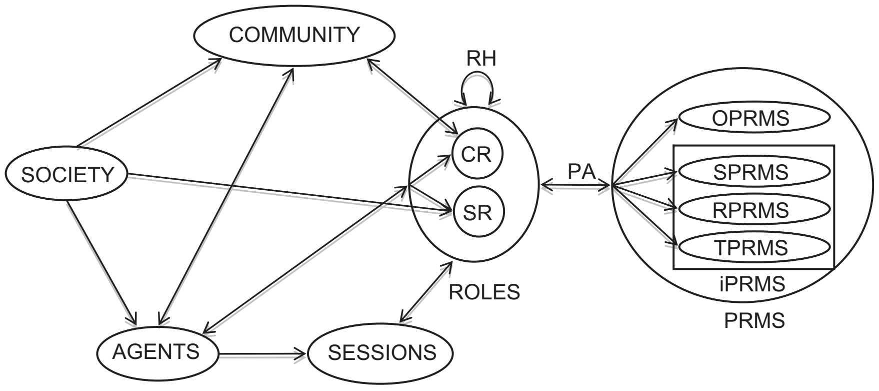


Fig. 3 – Overview of CRiBAC.

To present a specification language, we use the syntax of X-Grammar (Bhatti et al., 2005), a BNF-like grammar, instead of presenting XML schemas. Using BNF notion and XML's tagging notation allows better readability and presentation.

- RiBAC-B: This is the basic model which deals with accesses to agents or to tasks belong to those agents based on agents' roles. The formal definition is shown in the following Table 2 and the corresponding specification is shown in the Appendix A-1.
- RiBAC-H: For better permission management and inheritance, in RiBAC-H, permissions including object-oriented and interaction permissions can be inherited through a role hierarchy and override the authorization functions in RiBAC-B. We define the role hierarchy (*RH*) as shown in Table 2. In the specification of RiBAC-H, there is a change in the role definition only, as presented in the Appendix A-2(a).
- RiBAC-C: This model adds separation of duty (*SoD*) and cardinality constraints to RiBAC-B as presented in Table 2. SoD constraints have been examined in the RBAC literature; they serve as a mechanism to minimize the likelihood of fraud and major errors through simultaneous access by agents to key organizational tasks or deliberate collusion by agents. Community computing environments have similar vulnerabilities. As a remedy, we propose the static and dynamic SoD constraints for RiBAC. In the static SoD (*SSoD*), no agent can be assigned to more than a specific number of roles in a role set. In contrast to SSoD, the dynamic SoD (*DSoD*) enforces the SoD constraint on role activations instead of agent-role assignments (*AA*). As a consequence, an agent cannot activate certain roles together in one session. Also, RiBAC-C supports cardinality constraints that limit the number of agents that may be assigned to a role. The cardinality constraints can be static or dynamic. Static cardinality constraints are applicable to the AA relationship, while dynamic cardinality constraints are imposed on active roles in agents' sessions. Moreover, cardinality constraints can be considered as minimum and maximum limitations. We define four different cardinality constraints: *SSoD_Min_Cardinality*, *SSoD_Max_Cardinality*, *DSoD_Min_Cardinality*, and *DSoD_Max_ Cardinality*. Note that care must be taken to ensure the consistency of the policy by avoiding definition of conflicting constraints. A static minimum cardinality of *m* should be less than a static maximum cardinality of *n* ($m < n$) for a role. The specification of RiBAC-C is presented in the Appendix A-2(b).
- RiBAC-CH: RiBAC-CH is formed by a combination of hierarchical and constrained RiBAC models. However, the implications of such a combination should be precisely captured.

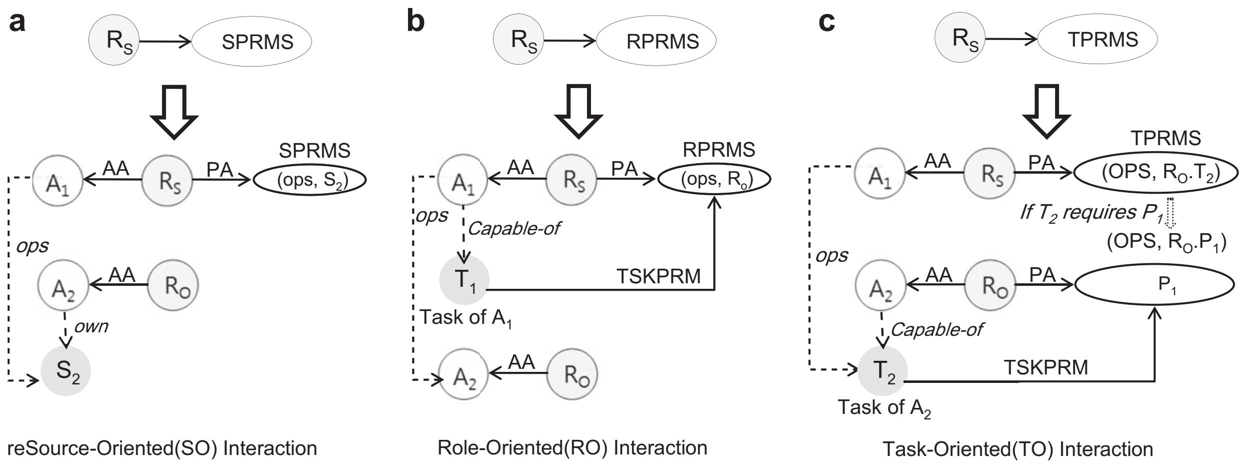


Fig. 4 – Interaction permissions detailed with relationships in CRiBAC.

Table 2 – Formal definition of the family of RiBAC models: RiBAC-B, RiBAC-H, RiBAC-C, and RiBAC-CH.

Model	Element	Definition	Description
RiBAC-B	A		The set of all agents in a system, an agent *a* has his/her own tasks. $a = \{TSK_a\}$, $TSK_a \in$ TSK is a set of *a*'s tasks.
	TSK	ACT ∪ OPS	The set of all tasks which agents can do.
	ACT		The set of all actions of roles which do not require any targets like walking and speaking.
	OPS		The set of all applicable operations of roles on OBJ. Some tasks need one or more permission(s) and such a relationship between an agent's task and permission is represented by TSKPRM.
	TSKPRM	$\subseteq TSK \times P$	A many-to-many task to permission relationship. This relationship represents a necessary permission(s) to perform a task.
	R		The set of all roles available in a society. The following function retrieves the authorized roles of an agent *a* according to the policy: *authorized_roles(a:A)*$\rightarrow 2^R$, the mapping of agent *a* to the set of its authorized roles that it can activate.
	OBJ	OBJ_s ∪ OBJ-ROLE ∪ OBJ-TSK	The set of all target objects of OPS and it can be a system object, role, or role's task.
	P	OPRMS ∪ RPRMS ∪ TPRMS	A set of permissions in a society. The following function retrieves the authorized permissions of a role *r* according to the policy: *authorized_prms(r:R)* $\rightarrow 2^{OPRMS \cup RPRMS \cup TPRMS}$, the mapping of role *r* to the set of its authorized permissions including object-oriented permissions and interaction permissions. Formally: *authorized_prms(r)* = *authorized_oprms(r)* ∪ *authorized_rprms(r)* ∪ *authorized_tprms(r)*
	OPRMS	$\subseteq OPS \times OBJ_s$	The set of all object-oriented permissions. Retrieval function: *authorized_oprms(r:R)*$\rightarrow 2^{OPRMS}$, the mapping of role *r* to the set of its authorized *oprms*.
	RPRMS	$\subseteq$ OPS × OBJ-ROLE	The set of all role-oriented permissions. Retrieval function: *authorized_rprms(r:R)* $\rightarrow 2^{RPRMS}$, the mapping of role *r* to the set of its authorized *rprms*.
	TPRMS	$\subseteq$ OPS × OBJ-TSK	The set of all task-oriented permissions. Through a corresponding TSKPRM relationship, it can be specified as TPRMS ⊆ OPS × OBJ-ROLE.OBJ-PRMS where OBJ-PRMS ⊆ P, (OBJ-ROLE, OBJ-PRMS) ⊆ PA, and (OBJ-TSK, OBJ-PRMS) ⊆ TSKPRM. Retrieval function: *authorized_tprms(r:R)* $\rightarrow 2^{TPRMS}$, the mapping of role *r* to the set of its authorized *tprms*.
	OBJs		The set of a system's objects/resources.
	OBJ-ROLE	⊆R	The set of roles that can be an object of an operation *ops*.
	OBJ-TSK	⊆TSK	The set of tasks which can be a target object of a *tprms*.
	PA	{OPA ∪ RPA ∪ TPA} ⊆ $R \times P$	A many-to-many role to permission assignment relationship.
	OPA	⊆R × OPRMS	A many-to-many role to object-oriented permission assignment relationship.
	RPA	⊆R × RPRMS	A many-to-many role to role-oriented permission assignment relationship.
	TPA	⊆R × TPRMS	A many-to-many role to task-oriented permission assignment relationship.
	S		The set of all sessions created for agents in a society. The following relationships capture the runtime state of access control through sessions: *SessionAgents*(s:S)→ A, the mapping of session *s* to its corresponding agent, and *SessionRoles*(s:S) → 2^R, the mapping of session *s* to the set of active roles in it.
RiBAC-H	RH	⊆$R \times R$	A partial order relationship for R, denoted as ≥, where *r* ≥ *r'* only if all permissions of *r'* are inherited by *r* and agents assigned to *r* can also activate *r'*.
RiBAC-C	SSoD	$\subseteq 2^R \times N$	A collection of pairs (*rs*, *n*) that defines SSoDs, where for each (*rs*, *n*) no agent should be assigned to *n* or more roles from the set of roles *rs*. Formally: $(rs, n) \in SSoD \Rightarrow \nexists\, a \in A, \lvert authorized\_roles(a) \cap rs\rvert \geq n$.
	DSoD	$\subseteq 2^R \times N$	A collection of pairs (*rs*, *n*) that defines DSoDs, where for each (*rs*, *n*) no agent can activate *n* or more roles from the set of roles *rs* together in one session. Formally: $(rs, n) \in DSoD \Rightarrow \nexists s \in SESSIONS, \lvert\{r \in SessionRoles\ (s) \mid r \in rs\}\rvert \geq n$. SMinCardinality ⊆ R × N, a collection of pairs (*r*, *n*) that defines static minimum

Table 2 – (*continued*)

Model	Element	Definition	Description
			cardinality for roles, where for each (r, n) at least n agents should be assigned to the role r.
	SMax – Cardinality	$\subseteq R \times N$	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: $(r, n) \in SMaxCardinality \Rightarrow \|\{a \in A\| r \in authorized\_roles(a)\}\| \leq n$
	DMin – Cardinality	$\subseteq R \times N$	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.
	DMax – Cardinality	$\subseteq R \times N$	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: $(r, n) \in DMaxCardinality \Rightarrow \|\{s \in S\| r \in SessionRoles(s)\}\| \leq n$.
RiBAC-CH	DSoD	$\subseteq 2^R \times N$	A collection of pairs (rs, n) that defines dynamic SoDs in presence of hybrid hierarchy, where for each (rs, n) no agent can activate or use permissions of n or more roles from the set rs together in one session. Formally: $(rs, n) \in DSoD \Rightarrow \nexists s \in S, \|\{r\| r' \geq_I r, r' \in rs, r' \in SessionRoles(s)\}\| \geq n$.
	DMin – Cardinality	$\subseteq R \times N$	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: $(r, n) \in DMinCardinality \Rightarrow \|\{s \in S\| r' \geq_I r, r' \in SessionRoles(s)\}\| \geq n$.
	DMax – Cardinality	$\subseteq R \times N$	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: $(r, n) \in DMaxCardinality \Rightarrow \|\{s \in S \|r' \geq_I r, r' \in SessionRoles(s)\}\| \leq n$.

For instance, assume that role r_1 has a dynamic maximum cardinality constraint of 3, and that role r_2 is senior to r_1 ($r_2 \geq r_1$). In such a configuration, if more than 3 agents activate 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, which makes the earlier interpretation too rigid. In order to provide more flexibility and to truly capture the behavior of constraints in the presence of role hierarchy, we adopt the notion of Hybrid Hierarchy (*HH*) that is originally defined in the context of Generalized Temporal RBAC (*GTRBAC*) (Joshi et al., 2005). In comparison with the standard RBAC hierarchy, hybrid hierarchy differentiates between permission usage and role activation semantics in a hierarchy, by taking into account three possible relationships: 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 an agent 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 ($r_1 \geq_{IA} r_2$). By leveraging the activation and permission inheritance relationships, we achieve more flexibility in policy specification. For instance, to resolve the problem in the aforementioned example, we can specify r_2 as an *A-senior* to r_1. Therefore, whenever an agent activates r_2, the cardinality constraint is respected, and an agent can also activate the role r_1 whenever needed but according to the cardinality constraint. In Appendix A-2(c), we show the specification of RiBAC-CH.

4. Community based RiBAC (CRiBAC)

As mentioned in Section 2.2, it is very important to address cooperation among agents beyond interaction between them. To address the requirements, we propose CRiBAC by extending RiBAC with the concept of community. In this section, we present its formal definition and two example scenarios.

4.1. *Formal definition*

CRiBAC aims to support cooperation among agents within a community, as well as decentralized interactions between agents who do or do not participate in a community. In CRiBAC, cooperation is required to achieve a community's goal and is regarded as a set of interactions among community members based on the roles that they assume. CRiBAC extends RiBAC to provide support for handling secure cooperation among agents using the various information and requirements related to individual communities. To do so, CRiBAC adopts the cooperation mechanism of CCM such that community creation by recruiting suitable agents is based on their context and capabilities. Fig. 3 provides an overview of the CRiBAC model and its formal definition is presented in Table 3.

In CRiBAC, a society consists of communities and agents. All agents and their communities cooperating in a society should be registered with their society. By registering in a society, each agent is assigned to one or more society roles (*SR*) depending on the agents' contexts and tasks. All *SR*s are defined by administrators at the time that the society is

Table 3 – Formal definition of the CRiBAC model.

Element	Definition	Description
RSC		The set of resources that agents own.
CONT	$CONT_s \cup CONT_c \cup CONT_a$	The set of all possible contexts which represent a society, communities, or agents.
TSK		The set of all tasks which agents can have. Some tasks need one or more permission(s) and such a relationship between an agent's task and permission is represented by *TSKPRM*.
a	$<RSC_a, CONT_a, TSK_a>$	An agent *a* in *A* has own resources ($RSC_a \subset RSC$), contexts ($CONT_a \subset CONT$), and tasks ($TSK_a \subset TSK$). Formally: $a = <RSC_a, CONT_a, TSK_a>$. *A* is the set of all agents in a society
SR		The set of all society roles
CR		The set of all community roles.
R	$SR \cup CR$	The set of all roles available in a society.
c	$<g_c, CR_c, A_c, CONT_c>$	A community has its goals ($g_c \in G$), a set of community roles ($CR_c \subset CR$), a set of agents ($A_c \subset A$), and own contexts ($CONT_c \subset CONT$). Formally: $c = <g_c, CR_c, A_c, CONT_c>$. *C* is the set of all possible communities in a society.
OBJ	$OBJ_s \cup$ OBJ-RSC $\cup$ OBJ-ROLE $\cup$ OBJ-TSK	The set of all target objects of *OPS* and it can be a social object, agents' resources, role, or role's task.
OBJ-RSC	OBJ-RSC $\subseteq$ RSC	The set of resources that can be an object of a role's operation.
OBJ-ROLE	OBJ-ROLE $\subseteq$ R	The set of roles that can be an object of a role's operation.
OBJ-TSK	OBJ-TSK $\subseteq$ TSK	The set of tasks that can be a target object of a task-oriented permission.
OPS		The set of all applicable operations on OBJ, where *OPS* $\subseteq$ *TSK*.
S	$<C, A, OBJ_s, CONT_s>$	A society *s* represents an entire cooperative system and it has a set of communities (*C*), a set of agents (*A*), a set of society objects (OBJ_s), and a set of society contexts ($CONT_s$). $OBJ_s \in OBJ$ is a set of objects which belong to a society and the society controls accesses from agents to society objects by *PA* to OBJ_s. $CONT_s \in CONT$ is a set of context information representing a society and it is available to any agents in a society.
OPRMS	OPS $\times$ OBJ	The set of all object-oriented permissions.
SPRMS	OPS $\times$ RSC	The set of all resource-oriented permissions.
RPRMS	OPS $\times$ OBJ-ROLE	The set of all role-oriented permissions, where *OBJ-ROLE* $\subseteq$ *R*.
TPRMS	OPS $\times$ OBJ-TSK	The set of all task-oriented permissions. Through the corresponding *TSKPPRM* relationship, it can be redefined as *TPRMS* $\subseteq$ *OPS* $\times$ *OBJ-ROLE.OBJ-PRMS* where *(obj-role, obj-prms)* $\subseteq$ *PA* and *(obj-tsk, obj-prms)* $\subseteq$ *TSKPRM*.
TSKPRM	TSK $\times$ *P*	A many-to-many task to permission relationship. This relationship represents a necessary permission(s) to perform a task.
P	OPRMS $\cup$ RPRMS $\cup$ TPRMS	A set of permissions in a society.
PA	{OPA $\cup$ SPA $\cup$ RPA $\cup$ TPA} $\subseteq$ R $\times$ *P*	A many-to-many role to permission assignment relationship, where *OPA* $\subseteq$ *R* $\times$ *OPRMS, SPA* $\subseteq$ *R* $\times$ *SPRMS, RPA* $\subseteq$ *R* $\times$ *RPRMS, and TPA* $\subseteq$ *R* $\times$ *TPRMS*.
AA	SRA $\cup$ CRA	A many-to-many agent to *R* assignment relationship, where *SRA* $\subseteq$ *A* $\times$ *SR* and *CRA* $\subseteq$ *A* $\times$ *CR*
SS		The set of all sessions created for agents in a society.

established. A society has society objects (OBJ_s), similar to objects in traditional RiBAC, and society contexts ($CONT_s$) that represent information relate to a society. An agent needs to register with the society to gain access to a society's contexts or objects, or interact with other agents in the society. However, any society role assigned to an agent is revoked when the agent leaves the society.

In CRiBAC, an agent can participate in one or more communities to help achieve the communities' goals by taking roles from *CR*, for as long as it belongs to the corresponding society. A community *c* consists of the community roles (CR_c), agents playing community roles (A_c), and the community context ($CONT_c$). $CONT_c$ represents information related to community *c* such as the community's type, the creation time, the number of members, and so on. In the XML-based specification shown in the Appendix B, a community type defines its goal and necessary roles (*CR*). The constraints on corresponding *CR* assignments (*CRA*) are described in a community type definition. According to those constraints, a community invites suitable agents for each *CR* to accomplish its goal. The community then selects the most appropriate agents based on contexts and tasks of agents. After receiving an invitation from a community, every agent must decide whether or not it wants to participate. This decision is made based on an agent's ability and its preference. The *CR*s are revoked from agents when the community is terminated or an agent leaves the community. When an administrator assigns roles to agents, he/she should consider the relationships between the agent's tasks and permissions. In order to perform some agent's tasks, particular permissions may be required. Accordingly, an administrator should be careful to only assign the necessary permissions to the agents' tasks when he/she administers the role assignments. Note that CRiBAC does not allow changes in the assignment relationships of community roles after a community's cooperation has been initiated.

An agent has its own resources, contexts, and tasks that it can perform. An agent's resources are a set of objects which belong to the agent and it is controlled by that agent. An agent also has its own contexts that capture specific information such as status and identification, and its tasks to show what kinds of work it can do. As we mentioned previously, an agent can play one or more SRs and CRs. If a role is assigned to an agent based on the agent's tasks, then the assigned permissions to the role should include the permissions necessary to perform the agent's tasks.

Permissions in CRiBAC are of two types: *traditional OPRMS* and *interaction permissions*. The *interaction* permissions include the resource-oriented permission (*SPRMS*), *RPRMS*, and *TPRMS*, as shown in Fig. 4. A *sprms* is a permission that allows access to agents' resources; the remaining two interaction permissions are the same as those in RiBAC. Agents can interact with each other by having interaction permissions. Actually, some permissions are parameterized based on roles, not specific objects. In such a case, the parameterized permissions should be interpreted with real agents who are assigned to the corresponding role, after the role assignment. Additionally, in this model, we assume that cooperating agents share their information such as their tasks, contexts, and permissions during interactions by using a centralized server which belongs to a community. Similar to RiBAC family, CRiBAC has four models: CRiBAC-B, CRiBAC-H, CRiBAC-C, and CRiBAC-CH. In this paper, we present the specification of CRiBAC-CH in the Appendix B.

4.2. *Example scenarios*

To emphasize the necessity for CRiBAC, we present two example scenarios calling for urgent cooperation: The University of Pittsburgh Medical Center (UPMC) and Disaster Relief.

4.2.1. *UPMC scenario*

We assume that there is a large-scale CCS for UPMC and that all the doctors and nurses who work for UPMC are categorized into several groups according to their specialties as shown in Fig. 5. We assume that all the employees have their own personal devices where an agent is hosted for supporting information sharing and cooperation among them.

Let's assume that there are five communities: BS, LS, A, N, BW and many agents including patients, caretakers, pharmacists, patrolmen, etc., in UPMC. At the time ti-1, some agents can interact with others. For example, Kevin, an examiner of bacteriological weapons (*BWE*), can culture bacteria and then preserve or kill the cultured bacteria. If *Bill* requests the cultivation or elimination of a type of bacteria, then *Kevin* can perform the requested task. However, he has to ignore requests to do this if the requests come from patients or patrolmen. In order to do so, we should allow only authorized agents to have access to an agent's tasks such as bacteria cultivation. Further, we need to ensure that only authorized agents can perform tasks on other agents. For example, a_4 is a patient who had brain surgery and *Bill* and *Jane* are in charge of his care. *Jane* is permitted to give an injection to a_4 and *Bill* is allowed to prescribe medicine for a_4. However, other agents must be prohibited from doing these tasks even if they are medical doctors or nurses. Note that it is necessary to guarantee the secure interactions among agents but existing models do not deal with interactions among roles. They focus on protecting only resources from unauthorized agents, but do not bar access to the agents' tasks or agents themselves.

To ensure security during cooperation in CCSs, we need to consider the community-related aspects as well. Consider an example scenario described as follows. The Emergency Brain and Lung Surgery community (*EBLS*) for *Bob*, who is the chief lung surgeon, is created as soon as *Bob* is injured. To organize the community, an administrator tries to employ three chief medical doctors for the Emergency Brain Surgeon (*EBS*) role, the Emergency Lung Surgeon (*ELS*) role, and the Emergency Anesthetist (*EA*) role, respectively. At this time, *Bill* and *Bob* are not available to play the emergency surgeon roles, *EBS* and *ELS*, in that community. *Bill* is the chief brain surgeon but he is participating in, say, the

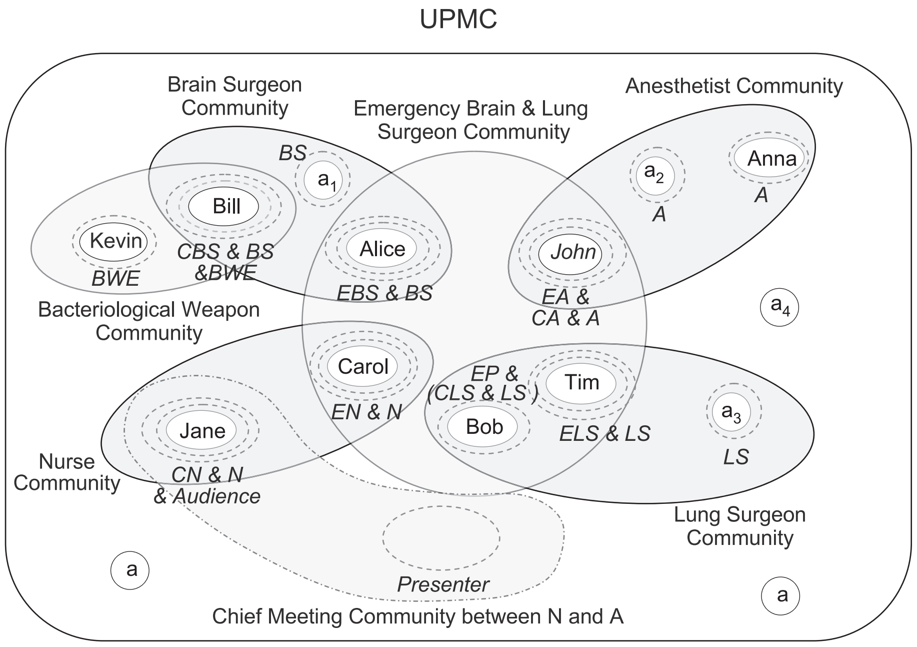


Fig. 5 – The UPMC Example of CRiBAC.

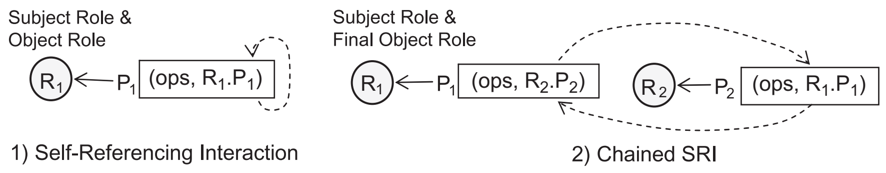


Fig. 6 – Examples of SRI problems.

bacteriological weapons community as a Bacteriological Weapon Examiner (*BWE*) at that time. Because of the threat of infection, people who are involved in the Bacteriological Weapon (*BW*) community should be prohibited from working as a surgeon; therefore, *Alice*, who is available, assumes the *EBS* role in the *EBLS* community instead of *Bill*. For the *ELS* role, *Tim* is chosen since *Bob*, who was a chief lung surgeon, is a patient. As can be seen in this example, information related to the community – such as existing assignments, relationships between agents and communities or community roles – is critical when trying to assign agents to create a new community. When an agent receives a request to participate in a community, that agent might also need information such as the critical importance of the proposed community, the suggested community role to be assumed, and the identification of the other participants. Knowing such information helps an agent to decide whether he/she wants to accept the request or not. For instance, *John* receives a participation request as a presenter from the chief meeting community of a chief nurse and a chief anesthetist (*CMNA*) after the creation of *EBLS* community. However, he declines the request since he already belongs to the *EBLS* community whose priority is higher than the priority of the *CMNA* community.

The community information is also important when an administrator controls access to resources or an agent decides whether or not he/she will allow an interaction access from others. For example, an administrator can allow only agents in the *EBLS* community to access *Bob*'s medical information. Accordingly, *Bob* can reject a request for an interaction from *Anna*, an anesthetist, since she does not belong to the same community. As can be seen, in many cases, the consideration and explicit specification about interaction and cooperation among agents are necessary for efficient access control in cooperative MASs. The CRiBAC specification of *UPMC* example is presented in the Appendix C-1.

4.2.2. *Disaster relief scenario*

This growth in online SNSs brings many changes in patterns of communication and human behavior. Cooperation among large number of globally connected people through online SNSs provides an unprecedented opportunity for significant social transformations. SNSs provide a huge pool of manpower and quick delivery of information, thereby, allowing a solid basis for immediate cooperation among users. For example, many people are using Twitter to broadcast information about their lost pets and ask for help finding them, such as *Lost Dog Found* and *Fidofinder*. Another example where the use online SNS has resulted in immediate cooperation among people relates to Healthcare domain (Emory Healthcare, 2011): a medical doctor who works for Emory Healthcare received a tweet from a man, Matthew, about an emergency situation involving his grandmother. A medical team communicated with Matthew via Twitter to instruct him on life-saving emergency first aid while emergency transportation was arranged. The doctor said, “Without the quickness of social media, the helicopter may have never been dispatched”. As can be seen from the aforementioned examples, online SNSs have an enormous potential for helping people by supporting dynamic and immediate cooperation among users.

Although these examples show significant promise for social cooperation in SNSs, there exist no cooperation models for SNSs that ensures effective and secure cooperation. To date, cooperation among users has been achieved only in an ad hoc manner. Meaningful cooperation cannot be guaranteed by such an approach. In addition, security issues must be considered. Allowing access to individuals' information and resources to the public during cooperation may raise serious privacy and access control problems. By using CRiBAC, we are able to resolve the aforementioned problems on existing cooperation on online SNSs. CRiBAC allows to specify cooperation among users and security policies necessary for protecting users' information and resources during cooperation. For better understanding of the usefulness of CRiBAC, we present an example of cooperation among users of an online SNS, in particular Facebook, for disaster relief as follows.

Let's assume that a man is injured. He posts on an online SNS using his mobile phone to ask for help. Many people who see the post spread it to let more people know his

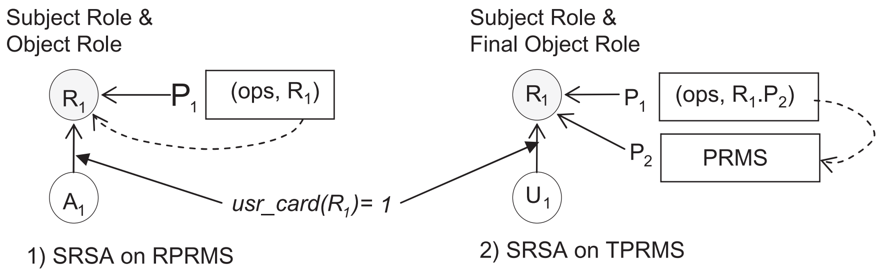


Fig. 7 – Examples of SRSA problems.

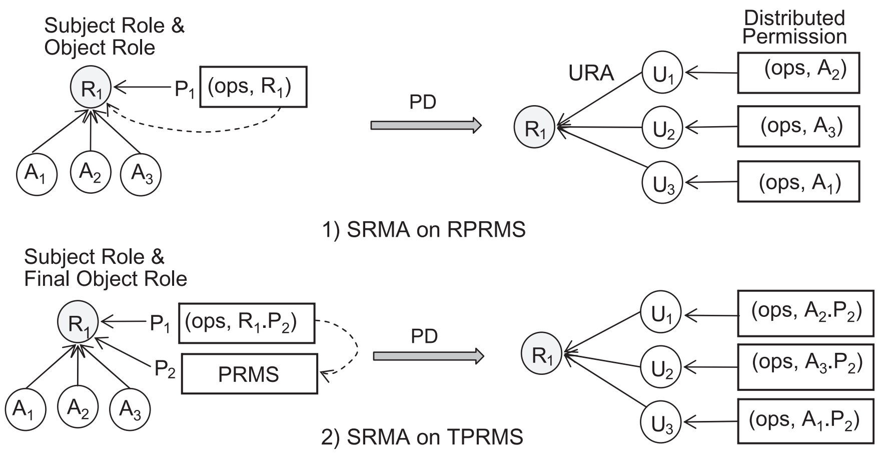
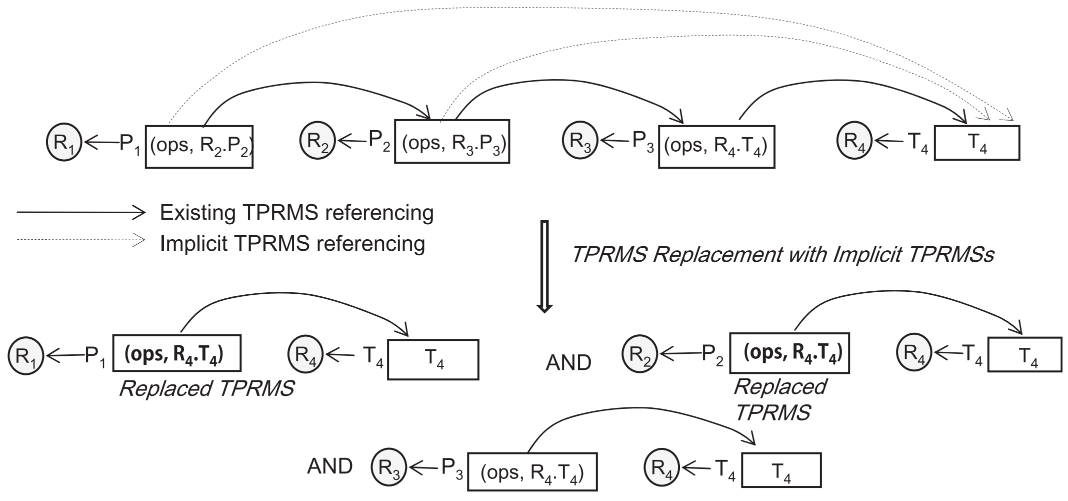
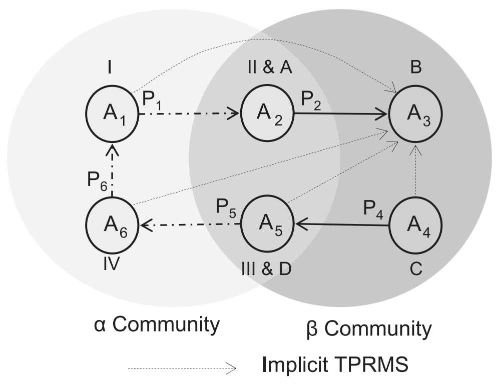


Fig. 8 – Examples of SRMA problems.

urgent situation and/or go out to save him. Shortly afterward, other victims, including a seriously injured woman, ask for help simultaneously in many different places. To successfully achieve the goal of this cooperation, the most suitable cooperative services are offered to every victim as soon as possible. In order to do so, it is important to organize a cooperative group of volunteer users who are close to a victim and are capable of giving necessary aid including


R1
P1
(ops, R2.P2)
R2
P2
(ops, R3.P3)
R3
P3
(ops, R4.T4)
R4
T4
Existing TPRMS referencing
Implicit TPRMS referencing
TPRMS Replacement with Implicit TPRMSs
Replaced TPRMS
AND
Replaced TPRMS
AND

1) An example of implicit TPRMS Problem in a TPRMS chain and TPRMS Replacement



2) An example of implicit TPRMS Problem across two communities

Fig. 9 – Examples of extracted implicit TPRMSs and corresponding TPRMS replacement.

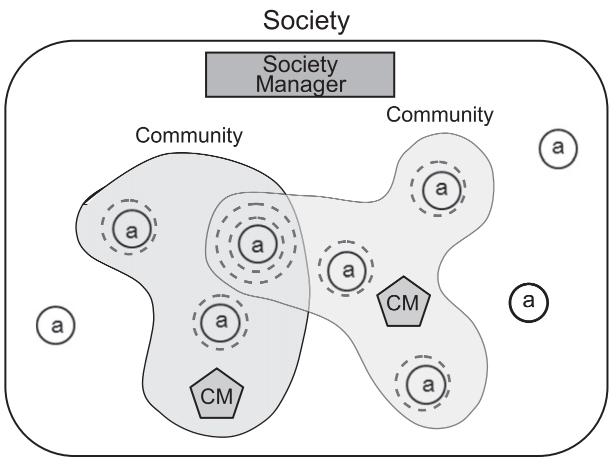


Fig. 10 – Administration of a CCS by SM and CM.

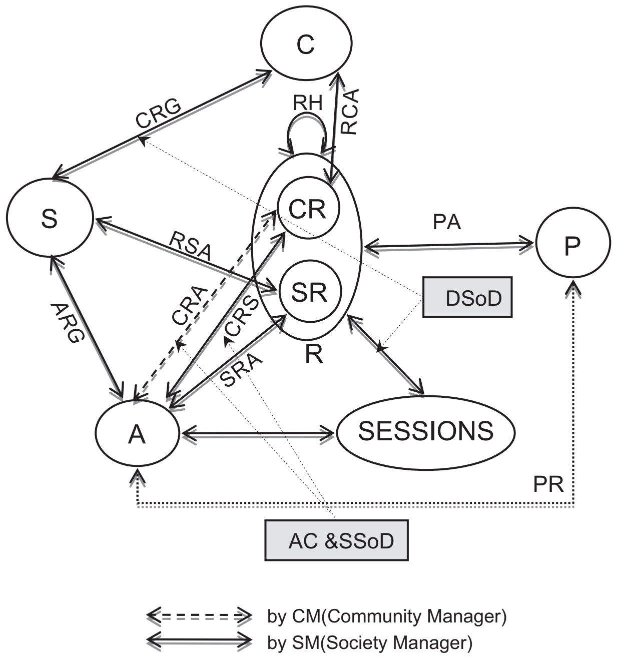


Fig. 11 – The administration model for CRiBAC (ACRiBAC).

specific medical aid. If volunteers flock to a few victims who have posted earlier, others' chances of being rescued become less likely. Even though a victim has many helpers, the lack of vital aid may lead to an overall failure in cooperation. For efficient and secure disaster relief, we can specify a cooperative SNS by employing CRiBAC as shown in the Appendix C-2.

5. Analysis on interaction permissions

In the literature, potential problems related to permissions such as concurrent execution of conflicting operations have been studied and some of them have been solved by constraints such as history-based constraints, SSoD or DSoD. We can apply constraints existing in RBAC to CRiBAC since CRiBAC includes all the properties of RBAC. However, the additional interaction permissions that CRiBAC includes have not been analyzed in the literature; thus, we will analyze the problems related to interaction permissions and provide solutions in this section.

Constrained CRiBAC supports the SoD constraints including *SSoD* and *DSoD*. The *SoD* constraints are the same as those in RiBAC-C. We refine the cardinality constraints of RiBAC-C into the Agent Cardinality (*AC*) constraint as shown in DEF 1. It aims to limit the number of agents that can be assigned to a certain role, and it is applied to the Agent-Role Assignment (*AA*), in the same way as the *SoD* constraint (For more details, see Fig. 11).

DEF 1. Agent cardinality (AC) constraint for a role.

There are several potential problems that may result from the interaction permissions or relationships among them. These include Self-Referencing Interaction (*SRI*), Single-Role and Single-Agent Interaction (SRSA), Single-Role and Multi-Agent Interaction (*SRMA*), and *Implicit TPRMS* Problem. The formal definitions of each problem are shown in DEF 2.

- *Self-Referencing Interaction (SRI)* – This problem is that a *tprms* recursively invokes itself. It happens when a *tprms* and its object permission (*obj-prms*) are actually the same. If the *obj-prms* of a *tprms* invokes to another *tprm*, a referencing chain is formed. If the end of the chained referencing refers to its beginning, the original *tprms*, is a *chained SRI*. If an agent has an interaction permission experiencing the *SRI* problem, it recursively carries out referencing. In Fig. 6, we can see two examples of the *SRI* problem. The first one is referred to as the *SRI* problem where a *tprms* invokes itself as its object permission. The latter is an example of the *chained SRI* problem. If an *SRI* problem is discovered, then the corresponding *PA* relationship(s) has to be modified.
- *Single-Role and Single-Agent Interaction (SRSA)* – The SRSA problem is that only one agent is assigned to both a subject role and an object role. It occurs when an agent is assigned to a subject role. Here, if the subject role refers to itself as its object role or its *TPRMS* eventually invokes itself as its object permission, then the *SRSA* problem occurs. The formal definition of the *SRSA* problem is as follows. As shown in

It restricts the number of agents assigned to a certain role at the time of agent-role assignment (*AA*), a union of *CRA* and *SRA*.
Cardinality can be expressed as an exact number or a range of numbers including a minimum and a maximum.
Formally: *agt_card(r:R)=cardinality, where* $1 \le cardinality \in I$ *or cardinality=(min,max),* $1 \le$ *min and max* $\in I$
For examples, *agt_card(emergency_surgeon_leader)*=1, *agt_card(emergency_brain_surgeon)*= (1,3)

Fig. 7, this problem leads to the abnormal situation where an agent interacts with itself; in truth, this is really not an interaction. This situation is not different from that of an agent who plays a subject role in executing its own tasks. This problem might not directly relate to serious security fraud or system error, but it is logically incorrect. To resolve it, an administrator should check the role-permission assignments (*PA*) and agent cardinality (*AC*) constraints. If an *SRSA* is found, the administrator should modify the corresponding policies.

- *Single-Role and Multi-Agent Interaction Problem (SRMA)* Problem – The *SRMA* problem is similar to *SRSA*, but in this problem, two or more agents are assigned to a subject role and an object role, simultaneously. Consider Fig. 8(1), which shows the *SRMA* problem on an *rprms*. R_1 has $P_1 \in RPRMS$, and P_1's object role is R_1. It looks like a *SRSR* problem, but R_1 has three agents. Fig. 8(2) presents the *SRMA* problem in *a tprms*. R_1 is a subject role and becomes its final object role which has the final object permission of P_1, P_2. CRiBAC interprets this problem as a situation where all of the agents who are assigned to a subject role have a corresponding interaction permission to interact with each other. In order to solve this problem, CRiBAC distributes an interaction permission having a *SRMA* problem to all of the agents who are assigned to the subject role of the interaction permission. We call it the Permission Distribution (*PD*). To better understand, we present the example solutions by *PD* of *SRMA* in Fig. 8. For the *SRMA* problem on *RPRMS*, an *rprms* $P_1 \in RPRMS$ is distributed to each agent as shown in Fig. 8(1). It means that all of the agents who are assigned to a subject role and an object role have P_1 to perform their operation *ops* on another agent who takes R_1. In case of the *SRMA* problem on a *tprms*, all agents have $P_1 \in TPRMS$ to invoke a task by other agents who play role R_1 as presented in Fig. 8(2). By *PD*, each agent who is assigned to R_1 can interact with other.
- *Implicit Tprms* Problem – If a *tprms* invokes other roles' tasks in a sequence, we can determine a new *tprms*. We refer to such a *tprms* obtained from a chain of *tprms*s as *implicit tprms*. If an *implicit tprms* is found, then it should be replaced with an existing *tprms*. We call it the TPRMS Replacement (*TR*). To guarantee the security of a system, an administrator should extract all *implicit tprms*s. If an agent participates in multiple communities, a chain of *tprms*s might be spread over several communities. In this case, an administrator should check all the *implicit tprms*s across communities. Fig. 10 indicates two examples of the *implicit tprms*. In Fig. 9(1), R_1 has $P_1 \in TPRMS$ but P_1 invokes another *tprms* P_2. Also, P_2 invokes $P_3 \in TPRMS$ and P_3 invoke P_4. It means that P_1, P_2, and P_3 eventually invoke P_4. We refer to this relationship among such connected *tprms*s – which are invoked one after another – as a *tprms chain*. In the example, four *tprms*s; P_1, P_2, P_3, P_4, form a *tprms chain* and P_1 and P_2 are replaced by an *implicit tprms* = (ops, $R_4.P_4$). A *tprms chain* can be made across communities as shown in Fig. 9(2). The α community has four members; $A_1(R_I)$, $A_2(R_{II})$, $A_5(R_{III})$, $A_6(R_V)$, and the β community includes $A_2(R_A)$, $A_3(R_B)$, $A_4(R_C)$, $A_5(R_D)$. As a member of α, A_1 has P_1 through the *PA* between *I* and P_1. Similarly, A_5 has P_5 and A_6 has P_6. In β, A_2 has P_2 and A_4 has P_4. As you can see in this example, although *tprms*s are assigned to different roles which are involved in different communities, they can form a *tprms chain*. We should extract all *implicit tprms*s across the communities, and then replace existing *tprms*s with *implicit tprms*s.

DEF 2. Problems on interaction permissions and corresponding solutions.

Problem	Object	Condition	Solution
SRI	TPRMS	If $tprm = (ops, obj\text{-}prms) \in TPRMS$ & $obj\text{-}prms \equiv tprm$ *where* $obj\text{-}prms \in TPRMS$	An administrator should modify the corresponding PA relationship(s).
SRSA	RPRMS	If $(sub\text{-}role, rprms) \in PA$ & $sub\text{-}role \equiv obj\text{-}role$ & $agt\_card(sub\text{-}role) = 1$ *where* $sub\text{-}role, obj\text{-}role \in R$ & $agt\_card(sub\_role) \in AC$	An administrator should modify the corresponding role-permission assignments (PA) and/or agent cardinality (AC) constraints.
	TPRMS	If $(sub\text{-}role, tprms) \in PA$ & $(sub\text{-}role, tprms.final\text{-}obj\text{-}prms) \in PA$ & $sub\text{-}role \equiv final\text{-}obj\text{-}role$ & $agt\_card(sub\text{-}role) = 1$ *where* $sub\text{-}role, obj\text{-}role \in R$ & $final\text{-}obj\text{-}prms \in P$ & $agt\_card(sub\_role) \in AC$	
SRMA	RPRMS	If $(sub\text{-}role, rprms) \in PA$ & $sub\text{-}role \equiv obj\text{-}role$ & $2 \le agt\_card(sub\text{-}role)$ &$\{(ai, sub\text{-}role) \mid 1 \le i \le, n \ge 2\} \in ARA$, *where* $sub\text{-}role, obj\text{-}role \in R$ & $agt\_card\ (sub\text{-}role) \in AC$	Permission Distribution (PD): $\{assign(ai,(ops, ai+1)) \mid 1 \le i \le n\text{-}1\}$ & $assign(an, (ops, a1))$
	TPRMS	If $sub\text{-}role \equiv final\text{-}obj\text{-}role$ & $(sub\text{-}role, tprms) \in PA$ & $(sub\text{-}role, tprms.final\text{-}obj\text{-}prms) \in PA$ & $2 \le agt\_card(sub\text{-}role)$ & $\{(a_i, sub\text{-}role) \mid 1 \le i \le n, n \ge 2\} \in ARA$, *where* $sub\text{-}role, obj\text{-}role \in R$ & $agt\_card(sub\text{-}role) \in AC$	Permission Distribution (PD): $\{assign(ai,(ops, ai+1.tprms)) \mid 1 \le i \le n\text{-}1\}$ & $assign\ (an, (ops,a1.tprms))$
Implicit TPRMS	TPRMS	If $\{(ri, pj) \mid 1 \le i \le n, 1 \le j \le m, 3 \le n\} \in PA$ & $\{pj \mid 1 \le j \le nm, 3 \le n\} \in TPRMS$, $\{(pj.obj\text{-}prms) \mid 1 \le j \le nm\text{-}2, 3 \le n\} \in TPRMS$, *and* $pj.\ final\text{-}obj\text{-}prms \notin TPRMS$	TPRMS Replacement (TR): $\{revoke(ri, pj) \mid 1 \le i \le n, 3 \le n, 1 \le j \le nm\text{-}1\}$ & $\{assign(ri, pip) \mid 1 \le i \le n\text{-}1, 3 \le n\}$, *where* $pip = (ops, final\text{-}obj\text{-}prms) \in TPRMS$ *is an implicit TPRMS*

6. Administration of CRiBAC

In order to apply CRiBAC to large-scale cooperative systems, an administration model is necessary. Existing administration models for RBAC are relatively well-defined, but they are not able to manage the interaction permission and cooperation of CRiBAC since RiBAC does not deal with them. In this paper, we therefore propose an administration model for CRiBAC, called ACRiBAC.

Before we specify the administration model in detail, we first give an overview of the administration model of CCS since cooperation in CRiBAC is based on it. The administration model of community computing is somewhat decentralized, having two different types of administrators: Society Manager (*SM*) and Community Manager (*CM*). *SM* is the more important and only one *SM* manages a CCS. *SM* supervises system-wide matters such as maintaining agent information and interactions among agents, constructing new communities, creating community managers, and so on. For each community, one *CM* takes responsibility for community matters such as recruitment of community members, cooperation management, community termination, and so on. An example of a CCS maintained by one *SM* and several *CMs* is shown in Fig. 10.

6.1. *Administration model of CRiBAC (ACRiBAC)*

To align with the administration model of community computing, ACRiBAC employs both *SM* and *CM*. A *SM* takes responsibility for controlling *agent registration* in a society (*ARG*), *Community Registration* in a society (*CRG*), Society Role-to-Society Assignment/Revocation (*RSA*), Agent-to-Society Role Assignment/Revocation (*SRA*), Community Role-to-Community Assignment/Revocation (*RCA*), Agent-to-Community Role Selection (*CRS*), Role-to-Permission Assignment (*PA*), *CM* creation/deletion, and *RH* maintenance. On the other hand, a *CM* has responsibility for the creation and termination of a community, Agent to Community Role Assignment (*CRA*), and Permission Realization (*PR*). The overview of ACRiBAC is shown in Fig. 11 and its formal definition is presented in Table 4.

The detailed description of relationships between different modeling elements in ACRiBAC is as follows.

- *CRG* (Community to a Society Registration): All communities existing in a society should register their information such as their goals, cooperation processes, necessary roles, and assigned agents with the *SM* of the society at the onset. As you can see in the Appendix B, the community's goal, cooperation process, and necessary roles are specified for each community type in the *Community_Types* part of *Society_Contexts* (see Appendix B(i)), for an instance, see the specification of the *EBLS* community type (see Appendix C-1(i). In addition, all member agents that are actually assigned to a community are specified in the *CR_Assignment* part of a community's specification, for example, see *EBLS* community (see Appendix C-1).
- *ARG* (Agent Registration to a society): All agents that join a society should register their information such as their own resources, contexts, and tasks with the *SM*. Information about all the agents is specified in the *Agents* part of CRiBAC description (see Appendix B(c), for an example, you can see the agent specifications for *Bob*, *Bill*, and *Kevin* in the Appendix C-1(b). To protect the privacy of agents, only the *SM* can access all the information about all agents such as an agent's identification, contexts, tasks, society role(s), community role(s), and so on. However, an agent can access a society's objects if the corresponding SPRMS are granted. Since every agent in a society must have one or more society roles, *SRA* always follows *ARG* for each agent.
- RSA (SR to Society Assignment): The *SM* can assign society roles to a society to achieve the long-term and global goal of the society. All society roles are specified in the *Society_Roles* part of *Society_Contexts* as shown in the Appendix B(h). For examples, some society roles of *UPMC* are specified in the Appendix C-1(d).
- *SRA* (Agent to *SR* Assignment): Every agent should be assigned to one or more society role(s) based on their tasks

Table 4 – The administration model for CRiBAC (ACRiBAC).

	Relation	Definition	Description
SM	PA	$OPA \cup SPA \cup RPA \cup TPA \subseteq R \times P$	A many-to-many role to permission assignment relationship, where $OPA \subseteq R \times OPRMS$, $SPA \subseteq R \times SPRMS$, $RPA \subseteq R \times RPRMS$, and $TPA \subseteq R \times TPRMS$.
	AA	$SRA \cup CRA$	A many-to-many agent to R assignment relationship.
	ARG	$\subseteq A \times S$	A many-to-one agent to society registration.
	ARA	$SRA \cup CRA \subseteq A \times R$	A many-to-many agent to role assignment relationship.
	SRA	$\subseteq A \times SR$	A many-to-many agent to SR assignment relationship.
	ACA	$\subseteq A \times C$	A many-to-many agent who a candidate member to community assignment relationship.
	RSA	$\subseteq SR \times S$	A many-to-one society role to society assignment relationship.
	RCA	$\subseteq CR \times C$	A many-to-many community role to community assignment relationship.
CM	CRA	$\subseteq A \times CR$	A many-to-many agent to CR assignment relationship.
	CRG	$C \times S$	A many-to-one community to society registration.
	PR	$\subseteq parameterized\ P \times P$	A one-to-many parameterized permission to real permission realization.

and contexts when they register with a society. All of *SRAs* are specified in the *SR-Assignment* part of the *Society_-Contexts* (see Appendix B(h). For example, a *SRA* for the society role *D* is shown in the Appendix C-1(h). By *SRA* and corresponding *PA*, each agent has the necessary permission(s) to play his/her society role(s).

- *RCA* (*CR* to Community Assignment): The *SM* should assign necessary roles for each community type. The *RCAs* for a community type are specified in the *Community_Roles* part of a *Community_Type* description (see the Appendix B(i)). For examples, you can see the corresponding RCAs for the *BS* and the *BW* community types in the Appendix C-1(c2). *RCA* should be executed before the *SM* enforces *CRS* for the community.
- *CRS* (Agent to *CR* Selection): *CRS* is performed as a basis for the agent to *CR* assignment (*ACA*). That is, *ACA* is performed by *RCA* and *CRS*. Before the assignment of agents to community roles (*CRA*), the *SM* selects candidate agents for every community role based on the agents' information, which is stored in *SM*, such as their contexts, tasks, society role(s), or community role(s) at the time. For privacy protection of agents, only the *SM* can access information about all agents and select proper candidates.
- *CRA* (Agent to *CR* Assignment): A *CM* assigns the most suitable agent(s) among candidate agents selected by *CRS* to each community role, according to the criteria for each assignment, which is specified in the *Community_Type* description. A *CRA* for each *CR* is specified in the *CR_Assignment* part of a *Community* description (see Appendix B(c)). For example, a *CRA* for the *EBS* community role is specified as shown in the Appendix C-1(c1).
- *PA* (Role to Permission Assignment): It is same as *PA* in NIST RBAC, and is enforced by *SM*. All *PAs* are specified in the *Permission_Assignments* description (see Appendix B(f)). For example, we present the part of *PAs* for a UPMC example in Appendix C-1(f).
- *PR* (Permission Realization): After a *CM* performs *CRA*, it interprets all the role-based parameterized permissions assigned to all the community roles and then realizes those permissions with real agents assigned to each community role.

A *SM* deals with *ARG*, *RSA*, *SRA*, *RCA*, *CRS*, and *PA*. A *CM* handles *CRA* and *CRG* for each community. To preserve the least privilege rule, unnecessary agents or roles should be removed or revoked.

6.2. *Administration of cooperation*

In this section, we show how agents' cooperation is supported by the ACRiBAC model. To maintain society-level matters, an SM executes several functions such as agent registration, community registration, establishing a community manager, *RSA*, *SRA*, *RCA*, *CRS*, and *PA*. In this section, we present the algorithms for society administration; *AReg*, *UnAReg*, *CMCreate*, and *CMTerminate*.

When a *SM* receives a request for registration within a society from an outside agent, it executes the *AReg* algorithm. In *AReg*, the *SM* assigns an agent to a suitable society role(s) depending on the agent's tasks and contexts and then saves information about the agent. If the agent leaves the society, the *SM* executes *unAReg* to revoke all permissions associated with agent's society role(s). The *CMCreate* algorithm is executed to generate a community manager agent when a *SM* receives a request to create a community from an agent who recognizes a need for a community. To complete a community manager agent (*CM*), the *SM* performs *RCA* according to the community's goal, and then selects candidates for each community role. After a community is dissolved, the *SM* deletes its *CM*. We describe those algorithms in Table 5.

To create a community, the *CM* created by the *SM* performs the *CommCreate* algorithm. In this algorithm, the *CM* assigns the most suitable agent(s) to each community role and then induces the agent's permissions with the assigned role members. Once a community achieves its goal or reaches a failure state, its *CM* terminates it by using algorithm *CommTerminate*. These two algorithms are presented in Table 6.

7. Implementation of CRiBAC prototype System

In this section, we describe our prototype system which was developed in order to test a variety of policies based on the proposed CRiBAC model. This system tests not only access control policies but also the process of policy enforcement in a CCS, and then simulates cooperation among agents to ensure enforcement of the specified policies. It is developed on the JDK 1.6 platform and also uses the JADE platform for the simulation of the CCS.

Table 5 – Algorithms for society administration.

```
AReg (agt.id) {
  agt _verification(agt.id);
  If (agt _verification) {
    request(agt.id, {agt.cont}, {agt.task});
    If (response(agt.id)) {
      assign_SRA(agent.id);
      SM.AgentTb ← InsertRow(agt.id,{agt.cont},{agt.task},{agt.sr});
      inform(agt.id, agt.sr);
    Else FAIL
  Else  FAIL  }
```

```
UnAReg(agt.id) {
  revoke_SRA(agt.id);
  SM.agentTb ← DeleteRow(agt.id); }
```

```
CMCreate(comm.goal) {
  cm.id ← CreateCMagent(comm.goal);
  cm.cr ← assign_RCA(comm.goal);
  For each cr, cr.candidates ← select_CRS(cm.cr); }
```

```
CMTerminate(cm.id) {
    DeleteCMagent(cm.id); }
```

Table 6 – Algorithms for community administration.

CommCreate (cm.id, cm.goal, {cm.cr}, {cr.candidates},{cr.criteria})
{ For each cr
    comm.agt ← *assign_CRA({cr.candidates}, cr.criteria);*
  For each comm.agt
    agt.effetprms← *PR({cr.prms});* }
	CommTerminate (cm.id) {
	  revoke_CRA({cr});
	  For each comm.agt
	    revoke_effetprms({agt.effetprms});
	}

7.1. *Functionalities*

1) Developing CCSs – To test its policies, we first need an operating CCS. To build CCSs on the developed prototype system, an administrator creates all the elements such as a society, communities, and members (agents), and describes cooperation among members in a community. At this time, we simulate the behavior of agents instead of actually implementing the behavioral actions of agents as our ultimate goal is to test CRiBAC policies rather than to test a CCS. After developing a CCS, a tester can see how the system operates through the user interface of our prototype system. Since the developed CCS is connected to the CRiBAC engine, all agents in CCS can acquire proper permissions to access resources or other agents through this engine for interacting and cooperating with agents.
2) CRiBAC Administration – By using the developed prototype, developers can easily design CRiBAC policies and test them by simulating a corresponding CCS. More specific functions for access control and object management are described as follows.

- *Role Management* – CRiBAC can manage a role set by creating, modifying, and deleting roles. For each role, all related information are defined including access control policies as well as the role's capabilities, contexts, objects, and cooperation.
- *Permission Management* – It can manage a set of permissions by creating, modifying, and deleting permissions. The permission set includes three types of permissions which are defined in CRiBAC: object-oriented (*OPRMS*), role-oriented (*RPRMS*), and task-oriented (*TPRMS*) permissions.
- *Agent Management* – It can manage a group of agents by creating, updating, and deleting agents. Agents in CRiBAC are identical to members in a CCS; therefore, every change in members in a CCS is reflected to agents in CRiBAC simultaneously.
- *Session Management* – It manages all sessions which are assigned to agents who play a certain role in a society.
- *Object Management* – In CRiBAC, the set of objects includes society objects and agent's objects; both can be a target of an access. Accordingly, it should be managed by two types of objects separately.

3) Policy Definition and Enforcement – A tester can define CRiBAC policies by utilizing a user interface and file operation of the prototype system, including assignment relationships such as the agent-role assignment (AA) and the role-permission assignment (PA). According to these policies, each access will be permitted or denied in a CCS.
4) Policy Analysis – The prototype system provides an analysis of the specified policies before they are applied to an actual system. Through this analysis, we can identify conflicts between policies or problems which can cause a serious security fault such as the self-referencing and chained referencing problem. By testing policies before deploying them, a developer can reduce the cost of development and maintenance.
5) Policy Enforcement and System Monitoring – The prototype system can enforce CRiBAC policies while a community computing system is operating. During the operation, a developer can monitor all events in a CCS through the agent interface such as community creation, member registration, and policy enforcement.

7.2. *System architecture*

For better understanding of our prototype system, we present its architecture in Fig. 12. The prototype system consists of three parts: user interface; CCS Framework including community computing system, CRiBAC engine, and knowledge-base; and JADE Agent platform. The detailed explanation about each element is as follows.

7.2.1. *CCS framework*

The CCS Framework consists of a CCS, CRiBAC Engine, and Knowledge-Base. An administrator can manage CRiBAC policies and Knowledge-Base as well as CCS through the user interface. It can also simulate a CCS on the Jade Agent Platform at runtime.

- CCS – It manages communities and agents in a CCS and maintains information related to Community Computing. The Role Repository stores information about all roles participating in the cooperation process as well as access control policies. We note that CCS and CRiBAC engines share this role repository. It means that the definition of roles is used for cooperation as well as access control.
- CRiBAC Engine – It aims to control all types of accesses according to the defined policies, object-oriented accesses as well as interaction accesses. A system administrator defines CRiBAC policies through the Policy Authority Point (*PAP*) Module, and those policies are stored in the *Policy Repository*. If an agent needs to get permission, it should send a request for the access to the Policy Enforcement Point (*PEP*). The *PEP* Module then delivers the agent's request to the Policy Decision Point (*PDP*) with information about the agent and its request. The *PDP* Module decides whether or not to grant the required permission. To make a decision, the *PDP* fetches the corresponding policies from *Policy Repository* and evaluates them by using the relevant context information retrieved from the *Context Server*. After reasoning about policies, *PDP* conveys the result to the *PEP*, and the *PEP* performs authorization by granting the requested permissions. The detailed CRiBAC Engine is shown in Fig. 13.

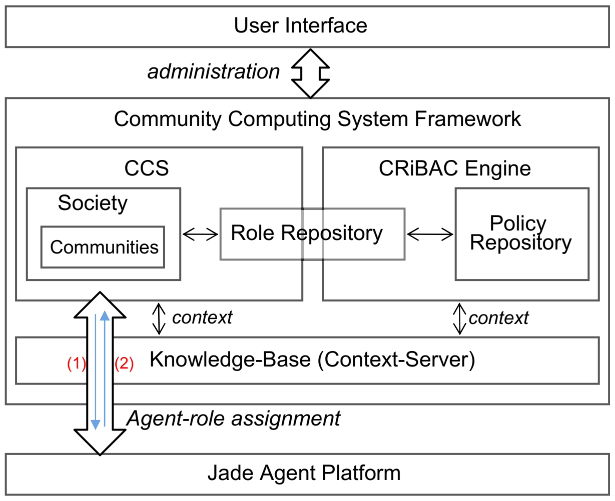


Fig. 12 – Architecture of CRiBAC prototype system.

- Knowledge-Base (Context Server) – It is in charge of maintaining and inferring context used for describing cooperation and defining and enforcing policies. In this module, context information is represented by a *key-value* model as follows: <*context-name, context-value*>. A variety of context information is required, so that CCS and the *CRiBAC engine* may perform many operations such as community creation, role assignment, permission assignment, and policy enforcement. The *Context Server* provides the necessary context information by gathering context information from the environment and agents and reasoning about those contexts.

7.2.2. *Jade agent platform*

To develop agents in a CCS, we have used the JADE agent platform. The JADE platform is one of the most popular agent platforms and is widely used for developing, simulating, and testing multi-agent systems. It follows FIPA standard and also supports diverse and unique features of the agents. In our previous work, we have developed a complete CCS based on the JADE platform. In this work, we simulate an agent's behaviors because the developed prototype system is more focused on testing access control policies than testing an agent's cooperation. That is, agents simply inform their actions or the action's results, but do not perform actual actions. In this prototype, a society is developed as a Jade agent container; each community and agent is implemented as a Jade agent. In addition, the administration of CCS through the user interface (such as community creation or agent registration) is simultaneously reflected in corresponding agent containers and agents in JADE platform.

7.2.3. *User interface (UI)*

The proposed prototype system has a variety of *UI* components to design and test CCSs and corresponding CRiBAC policies. The *UI* is roughly divided into four sections as shown in Fig. 14: Society tree, Permission Management Component, Assignment Management Component, and Display console.

- *Society tree* – presents a CCS as including a society, communities and agents. The society tree can be used for administrating entities in a CCS by adding, modifying, and removing elements of the society; such as communities and agents. At this time, communities and agents in the society tree are synchronized with the JADE agent GUI so that an administrator accomplishes tasks through the JADE GUI and resulting changes are reflected in the society tree.
- *Permission Management Component* – manages all permissions in a CCS Framework. Each of permissions is defined by a unique ID, permission type, subject role, and object role. Through the UI, an administrator can define all types of permissions that the CRiBAC model introduces.
- *Assignment Management Component* – an administrator can handle Agent-Role Assignments (*AA*) and Role-Permission assignments (*PA*). By performing *AA*, he can recruit suitable agents for each community role. In order to do so, the *AA* component automatically searches through all proper agents in a CCS to create easier and faster assignments. Moreover, the *PA* component rapidly finds candidate roles and a list of permissions; it helps an administrator to define conditions to get certain permission.
- *Display console* – shows all changes in a CCS. Therefore, an administrator can check the result of administrative behavior and also monitor the system. In order to avoid mistakes by the administrator, significant changes are presented in red or blue colored text. Through this console, all of the administrative behaviors, decisions on access requests, and cooperative processes among agents are shown. By monitoring the console, administrators can evaluate present policies and also make a plan for further securing the systems.

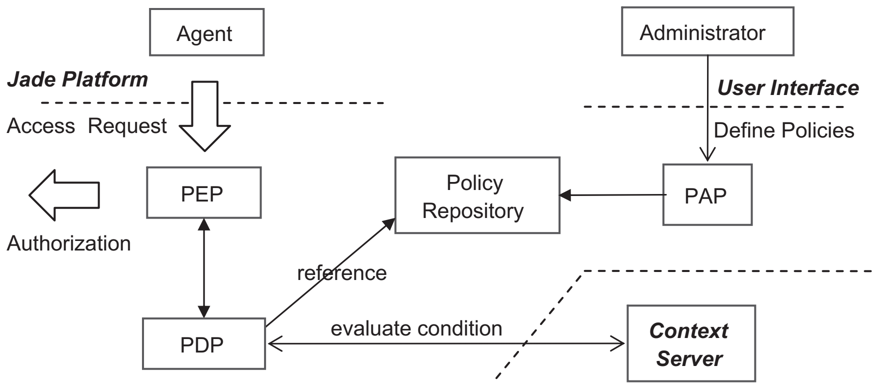
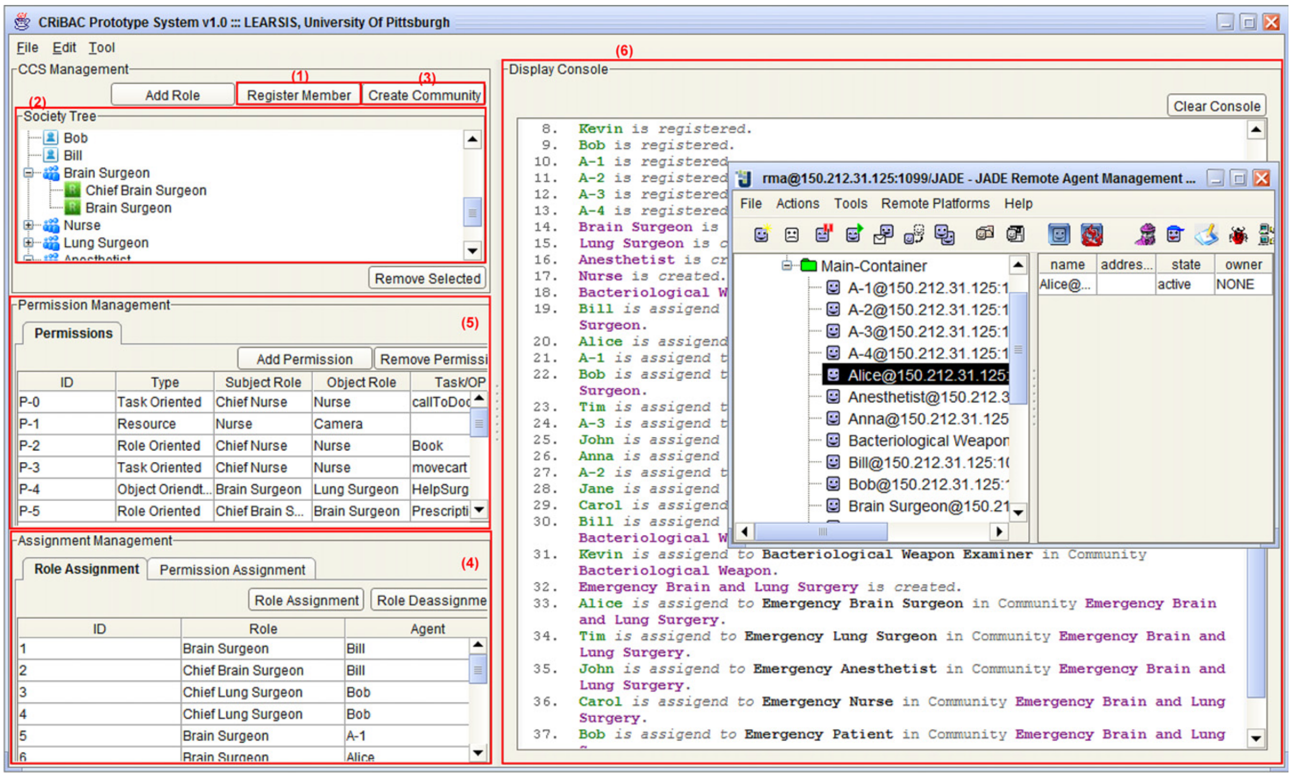


Fig. 13 – CRiBAC engine.



ID	Type	Subject Role	Object Role	Task/OP
P-0	Task Oriented	Chief Nurse	Nurse	callToDoc
P-1	Resource	Nurse	Camera	
P-2	Role Oriented	Chief Nurse	Nurse	Book
P-3	Task Oriented	Chief Nurse	Nurse	movecart
P-4	Object Oriendt...	Brain Surgeon	Lung Surgeon	HelpSurg
P-5	Role Oriented	Chief Brain S...	Brain Surgeon	Prescripti

ID	Role	Agent
1	Brain Surgeon	Bill
2	Chief Brain Surgeon	Bill
3	Chief Lung Surgeon	Bob
4	Chief Lung Surgeon	Bob
5	Brain Surgeon	A-1
6	Brain Surgeon	Alice

name	addres...	state	owner
Alice@...		active	NONE

```
 8.  Kevin is registered.
 9.  Bob is registered.
10.  A-1 is registered
11.  A-2 is registered
12.  A-3 is registered
13.  A-4 is registered
14.  Brain Surgeon is
15.  Lung Surgeon is c
16.  Anesthetist is cr
17.  Nurse is created.
18.  Bacteriological W
19.  Bill is assigend
     Surgeon.
20.  Alice is assigend
21.  A-1 is assigend t
22.  Bob is assigend t
     Surgeon.
23.  Tim is assigend t
24.  A-3 is assigend t
25.  John is assigend
26.  Anna is assigend
27.  A-2 is assigend t
28.  Jane is assigend
29.  Carol is assigend
30.  Bill is assigend
     Bacteriological W
31.  Kevin is assigend to Bacteriological Weapon Examiner in Community
     Bacteriological Weapon.
32.  Emergency Brain and Lung Surgery is created.
33.  Alice is assigend to Emergency Brain Surgeon in Community Emergency Brain
     and Lung Surgery.
34.  Tim is assigend to Emergency Lung Surgeon in Community Emergency Brain and
     Lung Surgery.
35.  John is assigend to Emergency Anesthetist in Community Emergency Brain and
     Lung Surgery.
36.  Carol is assigend to Emergency Nurse in Community Emergency Brain and Lung
     Surgery.
37.  Bob is assigend to Emergency Patient in Community Emergency Brain and Lung
```

Fig. 14 – Screenshot of the developed CRiBAC prototype system.

7.3. *Demonstration*

To verify the feasibility of CRiBAC and ACRiBAC, we implement a compact CRiBAC system based on the *UPMC* scenario presented in Section 4.2.1. For better understanding, we present a summary of our *UPMC* scenario-based demonstration in Table 7 and specifically describe operations of the whole system as follows.

At t_0, all agents that want to be involved register in the *UPMC Society* through the *UPMC Society Manager* (*SM*). In the CRiBAC prototype system, a system administrator can register an agent in a society by using the *Register Member* button in the system's *UI* (see Section 7.2.3 and Fig. 14(1)). The button triggers an execution of *AReg()* shown in Table 5. Then, an administrator is required to input an agent's information such as the agent's name, contexts, and tasks so that the Jade agent platform receives the information and creates a Jade agent incorporating all of the delivered information (see Section 7.2.2 and Fig. 12(1)). Jade informs the *SM* of a new agent's ID (see Fig. 12(2)). Such agents' information is specified by using the proposed XML-based specification language (For examples, *Bob*, *Bill*, and *Kevin* specification in the Appendix C-1(b)), and stored in the *Context Sever*. All registered agents are displayed in the *Society Tree* of *UI* (see Fig. 14(2)).

At t_1, five communities, *BS*, *LS*, *N*, *A*, and *BW*, are generated and registered by using the *Create Community* button for each community (see Fig. 14(3)). As the button triggers an execution of *CMCreate()*, a *CM* is created as a Jade agent with information about the corresponding community type, for example, the *EBLS* community type in the Appendix C-1(i). After being created, a *CM* communicates with all agents to recruit the most suitable agents for a community according to the *CRA_Constraint* and *criteria* specified in the *Community_Type* specification. At this time, *SM* imparts an agent's information to *CM* to assist with identifying suitable agents to achieve the community's goal (See *CRS* in Section 6.1). If *CM* gathers all necessary agents by executing *CommCreate()* shown in Table 6, the *CM* informs *SM* of information about the community. The delivered community information is also specified as an XML sheet, for example, *EBLS*, *BS*, and *BW* specification in the Appendix C-1(c). In a real-world CCS, the community creation process is automatically performed by *CM*; in our prototype system, an administrator manually assigns community roles to agents through the *Assignment Management Component* of *UI* (See Fig. 14(4)).

After creation, each community carries out its own cooperation at t_{i-1}. Each agent involved in the community has a set of permissions according to the CRiBAC policies and cooperates with each other. Therefore, the corresponding permission assignment (*PA*) to each community role (*CR*) should be done before a community creation. An administrator can define permissions through the *Permission Management* (See Fig. 14(5)), and all permissions are specified by the proposed specification language, for examples, *p*1 and *p*5 in the Appendix C-1(e). Then, an administrator assigns permissions to roles through the *Assignment Management Component* in *UI* and such assignments are specified in the *Permission_Assignments*. For examples, *pa*1 presented in the Appendix C-1(f) assigns *p*1, a *RPRMS* which performs *write_prescription* to a *patient* (*P*), to the *doctor* (*D*). According to *pa*1, *Bill* can give a prescription to a patient A_4 but *Carol* cannot because she is

Table 7 – Example UPMC (University of Pittsburgh Medical Center) community scenario.

Time	Situation description	Interaction among agents		Administration	
t_0	Agent registration with UPMC society			SM: For all agents, AReg();	
t_1	Community Creation and Registration with UPMC society	Brain Surgeon (BS), Lung Surgeon (LS), Anesthetist (A), Nurse (N), Bacteriological Weapon (BW)		1) SM: For each community, CMCreate(); 2) BS_CM: CommCreate(*BS*); LS_CM: CommCreate(*LS*); A_CM: CommCreate(*A*); N_CM: CommCreate(*N*); BW_CM: CommCreate(*BW*);	
t_{i-1}	In UPMC society, five communities exist and some agents are interacting or cooperating with each other.	Bill →[a]Kevin's task (bacteria cultivation)		Granted	
		A_4 → Kevin's task (bacteria elimination)		Denied	
		Bill's task (prescription) → A_4		Granted	
		Carol's task (prescription) → A_4		Denied	
		Community	Community Roles	Community Members	Administration
		BS	Chief Brain Surgeon (CBS)	Bill	ongoing cooperation
			Brain Surgeon (BS)	Bill, Alice, A_1	
		LS	Chief Lung Surgeon (CLS)	Bob	ongoing cooperation
			Lung Surgeon (LS)	Bob, Tim, A_3	
		A	Chief Anesthetist (CA)	John	ongoing cooperation
			Anesthetist (A)	John, Anna, A_2	
		N	Chief Nurse (CN)	Jane	ongoing cooperation
			Nurse (N)	Carol	
		BW	Bacteriological Weapon Examiner (BWE)	Bill, Kevin	ongoing cooperation
t_i	Bob is shot in the head and chest, so he needs to have an emergency surgery.	Same as those at t_{i-1}			SM: CMCreate(*EBLS*);
t_{i+1}	An "emergency surgery community" for Bob's brain and lung injuries is dynamically created	Emergency Brain and Lung Surgery (EBLS)	Emergency Brain Surgeon (EBS)	Alice	EBLS_CM: CommCreate(*EBLS*);
			Emergency Lung Surgeon (ELS)	Tim	
			Emergency Anesthetist (EA)	John	
			Emergency Nurse (EN)	Carol	
			Emergency Patient (EP)	Bob	
		LS	CLS	None[b]	LS_CM: Deactivate(*CLS*);
			LS	Tim, A_3	
		Other Four Communities	Same as those at t_{i-1}		Ongoing cooperation
t_{i+2}	A meeting community is organized between the nurse group and the anesthetist group but fails to take place	Chief Meeting Community between Nurses and Anesthetists (CMNA)	Presenter	None[c]	1) SM: CMCreate(*CMNA*); 2) CMNA_CM: CommCreate(*CMNA*); => FAIL
			Audience	Jane	
		EBLS			Ongoing cooperation
		Other four communities	Same as t_{i-1}		

a → Means an interaction access by another agent.
b Bob is not able to perform his roles due to his injury.
c John was expected, but he is not available.

not a *doctor*. Similar to this case, according to *pa*3, *Kevin* will do *cultivate_bacteria* operation if *Bill* asks to do but he will reject the request from A_4 because A_4 is not a *CBWE*.

At t_i, an *EBLS* community is required to handle the emergency situation that *Bob*, who is a chief in the Lung surgeon community (*LS*), sustains serious injuries. To do this, *SM* creates a *CM* for an *EBLS* community, called *EBLS_CM*, by operating *CMCreate(EBLS)*. The *EBLS_CM* organizes its community through the assignment of community roles to agents at t_{i+1}. At this moment, *LS_CM* perceives the change in *Bob*'s contexts and capabilities, for example, the change in the *health_condition* context from *good* to *injured*. Accordingly, *LS_CM* deactivates the *CRA* of *Bill* to *LS* because *Bill* is no longer satisfied with the assignment condition of *LS*. In our prototype system, an administrator manually deactivates a role through the *Role Assignment Component* (See Fig. 14(4)).

At t_{i+2}, *CMNA_CM* tries to create a *CMNA* community as scheduled but fails since *John* who takes the presenter role in the community is unavailable. At that time, *John* is working for an *EBLS* community and the community is a higher priority. Therefore, *John* denies the request for participation from *CMNA_CM*. Other communities keep cooperating with each other. All events such as the administrator's modification, community creation, and community's failure are described in the Console of *UI* (See Fig. 14(6)). An administrator can monitor all processes occurring in the system including cooperation among agents and results of policy enforcement.

8. Related work

All With the increasing interest in agent cooperation, several cooperative systems have been developed; for example, groupwares (Greenberg, 1991) of CSCW such as videoconferencing software or multi-agent-based cooperation systems such as Gaia (Zambonelli et al., 2003) and SuperSpace (Roman and Campbell, 2000). For practical use of these systems, it is necessary to apply an appropriate security model. In Tolone et al. (2005), the author specifies access control requirements for collaboration and compares existing access control models such as TMAC (Team-based Access Control Model) (Thomas, 1997), TBAC (Task-based Access Control Model) (Thomas and Sandhu, 1997), RBAC (Role-based Access Control Model) (Sandhu et al., 1996), and the context-aware access control model (Covington et al., 2001).

As pointed out in Tolone et al. (2005), TBAC has some drawbacks to being used in cooperation systems. The major drawback is the difficulty in dividing cooperation into tasks with usage counts. Second, TBAC specifies diverse but restricted context information related to tasks, activities, or workflows. However, cooperation systems need to capture much richer contexts. TMAC and Context-based TMAC (Georgiadis et al., 2001) include access control schemes for groups/teams of users who can take on different roles. However, they do not adequately handle dynamic and rich cooperation, because they do not have a flexible access control or administration models. In addition, they capture only limited context information. RBAC is an attractive model for cooperation systems because it encourages intuitive design, convenient administration, and can support more fine-grained and diverse access control requirements. However, current RBAC models lack flexibility with regard to dynamic changes associated with cooperation.

GTRBAC and GEO-RBAC (Damian et al., 2007) are extended RBAC models that address temporal and spatial contexts. DRBAC (Zhang and Parashar, 2004) allows real-time adjustment of user assignment as well as permission assignment by dynamic activation of roles or permissions based on environment contexts called ENVS. Covington et al. (2001) introduces context-aware role validation and activation using *Environment Roles* as well as user-role assignment in the GTRBAC model. Another context-aware access control model is CA-RBAC (Kulkarni and Tripathi, 2008) which was developed to support the needs of pervasive computing systems based on dynamically changing contexts during runtime. However, those models deal only with environmental contexts such as time, location, and temperature rather than an individual's private context such as height, age, gender, preference, medical history, and so on. However, in cooperative systems, individual users' contexts can be important. For example, an individual's task, status, or current roles in other groups can be critical when a cooperative group selects members to achieve its goal. If existing context-aware models try to capture individuals' contexts, they should be described as public environment contexts and the number of environmental contexts may grow explosively. In addition to an individual's context, consideration of group cooperation is required to control accesses in cooperative systems. GB-RBAC (Li et al., 2009) supports access for inter-group collaboration. It assumes that many groups already exist in a system, and concentrates on the administration of cooperative groups, known as Virtual Groups (VG). A *VG* is dynamically created at the outset by gathering roles and permissions from collaborating groups, and the group is terminated as soon as it accomplishes its goal. Each group administrator manages a *VG*. Since GB-RBAC is concerned with inter-group cooperation, it does not have a way to create cooperative groups or to manage them when there is no group. The more critical problem is that it does not support context-awareness. When it creates a *VG*, administrators from every collaborative group are elected as group administrators of a *VG*. It means that a *VG* has more than one administrator who makes decisions about User-Role assignments and access control during cooperation. However, there is no mention of how they decide when conflicting decisions arise. In addition, *DSet*, a set of default roles of a *VG*, is simply a union of those participating groups. It enables all members in a *VG* to have all permissions assigned to the collaborating groups' *DSet*. However, it could cause a serious security threat. Let's suppose a *VG* between a police group and a group of doctors are to take care of a patient involved in a traffic accident. Doctors participating in the *VG* have permissions assigned to *DSet* of the police group. For example, if the permission to read personal profiles of all citizens is assigned to a default role of the police group, doctors in the *VG* might be able to read someone's profile, a potential unsecured access to that information.

As you can see above, context-awareness and understanding of cooperation are simultaneously required for dynamic and efficient control in cooperative systems. In order to do that, we need an appropriate model reflecting both requirements at the same time. CRiBAC is one of such access control models that satisfy those requirements. It utilizes a range of contexts from public ones such as ambient contexts or community-related contexts to private contexts belonging to individuals. By using various contexts, CRiBAC can guarantee dynamic and fine-grained access control at runtime. Furthermore, it enables us to control interaction and cooperation among users in cooperative systems by reflecting the concept of community and role-based cooperation of Community Computing.

g-SIS (Group-Centric Secure Information Sharing) (Krishnan et al., 2009) also considers group-centric cooperation. It assumes that users and information come together for some common purpose much like that of a cooperative group. To determine authorization for shared information, it proposes some semantics of group operation, such as *join/leave* for users and *add/remove* for objects and the varieties of those operations. g-SIS allows detailed access control to shared information but it does not focus on context-aware authorization or interaction control. Therefore, it can be a complement to CRiBAC. CRiBAC can provide context-

Table 8 – Comparison of access control models for cooperation control.

	Ambient context	User's context	Context-awareness	User's object	Group aspect	Role interaction control
Environment roles	O	X	Medium	X	X	X
DRBAC	O	X	Medium	X	X	X
CA-RBAC	O	Δ (possible)	High	X	X	X
GB-RBAC	X	X	X	X	O	X
g-SIS	X	Δ (join/leave)	Low	X	O	X
RiBAC	O	X	High	O	X	O
CRiBAC	O	O	Very High	O	O	O

awareness and interaction control based on community, while g-SIS can offer ways to manage the membership of users and user's objects in detail.

Currently, there is no existing access control model which supports context-aware dynamic interaction or cooperation among users. We propose RiBAC and CRiBAC in this paper toward fulfilling this gap. In Table 8, we present a comparison between existing models and our RiBAC and CRiBAC in terms of context-awareness and cooperation.

9. Conclusion and future work

Multi-agent technology has been considered as one of the most promising research areas for the past few decades. An agent's cooperative and dynamic problem-solving behavior has been the key to providing intelligent services in a variety of application areas such as online business, e-government, e-healthcare, etc. In particular, an agents' ability to interact and cooperate with other agents has enabled many MASs to solve complex and large-scale problems (Zambonelli et al., 2003; Kumar et al., 2003; Jung and Kim, 2010). The rapid growth of technologies related to networking and smart devices has accelerated rich social interactions among agents. For more practical uses of MASs, the security for agents' interactions and cooperation is essential to allow only authorized social interactions in MASs. As aforementioned in Section 8, existing access control model do not however consider dynamic interaction or cooperation. To guarantee secure interaction and cooperation in MASs, in this paper, we propose two of role-interaction based access control models, the RiBAC model and the CRiBAC model, and provide an administration model and policy analysis methods. Here, we briefly summarize the major contributions of this paper as follows.

- RiBAC model – To ensure the security of agent interactions, we refined a family of RiBAC models proposed in 2009. In this model, an interaction among agents is also an entity to be protected as well as objects. In order to do so, we proposed a new type of permission, the interaction permission which includes the role-oriented interaction permission (RPRMS) and the task-oriented interaction permissions (TPRMS). For better readability and extensibility, in this paper, we refined the RiBAC models and proposed an XML-based specification language based on the notation of X-GTRBAC.
- CRiBAC model – we proposed the CRiBAC model to deal with the context-aware secure cooperation among agents as well as the secure interactions within a community setting. To do so, CRiBAC employs the community concept of Community Computing (CC) as a cooperation mechanism of MASs. In this model, the interaction permissions are classified into three types: RPRMS, TPRMS, and reSource-oriented interaction permission (SPRMS) for accessing an agent's resource. CRiBAC contributes to securely organize a group of agents and control accesses to participating agents during cooperation.
- ACRiBAC model – For the convenience of administration, we proposed ACRiBAC to support the administration of systems employing CRiBAC. In this model, we described the grant and revocation model of CRiBAC and several algorithms for administrating cooperation.
- CRiBAC policy analysis technique – we proposed the analysis techniques to verify the CRiBAC policies and also propose the corresponding solutions.
- Visual user interfaces – we implemented visual user interfaces for the convenient design and analysis of RiBAC and CRiBAC policies.
- CRiBAC prototype system – To test the feasibility and applicability of the proposed CRiBAC model, we implemented a prototype of CRiBAC system based on an emergency scenario in a hospital.

Although RiBAC and CRiBAC resolve the problem of unauthorized access to agents' resources, tasks, and agents themselves during interaction and cooperation in MASs, a few things still remain to be investigated:

- Trust or reputation-based access control on interactions and cooperation among agents. The proposed models deal with unauthorized interaction and cooperation based on roles assigned to agents. However, trust or reputation can be good criteria to allow interaction and cooperation among agents.
- Development of context ontology for CRiBAC. For practical context-awareness of the CRiBAC model, we plan to develop some domain-specific context ontology.
- Development of diverse application systems for practical verification of RiBAC and CRiBAC. To explore the feasibility of the proposed models, it is necessary to demonstrate more application systems working areas across of diverse group of subject areas.

Acknowledgements

Work of James Joshi has been supported by the US National Science Foundation grant IIS-0545912. We would like to thank the anonymous reviewers for their helpful comments.

Appendix A. The Specification of the Family of RiBAC models

A-1. The Specification of the Basic RiBAC model (RiBAC-B)

```
<!—RiBAC-B Definition>::=
<RiBAC-B>
  <!—Agent Definitions>
  <!—Role Definitions>
  <!—Permission Definitions>
  <!—Agent_Assignment Definitions>
  <!—Permission_Assignment Definitions>
</RiBAC-B>
```

a) RiBAC-B

```
<!—Agent Definitions>::=
<Agents>
  {<!—Agent Definition>}+
</Agents>

<!—Agent Definition>::=
<Agent Agent_ID=(id)>
  <Contexts> {<!—Context
   Definition>}+
  </Contexts>
  <Tasks> {<!—Task Definition>}*
  </Tasks>
</Agent>

<!—Context Definition>::=
<Context Context_Name=
(context name)>
   (context value)
</Context>
<!—Task Definition>::=
<Task Task_Name=(task
   name)/>
```

b) Agent Definition

```
<!—Role Definitions>::=
<Roles>
  {<!—Role Definition>}+
</Roles>

<!—Role Definition>::=
<Role Role_Name=
 (role name)/>
```

c) Role Definition

```
<!—Permission Definitions>::=
<Permissions>
  {<!—Permission Definition>}+
</Permissions>

<!—Permission Definition>::=
<Permission Permission_ID=(permission id)
   Permission_Type={OPRMS|RPRMS|TPRMS}>
  {<!—OPRMS Definition>|
   <!—RPRMS Definition>|
   <!—TPRMS Definition>}
</Permission>

<!— OPRMS Definition>::=
<OPRMS>
  <Operation> (operation name)
  </Operation>
 {<Object Object_ID=(object id)/>}+
</OPRMS>

<!— RPRMS Definition>::=
<RPRMS>
  <Operation> (operation name)
  </Operation>
  {<Obj_Role Role_Name=(role name)/>}+
</RPRMS>

<!— TPRMS Definition>::=
<TPRMS>
  <Operation> (operation name)
  </Operation>
 {<Obj_Task Obj_Role=(role name)
   Obj_Task_Name=(task name)/>}+
</TPRMS>
```

d) Permission Definition

```
<!—Agent Assignment Definitions>::=
<Agent_Assignments>
  {<!—<AA Definition>}*
</Agent_Assignments>

<!—AA Definition>::=
< AA AA_ID=(id) Role_Name=(role name)>
  <!—Constraint Definition>
  <Assigned_Agents>
    {<Agent Agent_ID=(id)/>}+
  </Assigned_Agents>
</AA>

<!— Constraint Definition>::=
<Constraint [op={AND|OR|NOT}]>
  <Conditon>
    {<!—Logical_Expr Definition>}+
  </Condition>
</Constraint>
<!— Logical_Expr Definition>::=
<Logical_Expr [op={AND|OR|NOT}]>
  {<!—Predicate Definition>}+
</Logical_Expr>

<!— Predicate Definition>::=
<Predicate>
{{<Operator> {GT|LT|EQ|NEQ} </Operator>
  <Para_Name Type={Cont|Tsk|Role}> (para name)
  </Para_Name>
  <Para_Value> (para value) </Para_Value>}
  |
  <!—Logical Expr>}
</Predicate>
```

e) Agent Assignment Definition

```
<!—Permission Assignment Definitions>::=
<Permission_Assignments>
  {<!—<PA Definition>}*
</Permission_Assignments>

<!—PA Definition>::=
< PA PA_ID=(id) Role_Name=(role name)>
  <Assigned_Permissions>
    {<Permission Permission_ID=(id)/>}
  </Assigned_Permissions>
</PA>
```

f) Permission Assignment Definition

A-2. Specification of the Hierarchical RiBAC Model (RiBAC-H) and Constrained RiBAC Model (RiBAC-C)

```
<!—Role Definition>::=
<Role Role_Name=(role name)>
  <Senior_Roles>
    {<Senior Role_Name= (role name)/>}*
  </Senior_Roles>
  <Junior_Roles>
    {<Junior Role_Name= (role name)/>}*
  </Junior_Roles>
</Role>
```

a) Role Definition of RiBAC-H with Role Hierarchy

```
<!—RiBAC-C Definition>::=
<RiBAC-C>
  <!—Agent Definitions>
  <!—Role Definitions>
  <!—Permission Definitions>
  <!—Agent_Assignment Definitions>
  <!—Permission_Assignment Definitions>
  <!—SoD_Constraint Definitons>
</RiBAC-C>
<!—Role Definition>::=
<Role Role_Name=(role name)>
  [<!—Constraints Definition>]
</Role>
<!—Constraints Definition>::=
<Constraints>
  [<Cardinality>
     [<Min> (number) </Min>]
     [<Max> (number) </Max>]
  </Cardinality>]
  [<!—SoD Definition>]
</Constraints>

<!—SoD Definition>::=
<SoD>
  <SSoD_Set>
    {<SSoD SSoD_ID=(id)>}+
  </SSoD_Set>
  <DSoD_Set>
    {<DSoD DSoD_ID=(id)>}+
  <DSoD_Set>
</SoD>
<!—SoD_Constraint Definitions>::=
<SoD_Constraints>
  <!—SSoD Constraint Definitions>
  <!—DSoD Constraint Definitions>
</SoD_Constraints>

<!—SSoD Constraint Definition>::=
<SSoD_Constraints>
  {<!—SSoD Definition>}*
</SSoD_Constraints>

<!—DSoD Constraint Definition>::=
<DSoD_Constraints>
  {<!—DSoD Definition>}*
</DSoD_Constraints>

<!—SSoD Definition>::=
<SSoD SSoD_ID=(id)>
 {<SSoD_Role
    Role_Name=(role name)
    [SSoD_Min_Cardinality=(number)]
    [SSoD_Max_Cardinality=(number)]/>}*
</SSoD>
<!—DSoD Definition>::=
<DSoD DSoD_ID=(id)>
 {<DSoD_Role
    Role_Name=(role name)
    [DSoD_Min_Cardinality=(number)]
    [DSoD_Max_Cardinality=(number)]/>}*
</DSoD>
```

b) RiBAC-C with Cardinality and SoD Constraints

```
<!—RiBAC-CH Definition>::=
<RiBAC-CH>
  <!—Agent Definitions>
  <!—Role Definitions>
  <!—Permission Definitions>
  <!—Agent_Assignment Definitions>
  <!—Permission_Assignment Definitions>
  <!—SoD_Constraint Definitons>
</RiBAC-C>

<!—Role Definition>::=
<Role Role_Name=(role name)>
  <Senior_Roles>
    {<Senior Role_Name= (role name) Type={I-Senior|A-Senior|IA-Senior}/>}*
  </Senior_Roles>
  <Junior_Roles>
    {<Junior Role_Name= (role name)/>}*
  </Junior_Roles>
  [<Constraints> <!—Constraints Definition> </Constraints>]
</Role>
```

c) RiBAC-CH with Hybrid Role Hierarchy and Constraints

Appendix B. The Specification of the Constrained Hierarchical CRiBAC Model (CRiBAC-CH)

```
<!—CRiBAC Definition>::=
<Society Society_Name=(society name)>
  <!—Community Definitions>
  <!—Agent Definitions>
  <!—Role Definitions>
  <!—Permission Definitions>
  <!—Permission_Assignment Definitions>
  <!—SoD_Constraint Definitions>
  <!—Society_Context Definitions>
  <!—Society_Object Definitions>
</Society>
```

a) CRiBAC-CH

```
<!—Agent Definitions>::=
<Agents>
  {<!—Agent Definition>}+
</Agents>
```

```
<!—Agent Definition>::=
<Agent Agent_ID=(id)>
  <Resources> {<!—Resource Definition>}*
  </Resources>
  <Contexts> {<!—Context Definition>}+
  </Contexts>
  <Tasks> {<!—Task Definition>}*
  </Tasks>
</Agent>
```

```
<!—Context Definition>::=
<Context Context_Name=(context name)>
  (context value)
</Context>
<!—Task Definition>::=
<Task Task_Name=(task name)/>

<!—Resource Definition>::=
< Resource Type=(resource type)
   Resource _ID=(id)/>
```

b) Agent Definition

```
<!—Community Definitions>::=
<Communities>
  {<!—Community Definition>}+
</Communities>


<!—Community Definition>::=
<Community Community_ID=(id)
   Community_TypeID=(type id)>
  <!—CommunityRole_Assignment Definitions>
</Community>
```

```
<!—CommunityRole_Assignment Definitions>::=
<CR_Assignments>
  {<!—CR_Assignment Defniniton>}+
</CR_Assignments>


<!—CR_Assignment Defniniton>}::=
<CR_Assignment CRA_ID=(id)
   CR_Name=(role name)>
  <!—CR_Member Definitions>
</CR_Assignment>
```

```
<!—CR_Member Definitions>::=
<CR_Members>
  {<Agent Agent_ID=(id)>}+
</CR_Members>
```

c) Community Definition

```
<!—Role Definitions>::=
<Roles>
  {<!—Role Definition>}+
</Roles>
<!—Role Definition>::=
<Role Role_Name=(role name)
   Role_Type={SR|CR}>
  <!—Senior_Role Definitons>
  <!—Junior_Role Definitons>
 [<!—Constraints>]
</Role>
```

d) Role Definition

```
<!—Permission Definitions>::=
<Permissions>
  {<!—Permission Definition>}+
</Permissions>


<!—Permission Definition>::=
<Permission Permission_ID=(permission id)
   Permission_Type={OPRMS|SPRMS|RPRMS|TPRMS}>
  {<!—OPRMS Definition>|<!—SPRMS Definition|
   <!—RPRMS Definition>|<!—TPRMS Definition>}
</Permission>
```

```
<!— OPRMS Definition>::=
<OPRMS>
  <Operation> (operation name)
  </Operation>
 {<Object Object_ID=(object id)/>}+
</OPRMS>
<!— SPRMS Definition>::=
<SPRMS>
  <Operation> (operation name)
  </Operation>
 {<Resource Obj_Agent=(agent id) Resource_ID=(id)/>|
  <Resource Obj_Role=(role id) Type=(type)/>}+
</SPRMS>
```

```
<!— RPRMS Definition>::=
<RPRMS>
  <Operation> (operation name)
  </Operation>
  {<Obj_Role Role_Name=(role name)/>}+
 </RPRMS>
<!— TPRMS Definition>::=
<TPRMS>
  <Operation> (operation name) </Operation>
  {<Obj_Task Obj_Role=(role name)
    Obj_Task_Name=(task name)/>}+
 </TPRMS>
```

e) Permission Definition

```
<!—Permission Assignment Definitions>::=
<Permission_Assignments>
  {<!—<PA Definition>}*
</Permission_Assignments>
```

```
<!—PA Definition>::=
< PA PA_ID=(id) Role_Name=(role name)>
  <Assigned_Permissions>
    {<Permission Permission_ID=(id)/>}
  </Assigned_Permissions>
</PA>
```

f) Permission Assignment Definition

```
<!—SoD_Constraint Definitions>::=
<SoD_Constraints>
  <!—SSoD Constraint Definitions>
  <!—DSoD Constraint Definitions>
</SoD_Constraints>
```

g) SoD Constraint Definition

```
<!—Society_Contexts Definition>::=
<Society_Contexts>
  <!—Society_Role Definitions>
  <!—Community_Type Definitions>
</Society_Contexts>

<!—Society_Role Definitions>::=
<Society_Roles>
  {<!—Society_Role Definition>}+
</Society_Roles>
```

```
<!—Society_Role Definition>::=
<Society_Role SR_Name=(SR name)>
  <!—SR_Assignment Definition>
</Society_Role>


<!—SR_Assignment Definition>::=
<SR_Assignment SRA_ID=(id)>
  [<SRA_Constraints op={AND|OR|NOT} type=(!—
   ConstraintType)> (constraint)
  </SRA_Constraints>]
  <!—SR_Member Definitions> </SR_Assignment>
```

```
<!—SR_Member Definitions>::=
<SR_Members>
  {<Agent Agent_ID=(id)/>}+
</SR_Members>
```

h) Society Role Definition

```
<!—Community_Type Definitions>::=
<Community_Types>
  <!—Community_Type Definition>
</Community_Types>


<!—Community_Type Definition>::=
<Community_Type CommunityTypeID=(id)>
  <Goal> (goal description) </Goal>
  <!—Community_Role Definitons>
</Community_Type>
```

```
<!—Community_Role Definitons>::=
<Community_Roles>
  {<!—Community_Role Definiton>}+
</Community_Roles>


<!—Community_Role Definiton>::=
<Community_Role
   CR_Name=(role name)
   [criteria={MAX|MIN}(para_name)]>
  <!—CRA_Constraint Definition>
 [<!—Cardinality Definition>]
</Community_Role>
```

```
<!—CRA_Constraint Definition>::=
 <CRA_Constraint type=(!—ConstraintType)>
   (Constraints)
 </CRA_Constraint>


<!—Cardinality Definition>::=
<Cardinality>
  [<MIN> (number) </MIN>]
  [<MAX> (number) </MAX>]
</Cardinality>
```

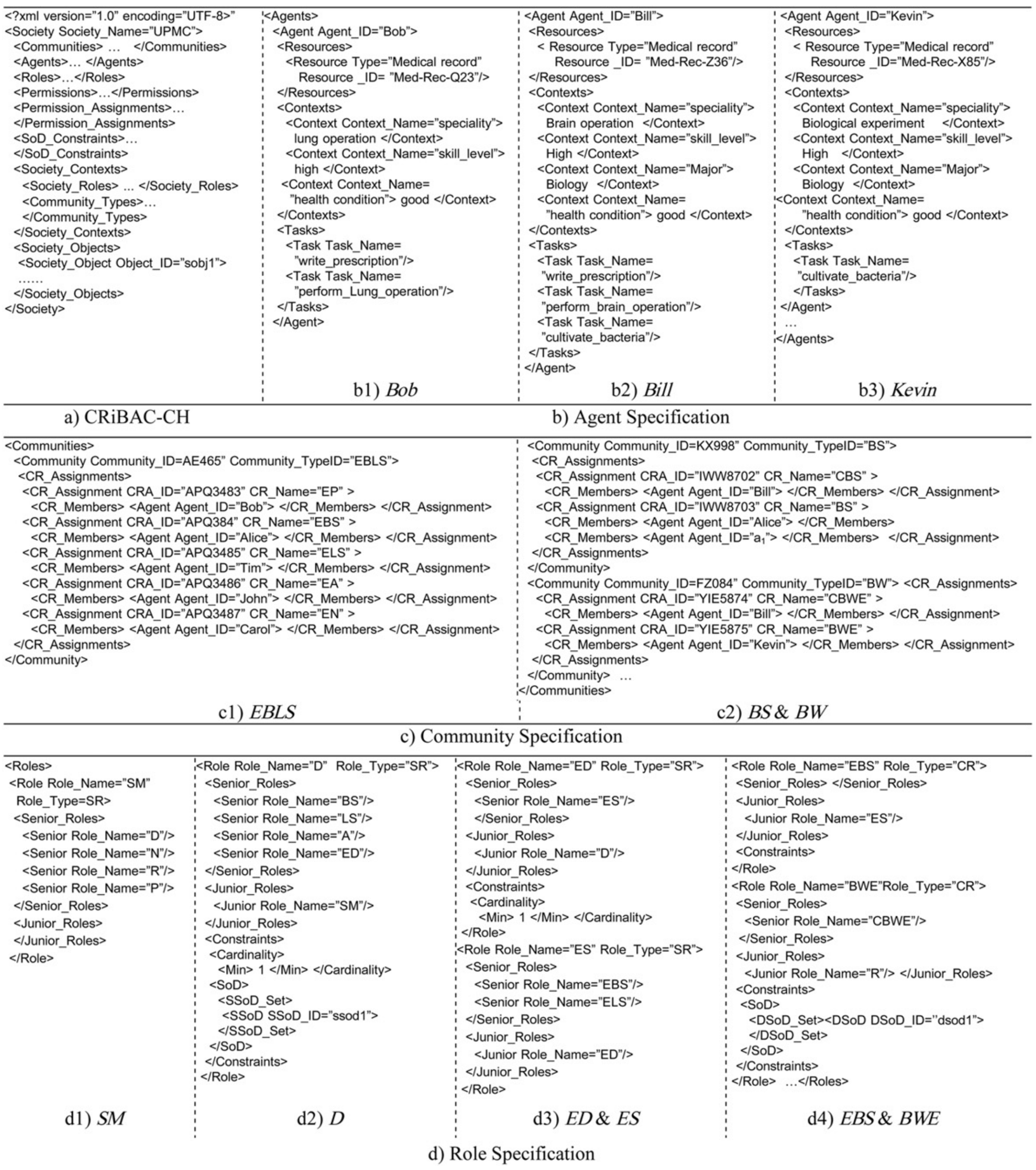
i) Community Type Definition

```
<!—Society_Object Definitions>::=
<Society_Objects>
  {<Society_Object Object_ID=(id)>}*
</Society_Objects>
```

j) Society Object Definition

Appendix C. The Specification of Example Scenarios: *UPMC* and *Disaster Relief*

C-1. *A Part of UPMC Example Specification*



```
<?xml version="1.0" encoding="UTF-8>"
<Society Society_Name="UPMC">
  <Communities> ...   </Communities>
  <Agents>... </Agents>
  <Roles>...</Roles>
  <Permissions>...</Permissions>
  <Permission_Assignments>...
  </Permission_Assignments>
  <SoD_Constraints>...
  </SoD_Constraints>
  <Society_Contexts>
    <Society_Roles> ... </Society_Roles>
    <Community_Types>...
    </Community_Types>
  </Society_Contexts>
  <Society_Objects>
   <Society_Object Object_ID="sobj1">
   ......
  </Society_Objects>
</Society>
```

a) CRiBAC-CH

```
<Agents>
 <Agent Agent_ID="Bob">
  <Resources>
    <Resource Type="Medical record"
       Resource _ID= "Med-Rec-Q23"/>
  </Resources>
  <Contexts>
    <Context Context_Name="speciality">
     lung operation </Context>
    <Context Context_Name="skill_level">
     high </Context>
   <Context Context_Name=
    "health condition"> good </Context>
  </Contexts>
  <Tasks>
    <Task Task_Name=
     "write_prescription"/>
    <Task Task_Name=
     "perform_Lung_operation"/>
  </Tasks>
 </Agent>
```

b1) *Bob*

```
<Agent Agent_ID="Bill">
 <Resources>
  < Resource Type="Medical record"
     Resource _ID= "Med-Rec-Z36"/>
 </Resources>
 <Contexts>
  <Context Context_Name="speciality">
   Brain operation  </Context>
  <Context Context_Name="skill_level">
   High </Context>
  <Context Context_Name="Major">
   Biology  </Context>
  <Context Context_Name=
   "health condition"> good </Context>
 </Contexts>
 <Tasks>
  <Task Task_Name=
   "write_prescription"/>
  <Task Task_Name=
   "perform_brain_operation"/>
  <Task Task_Name=
   "cultivate_bacteria"/>
 </Tasks>
</Agent>
```

b2) *Bill*

```
<Agent Agent_ID="Kevin">
 <Resources>
  < Resource Type="Medical record"
     Resource _ID="Med-Rec-X85"/>
 </Resources>
 <Contexts>
  <Context Context_Name="speciality">
   Biological experiment    </Context>
  <Context Context_Name="skill_level">
   High   </Context>
  <Context Context_Name="Major">
   Biology  </Context>
<Context Context_Name=
   "health condition"> good </Context>
 </Contexts>
 <Tasks>
  <Task Task_Name=
  "cultivate_bacteria"/>
  </Tasks>
</Agent>
 ...
</Agents>
```

b3) *Kevin*

b) Agent Specification

```
<Communities>
 <Community Community_ID=AE465" Community_TypeID="EBLS">
  <CR_Assignments>
   <CR_Assignment CRA_ID="APQ3483" CR_Name="EP" >
    <CR_Members> <Agent Agent_ID="Bob"> </CR_Members> </CR_Assignment>
   <CR_Assignment CRA_ID="APQ384" CR_Name="EBS" >
    <CR_Members> <Agent Agent_ID="Alice"> </CR_Members> </CR_Assignment>
   <CR_Assignment CRA_ID="APQ3485" CR_Name="ELS" >
    <CR_Members> <Agent Agent_ID="Tim"> </CR_Members> </CR_Assignment>
   <CR_Assignment CRA_ID="APQ3486" CR_Name="EA" >
    <CR_Members> <Agent Agent_ID="John"> </CR_Members> </CR_Assignment>
   <CR_Assignment CRA_ID="APQ3487" CR_Name="EN" >
    <CR_Members> <Agent Agent_ID="Carol"> </CR_Members> </CR_Assignment>
 </CR_Assignments>
</Community>
```

c1) *EBLS*

```
 <Community Community_ID=KX998" Community_TypeID="BS">
  <CR_Assignments>
   <CR_Assignment CRA_ID="IWW8702" CR_Name="CBS" >
     <CR_Members> <Agent Agent_ID="Bill"> </CR_Members> </CR_Assignment>
   <CR_Assignment CRA_ID="IWW8703" CR_Name="BS" >
     <CR_Members> <Agent Agent_ID="Alice"> </CR_Members>
     <CR_Members> <Agent Agent_ID="a1"> </CR_Members>  </CR_Assignment>
  </CR_Assignments>
 </Community>
 <Community Community_ID=FZ084" Community_TypeID="BW"> <CR_Assignments>
   <CR_Assignment CRA_ID="YIE5874" CR_Name="CBWE" >
     <CR_Members> <Agent Agent_ID="Bill"> </CR_Members> </CR_Assignment>
   <CR_Assignment CRA_ID="YIE5875" CR_Name="BWE" >
     <CR_Members> <Agent Agent_ID="Kevin"> </CR_Members> </CR_Assignment>
  </CR_Assignments>
 </Community>  ...
</Communities>
```

c2) *BS & BW*

c) Community Specification

```
<Roles>
 <Role Role_Name="SM"
  Role_Type=SR>
  <Senior_Roles>
    <Senior Role_Name="D"/>
    <Senior Role_Name="N"/>
    <Senior Role_Name="R"/>
    <Senior Role_Name="P"/>
  </Senior_Roles>
  <Junior_Roles>
  </Junior_Roles>
 </Role>
```

d1) *SM*

```
<Role Role_Name="D"  Role_Type="SR">
 <Senior_Roles>
   <Senior Role_Name="BS"/>
   <Senior Role_Name="LS"/>
   <Senior Role_Name="A"/>
   <Senior Role_Name="ED"/>
 </Senior_Roles>
 <Junior_Roles>
   <Junior Role_Name="SM"/>
 </Junior_Roles>
 <Constraints>
  <Cardinality>
    <Min> 1 </Min> </Cardinality>
  <SoD>
    <SSoD_Set>
     <SSoD SSoD_ID="ssod1">
    </SSoD_Set>
  </SoD>
 </Constraints>
</Role>
```

d2) *D*

```
<Role Role_Name="ED" Role_Type="SR">
 <Senior_Roles>
   <Senior Role_Name="ES"/>
   </Senior_Roles>
 <Junior_Roles>
   <Junior Role_Name="D"/>
 </Junior_Roles>
 <Constraints>
  <Cardinality>
    <Min> 1 </Min> </Cardinality>
</Role>
<Role Role_Name="ES" Role_Type="SR">
 <Senior_Roles>
   <Senior Role_Name="EBS"/>
   <Senior Role_Name="ELS"/>
 </Senior_Roles>
 <Junior_Roles>
   <Junior Role_Name="ED"/>
 </Junior_Roles>
</Role>
```

d3) *ED & ES*

```
<Role Role_Name="EBS" Role_Type="CR">
 <Senior_Roles> </Senior_Roles>
 <Junior_Roles>
   <Junior Role_Name="ES"/>
 </Junior_Roles>
 <Constraints>
</Role>
<Role Role_Name="BWE"Role_Type="CR">
 <Senior_Roles>
   <Senior Role_Name="CBWE"/>
  </Senior_Roles>
 <Junior_Roles>
   <Junior Role_Name="R"/> </Junior_Roles>
 <Constraints>
  <SoD>
    <DSoD_Set><DSoD DSoD_ID="dsod1">
    </DSoD_Set>
  </SoD>
 </Constraints>
</Role>  ...</Roles>
```

d4) *EBS & BWE*

d) Role Specification

```
<Permissions>
 <Permission Permission_ID="p1" Type="RPRMS">
  <RPRMS>
   <Operation> write_prescription </Operation>
   <Obj_Role Role_Name="P"/>
  </RPRMS>
</Permission>
<Permission Permission_ID="p2" Type="SPRMS">
  <SPRMS>
   <Operation> read  </Operation>
   <Resource Obj_Role="EP" Type="medical record"/>
  </SPRMS>
</Permission>
<Permission Permission_ID="p3" Type="RPRMS">
 <RPRMS>
  <Operation> perform_emergency_operation </Operation>
  <Obj_Role Role_Name="EP"/>
 </RPRMS>
</Permission>
<Permission Permission_ID="p4" Type="OPRMS">
 <OPRMS>
  <Operation> use </Operation>
  <Object Object_ID={surgery room1|surgery room2}/>
 </OPRMS>
</Permission>
<Permission Permission_ID="p5" Type="TPRMS">
  <TPRMS>
   <Operation> command </Operation>
   <Obj_Task Obj_Role="BWE"
     Obj_Task_Name="cultivate_bacteria"/>
  </TPRMS>
 </Permission>
 ...
</Permissions>
```

e) Permission Specification

```
<Permission_Assignments>
 <PA PA_ID="pa1" Role_Name="D">
  <Assigned_Permissions>
    <Permission Permission_ID="p1"/>
     ....
  </Assigned_Permissions>
 </PA>
 <PA PA_ID="pa2" Role_Name="ED">
  <Assigned_Permissions>
    <Permission Permission_ID="p2"/>
     ....
  </Assigned_Permissions>
 </PA>
 <PA PA_ID="pa3" Role_Name="ES">
  <Assigned_Permissions>
   <Permission Permission_ID="p3"/>
   <Permission Permission_ID="p4"/> ....
  </Assigned_Permissions>
 </PA>
 <PA PA_ID="pa3" Role_Name="CBWE">
  <Assigned_Permissions>
   <Permission Permission_ID="p5"/>
    ...
  </Assigned_Permissions>
 </PA>
```

f) Permission Assignment Specification

```
<SoD_Constraints>
 <SSoD_Constraints>
  <SSoD SSoD_ID="ssod1">
   <SSoD_Role Role_Name="EP"
   <SSoD_Role Role_Name="D"> </SSoD>
 </SSoD_Constraints>
 <DSoD_Constraints>
  <DSoD DSoD_ID="dsod1">
   <DSoD_Role Role_Name="BWE">
   <DSoD_Role Role_Name="S"> </DSoD>
 </DSoD_Constraints>
</SoD_Constraints>
```

g) SoD Constraint Specification

```
<Society_Roles>
 <Society_Role SR_Name="D" >
  <SR_Assignment SRA_ID="sra1">
   <SRA_Constraints">
    <Constraint>
     <Conditon>
      <Logical_Expr>
       <Predicate>
        <Logical_Expr op="AND">
         <Predicate>
          <Operator> EQ </Operator>
          <Para_Name Type="Cont">
            M.D license </Para_Name>
          <Para_Value> certified
          </Para_Value>
         </Predicate>
         <Predicate>
          <Operator> EQ </Operator>
          <Para_Name Type="Tsk">
            task name </Para_Name>
          <Para_Value> write_prescription
          </Para_Value>
         </Predicate>
        </Logical_Expr>
       </Predicate>
      </Logical_Expr>
     </Condition>
    </Constraint>
   </SRA_Constraints>
   <SR_Members>
    <Agent Agent_ID="a_1"/>
    <Agent Agent_ID="a_2"/>
    <Agent Agent_ID="a_3"/>
    <Agent Agent_ID="Alice"/>
    <Agent Agent_ID="Anna"/>
    <Agent Agent_ID="Bill"/>
    <Agent Agent_ID="Bob"/>
    <Agent Agent_ID="John"/>
    <Agent Agent_ID="Tim"/>
   </SR_Members>
  </SR_Assignment>
 </Society_Role>
 ...
</Society_Roles>
```

h) Society Role Specification - *D*

```
<Community_Types>
 <Community_Type
  CommunityTypeID="EBLS">
  <Goal> perform an emergency surgery on a
    patient's brain and lung </Goal>
  <Community_Roles>
   <Community_Role CR_Name="CLS"
     Criteria="MAX(skill-level)">
    <CRA_Constraint>
     <Conditon>
      <Logical_Expr op="AND">
       <Predicate>
        <Logical_Expr op="AND">
         <Predicate>
          <Operator> EQ </Operator>
          <Para_Name Type="Role">
            role name </Para_Name>
          <Para_Value> LS </Para_Value>
         </Predicate>
         <Predicate>
          <Operator> EQ </Operator>
          <Para_Name Type="Cont">
            emergency call</Para_Name>
          <Para_Value> on duty </Para_Value>
         </Predicate>
        </Logical_Expr>
       </Predicate>
       <Predicate>
        <Operator> GT </Operator>
        <Para_Name Type="Cont"> skill-level
        </Para_Name>
        <Para_Value> mid </Para_Value>
       </Predicate>
      </Logical_Expr>
     </Condition>
    </CRA_Constraint>
    <Cardinality>
     <MIN> 1 </MIN>
     <MAX> 2 </MAX>
    </Cardinality>
   </Community_Role>
   ....
  </Community_Roles>
 </Community_Type>
</Community_Types>
```

i) Community Type Specification - *EBLS*

C-2. A part of the Disaster Relief Example Specification

```
<?xml version="1.0" encoding="UTF-8>"
<Society Society_Name="Disaster Relief">
  <Communities> …   </Communities>
  <Agents>… </Agents>
  <Roles>…</Roles>
  <Permissions>…</Permissions>
  <Permission_Assignments>…
  </Permission_Assignments>
  <SoD_Constraints>… </SoD_Constraints>
  <Society_Contexts>
    <Society_Roles> ... </Society_Roles>
    <Community_Types>…  </Community_Types>
  </Society_Contexts>
  <Society_Objects>..  </Society_Objects>
</Society>
```

a) Disaster Relief with CRiBAC

```
<Agents>
  <Agent Agent_ID="$V_1$">
   <Resources>
     <Resource Type="Medical record"
       Resource _ID= "Med-Rec-MK8"/>
   </Resources>
   <Contexts>
     <Context Context_Name="location">
      ($V_1$'s GPS data) </Context>
   </Contexts>
  </Agent>
```

b1) V_1

```
<Agent Agent_ID="$C_6$">
  <Resources> .. </Resources>
  <Contexts>
    <Context Context_Name="speciality">
      heart disease </Context>
    <Context Context_Name="skill_level">
      High </Context>
  </Contexts>
  <Tasks>
    <Task Task_Name=
    "treat_heartdisease"/>
    <Task Task_Name=
    "provide_firstaid_instruction"/>
  </Tasks>
</Agent>
```

b2) C_6

```
<Agent Agent_ID="$H_5$">
  <Resources> .. </Resources>
  <Contexts>
    <Context
   Context_Name="location">
      ($H_5$'s GPS data) </Context>
  </Contexts>
  <Tasks>
    <Task Task_Name=
      "provide_firstaid"/>
    </Tasks>
 </Agent>
  …
</Agents>
```

b3) H_5

b) Agent Specification

```
<Communities>
  <Community Community_ID=R83" Community_TypeID=
   "Rescue-cardiac-patient">
   <CR_Assignments>
    <CR_Assignment CRA_ID="W33" CR_Name="Victim" >
      <CR_Members> <Agent Agent_ID="$V_1$"> </CR_Members>
    </CR_Assignment>
    <CR_Assignment CRA_ID="P82" CR_Name="Cardiologist" >
      <CR_Members> <Agent Agent_ID="$C_6$"> </CR_Members>
    </CR_Assignment>
    <CR_Assignment CRA_ID="K09" CR_Name="Helper" >
      <CR_Members>
        <Agent Agent_ID="$H_5$"> <Agent Agent_ID="$H_{11}$">
      </CR_Members>
     </CR_Assignment>
   </CR_Assignments>
  </Community>
 …..
<Communities>
```

c) Community Specification - *Rescue-cardiac-patient*

```
<Permission_Assignments>
 <PA PA_ID="pa11" Role_Name="Cardiologist">
   <Assigned_Permissions>
     <Permission Permission_ID="p11"/>
     <Permission Permission_ID="p12"/>
     <Permission Permission_ID="p13"/>
   </Assigned_Permissions>
 </PA>
 <PA PA_ID="pa2" Role_Name="Helper">
   <Assigned_Permissions>
     <Permission Permission_ID="p12"/>
     <Permission Permission_ID="p14"/>
   </Assigned_Permissions>
 </PA>
 ….
</Permission_Assignments>
```

d) Permission Assignment Specification

```
<SoD_Constraints>
 <SSoD_Constraints>
  <SSoD SSoD_ID="ssod11">
    <SSoD_Role Role_Name="Victim"
    <SSoD_Role Role_Name="Helper"> </SSoD>
 </SSoD_Constraints>
 …..
</SoD_Constraints>
```

e) SoD Constraint Specification

```
<Permissions>
 <Permission Permission_ID="p11" Type="SPRMS">
  <SPRMS>
   <Operation> read  </Operation>
   <Resource Obj_Role="Victim" Type="medical record"/>
  </SPRMS>
 </Permission>
<Permission Permission_ID="p12" Type="SPRMS">
  <SPRMS>
   <Operation> read  </Operation>
   <Resource Obj_Role="Victim" Type="location"/>
  </SPRMS>
 </Permission>
<Permission Permission_ID="p13" Type="TPRMS">
    <TPRMS>
     <Operation> command </Operation>
     <Obj_Task Obj_Role="Helper"
       Obj_Task_Name="provide_firstaid"/> </TPRMS>
  </Permission>
  <Permission Permission_ID="p14" Type="TPRMS">
    <TPRMS>
     <Operation> ask </Operation>
     <Obj_Task Obj_Role="Cardiologist" Obj_Task_Name="provide_firstaid_instruction"/>  </TPRMS>
  </Permission>   …
</Permissions>
```

f) Permission Specification

```
<Community_Types>
  <Community_Type CommunityTypeID="Rescue-
  cardiac-patient">
   <Goal> Rescue a victim who has cardiac disease
   </Goal>
   <Community_Roles>
    <Community_Role Role_Name="'Victim">
     <CRA_Constraint>
      <Conditon>
       <Logical_Expr>
        <Predicate>
         <Logical_Expr op="AND">
          <Predicate>
           <Operator> EQ </Operator>
           <Para_Name Type="Cont">
             body condition </Para_Name>
           <Para_Value> injured </Para_Value>
          </Predicate>
          <Predicate>
           <Operator> EQ </Operator>
           <Para_Name Type="Cont">
             disease</Para_Name>
           <Para_Value> cardiac </Para_Value>
          </Predicate>
         </Logical_Expr>
        </Predicate>
       </Logical_Expr>
      </Condition>
     </CRA_Constraint>
     <Cardinality>
      <MIN> 1 </MIN>
     </Cardinality>
    </Community_Role>
    <Community_Role Role_Name="'Cardiologist">
     <CRA_Constraint>
      <Conditon>
       <Logical_Expr>
        <Predicate>
         <Logical_Expr op="AND">
          <Predicate>
           <Operator> EQ </Operator>
           <Para_Name Type="Role">
             role name </Para_Name>
           <Para_Value> D </Para_Value>
          </Predicate>
          <Predicate>
           <Operator> EQ </Operator>
           <Para_Name Type="Cont">
             speciality </Para_Name>
           <Para_Value> heart disease </Para_Value>
          </Predicate>
         </Logical_Expr>
        </Predicate>
       </Logical_Expr>
      </Condition>
     </CRA_Constraint>
     <Cardinality>
      <MIN> 1 </MIN>
     </Cardinality>
    </Community_Role>
    <Community_Role Role_Name="'Helper">
     <CRA_Constraint>
      <Conditon>
       <Logical_Expr>
        <Predicate>
         <Logical_Expr>
          <Predicate>
           <Operator> EQ </Operator>
           <Para_Name Type="Cont">
             location </Para_Name>
           <Para_Value> (victim's locaton)
           </Para_Value>
          </Predicate>
         </Logical_Expr>
        </Predicate>
       </Logical_Expr>
      </Condition>
     </CRA_Constraint>
     <Cardinality>
      <MIN> 1 </MIN>  <MAX> 3 </MAX>
     </Cardinality>
    </Community_Role>
   </Community_Roles>
  </Community_Type>
 ……
</Community_Types>
```

g) Community Type Specification - *Rescue-cardiac-patient*

REFERENCES

Bhatti R, Ghafoor A, Bertino E, Joshi J. X-GTRBAC: an XML-based policy specification framework and architecture for enterprise-wide access control. ACM Transactions on Information and System Security (TISSEC) 2005;8(2):187–227. doi:10.1145/1065545.1065547.

Covington MJ, Long W, Srinivasan S, Dey AK, Ahamad M, Abowd GD. Securing context-aware applications using environment roles. In: 6th ACM symposium on access control models and technologies(SACMAT '01); 2001. p. 10–20. doi:10.1145/373256.373258.

Damian ML, Bertino E, Catania B, Perlasca P. GEO-RBAC: a spatially aware RBAC. ACM Transactions on Information and System Security (TISSEC) 2007;10(1):1–42.

Emory Healthcare. Can Twitter help save lives? A health care social media case study-part I, http://advancingyourhealth.org/highlights/2011/04/27/can-twitter-help-save-lives-a-health-care-social-media-case-study-part-i/; May 2011.

Ferrini R, Bertino E. Supporting RBAC with XACML+OWL. In: 14th ACM symposium on access control models and technologies (SACMAT '09); 2009. p. 145–54. doi:10.1145/1542207.1542231.

Georgiadis CK, Mavridis I, Pangalos G, Thomas RK. Flexible team-based access control using contexts. In: 6th ACM symposium on access control models and technologies (SACMAT '01); 2001. p. 21–7.

Greenberg S. Computer-supported cooperative work and groupware. London: Academic Press Ltd; 1991.

Joshi JBD, Bertino E, Latif U, Ghafoor A. A generalized temporal role-based access control model. IEEE Transactions on Knowledge and Data Engineering 2005;17(1):4–23.

Jung Y, Kim M. Situation-aware community computing model for developing dynamic ubiquitous computing systems. Journal of Universal Computer Science 2010;16(15):2139–74.

Jung Y, Masoumzadeh A, Joshi JBD, Kim M. RiBAC: role interaction based access control model for community computing. Lecture notes of the institute for computer sciences. Social Informatics and Telecommunications Engineering 2009;10: 304–21.

Jung Y, Kim M, Masoumzadeh A, Joshi JBD. A survey of security issue in multi-agent systems. Review 2011;published online. Artificial Intelligence; June 2011. doi:10.1007/s10462-011-9228-8.

Krishnan R, Sandhu RS, Niu J, Winsborough W. Towards a framework for group-centric secure collaboration. In: 5th international conference on collaborative computing: networking, applications and worksharing; 2009. p. 1–10.

Kulkarni D, Tripathi A. Context-aware role-based access control in pervasive computing systems. In: 13th ACM symposium on access control models and technologies; 2008. p. 113–22.

Kumar M, Shirazi BA, Das SK, Singhal M, Sung BY, Levine D. Pervasive information communities organization PICO: a Middleware framework for pervasive computing. IEEE Pervasive Computing 2003;2(3):72–9.

Li Q, Zhang X, Xu M, Wu J. Towards secure dynamic collaborations with group-based RBAC model. Computers & Security 2009;28(5):260–75.

Roman M, Campbell RH. GAIA: enabling active spaces. In: 9th ACM SIGOPS European workshop; 2000. p. 229–34. Kolding, Denmark.

Sandhu RS, Coyne EJ, Feinstein HF, Youman CE. Role-based access control models. Computer 1996;29(2):38–47.

Thomas RK. Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments. In: 2nd ACM workshop on role-based access control (RBAC '97); 1997. p. 13–9.

Thomas RK, Sandhu RS. Task-based authorization controls (TBAC): a family of models for active and enterprise-oriented authorization management. In: IFIP TC11 WG11.3 11th international conference on database security XI: status and prospects; 1997. p. 166–81. London, UK.

Tolone W, Ahn G, Pai T, Hong S. Access control in collaborative systems. ACM Computing Surveys 2005;37(1):29–41.

XACML (eXtensible Access Control Markup Language) v.3.0 Core and Hierarchical RBAC Profile v.1.0, OASIS, 10 August 2010, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cs-01-en.html.

XACML v.3.0, OASIS, 10 August 2010, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.html.

Zambonelli F, Jennings NR, Wooldridge M. Developing multiagent systems: the Gaia methodology. ACM Transactions on Software Engineering and Methodology 2003;12(3):317–70.

Zhang G, Parashar M. Context-aware dynamic access control for pervasive applications. In: Conference on the communication networks and distributed systems modeling and simulation; 2004. San Diego, CA, USA.

Youna Jung received the PhD degree from Ajou University in 2007. She worked in the Laboratory of Education and Research on Security Assured Information Systems (LERSAIS) at the University of Pittsburgh as a postdoctoral researcher. She is currently a research scientist in the Advanced Computing and Information Systems (ACIS) Laboratory in University of Florida. Her research interests include situation-aware computing, cooperative computing, community computing, security of multi-agent systems, and security of social computing system.

James B.D Joshi is an associate professor and the director of the Laboratory for Education and Research on Security Assured Information Systems (LERSAIS) in the School of Information Sciences at the University of Pittsburgh. He received his MS in Computer Science and PhD in Computer Engineering from Purdue University in 1998 and 2003, respectively. His research interests include role-based access control, trust management, and secure interoperability. He is a member of IEEE and ACM.