Dependencies and Separation of Duty Constraints in GTRBAC

James B D Joshi
School of Electrical and
Computer Engineering,
Purdue University,
West Lafayette, IN, USA
joshij@purdue.edu

Elisa Bertino
Dipartmento di Scienze
dell''Informazione,
Universita' di Milano,
Milan, Italy
bertino@dsi.unimi.it

Basit Shafiq
School of Electrical and
Computer Engineering,
Purdue University,
West Lafayette, IN, USA
shafiq@purdue.edu

Arif Ghafoor
School of Electrical and
Computer Engineering,
Purdue University,
West Lafayette, IN, USA
ghafoor@purdue.edu

ABSTRACT

A Generalized Temporal Role Based Access Control (GTRBAC) model that captures an exhaustive set of temporal constraint needs for access control has recently been proposed. GTRBAC's language constructs allow one to specify various temporal constraints on role, user-role assignments and role-permission assignments. In this paper, we identify various time-constrained cardinality, control flow dependency and separation of duty constraints (SoDs). Such constraints allow specification of dynamically changing access control requirements that are typical in today's large systems. In addition to allowing specification of time, the constraints introduced here also allow expressing access control policies at a finer granularity. The inclusion of control flow dependency constraints allows defining much stricter dependency requirements that are typical in workflow types of applications.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access control; H.2.7 [**Database Administration**] Security, integrity, and protection.

General Terms

Security, Theory.

Keywords

Role based access control, security, separation of duty, temporal constraints, cardinality constraint

1. INTRODUCTION

Role based access control (RBAC) has emerged as a promising alternative to traditional discretionary and mandatory access control (DAC and MAC) models [7, 11, 15, 16], which have some inherent limitations [11]. Several key features such as policy neutrality, support for least privilege, efficient access control management, are associated with RBAC models [7, 11, 16]. Such

features make RBAC better suited for handling access control requirements of diverse organizations. Furthermore, the concept of role is associated with the notion of functional roles in an organization, and hence RBAC models provide intuitive support for expressing organizational access control policies [5]. RBAC models have also been found suitable for addressing security issues in the Internet environment [2, 11], and have shown prospects for supporting secure interoperation in a heterogeneous multidomain environment [10].

One of the important aspects of access control is that of time constraining accesses to limit resource use. Such constraints are essential for controlling time-sensitive activities that may be present in various applications such as workflow management systems (WFMSs). Tasks in a WFMS may be time dependent and need to be executed in some order [3]. To address general time-based access control needs, Bertino *et al*. propose a Temporal RBAC model (TRBAC) [4], which has been recently generalized by Joshi *et al.* [12]. The Generalized-TRBAC (GTRBAC) model [12] incorporates a set of language constructs for the specification of various temporal constraints on roles, including constraints on their activations as well as on their enabling times, user-role assignment and role-permission assignment. In particular, GTRBAC makes a clear distinction between role *enabling* and role *activation*. A role is *enabled* if a user can acquire the permissions assigned to it. An *enabled* role becomes *active* when a user acquires the permissions assigned to it in a session. An open issue in the GTRBAC model, as well as in the TRBAC model [4] is the specification and enforcement of time-constrained cardinality, control flow dependency and separation of duty (SoD) constraints.

Cardinality and SoD constraints are crucial for securing many applications in a commercial environment. Many researchers have highlighted the importance and use of cardinality and SoD constraints in RBAC models [8, 17, 18]. However, no one has addressed the time-based cardinality and SoD constraints. Use of a particular constraint for a period of time or duration is important for emerging applications as access requirements frequently change with time. Dependency constraints are relevant to role based systems as roles often embody organizational functions that may be inter-dependent. For instance, a doctor in training may be allowed to work *only if* some senior doctor who can supervise him, is also on duty. Some aspect of dependency constraints such as history based access control, operational SoD, etc., have been mentioned in general access control literature [3, 18], but they have not been adequately addressed for general RBAC systems. In

this paper, we focus on these constraints within the GTRBAC modeling framework [12]. The key contributions of this paper are as follows:

- We introduce a generic framework for expressing a wide range of time-based cardinality constraints with the help of GTRBAC status predicates, a function to evaluate these predicates and a projection operator that extracts a set of elements from the evaluation of the function.

- We develop an elaborate trigger expression that can capture complex dependencies among events and conditions. In particular, we define CFD constraints that can be used to express stricter control flow dependencies. Furthermore, we show that the trigger framework and the CFD constraint expressions can be easily extended to provide an elaborate time based RBAC model for context-based access control.

- We identify a large set of possible SoD constraints using the GTRBAC status predicates. These SoDs subsumes the SoDs that have been identified in the RBAC literature, and at the same time provide much finer modeling capability.

The paper is organized as follows. In section two, we briefly present the constraints of GTRBAC. In section three, we present the status predicates for a GTRBAC system and the cardinality constraints. In section 4, we present the GTRBAC triggers and the control flow dependency constraints. The time-constrained SoD constraints are presented next, in section 5. The related work is presented in section 6. Section 7 concludes the paper and provides some future directions.

2. THE GTRBAC MODEL

The GTRBAC model provides a temporal framework for specifying an extensive set of temporal constraints [12]. This model is an extension of the TRBAC model [4] and uses a language-based framework. GTRBAC allows various types of temporal constraints such as *temporal constraints on role enabling/disabling*, *temporal constraints on user-role and role-permission assignments/de-assignments*, *role activation-time constraints, etc*. GTRBAC's *administrative run-time events* allow an administrator to dynamically initiate events. Another set of run-time events allows users to make activation requests to the system. Furthermore, *constraint-enabling expressions* include events that enable or disable duration constraints and role activation constraints. The GTRBAC *triggers* allow the expression of dependencies among GTRBAC events, and capturing past events. GTRBAC can capture the dynamically changing access control needs of a system [12, 13]. The periodic expressions are written as (I, P), where I is an interval and P is a set of infinite number of intervals. (I, P) represents the set of all the intervals of P that are contained in I. For example, $(I, P) =$ ([1/1/2002, 12/31/2002], *Mondays*) considers all the *Mondays* of the year 2002. D is used to express the duration specified for a duration constraint. Temporal constraints are expressed by a generic form, (I, P, E), where (I, P) is the periodic expression, or a duration constraint $c = ([I, P| D], D_x, E)$, where D_x specifies the duration in which the event E is valid, and D or (I, P) specifies the duration/interval in which the duration constraint c is valid. The periodic expressions (I, P) used in the constraint expressions are based on those in [4]. For more details, we refer the readers to

[12, 13]. An example of a GTRBAC policy for a medical information system is illustrated below.

Example: Table 1 contains the GTRBAC policy for a hospital. The periodicity constraint 1*a* specifies the enabling times of DayDoctor and NightDoctor roles. For simplicity, we use *DayTime* and *NightTime* instead of their (I, P) forms. The periodicity constraint 1*b* allows the DayDoctor role to be assigned to *Adams* on *Mondays*, *Wednesdays* and *Fridays*, and to *Bill* on *Tuesdays*, *Thursdays, Saturdays* and *Sundays*. Similarly, *Alice* and *Ben* are assigned to the NightDoctor role on the different days of the week. Furthermore, the assignment in 1*c* allows *Carol* to assume the DayDoctor role everyday between 10am and 3pm. In 2a, *Ami* and *Elizabeth* are assigned to roles NurseInTraining and DayNurse respectively with no temporal restriction, i.e., the assignment is valid at all times. 2*b* specifies a duration constraint of 2 *hours* on the enabling time of the NurseInTraining role, but this constraint is valid for only 6 *hours* after the constraint *c*1 has been enabled. Because of this, *Ami* will be able to activate the NurseInTraining role at the most for two hours whenever the role is enabled. In row 3, we have a set of triggers. Trigger 3*a* indicates that constraint *c*1 is enabled when the DayNurse is enabled, which means, now, the NurseInTraining role can be enabled within the next 6 *hours*. Trigger 3*b* indicates that 10 *min* after *Elizabeth* activates the DayNurse role, the NurseInTraining role is enabled for a period of 2 *hours*. This shows that a nurse in training will have access to the system only if *Elizabeth* is present in the system, that is, she may be acting as a training supervisor. It is possible that *Elizabeth* activates the DayNurse role a number of times in 6 hours after the DayNurse role has been enabled, and each time the NurseInTraining role will also be enabled if these activations (by *Elizabeth*) are more than 2 hours apart. This will allow *Ami* to activate the NurseInTraining role a number of times. The remaining triggers in 3 show that the DayNurse and NightNurse roles are enabled (disabled) 10 *min* after the DayDoctor and NightDoctor roles are enabled (disabled).

Table 1: Example GTRBAC access control policy for a medical information System

1	a.	(*DayTime*, enable DayDoctor), (*NightTime*, enable NightDoctor)
	b.	((M, W, F), assign_U *Adams* to DayDoctor), ((T, Th, S, Su), assign_U *Bill* to DayDoctor); ((M, W, F), assign_U *Alice* to NightDoctor), ((T, Th, S, Su), assign_U *Ben* to NightDoctor)
	c.	([10am, 3pm], assign_U *Carol* to DayDoctor)
2	a.	(assign_U *Ami* to NurseInTraining); (assign_U *Elizabeth* to DayNurse)
	b.	$c1 = (6\ hours, 2\ hours,\ $ enable NurseInTraining)
3	a.	(enable DayNurse → enable $c1$)
	b.	(activate DayNurse for *Elizabeth* → enable NurseInTraining after 10 *min*)
	c.	(enable NightDoctor → enable NightNurse after 10 *min*);
	d.	(enable DayDoctor → enable DayNurse after 10 *min*); (disable DayDoctor → disable DayNurse after 10 *min*)

3. STATUS EXPRESSIONS AND CARDINALITY CONSTRIANTS

Table 1 lists various GTRBAC status predicates. The predicate set extends the set of status predicates defined in our previous work [13]. Such an extension was needed in order to provide a finer modeling capability required to represent various temporal constraints that have been introduced in this paper. The non-temporal counterparts of each predicate can be simply obtained by removing the time parameter. A non-temporal predicate s simply indicates that its corresponding temporal predicate s_t applies at all times, i.e., $s \rightarrow \forall t, s_t$. Inversely, s_t means that status predicate s holds at time t. The second column of Table 2 specifies the evaluation domain for the predicates in the first column. The third column describes the semantics of the predicate.

Table 2: Various status predicates

Predicate(s_t)	Evaluation Domain(*DOM*)	Semantics
P:permission set, R:role set, U:user set, S:set of sessions, T:time instants, and $r \in R, p \in P, u \in U, s \in S, t \in T$		
enabled(*r, t*)	$R \times T$	*r is enabled at time t*
u_assigned(*u, r, t*)	$U \times R \times T$	*u is assigned to r at time*
p_assigned(*p, r, t*)	$P \times R \times T$	*p is assigned to r at t*
can_activate (*u, r, t*)	$U \times R \times T$	*u can activate r at t*
can_acquire (*u, p, t*)	$U \times P \times T$	*u can acquire p at t*
r_can_acquire (*u,p, r, t*)	$U \times P \times R \times T$	*u can acquire p through r at t*
can_be_acquired(*p, r,t*)	$P \times R \times T$	*p can be acquired through r at t*
active(*u, r, t*)	$U \times R \times T$	*r is active in u's session at t*
s_active(*u, r, s, t*)	$U \times R \times S \times T$	*r is active in u's session s at t*
acquires(*u, p, t*)	$U \times P \times T$	*u acquires p at t*
r_acquires(*u, p, r, t*)	$U \times P \times R \times T$	*u acquires p through r at t*
s_acquires(*u, p, s, t*)	$U \times P \times S \times T$	*u acquires p in session s at t*
rs_acquires(*u, p, r, s, t*)	$U \times P \times R \times S \times T$	*u acquires p through r in session s at t*

Predicate enabled(*r, t*), u_assigned(*u, r, t*) and p_assigned(*p, r, t*) refer to the status of roles, and user-role and role-permission assignments at time *t*. Predicate can_activate(*u, r, t*) implies that user *u* can activate role *r* at time *t*. It allows us to capture the fact that a user *u* may be able to activate role *r* without being explicitly assigned to it, as it is possible in a hierarchy that incorporates the *activation-inheritance* semantics [13]. In other words, "*u can activate r*" implies that user *u* is implicitly or explicitly assigned to role *r*. It does not rule out the possibility that some activation or SoD constraints prevent the actual activation of *r* by *u* at time *t*. Predicate can_acquire(*u, p, t*) implies that "*u can acquire permission p*"

at time *t*. Predicate r_can_acquire(*u, p, r, t*) provides much finer level of information than can_acquire(*u, p, t*) and indicates that "*u can acquire permission p through role r*" at time *t*. can_acquire(*u, p, t*) can be semantically defined in terms of r_can_acquire(*u, p, r, t*) as shown in Table 3. can_be_acquired(*u, r, t*) implies that permission "*p can be acquired through role r*" at time *t*.

Table 3: Semantic relation between GTRBAC status

1	can_acquire (*u, p, t*) $\leftrightarrow \exists r \in R$, r_can_acquire (*u, p, r, t*)
2	active (*u, r, t*) $\leftrightarrow \exists s \in S$, active (*u, r, s, t*)
3	acquires (*u, p, t*) $\leftrightarrow \exists r \in R$, r_acquires (*u, p, r, t*)
4	acquires (*u, p, t*) $\leftrightarrow \exists s \in S$, s_acquires (*u, p, s, t*)
5	acquires (*u, p, r, t*) $\leftrightarrow \exists s \in S$, rs_acquires (*u, p, r, s, t*)
6	acquires (*u, p, s, t*) $\leftrightarrow \exists r \in R$, rs_acquires (*u, p, r, s, t*)

It is important to note that can_activate(*u, r, t*), can_acquire(*u, p, t*), r_can_acquire(*u, p, r, t*) and can_be_acquired(*u, r, t*) predicates do not assume anything about the state of a role. That is, they do not say in which state role *r* is at time *t*. For example, if can_activate(*u, r, t*) and enabled(*r, t*) hold, then a user *u*'s request to activate *r* at time *t* is granted provided there are no other activation or SoD constraints prohibiting it. However, if can_activate(*u, r, t*) holds but not enabled(*r, t*), then *u*'s request to activate *r* at time *t* is denied. Thus, these predicates indicate possibility rather than what actually occurs.

Predicates active(*u, r, t*), s_active(*u, r, s, t*), acquires(*u, p, t*), r_acquires(*u, p, r, t*), s_acquires(*u, p, s, t*) and rs_acquires(*u, p, r, s, t*) refer to what actually occurs at time instant *t*. active(*u, r, t*) indicates that role *r* is active in a user *u*'s session at time *t* and can be expressed using predicate s_active(*u, r, t*) as shown in the Table 3. acquires(*u, p, t*) implies that a user "*u* acquires permission *p* at time *t*" and can be expressed in terms of r_acquires(*u, p, r, t*) and s_acquires(*u, p, s, t*), which in turn can be defined in terms of rs_acquires(*u, p, r, s, t*). The following axioms, as introduced in [13], capture the key relationships among various predicates in Table 1 and provide the basis for defining precisely the permission-acquisition and role-activation semantics of a GTRBAC system.

Axioms: *For all $r \in R$, $u \in U$, $p \in P$, $s \in S$, and time instant $t \in T = \{0, \infty\}$, the following implications hold*:

1. assigned(*p, r, t*) $\rightarrow$ can_be_acquired(*p, r, t*)

2. assigned(*u, r, t*) $\rightarrow$ can_activate (*u, r, t*)

3. can_activate (*u, r, t*) $\wedge$ can_be_acquired(*p, r, t*) $\rightarrow$ can_acquire (*u, p, t*)

4. active(*u, r, t*) $\wedge$ can_be_acquired(*p, r, t*) $\rightarrow$ acquires(*u, p, t*)

Axiom (1) states that if a permission is assigned to a role, then it "*can be acquired*" through that role. Axiom (2) states that all

users assigned to a role *can activate* that role. Axiom (3) states that if a user *u can activate* a role *r*, then all the permissions that *can be acquired* through *r can be acquired* by *u*. Thus, for the case where user *u* and permission *p* are assigned to *r*, the axioms imply that *u can acquire p*. Similarly, axiom (4) states that if there is a user session in which a user *u* has activated a role *r* then *u acquires* all the permissions that *can be acquired* through role *r*. We note that axioms (1) and (2) indicate that permission acquisition and role activation semantics is governed by explicit user-role and role permission assignments. Next, we define a predicate evaluation function eval over the status predicates and a projection operation $\Pi_{\pi 1, \pi 2, .., \pi m}$ over the evaluation of a predicate as follows.

Definition 3.1(eval, Π_i): *Let $s_t(alist)$ be a status predicate, where alist is a list of arguments $a_1, \ldots, a_i, \ldots, a_n$ associated with domains $D_1, \ldots, D_i, \ldots, D_n$, respectively ($\forall j \in \{1, ..n\}, D_j \in \{R, P, U, S, T\}$) . If DOM is the evaluation domain of $s_t(alist)$, then, we define evaluation function* eval *and projection operator* $\Pi_{\pi 1, \pi 2, .., \pi m}$ *as follows:*

- eval($s_t(alist)$)={$(x_1, \ldots, x_i, \ldots, x_n) \mid ((x_1, \ldots, x_i, \ldots, x_n) \in DOM) \wedge s_t(x_1, \ldots, x_i, \ldots, x_n)$}

- $\Pi_{\pi 1, \pi 2, .., \pi m}$eval($s_t(alist)$)={$(x_{\pi 1}, x_{\pi 2} .., x_{\pi m}) \mid \{\pi_1, \pi_2, \ldots, \pi_m\} \subseteq \{1, 2, \ldots, n\}$; $\forall x_{\pi i} \in D_{\pi i}$; and for all pairs $(x_1, x_2 \ldots, x_n), (y_1, y_2 \ldots, y_n) \in$ eval($s_t(a_1, \ldots a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$), $x_j = y_j$ for all all $j \in \{1, 2, \ldots, n\}/\{\pi_1, \pi_2, \ldots, \pi_m\}$; moreover, for all such *j* we replace the argument by its constant value in quotes, i.e., we denote a constant value $x \in D$ by "x" in the argument list}

Evaluation function eval returns the subset of the evaluation domain corresponding to the predicate that it evaluates. For instance, eval ($enabled(r, t)$) is a subset of domain ($R \times T$). Similarly, $\Pi_{\pi 1, \pi 2, .., \pi m}$ allows us to project the evaluation of a predicate over a particular argument indexed by *i*. For instance, $\Pi_1$eval($enabled(r, "t")$) returns the set of all roles that are enabled at time "*t*". Similarly, $\Pi_2$eval($enabled("r", t)$) returns the set of all time instants at which role "*r*" is enabled. Let us denote the set of all projection functions over the predicates defined in Table 2 as Π. Note that we can also have evaluation of the negation of the predicates of Table 2, for instance, $\Pi_1$eval ($\neg enabled(r, "t")$). Π^{-1} denotes the set of projection operators over negated predicates. Based on these projection operators and the original set of set elements Orig={R, U, P, S, T}, we build a framework for expressing exhaustive set of cardinality constraints as follows. Let OP $\in \{\cup, \cap, /\}$ be a set operation, then we have a generic set function *f* as follows:

1. $f \in (\Pi \cup \Pi^{-1})$;
2. $f = (f$ OP $X)$, where $X \subseteq E \in$ Orig;
3. $f = (f_1$ OP $f_2)$, where f_1 and f_2 are generic set functions.

We can express a cardinality constraint as ($|f|$ cop n), where $|f|$ is the number of elements in set *f*, cop $\in \{=, \neq, <, >, \geq, \leq\}$ is a comparator operator, and *n* is a positive number. Some examples of the cardinality constraint expressions are shown in Table 4. It is to be noted that while projection operators in Π make sense in general context (as shown in Table 4), those in Π^{-1} may not have a

clear meaning. Therefore, care should be taken in constructing cardinality constraints based on them. For example, the function $\Pi_2$eval($\neg active("u", r, "t")$) refers to a set of roles that are not active in any of user *u*'s sessions at time *t*. Hence, $|\Pi_2$eval($\neg active("u", r, "t")$)$| \leq n$ states that the number of roles *not* active in any of user *u*'s sessions at time *t* cannot be more than *n*. However, it is not clear whether it is *n* out of those that *u* can activate or that are in *R*. Depending upon application, distinction may need to be made a priori. For instance, we can say that, "*by default, out of those that u can activate*". Periodicity and duration constraints on a cardinality constraint $C = (|f|$ cop $n)$ can be simply defined using the GTRBAC temporal framework as (*I, P, C*), which indicates that the cardinality constraint is valid for each instant in intervals defined by (*I, P*), and as ([*I, P,| D*], $D_x C$), with D_x indicating the duration in which the cardinality constraint is valid.

Table 4. Examples of cardinality constraints

1	$|\Pi_1$eval($enabled(r, "t")$)$| \geq n$	Number of roles enabled at time "*t*" cannot be less than *n*
2	$|\Pi_1$eval($\neg enabled(r, "t")$)$| \leq n$	Number of roles disabled at time "*t*" cannot be more than *n*.
3	$|\Pi_2$eval($u\_assigned("u", r, "t")$)$| \leq n$	Number of roles assigned to "*u*" at time "*t*" cannot be more than *n*
4	$|\Pi_2$eval($can\_activate("u", r, "t")$)$| \leq n$)	Set of roles that *u* can activate at time *t* cannot be more than *n*.
5	(*Daytime*, $|\Pi_1$eval($u\_assignedSet(u,$ "Nurse", t)$| \leq n$) *indicates* Number of users assigned to Nurse role in *Daytime* cannot exceed *n*	

We note that some cardinality constraints of type $C = (|\Pi_{\pi 1, \pi 2, .., \pi m}$eval($s_t(plist)$)$|$ cop n) may not have direct application in a general RBAC framework. For example, $\Pi_1$eval($s\_active(u, "r", "s", "t")$) (set of users that have activated *r* in session *s* at time *t*) associates multiple users with the same session. Such cases may be useful if we consider a collaborative system where a session is created with multiple active users.

4. GTRBAC TRIGGERS AND CONTROL FLOW DEPENDENCCY CONSTRAINTS

Another set of constraints that are often needed in the commercial systems is that of dependencies between roles and other events associated with RBAC entities. GTRBAC provides a trigger mechanism that can be used to express some dependency constraints. However, there are much stricter forms of dependency constraints known as control flow dependency (CFD) constraints, which are needed in various applications. In this section, we extend the original GTRBAC triggers and define the CFD constraints using extended triggers

4.1 Extended-GTRBAC Triggers

The basic trigger expression of GTRBAC is of the form: ($E_1, \ldots, E_m, C_1, \ldots, C_k \rightarrow pr:E$ after Δt), where E_i is an event and C_i is a status condition [12]. Semantically, it means that the prioritized event *pr:E* with priority *pr* can take place Δt time units after the trigger fires. The definition, however, is limiting in the

following ways: (1) it only allows scenarios in which all the antecedent events $E_1,\ldots,$ and E_m occur at the same time and all the conditions $C_1,\ldots,$ and C_k hold; it does not allow capturing history information in which events are spread in the temporal dimension; (2) it does not allow specifying temporal intervals in which the occurence of an event E_i can take place, or a condition C_i is satisfied; (3) it is possible that in some cases a condition C_i must be valid for a specified duration before triggering the event E; such a requirement is also not captured by the current triggers; and (4) the current trigger considers that $E \neq s{:}\texttt{activate } r$ $\texttt{for } u$; this needlessly prevents specifying any preconditions for activation events. In some cases, an activation request may need to be granted only if certain conditions have been satisfied. We define the extended trigger form, which is temporally more expressive than the current GTRBAC triggers and does not have the above limitations, as follows:

Definition 4.1 (Extended Triggers): *The extended trigger expression has the following form*:

$$(E_1 \texttt{ in } \pi_1) \ \mathsf{op}_1 \ldots \mathsf{op}_{m-1} \ (E_m \texttt{ in } \pi_m) \ \mathsf{op}_m \ (C_1 \texttt{ in } \tau_1 \texttt{ for } d_1)$$
$$\mathsf{op}_{m+1} \ldots \mathsf{op}_{m+n-1} \ (C_n \texttt{ in } \tau_n \texttt{ for } d_n)$$
$$\rightarrow pr{:}E \texttt{ after } \Delta t \texttt{ for } \Delta d, \textit{ where}$$

- *E_is are simple event expressions or run time requests; and C_is are GTRBAC status expressions,*

- *$pr{:}E$ is a prioritized event expression with priority pr.*

- *If ($E = s{:}\texttt{activate } r \texttt{ for } u$) is an activation request at time $t_a \geq (t + \Delta t)$ then $\texttt{active}(u, r, s, t)$ is true in the interval $(t_a, t_a + \Delta d)$, provided that the trigger fires at time t,*

- *the trigger is fired if τ_i (π_i) is an interval such that there exists a $t \in \tau_i$ (π_i) at which C_i (E_i) becomes valid, and C_i remains valid for duration d_i; we simply write "C_i in τ_i" to mean that there exists a $t \in \tau_i$ at which C_i is valid for some duration; we write "C_i at t" ("E_i at t") instead of "C_i in τ_i" ("E_i in t") when τ_i (π_i) = (t, t); we write "C_i for d_i" to mean that C_i is valid for some duration d_i.*

- *Δt is the duration between the firing of the trigger and the occurence of the event E, and Δd is the duration for which the event E remains valid. If not specified, $\Delta t = 0$, and $\Delta d = \infty$ $\mathsf{op}_i \in \{\vee, \wedge\}$ and $\wedge$ has precedence over $\vee$. For simplicity, we use "," to denote the $\wedge$ operator and "|" to denote the $\vee$ operator.*

We note that the old trigger form cannot be used to specify the temporal information such as "E_i in π_i" or "C_i in τ_i". The earlier form is actually a special case of the extended form, in which all the antecedent events and conditions are associated with the same time instant. That is, for any t,

$$E_1 \texttt{ at } t,.., E_m \texttt{ at } t, C_1 \texttt{ at } t,.., C_k \texttt{ at } t$$
$$\rightarrow pr{:}E \texttt{ after } \Delta t, \qquad (a)$$

The duration information Δd associated with the triggered event E in the extended trigger simplifies specification but does not increase the expressive power over the earlier form. The following trigger:

$$(E_1 \texttt{ at } t,\ldots, E_m \texttt{ at } t, C_1 \texttt{ at } t,\ldots, C_k \texttt{ at } t \rightarrow$$
$$pr{:}E \texttt{ after } \Delta t \texttt{ for } \Delta d) \qquad (b)$$

is semantically equivalent to the combination of the following two old triggers

1. $E_1,\ldots, E_m, C_1,\ldots, C_k \rightarrow pr{:}E \texttt{ after } \Delta t$,

2. $E \rightarrow pr'{:}\mathsf{Conf}(E) \texttt{ after } \Delta d$, where $\mathsf{Conf}(E)$ is the conflicting event of E and $pr' \geq pr$.

We note that the triggers of form (*a*) (one with "$\texttt{at } t_n$" phrase) can represent the extended form (one with "$\texttt{in } \pi_m$" phrase), however, it is easy to see that the extended form achieves compaction in expression over the form (*a*). For instance, the extended trigger form without the "$\texttt{for } d_1$" part can only be represented by using multiple triggers of form (*a*), each with a permutation of time instants from $\pi_1, \pi_2,\ldots, \pi_m, \tau_1, \tau_2,\ldots, \tau_n$. Similar compaction is achieved by the use of the two logical operators.

Note that triggers allow GTRBAC events and status conditions only [12, 13]. However, it can easily be extended to include other events and conditions. For instance, condition C_i can be any predicate expression that evaluates contextual information that affects access control decisions. Consider the following trigger;

$$(Location(x) = \text{``}EmergencyRoom\text{''}) \mid (situation \text{ ()} = \text{``}LifeThreatening\text{''}) \rightarrow pr{:}E \texttt{ enable } \mathsf{EmergencyDoctor}$$

Here, if the room, indicated by variable x, is *EmergencyRoom*, or the current *situation* is *LifeThreatening* then the *EmergencyDoctor* role is enabled, thus capturing environmental context. Similarly, we can allow event E to be any system related event. With a predefined set of predicates to capture static as well as dynamic environmental conditions and events, the extended GTRBAC trigger framework can easily provide a very elaborate support for context-based access control.

4.2 Control Flow Dependency Constraints

Control flow dependency (CFD) constraints often occur in task-oriented systems and are stricter forms of dependency constraints than those that can be expressed using GTRBAC triggers. The following example illustrates such CFD constraints.

Example: Consider the following requirements: (1) a junior employee of an office is allowed to activate the Junior_Employee role in the system *only if* his manager has activated the Manager role; (2) whenever a system administrator makes some changes in the system, the activation of the SysAdmin role that he uses *must* enable the SysAudit role so that another user can activate the SysAudit role and log those changes. The SysAudit role may need to be activated by the user within the next τ minutes; (3) everyday, if both the roles SysAdmin and SysAudit are activated, then the SysAdmin role must be activated before the SysAudit role.

The three requirements imply (1) *pre-condition*, (2) *post-condition* and (3) *precedence* constraints. Next, we show that GTRBAC does not adequately model these constraints, but, we can semantically define CFDs in terms of these triggers.

4.2.1 Pre-condition Constraints

A *pre-condition* constraint between two events essentially implies that an event can occur *only if* the other event has already occurred and/or the required conditions have already become true, as in the first case above. The following trigger closely resembles the pre-condition constraint (1):

s:activate Manager for John → enable
Junior_Employee
(assume John is the manager)

However, the "*only if*" semantics of the *pre-condition* constraint requires that there be no other events that will enable the Junior_Employee role, i.e., the Junior_Employee role is not enabled if John does not activate the Manager role. This means the above trigger can enforce the *pre-condition* constraint only if we also enforce additional restriction that no other constraint or trigger allows the enabling of the Junior_Employee role. However, GTRBAC's trigger mechanism currently does not imply such an additional restriction, hence, it falls short in providing support for the *pre-condition* constraint. For instance, in addition to the above trigger, assume that we also have the following periodicity constraint:

Everyday between 9am and 6pm, enable Junior_Employee

Presence of this periodicity constraint does not allow the above trigger to enforce the *pre-condition* constraint as it allows the role to be enabled even if the Manager role is not enabled.

4.2.2 Post-condition Constraints

A *post-condition* constraint between two events essentially implies that *if* an event occurs or a condition is satisfied then the other event also *must* occur, as indicated in the second case in the example above. Here, *if* the SysAdmin role is enabled then the SysAudit role *must* also be enabled, otherwise, it may incur certain security risks. However, the activation of the SysAudit role may also be triggered by other events in the system. In essence, the *post-condition* constraint will not be enforced if there are some other triggers or constraints that do not allow the SysAudit role to be enabled even though the SysAdmin role has been enabled. Thus, it is easy to see that the following trigger:

enable SysAdmin → enable SysAudit

enforces the *post-condition* constraint *only if* the system additionally makes sure that there are no other constraints or triggers that prohibits enabling of the SysAudit role when this trigger fires; this cannot be expressed using GTRBAC triggers

4.2.3 Precedence Constraints

A *precedence* constraint is said to exist between two events if there is a condition that *if the two events occur then one must always precede the other*, as shown in requirement (3). Another real world scenario in which such a *precedence* semantics applies is a pair of tasks involving *authorizing a check* and *cashing* it. It is easy to see that such *precedence* semantics is not enforced by triggers alone.

4.2.4 Definitions of CFD Constraints

Next, we formalize syntax and semantics of the CFD constraints in GTRBAC using triggers. In the definitions we will use (t_s, t_e),

such that (t_s, t_e) is in an interval of (I, P), or (t_s, t_e) is some duration D. For $(t_s, t_e) = D$, t_s is the time instant when D starts and is non-deterministic. A constraint $c = (D, C)$ needs to be enabled by a trigger or a runtime event [12]. Assume that T is the set of all GTRBAC constraints and Causes($c, pr:E, t$) is a predicate that evaluates to *true* if there is a constraint c in T which causes event $pr:E$ to fire at time t. Furthermore, we use Y to denote the left hand side of a trigger expression, *i.e.,*

$$Y =$$

E_1 in $\pi_1, ..., E_m$ in π_m, C_1 in τ_1 for $d_1,..., C_n$ in τ_n for d_n

The following precedence rule is applied in a GTRBAC system - if there are conflicting pairs of events (e.g., assign and deassign, activate and deactivate, etc.) then the negative event takes precedence (e.g., deassign takes precedence over assign) if the priority of the two events are the same; otherwise the higher priority event takes precedence.

Definition 4.2 (*Pre-condition constraint*): *The pre-condition constraint is expressed as* ([$I, P|D,$] pre, $Y, pr:E$ after Δt for Δd). *Semantically, to say that* ([$I, P|D,$] pre, $Y, pr:E$ after Δt for Δd) $\in T$ *is equivalent to saying that:*

(1) $(Y \rightarrow pr:E$ after Δt for $\Delta d)_t \in T$ *is an* extended-trigger, *and*

(2) $\neg\exists\ c \in T$ *s.t.* $(\forall t_x \in (t + \Delta t,\ t + \Delta t + \Delta d)$ *and* $pr' \geq pr$, Causes($c, pr': E, t_x$)) *is true for* $pr' \geq pr$.

Definition 4.3 (*Post-condition constraint*): *The post-condition constraint is expressed as* ([$I, P|D,$] post, $Y, pr:E$ after Δt). *Semantically, to say that* ([$I, P|D,$] post, $Y, pr:E$ after Δt for Δd) *is in T is equivalent to saying that*:

(1) $(Y \rightarrow pr:E$ after Δt for $\Delta d)_t \in T$ *is an* extended-trigger;

(2) (2) $\neg\exists\ c \in T$ *s.t.* $\forall\ t_x \in (t + \Delta t,\ t + \Delta t + \Delta d)$, Causes($c, pr':$ Conf(E), t_x) *is true for* $pr' \geq pr$.

Note that condition (2) in each definition ensures that the additional condition required for the two CFDs, as discussed earlier, are enforced. Next, we define a precedence constraint, which relates two events. We can also define a CFD with "*if and only if*" by combining above two constraints. Next we define precedence constraint that essentially relates two events.

Definition 4.4 (*Precedence constraint*): *Let $pr_1:E_1$ and $pr_2:E_2$ be prioritized events such that* ([$I, P|D,$] prec, $pr_1:E_1, pr_2:E_2$ after Δt), *i.e. $pr_2:E_2$ is precedent on $pr_1:E_1$. Then, for all t such that $t \in (t_s, t_e)$*:

([$I, P|D,$] precedence, $pr_1:E_1, pr_2:E_2$) $\rightarrow$

(for each pair $c_1, c_2 \in T$, Causes($c_1, pr:E_1, t_1$) $\wedge$ Causes($c_2, pr:E_2, t_2$) $\rightarrow (t_e \leq t_1 + \Delta t \leq t_2 \leq t_s))$

The safety notion introduced in TRBAC identifies scenarios that have ambiguous execution semantics, for example, the existence of a cycling dependency among events through triggers [12]. The safety checking algorithm can be easily extended to identify the violation of the CFD constraint by introducing extra checks to ensure that additional restrictions are enforced for the CFDs.

5. TIME BASED SEPRATION OF DUTY

Separation of Duty (SoD) policies have been found to be very crucial for securing commercial applications. Role-based systems are particularly very beneficial for expressing and enforcing such policies. Various SoDs have been identified in the literature [8, 14, 17, 18]. However, all earlier researches focus on SoDs in a non-temporal environment. The CFD constraints introduced in previous section can also be used to define SoD constraints that are based on access history, such as history-dependent SoD, order-dependent SoD, object-based SoD, which are identified in [18], as the triggers can capture timing information. In this section, we define various SoD constraints that cannot be captured

Table 5. Enabling time and Assignment SoDs

SoD Type	Expression (*SoD*)	Semantics $\forall u, u_1, u_2 \in U, \forall r, r_1, r_2 \in R, \forall p, p_1, p_2 \in R, \forall t \in Sol(I, P),\ (u_1 \neq u_2),\ (r_1 \neq r_2) \text{ and } (p_1 \neq p_2)$ *the following holds*:
Enabling/Disabling SoD		
EN-SoD	$(I, P, \mathsf{EN}, \mathsf{R})$	$SoD \wedge \texttt{enabled}(r_1, t) \rightarrow \neg\texttt{enabled}(r_2, t)$
		No two roles in R can be *enabled* at the same time in interval *(I, P)*
(DIS-SoD)	$(I, P, \mathsf{DIS}, \mathsf{R})$	$SoD \wedge \texttt{disabled}(r_1, t) \rightarrow \neg\texttt{disabled}(r_2, t)$
		No two roles in R can be *disabled* at the same time in interval *(I, P)*
User-Role assignment/de-assignment SoDs		
UAS-SoD$_1$	$(I, P, \mathsf{UAS}_1, \mathsf{U}, \mathsf{R})$	$SoD \wedge \texttt{u\_assigned}(u, r_1, t) \rightarrow \neg\texttt{u\_assigned}(u, r_2, t)$
		No two roles in R can be *assigned* to a user in U at the same time in interval *(I, P)*
UAS-SoD$_2$	$(I, P, \mathsf{UAS}_1, \mathsf{U}, \mathsf{R})$	$\forall r \in R,\ SoD \wedge \texttt{u\_assigned}(u_1, r, t) \rightarrow \neg\texttt{u\_assigned}(u_2, r, t)$
		No two users in U can be *assigned* to a role in R at the same time in interval *(I, P)*.
UAS-SoD$_3$	$(I, P, \mathsf{UAS}_3, \mathsf{U}, \mathsf{R})$	$SoD \wedge \texttt{u\_assigned}(u_1, r_1, t) \rightarrow \neg\texttt{u\_assigned}(u_2, r_2, t)$
		Different users in U cannot be *assigned* different roles in R at the same time in interval *(I, P)*
UAS-SoD$_4$	$(I, P, \mathsf{UAS}_4, \mathsf{U}, \mathsf{R})$	$SoD \leftrightarrow \text{UAS-SoD}_2 \wedge \text{UAS-SoD}_3$
		Roles in R can be assigned to only one of the users in U at the same time in interval *(I, P)*
UAS-SoD$_5$	$(I, P, \mathsf{UAS}_5, \mathsf{U}, \mathsf{R})$	$SoD \leftrightarrow \text{UAS-SoD}_1 \wedge \text{UAS-SoD}_3$
		Users in U can be *assigned* only one of the roles in R at the same time in interval *(I, P)*
UAS-SoD$_6$	$(I, P, \mathsf{UAS}_6, \mathsf{U}, \mathsf{R})$	$SoD \leftrightarrow \text{UAS-SoD}_1 \wedge \text{UAS-SoD}_2$
		A role in R can be assigned to only one user in U (and vice versa) at the same time in interval *(I, P)*
Role-Permission assignment/de-assignment SoDs		
PAS-SoD$_1$	$(I, P, \mathsf{PAS}_1, \mathsf{P}, \mathsf{R})$	$\forall p \in P,\ SoD \wedge \texttt{p\_assigned}(p, r_1, t) \rightarrow \neg\texttt{p\_assigned}(p, r_2, t)$
		No two roles in R can be *assigned* a permission in P at the same time in interval *(I, P)*
PAS-SoD$_2$	$(I, P, \mathsf{PAS}_2, \mathsf{P}, \mathsf{R})$	$\forall r \in R,\ SoD \wedge \texttt{p\_assigned}(p_1, r, t) \rightarrow \neg\texttt{p\_assigned}(p_2, r, t)$
		No two permissions in P can be *assigned* to a *role* in R at the same time in interval *(I, P)*.
PAS-SoD$_3$	$(I, P, \mathsf{PAS}_3, \mathsf{P}, \mathsf{R})$	$\forall p_1, p_2 \in P,\ SoD \wedge \texttt{p\_assigned}(p_1, r_1, t) \rightarrow \neg\texttt{p\_assigned}(p_2, r_2, t)$
		Different permissions in P cannot be *assigned* to different roles in R at the same time in interval *(I, P)*
PAS-SoD$_4$	$(I, P, \mathsf{PAS}_4, \mathsf{P}, \mathsf{R})$	$SoD \leftrightarrow \text{PAS-SoD}_2 \wedge \text{PAS-SoD}_3$
		Roles in R can be assigned only one of the permissions in P at the same time in interval *(I, P)*.
PAS-SoD$_5$	$(I, P, \mathsf{PAS}_5, \mathsf{P}, \mathsf{R})$	$SoD \leftrightarrow \text{PAS-SoD}_1 \wedge \text{PAS-SoD}_3$
		Permissions in P can be assigned to only one of the roles in R at the same time in interval *(I, P)*
PAS-SoD$_6$	$(I, P, \mathsf{PAS}_6, \mathsf{P}, \mathsf{R})$	$SoD \leftrightarrow \text{PAS-SoD}_1 \wedge \text{PAS-SoD}_2$
		Permissions in P can be assigned to only one of the roles in R at the same time in interval *(I, P)*

by such CFD constraints, and some of which correspond to those already identified in the literature. These SoDs will be defined with respect to GTRBAC status predicates introduced in Table 2.

5.1 Role Enabling/Disabling Time SoDs
(predicates: `enabled`/`disabled`)

The SoD constraints related to enabling and disabling events are shown in Table 5. *EN-SoD* indicates that roles from a given role set cannot be *enabled* at the same time. If there are role enabling events that attempt to enable more than one role at the same time then the enforcement mechanism must use some criteria to enable only one of the roles. SoD *DIS-SoD* is defined with respect to the role disabling event. The difference between them is that *EN-SoD* does not allow all the roles to be enabled at the same time but allows them to be disabled at the same time, whereas *DIS-SoD* allows all to be enabled at the same time but does not allow them to be disabled at the same time. Role enabling SoDs (*EN-SoD*) are cases where limiting access is a primary concern. Similarly, role disabling SoD (*DIS-SoD*) is more useful in cases where availability is the key concern, for example, in a hospital, a requirement may state that "*Both the* Nurse *and* Doctor *roles cannot be disabled at the same time*". These SoDs can be expressed in the form of cardinality constraints introduced earlier, e.g., *EN-SoD* can be expressed as $|\Pi_1$eval$(enabled(r, t)) \cap R| \leq 1$. Similarly, other SoDs defined below can also be expressed in this form; however, we use uniformly the implication rule to provide the semantics of these SoDs.
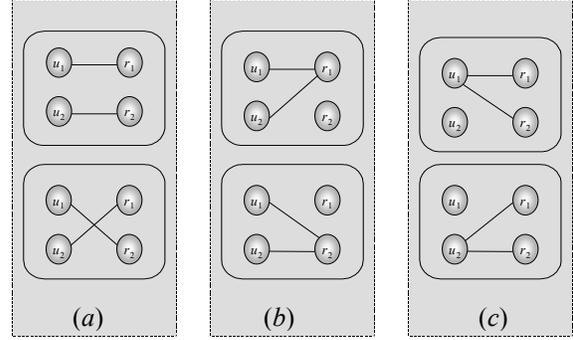
5.2 Assignment SoDs
(predicates: `u_assigned`, `p_assigned`)

Table 5 defines various user-role and role-permission assignment time constraints. *UAS-SoD$_1$* indicates that multiple roles from R cannot be *assigned* to a user in U at the same time. Accordingly, the roles from R can be assigned to any user not included in U. In other words, this implies that the role set R has conflicting semantics only with respect to the user set U. Allowing specification of such a set of conflicting roles *with respect to* a particular user set provides the benefit of expressing fine-grained SoD constraints. *UAS-SoD$_2$* states that different users of U cannot be assigned to a role in R, i.e., the users in U are conflicting with respect to role set R. Figure 1 depicts various assignment combinations that are not allowed by the user-role assignment constraints for U = $\{u_1, u_2\}$ and R = $\{r_1, r_2\}$. Here, a line from *u* to *r* indicates that *u* is assigned to *r*. In general, set U can be expected to be the set Users. In figure 1, *UAS-SoD$_1$* does not allow assignment combinations depicted in (*c*), whereas, *UAS-SoD$_2$* does not allow assignment combinations shown in (*b*).

UAS-SoD$_3$ states that different users from set R cannot be assigned to different roles. Here, U and R have conflicting semantics with respect to each other. Note that the notion of conflict here is slightly different from that of *UAS-SoD$_1$* and *UAS-SoD$_2$*. However, this constraint allows a single user from U to be assigned to multiple roles of R, and a single role from R to be assigned to multiple users. *UAS-SoD$_3$* does not allow assignment scenario depicted in (*a*). In real world scenario, U of *UAS-SoD$_3$* may refer to a set of employees who are related. Assignment of any two of these employees to different roles will allow them to commit fraud. If, for instance, one employee is assigned to role

AthorizationManager (that authorizes checks) and another is assigned to role CashingClerk (for cashing authorized checks), they can easily commit fraud. Another practical scenario in which this constraint can be applicable is when a set of roles represents subtasks of a bigger task, with the constraint that the different users of U cannot carry out different subtasks.

SoD	Doesn't Allow	SoD	Doesn't Allow
UAS-SoD$_1$	(*c*)	*UAS-SoD$_4$*	(*a*), (*b*)
UAS-SoD$_2$	(*b*)	*UAS-SoD$_5$*	(*a*), (*c*)
UAS-SoD$_3$	(*a*)	*UAS-SoD$_6$*	(*b*), (*c*)

Figure 1. User-assignment SoDs with U=$\{u_1,u_2\}$ and R=$\{r_1,r_2\}$

UAS-SoD$_4$, *UAS-SoD$_5$* and *UAS-SoD$_6$* can be derived as combinations of earlier SoDs, as shown in Table 5. We note that, although *UAS-SoD$_3$* allows defining constraints such as all the subtask roles need be assigned to the same user, it also allows the assignment scenario (*b*), which may not be relevant with regards to such a requirement. *UAS-SoD$_4$*, for instance, omits the possibility of assigning all the users to the same subtask role rendering the overall task un-accomplishable. As shown in the figure, *UAS-SoD$_5$* prevents the set of assignments of type shown in (*a*) and (*c*) – i.e. it allows multiple users to be assigned to only one of the roles, such as those in Figure 1(b). That is, as soon as one of the roles, say role *r*, is assigned to a user then none of the users can be assigned to any other roles; however, role *r* can be assigned to any number of users. An example of application of *UAS-SoD$_5$* is the assignment of a given set of consultants (set U) to the same consultancy duty (assignment of all the users to the role ConsultantOfCompanyA).

The role-permission assignments have semantics similar to that of the user-role assignments. Note that, here, we are using the notion of conflicting permission, for example in *PAS-SoD$_2$*.

5.3 Role Activation Time SoDs
(predicate: `active`)

Activation time constraints are listed in Table 6. *ACT-SoD$_1$* implies that activation of conflicting roles at the same time in the same session or different sessions by a user in U is not allowed. Figure 2 depicts the scenarios for U = $\{u_1, u_2\}$ and R = $\{r_1, r_2\}$ when both the roles are active. Here, *s*: $u_1(r_1, r_2)$ indicates that roles r_1 and r_2 are active in u_1's session *s*. *ACT-SoD$_1$* does not
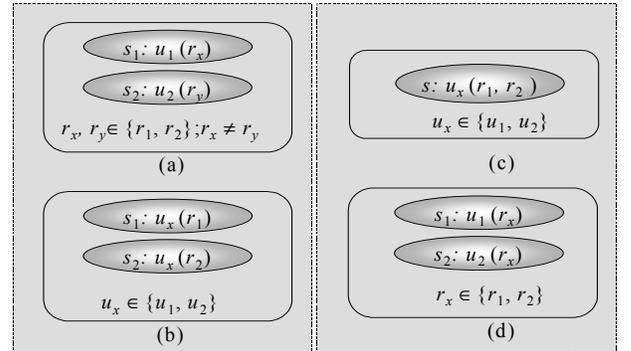
Table 6. Activation time SoDs

Type	*SoD*	Semantics $\forall u, u_1, u_2 \in U, \forall s, p_1, p_2 \in R, \forall r, r_1, r_2 \in R, \forall t \in Sol(I, P), (u_1 \neq u_2), (r_1 \neq r_2)$ and $(p_1 \neq p_2)$, *the following holds*:
Activation Time SoDs		
ACT-SoD$_1$	$(I, P, \text{ACT-SoD}_1, U, R)$	$\forall u \in U, SoD \wedge \texttt{active}(u, r_1, t) \rightarrow \neg\texttt{active}(u, r_2, t)$
	No two roles in R can be in *active* state in session(s) of a user in U at the same time in interval *(I, P)*	
ACT-SoD$_2$	$(I, P, \text{ACT-SoD}_2, U, R)$	$\forall u \in U, SoD \wedge \texttt{active}(u_1, r, t) \rightarrow \neg\texttt{active}(u_2, r, t)$
	No two users in U can have a role in R *active* at the same time in interval *(I, P)*	
ACT-SoD$_3$	$(I, P, \text{ACT-SoD}_3 \ U, R)$	$\forall u_1, u_2 \in U, SoD \wedge \texttt{active}(u_1, r_1, t) \rightarrow \neg\texttt{active}(u_2, r_2, t)$
	No two users in U can have two different roles in R *active* at the same time in interval *(I, P)*	
ACT-SoD$_4$	$(I, P, \text{ACT-SoD}_4, U, R)$	$\forall u \in U, \forall s \in S, SoD \wedge \texttt{active}(u, r_1, s, t) \rightarrow \neg\texttt{active}(u, r_2, s, t)$
	Two roles in R cannot be in *active* state at the same time in a single session *of* a user in U in interval *(I, P)*	
ACT-SoD$_5$	$(I, P, \text{ACT-SoD}_5, U, R)$	$\forall u \in U, \forall s_1, s_2 \in S, SoD \wedge \texttt{active}(u, r_1, s_1, t) \rightarrow \neg\texttt{active}(u, r_2, s_2, t)$
	No two sessions of a user in U can have two roles in R *active* at the same time in interval *(I, P)*	
ACT-SoD$_6$	$(I, P, \text{ACT-SoD}_6, U, R)$	$SoD \leftrightarrow \text{ACT-SoD}_2 \wedge \text{ACT-SoD}_4$
	Roles in R can be *active* in a session(s) of only one of the users in U at the same time in interval *(I, P)*	
ACT-SoD$_7$	$(I, P, \text{ACT-SoD}_7, U, R)$	$SoD \leftrightarrow \text{ACT-SoD}_5 \wedge \text{ACT-SoD}_6$
	Roles in R can be *active* in a single session of only one of the users U at the same time in interval *(I, P)*	

allow activation combinations depicted in 2(*b*) and 2(*c*). *ACT-SoD$_2$* does not allow activation of a role by conflicting users at the same time. Similarly, *ACT-SoD$_3$* does not allow conflicting roles to be active in different users' sessions, as depicted in 2(*a*).

ACT-SoD$_4$ does not allow 2(*c*), i.e. it prevents activation of the conflicting roles in the same session simultaneously. *ACT-SoD$_5$* does not allow scenarios depicted in 2(*b*), i.e. it prevents activation of the conflicting roles in the different sessions of the same user simultaneously. *ACT-SoD$_6$* and *ACT-SoD$_7$* are combinations of the earlier SoDs, as indicated in the table.

Figure 3 illustrates the usefulness of SoD constraints *ACT-SoD$_1$* - *ACT-SoD$_5$*. In Figure 3(a), roles r_1 and r_2 have a common set of permissions. Now suppose, we allow users u_1 and u_2 assigned to roles r_1 and r_2, respectively, to activate the respective roles at the same time. As the *read* permission on the object O_x is available to both the roles, the information that each role writes to object O_x is visible to the other. Hence, O_x opens up the information flow channel between the two users. Common permissions like these may occur explicitly, in a non-hierarchical case, or implicitly through an *I*-hierarchy relation such as in the one shown in Figure 3(*a*). In a non-hierarchical case, declaring the two roles as conflicting and applying *ACT-SoD$_3$* is a straightforward solution if such information flow needs to be contained. Kuhn [14] indicates that roles that have common permissions cannot form a conflicting pair. We believe that such semantics is too restrictive. Moreover, with that semantics, we indirectly impose mutual exclusion on the permission sets of the two roles. This may not be

what is required in the practice. For example, there may be situations where only the private permissions of a pair of roles are conflicting, but the roles may have a common set of permissions.

SoD	Does not allow	SoD	Does not allow
ACT-SoD$_1$	(*b*), (*c*)	*ACT-SoD$_5$*	(*b*)
ACT-SoD$_2$	(*d*)	*ACT-SoD$_6$*	(*a*), (*d*)
ACT-SoD$_3$	(*a*)	*ACT-SoD$_7$*	(*a*), (*b*) , (*d*)
ACT-SoD$_4$	(*c*)		

Figure 2. Activation time SoDs for $U = \{u_1, u_2\}$ and $R = \{r_1, r_2\}$

In such scenarios, conflicting roles imply conflicting set of private permissions only. For example, an Employee role in general can be used to group the basic set of permissions available to all the employees of an organization. We may have two roles such as AuthorizationManger and CashingClerk, which are both senior to Employee but are considered to be conflicting; however, conflicting semantics is obviously limited to their private permissions rather than the common permissions inherited from Employee. Kuhn's strict mutual exclusion semantics necessitates partitioning even such basic roles in order to enforce mutual exclusion over the total sets of permissions associated with the two roles. However, sometimes in such a scenario, common permissions may create information flow when the private permissions of the two roles conflict. In Figure 3(a), for example, when user u_1 activates role r_1, and u_2 activates role r_2 at the same time, they can exchange information contained in O_{1i} and O_{2j} to each other. *ACT-SoD*$_3$ prevents such possibilities.
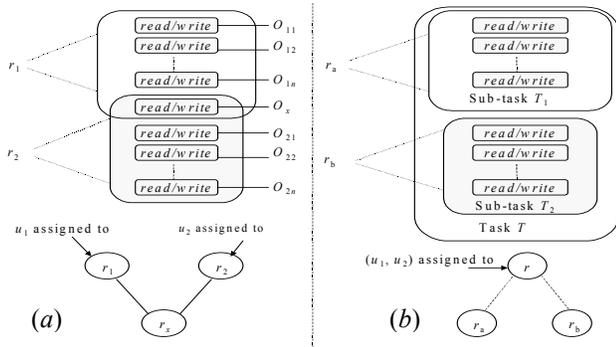

Figure 3. Session time SoD examples

ACT-SoD$_1$ can be used in cases where a user needs to be restricted from acquiring permissions that give him/her enough power to carry out some activities. For example, Figure 3(*b*) shows two roles that contain permissions for the subtasks of a bigger task. If we want that the same user do not carry out the two subtasks, then we can employ *ACT-SoD*$_1$ constraint. Furthermore, the roles may be organized as an *A*-hierarchy, where role *r* represents the actual

task role and is the senior of roles r_1 and r_2 that represent sub-tasks 1 and 2. If users from U are assigned to *r* and the *ACT-SoD*$_1$ is defined with respect to R = {r_1, r_2}, then the task can only be performed by two different users of U working at the same time.

ACT-SoD$_2$ can be used to enforce the requirement that a particular task can be performed by only one person at a time by assuming the task role. *ACT-SoD*$_4$ limits the access capability of a user by not allowing the conflicting roles to be active in a single user session. Its usefulness comes from the fact that a session in RBAC system is semantically the same as a subject in traditional access control models (DAC, MAC, etc.) [17]. Similarly, *ACT-SoD*$_5$ prevents a user from simultaneously acting as two subjects.

5.4 Possibilistic Activation Time SoDs

   (**predicates**: `can_activate`, `can_be_acquired`)
We also define SoDs based on the `can_activate` predicate, as shown in Table 7. *CACT-SoD*$_1$ prevents all possible activation of conflicting roles by users in U. Note that the purpose of *UAS-SoD*$_1$ is essentially to prevent activation of conflicting roles by a user by not allowing explicit assignments to conflicting roles in the first place. For example, when U = {u_1, u_2} and R = {r_1, r_2}, we can prevent the possibility of activation of both the roles by a user by explicitly denying assignments to conflicting roles by using *UAS-SoD*$_1$. Let's assume that because of this constraint u_1 is assigned to r_1 but not r_2. Now, assume that there is a role *x* such that *x* is senior to r_2 with respect to an *A*-hierarchy, i.e. any user assigned to *x* can also activate role r_2, as depicted in Figure 4(a). Now, if we allow the assignment of u_1 to *x*, the purpose of preventing u_1 from activating both r_1 and r_2 at the same time is not fulfilled. This is because, the *A*-hierarchy between *s* and r_2 makes the predicate `can_activate`(u_1, r_2, *t*) true [13], hence allowing u_1 to activate r_2 even when u_1 is already assigned to r_1. Therefore, when we have role hierarchies, implicit assignment may be possible through the use of `can_activate`(*u*, *r*, *t*) predicate (we refer to [13] for more details), which may make it possible for a user to activate conflicting roles even if the constraint *UAS-SoD*$_1$ is already employed. *CACT-SoD*$_1$ prevents such scenarios, i.e. it prevents both implicit and explicit assignments of a user to

Table 7. Possibilistic role activation SoDs

Possibilistic Activation (`can_activate`) SoDs		
Type	**SoD**	**Semantics** $\forall u, u_1, u_2 \in$ U, $\forall r, r_1, r_2 \in$ R, $\forall t \in Sol(I, P)$, $(u_1 \neq u_2)$ and $(r_1 \neq r_2)$, *the following holds*:
CACT-SoD₁	(*I*, *P*, CACT-SoD$_1$, U, R)	$\forall u \in U$, $SoD \wedge$ `can_activate`(*u*, r_1, *t*) $\rightarrow \neg$`can_activate`(*u*, r_2, *t*)
	No two roles in R can be activated by a user in U at the same time in interval *(I, P)*.	
CACT-SoD₂	(*I*, *P*, CACT-SoD$_1$, U, R)	$\forall u_1, u_2 \in U$, $SoD \wedge$ `can_activate`(u_1, r_1, *t*) $\rightarrow \neg$`can_activate`(u_2, r_2, *t*)
	No two users in U can activate two roles in R at the same time in interval *(I, P)*	
CACT-SoD₃	(*I*, *P*, CACT-SoD$_3$, U, R)	$SoD \leftrightarrow$ CACT-SoD$_1$ $\wedge$ CACT-SoD$_2$
	Users in U *can activate* only one of the roles in R at the same time in interval *(I, P)*	
CACT-SoD₄	(*I*, *P*, CACT-SoD$_4$, U, R)	$\forall u \in U$, $SoD \wedge$ `can_activate`(*u*, r_1, *s*, *t*) $\rightarrow \neg$`can_activate`(*u*, r_2, *s*, *t*)
	No two roles in R can be activated by a user in U in a single session *s* at the same time in interval *(I, P)*.	
	Users in U *can activate* only one of the roles in R in a single session *s* at the same time n interval *(I, P)*	

Table 8. Possibilistic permission acquisition SoDs

Possibilistic User-Permission Acquisition (`can_be_acquired` and `can_acquire`) SoDs		
Type	**SoD**	**Semantics** $\forall u, u_1, u_2 \in U, \forall r, r_1, r_2 \in R, \forall p, p_1, p_2 \in P, \forall t \in Sol(I, P), (u_1 \neq u_2), (r_1 \neq r_2)$ and $(p_1 \neq p_2)$ *the following holds*:
CACQ_SoD$_1$	$(I, P, \text{CACQ}_1, U, P, R)$	$SoD \wedge \texttt{can\_acquire}(u_2, r_1, t) \rightarrow \neg \texttt{can\_acquire}(u, p, r_2, t)$
	A permission in P cannot be acquired by a user in U through different roles in R at the same time	
CACQ_SoD$_2$	$(I, P, \text{CACQ}_2, U, P, R)$	$\forall u \in U, \forall p_1, p_2 \in P, SoD \wedge \texttt{can\_acquire}(u, p_1, r_1, t) \rightarrow \neg \texttt{can\_acquire}(u, p_2, r_2, t)$
	No two permissions in P can be acquired by a user in U through roles in R at the same time	
CACQ_SoD$_3$	$(I, P, \text{CACQ}_3, U, P, R)$	$\forall u_1, u_2 \in U, \forall p \in P, SoD \wedge r\_\texttt{can\_acquire}(u_1, p, r_1, t) \rightarrow \neg \texttt{can\_acquire}(u_2, p, r_2, t)$
	No two users in U can acquire a permission in P through different roles at the same time	
CACQ_SoD$_4$	$(I, P, \text{CACQ}_4, U, P, R)$	$\forall u_1, u_2 \in U, \forall p_1, p_2 \in P, SoD \wedge r\_\texttt{can\_acquire}(u_1, p_1, r_1, t) \rightarrow \neg r\_\texttt{can\_acquire}(u_2, p_2, r_2, t)$
	No two users in U can acquire different permissions in U through two roles at the same time	
CACQ_SoD$_5$	$(I, P, \text{CACQ}_5, U, P)$	$\forall p, SoD \wedge \texttt{can\_acquire}(u_1, p, t) \rightarrow \neg \texttt{can\_acquire}(u_2, p, t)$
	No two users in C *can acquire* a *permission* in P at the same time.	
CACQ_SoD$_6$	$(I, P, \text{CACQ}_6, U, P)$	$\forall p_1, p_2 \in P, SoD \wedge \texttt{can\_acquire}(u_1, p_1, t) \rightarrow \neg \texttt{can\_acquire}(u_2, p_2, t)$
	No two permissions in P can be *acquired* by the different users in U at the same time	
CACQ_SoD$_7$	$(I, P, \text{CACQ}_7, U, P, R)$	$\forall p \in P, \forall r \in R, SoD \wedge \texttt{can\_acquire}(u_1, p, r, t) \rightarrow \neg acquires(u_2, p, r, t)$
	A permission in P cannot be *acquired* by different users in U through the same role in R at the same time.	
CACQ_SoD$_8$	$(I, P, \text{CACQ}_8, U, P, R)$	$\forall p_1, p_2 \in P, \forall r \in R, SoD \wedge \texttt{can\_acquire}(u_1, p_1, r, t) \rightarrow \neg acquires(u_2, p_2, r, t)$
	No two permissions in *P* can be *acquired* by two users through the same role in *R* at the same time.	
CACQ_SoD$_9$	$(I, P, \text{CACQ}_9, R, P)$	$\forall r \in R, SoD \wedge \texttt{can\_be\_acquired}(p_1, r, t) \rightarrow \neg \texttt{can\_be\_acquired}(p_2, r, t)$
	No two permissions in P can be *acquired* through a *role* in R at the same time.	
CACQ_SoD$_{10}$	$(I, P, \text{CACQ}_{10}, R, P)$	$\forall r_1, r_2 \in R, SoD \wedge \texttt{can\_be\_acquired}(p_1, r_1, t) \rightarrow \neg \texttt{can\_be\_acquired}(p_2, r_2, t)$
	Different *permissions* in P cannot be *acquired* through different roles in R at the same time.	
CACQ_SoD$_{11}$	$(I, P, \text{CACQ}_{11}, U, P)$	$\forall u \in U, SoD \wedge \texttt{can\_acquire}(u, p_1, t) \rightarrow \neg \texttt{can\_acquire}(u, p_2, t)$
	A user in U cannot acquire different *permissions* in P at the same time.	
CACQ_SoD$_{12}$	$(I, P, \text{CACQ}_{12}, U, P)$	$\forall u_1, u_2 \in U, SoD \wedge \texttt{can\_acquire}(u_1, p_1, t) \rightarrow \neg \texttt{can\_acquire}(u_2, p_2, t)$
	No two users in U can acquire different *permissions* in P at the same time.	
CACQ_SoD$_{13}$	$(I, P, \text{CACQ}_{13}, U, P, R)$	$\forall r \in R, \forall u, SoD \wedge \texttt{can\_acquire}(u, p_1, r, t) \rightarrow \neg \texttt{can\_acquire}(u, p_2, r, t)$
	Permissions in P cannot be *acquired* by a *user* in U through a role in R at the same time.	

conflicting roles. Furthermore, *CACT-SoD$_2$* is an activation-time counterpart of *UAS-SoD$_3$*, and *CACT-SoD$_3$* is the activation-time counterpart of *UAS-SoD$_5$*. *CACT-SoD$_4$* is a session specific counterpart of *CACT-SoD$_1$*.

Note that one way to prevent the scenarios depicted in Figure 4(a) is to consider that r_1 is in conflict with all the roles hierarchically superior to r_2, as in [8, 14]. However, this approach is very restrictive, and makes the task of properly designing a role hierarchy very difficult.

5.5 Possibilistic Permission Acquisition SoDs
(**predicates**: `can_acquire`, `can_be_acquired`)
Table 8 lists the possibilistic permission acquisition SoDs. *CACQ-SoD$_1$* prevents the acquisition of permissions through the conflicting roles that will not be caught by *PAS-SoD$_3$*, similar to the way *CACT-SoD$_1$* prevents the activation of conflicting roles not prevented by *ACT-SoD$_1$*. That is, constraint *CACQ-SoD$_1$* employs the "*can acquire*" semantics and hence captures both explicit and implicit role-permission assignments. Note that *PAS-*

SoD_3 can prevent the acquisition of permissions through the conflicting roles by a user by restricting explicit role-permission assignment. However, permission acquisition may also be allowed through the implicit role-permission assignment because of some hierarchical relations. For example, let us consider $P = \{p_1, p_2\}$ and $R = \{r_1, r_2\}$. Suppose, we have the SoD constraint *PAS-SoD₃*, then the same permission in P cannot be assigned to the two roles. Provided there are no hierarchies in the system, the effect (and hence the purpose) of this SoD constraint is that the same permission is not acquired through two roles even if a user is allowed to activate them both. Now, assume there is a role x such that r_2 is senior of x with respect to an *I*-hierarchy, as shown in Figure 4(b). Suppose we allow the assignment of p_1 to x. Furthermore, suppose we have the following assignment: p_1 is assigned to r_1, and hence p_1 is not assigned to r_2 by virtue of the constraint *PAS-SoD₂*. But, as p_1 is also assigned to x, and $(r_2 \geq^I x)$, p_1 is also implicitly assigned to r_2, the SoD constraint *PAS-SoD₂* does not prevent p_1 being acquired through role r_2 using hierarchy semantics. *CACQ-SoD₁* prevents such permission-acquisitions through implicit assignments. *CACQ-SoD₂* is to *CACQ-SoD₁* the way *UAS-SoD₃* was to *UAS-SoD₁*.
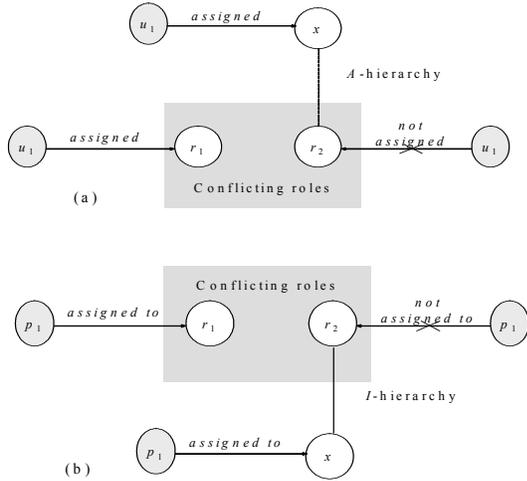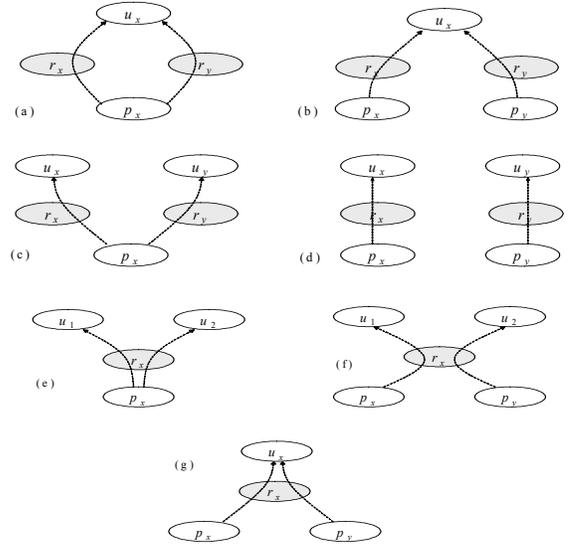
Figure 4. Implication of SoDs in presence of hierarchy

CACQ-SoD₃ allows acquisition of permissions in R by users through only one of the conflicting roles, whereas *CACQ-SoD₄* does not allow different users to acquire different permissions through the conflicting roles. Similar to the *CACQ-SoD₁* constraint, *CACQ-SoD₅* prevents acquisition of permissions that is allowed by both explicit and implicit assignments. As depicted in the table of Figure 4, *CACQ-SoD₅* prevents conflicting users from acquiring a permission of P through the same role, as in (a), or though different roles, as in (b). *CACQ-SoD₆*, on the other hand does not allow two separate permissions to be acquired by conflicting users neither through the same role (as in (c)) nor the different roles (as in (d)). *CACQ-SoD₇* prevents conflicting users from acquiring a permission in P through the same role in R at the same time and hence does not allow case (a). Similarly, the table in Figure 5 shows the cases depicted in Figure 5 that are not allowed by *CACQ-SoD₈*, *CACQ-SoD₃* and *User-SoD₄*. Various combinations of these SoDs define the SoDs from *CACQ-SoD₇* to *CACQ-SoD₁₃*.

SoD	Does not allow	SoD	Does not allow
CACQ-SoD₁	(*i*)	*CACQ-SoD₈*	(*iv*), (*vi*)
CACQ-SoD₂	(*ii*)	*CACQ-SoD₉*	(*vi*), (*vii*)
CACQ-SoD₃	(*iii*)	*CACQ-SoD₁₀*	(*ii*), (*iv*)
CACQ-SoD₄	(*iv*)	*CACQ-SoD₁₁*	(*ii*), (*vii*)
CACQ-SoD₅	(*v*)	*CACQ-SoD₁₂*	(*iv*), (*vi*)
CACQ-SoD₆	(*vi*)	*CACQ-SoD₁₃*	(*vii*)
CACQ-SoD₇	(*iii*), (*v*)		

Figure 4. Implication of SoDs in presence of hierarchy

5.6 Comparison with other SoDs

Table 9 shows the correspondence between the major SoDs identified in the literature and the ones proposed here. First, we note that our SoDs take into account time that has not been considered earlier. Secondly, we can express all the SoDs in [1] with our constraint expressions or their combinations. The table also shows how our SoDs correspond to those proposed in [18]. We note that the SoDs in rows 10 through 15 are more task oriented. However, with the help of the triggers and dependency constraint along with some transformation of the problem to map into RBAC domain, our framework can easily express them. Since previously identified SoDs are non-temporal, they correspond to the special case of the time-constrained SoDs proposed here, where $(I, P) = all$ and any occurrence of U, R or P in GTRBAC SoDs refer to the complete sets Users, Roles and Permissions. Furthermore, by using the GTRBAC status predicates, several new SoDs have been identified.

6. RELATED WORK

Mainly two kinds of cardinality constraints are often mentioned in the literature - user cardinality and role cardinality [6, 9, 15]. However, our approach of using status predicate, an evaluation function and a projection operator to define cardinality constraints allows one to express cardinality constraints associated with all states - role enabling, user or permission assignment as well as activation time including session.

Table 9. Comparison with SoDs proposed elsewhere

SH: Simon-Zurko's SoD list [18]; AH: Ahn's SoDs list [1].			GTRBAC (non-temporal forms)
1	**SZ**	Strong *SSoD* (*no user can be assigned to conflicting roles*)	$UAS\text{-}SoD_1$
	AH	**SSoD-CR** (*no user should be (implicitly and explicitly) assigned to conflicting roles, i.e., no user can-activate conflicting roles*)	$CACT\text{-}SoD_1$
2	**AH**	**SSoD-CP** (*a user cannot acquire conflicting permissions*)	$CACTs\text{-}SoD_9$
3	**AH**	*Variation of 2* (*2 + conflicting permissions cannot be acquired through a role*)	$CACTs\text{-}SoD_9 \wedge CACQ\text{-}SoD_{4.1}$
4	**AH**	*Variation of 1* (*1 + conflicting permissions cannot be acquired through a role + conflicting permissions cannot be assigned to a role*	$CACT\text{-}SoD_1 \wedge CACQ\text{-}SoD_{13} \wedge PAS\text{-}SoD_2$
5	**AH**	**SSoD-CU** (*1` + conflicting users cannot be assigned to a role*)	$CACT\text{-}SoD_1 \wedge CACQ\text{-}SoD_1 \wedge UAS\text{-}SoD_2$
6	**AH**	*Variation:* **(4) ∧ (5)**	**(4) ∧ (5)** above
7	**SZ**	*Simple* DSoD	$ACT\text{-}SoD_1$
	AH	*User-based* **DSoD** (*Conflicting roles cannot be active at the same time for a user*)	
8	**AH**	*User-based* **DSoD** *with* **CU** (*Conflicting roles cannot be active at the same time for a user*) *only difference between this and (7) is that here we are taking a conflicting user set other wise it is the same*	*Same as 7 but U is also a conflicting set*
9	**AH**	*Session-based* **DSoD** (*Conflicting roles cannot be active at the same in the same user session*)	$ACT\text{-}SoD_4$
10	**AH**	*Session-based* **DSoD** *with* **CR** (*Conflicting roles cannot be active at the same in the same user session*) *Only difference from 9 is that it has conflicting set of users*	*Same as 9 but R is also a conflicting set*
11	**SZ**	*Object-based* **DSoD** (*no user may act upon a target that that user has previously acted upon*)	*Can be rephrased as: if a user acquires a permission then he cannot acquire it again. Post-condition constraint can be used here.*
12	**SZ**	*Operational* **DSoD** (*no user may assume a set of roles that have capability for a complete business job*)	*Task oriented: if the task can be represented by atleast two roles (sub-tasks) then it can be easily represented using $UAS\text{-}SoD_1$ or $ACT\text{-}SoD_1$*
13	**SZ**	*History-based* **DSoD** (*no user is allowed to perform all the actions in a business task in the same target or collection of targets*)	*Comment similar to 12 can be made here, too.*
14	**SZ**	*Order-dependent* **SoD** (*The roles must perform their actions in a particular order*)	*It can be expressed as a sequence of precedence constraints*
15	**SZ**	*Order-independent* **SoD** (*Order does not matter as long as both happen*)	*Use of triggers of form ($\neg y$, $x \to y$ after Δt), ($\neg x$ $y \to x$ after Δt) can be used to enforce this.*

Several papers in the literature deal with separation of duty constraints, with efforts focused on identifying various forms of SoDs as well as to categorize them [1, 8, 14, 18]. Simon and Zurko [18] discuss informally a wide variety of SoD constraints that are required in systems. Gligor *et. al.* [8] provide a formalism for these SoDs. One limitation of this work, however, is that it does not consider the session-based dynamic SoD needed for simulating lattice-based access control and Chinese Walls in RBAC [15, 17]. Another limitation is that the SoDs defined do not capture the hierarchical semantics. Improvements along these lines can be seen in the SoDs listed by Ahn *et. al.* [1]. Unlike these approaches, we follow a predicate-based definition of general exclusion and inclusion of various kinds of assignments and activations to define the SoD properties in GTRBAC. This approach, while subsumes the SoDs defined in the above-mentioned literature, also provides the overall capability of an RBAC model to capture the separation of duty constraints that may exist.

Dependency constraints form a less explored aspect in RBAC. While some form of dependency is implied by role triggers introduced in TRBAC [4] and GTRBAC [12], the control flow dependency constraints, where a strict dependencies are implied, have not been included within an RBAC framework. Such control flow dependencies are typically used in workflow type of systems to define inter-dependencies between workflow tasks [3]. We believe that using such dependency constraints, GTRBAC can better handle access control requirements in time-sensitive, workflow types of applications, by providing a much broader framework for mapping tasks into roles and using these constraints to capture the interdependencies between the tasks.

No earlier work has addressed the issue of time-based cardinality, SoD and dependency constraints. Applying periodicity/duration constraints for these SoDs is more suitable for supporting access control needs of dynamically evolving systems that are prevalent today.

7. CONCLUSION AND FUTURE WORK

We have presented constraints for GTRBAC model including cardinality constraints, control flow dependency constraints, and separation of duty constraints. We used an evaluation function and a projection operator associated with a set of GTRBAC status predicates to build an elaborate framework for expressing cardinality constraints. GTRBAC's trigger has been extended so that more complex time-based past information can be captured. A set of control flow dependency constraints have been introduced using the trigger framework to enforce much stricter dependency constraints than those that can be expressed using triggers. We also showed that by generalizing to system events and conditions, the triggers and CFD framework provides an elaborate model for capturing context-based access requirements. Our approach to separation of duty constraints is based on the fact that the notion of conflict between elements in a set is often associated with another set. This allows us to consider SoDs that are of much finer-granularity. We have shown that the separation duty constraints identified in the literature can be easily expressed by a subset of our separation duty constraint expressions. One key future work that we plan to pursue is to develop a SQL or XML like language to specify the GTRBAC constraints. Another direction we plan to investigate is to use GTRBAC for workflow type of systems.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. ACM Transactions on Information and System Security (TISSEC), v.3 n.4, November 2000,

[2] J. Barkley, A. Cincotta, D. Ferraiolo, S. Gavrila, and D.R. Kuhn. Role based access control for the World Wide Web. In Proceedings of 20th National Information System Security Conference, NIST/NSA, 1997.

[3] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. ACM Transactions on Information and System Security, 2(1), 1999, p. 65-104.

[4] E. Bertino, P. A. Bonatti, E. Ferrari. TRBAC: A temporal role-based access control model. ACM Transactions on Information & System Security, 4(3), Aug.2001, p.191-233.

[5] D. F. Ferraiolo, D. M. Gilbert, N Lynch. An examination of federal and commercial access control policy needs. In

Proceedings of NISTNCSC National Computer Security Conference, Baltimore, MD, Sep 20-23 1993, p. 107-116.

[6] D. F. Ferraiolo, R. Sandhu , S. Gavrila , D. Richard Kuhn , R. Chandramouli. Proposed NIST standard for role-based access Control. ACM Transactions on Information and System Security (TISSEC) Volume 4, Issue 3, August 2001.

[7] L. Giuri. Role-based access control: A natural approach. In Proceedings of the 1st ACM Workshop on Role-Based Access Control. ACM, 1997.

[8] Gligor, V.D., S.I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. Proceedings 1998 IEEE Computer Society Symposium on Research in Security and Privacy (Oakland, Calif., May 1998), IEEE Computer Society, p. 172- 183.

[9] T. Jaeger, J. E. Tidswell. Practical safety in flexible access control models. ACM Transactions on Information System Security, 4(2), May 2001.

[10] J. B. D. Joshi, A. Ghafoor, W. Aref, E. H. Spafford. Digital government security infrastructure design challenges. IEEE Computer, Vol. 34, No. 2, Feb.2001, p. 66-72.

[11] J. B. D. Joshi, W. G. Aref, A. Ghafoor and E. H. Spafford. Security models for web-based applications. Communications of the ACM, 44 (2), Feb. 2001, p. 38-72.

[12] J. B. D. Joshi, E. Bertino, U. Latif, A. Ghafoor. Generalized temporal role based access control model (GTRBAC) (Part I)– specification and modeling. CERIAS TR 2001-47, Purdue University.

[13] J. B. D. Joshi, E. Bertino, A. Ghafoor. Temporal hierarchy and inheritance semantics for GTRBAC. 7th ACM Symposium on Access Control Models and Technologies. Monterey, CA, June 3-4, 2002.

[14] D. Richard Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. Proceedings of the second ACM workshop on Role-based access control, p.23-30, November 06-07, 1997, Fairfax, Virginia, United States

[15] S. Osborn, R. Sandhu, Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. ACM Transactions on Information and System Security, 3(2), May 2000, 85-106.

[16] R. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman. Role-based access control models. IEEE Computer 29(2), IEEE Press, 1996,p 38-47.

[17] R. Sandhu. Role hierarchies and constraints for lattice-based access controls. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo Eds., Computer Security - Esorics'96, LNCS N. 1146, Rome, Italy, 1996, p. 65-79.

[18] R. Simon and M. Zurko. Separation of duty in role-based environments. In Proceedings of 10th IEEE Computer Security Foundations Workshop, Rockport, Mass., June 1, p. 183-194.