CERIAS Tech Report 2001-47

GENERALIZED TEMPORAL ROLE BASED ACCESS CONTROL MODEL (GTRBAC) PART I
Specification and Modeling

by James B. D. Joshi, Elisa Bertino, Usman Latif,
Arif Ghafoor

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907

Generalized Temporal Role Based Access Control Model (GTRBAC) Part I *Specification and Modeling*

James B. D. Joshi[#], Elisa Bertino[*], Usman Latif[@], Arif Ghafoor[#]

[#]CERIAS and School of Electrical and Computer Engineering,
[@]CERIAS and Department of Computer Science,
Purdue University, West Lafayette, IN, USA
{*joshij, ghafoor*}*@ecn.purdue.edu,*
usman@purdue.edu
[*]Dipartimento di Scienze dell' Informazione, Universita' di Milano,
Milano, Italy
berino@dsi.unimi.it

Abstract

A temporal RBAC (TRBAC) model has recently been proposed that addresses the temporal aspects of roles and trigger-based role enabling. However, it is limited to constraints on enabling of roles only. We propose a Generalized Temporal Role Based Access Control model (GTRBAC) that is capable of expressing a wider range of temporal constraints. GTRBAC is capable of expressing periodic as well as duration constraints on roles, user-role assignments and role-permission assignments. In GTRBAC, temporal constraints on role enablings and role activations can be separately specified. A user-activated role can further be restricted to various activation constraints such as cardinality constraint or maximum active duration constraint within a specified interval. The GTRBAC model extends the syntactic structure of TRBAC model and its event and trigger expressions subsume those of TRBAC.

Portions of this work were supported by the sponsors of the Center for Education and Research in Information Assurance and Security (CERIAS)

1 Introduction

Role based access control (RBAC) models have generated great interest in the security community as a powerful and generalized approach to security [2, 13, 16, 18, 19]. In RBAC, users are assigned memberships to roles and these roles are in turn assigned permissions. A user can acquire all the permissions of a role of which he is a member. RBAC approach naturally fits into an organizational context as users are assigned to organizational roles that have well-defined responsibilities and are used to define access control requirements of the organization [7]. RBAC models have been shown to be policy-neutral [16, 17, 20], and can be used to express a very wide range of security policies including discretionary and mandatory, as well as user-defined or organizational specific policies [14]. Many benefits of an RBAC approach include its support for security management and the principle of least privilege [20]. For example, we can easily manage a change in a user's responsibility or role within his organization by assigning him the new role and removing him from the old one. Furthermore, use of role hierarchies and grouping of objects into object classes based on responsibility associated with a role makes the management of permissions very easy. By configuring the assignment of the least set of privileges from a role set assigned to a user when he activates the role, inadvertent damage can be minimized in a system.

Because of its relevance and above-mentioned benefits it provides, RBAC has been widely investigated and several extensions to it have been proposed [2, 8, 13, 16, 18, 19]. Although RBAC has today reached a good level of maturity, there are still relevant application requirements not addressed by current RBAC models. One such requirement is related to the temporal dimension that roles may have. In many organizations, functions may have limited or periodic temporal duration. Consider, for instance, the case of a part-time staff in a company and assume that any part-time staff is to be authorized to work within the company only on working days between 9 AM and 1 PM. If a part-time staff is represented by a role, enforcing such a requirement entails making sure that users log in under such a role only during the specified temporal intervals. A way to support such a requirement is to specify times when the role can be enabled so that a legitimate user can activate it. Roles can thus be enabled at certain time periods and disabled at others. Additionally, the part-time role may need to be further restricted to only two hours of active time in one session. Furthermore, depending upon organizational needs, the number of part-time staff that can activate the role during the daytime may need to be different from the number of part-time staff needed during night.

Bertino *at. el.* [5] have recently proposed Temporal-RBAC (TRBAC) that addresses some of the temporal issues related to RBAC. TRBAC is an extension of the RBAC model that is able to support temporal constraints on roles, particularly role enabling. The main features it provides include the periodic enabling/disabling of roles and temporal dependencies among them expressed by means of role triggers**,** which are active rules that are automatically executed based on the enabling and/or disabling of roles. Priorities are associated with both the triggers and periodic enabling/disabling of roles to handle possible conflicts that can arise, when the simultaneous enabling/disabling of a role is required. In such cases, a combination of priority and *denials-take-precedence* rule are used to resolve the conflicts. TRBAC further allows an administrator to issue run-time requests for enabling and disabling a role and restricted handling of role activations by a user. The model, however, is not able to handle several useful temporal constraints. In particular,

1. TRBAC does not include temporal constraints on user-role and role-permission assignments. It thus assumes that only roles can be transient, i.e., only they are enabled and disabled at different time intervals. In this paper, we show that in some applications, roles are more static in that they are enabled at all times, and users and permissions assigned to them are transient instead.

2. TRBAC only handles the temporal constraints on role enabling and does not include any constraints on the actual activations of roles by users. Thus, TRBAC does not use a well-defined, separate notion of role enabling and role activation. Because of this, TRBAC cannot handle many constraints that are related to the activations of a role such as the constraints on the maximum active duration allowed to a user, the maximum number of activations of a role by the same user within a particular interval of time, etc. Although TRBAC has a limited capability of restricting a user from activating a role, it is only handled as a run-time request that an administrator makes.

3. As TRBAC does not consider duration constraints and constraints on actual activations of roles, it does not include the notion of enabling and disabling of constraints. As activation constraints need to be well defined with respect to the enabled time of a role, we introduce the notion of constraint enabling/disabling in this paper.

A closely related work is the recently proposed constraint specification language called *RCL2000* by Ahn *et. al*. [1]. However, they do not incorporate the temporal constraints in their specification language. They also do not model the separate notions of role enabling and role activations that is essential for separately handling static and dynamic constraints.

In this paper, we illustrate that the constraints mentioned above are useful for several reasons and a complete TRBAC model should allow them. We propose a Generalized TRBAC (GTRBAC) model that subsumes TRBAC and can handle all the constraints mentioned above. We distinguish between the notions of role activation from that of role enabling to incorporate various activation constraints on role activations. We also extend the safety notion and the system architecture introduced in [5] to establish the applicability of our model.

The paper is organized as follows. In section two, we introduce the notion of periodic expressions and general background on the RBAC model. Section three presents the description of the temporal constraints. These are syntactically and semantically formalized in section four where we introduce the GTRBAC model. Additionally, we introduce the notion of safe temporal constraints and activation base (TCAB) as an extension of safe role activation base (RAB) of [5]. In section five, we present the system architecture for implementing a GTRBAC system. We discuss related work in section six and present our conclusions and future work in section seven.

2 Preliminaries

In this section, we provide a brief background on the RBAC model we refer to in this paper.

2.1 RBAC Model

The RBAC model as proposed by Sandhu *et. al.* in [20] consists of the following four basic components: a set of users `Users`, a set of roles `Roles`, a set of permissions `Permissions`, and a set of sessions `Sessions`. A user is a human being or an autonomous agent. A role is a collection of permissions needed to perform a certain job function within an organization. A permission is an access mode that can be exercised on objects in the system and a session relates a user to possibly many roles. When a user logs in the system he establishes a session and, during the session, he can request to activate some subset of roles he is authorized to play. An activation request is granted only if the corresponding role is enabled at the time of the request and the user is entitled to activate the role at that time. If the activation request is satisfied, the user issuing the request obtains all the permissions associated with the role he has requested to activate. On the sets `Users, Roles, Permissions`, and `Sessions`, several functions are defined. The *user role assignment* (UA) and the *role permission assignment* (PA) functions model the assignment of users to roles and the assignment of permissions to roles respectively. A user can be a member of many roles and a role can have many members. Moreover, a role can have many permissions and the same permissions can be associated with many roles. The *user* function maps each session to a single user, whereas function *role* establishes a mapping between a session and a set of roles (that is, the roles which are activated by the corresponding user in the session). On `Roles`, a hierarchy is defined, denoted by ≥. If $r_i \geq r_j$, r_i, $r_j \in$ `Roles` then r_i inherits the permissions of r_j. In such a case, r_i is a senior role and r_j a junior role. The following definition formalizes the above discussion.

Definition 2.1.1 (RBAC Model) [5] *The RBAC model consists of the following components:*

- *Sets* `Users, Roles Permissions` *and* `Sessions` *representing the set of users, roles, permissions, and sessions, respectively;*
- *PA*: `Roles` $\rightarrow$ `Permissions,` *the permission assignment function, that assigns permissions to roles;*
- *UA*: `Users` $\rightarrow$ `Roles,` *the user assignment function, that assigns users to roles;*
- *user*: `Sessions` $\rightarrow$ `Users,` *which maps each session to a single user*;
- *role*: `Sessions` $\rightarrow 2^{\text{Roles}}$ *that maps each session to a set of roles*;
- *RH* $\subseteq$ `Roles` $\times$ `Roles`, *a partially ordered role hierarchy* (written ≥). □

2.2 Periodic Expression

Periodic time is represented by means of a symbolic, user-friendly formalism [11, 3]. Under such formalism periodic time is represented by pairs ⟨[`begin`, `end`], `P`⟩, where `P` is a *periodic expression* denoting an infinite set of periodic time instants, and [`begin`, `end`] is a time interval denoting the lower and upper bounds that are imposed on instants in `P`.

The formalism for periodic expressions is based on the one proposed in [11], and relies on the notion of *calendars*. A calendar is defined as a countable set of contiguous intervals,[1] numbered by integers called indexes of the intervals.

A subcalendar relationship can be established between calendars. Given two calendars C_1 and C_2, we say that C_1 is a subcalendar of C_2, (written $C_1 \sqsubseteq C_2$), if each interval of C_2 is exactly covered by a finite number of intervals of C_1. New calendars can be dynamically generated from the

[1] Two intervals are contiguous if they can be collapsed into a single one (e.g., [1, 2] and [3, 4]).

existing ones, by means of a function *generate*() (cf. 3] for a formal definition), a *reference time instant*, and a *basic calendar* (the tick of the system), denoted by τ. In the following, we assume the existence of a set of calendars containing the calendars *Hours*, *Days*, *Weeks*, *Months*, and *Years*, where *Hours* is the calendar with the finest granularity, i.e., it is the *basic calendar*.

Calendars can be combined to represent more general *periodic expressions*, denoting periodic instants not necessarily contiguous, like, for instance, the set of *Mondays* or the set of *the third hour of the first day of each month*. Periodic expressions are formally defined as follows.

Definition 2.2.1 (**Periodic Expression**) [3]: *Given calendars* C_d, C_1, …, C_n, a *periodic expression* P *is defined as*:

$$\mathsf{P} = \sum_{i=1}^{n} O_i.C_i \ \rhd \ x.C_d$$

where $O_1 = \mathtt{all}$, $O_i \in 2^{\mathbb{N}} \cup \{all\}$, $C_i \sqsubseteq C_{i-1}$ for $i = 2,.., n$, $C_d \sqsubseteq C_n$, and $x \in \mathbb{N}$. □

Symbol $\rhd$ separates the first part of the periodic expression that identifies the set of starting points of the intervals it represents, from the specification of the duration of each interval in terms of calendar C_d. For example, $\mathtt{all}$.*Years* + {3, 7}.*Months* $\rhd$ 2.*Months* represents the set of intervals starting at the same instant as the third and seventh month of every year, and having a duration of 2 months. In practice, O_i is omitted when its value is *all*, whereas it is represented by its unique element when it is a singleton. $x.C_d$ is omitted when it is equal to $1.C_n$.

The infinite set of time instants corresponding to a periodic expression P is denoted by $\prod(\mathsf{P})$. Function $\prod(\mathsf{P})$ is formally defined as follows.

Definition 2.2.2 (**Function** $\prod$()) [3]: *Let* $\mathsf{P} = \sum_{i=1}^{n} O_i.C_i \ \rhd \ x.C_d$ *be a periodic expression, then* $\prod(\mathsf{P})$ *is a set of time intervals whose common duration is* C_d*, and whose set S of starting points is computed as follows*:

- *If* $n=1$, *S contains all the starting points of the intervals of calendar* C_1.
- *If* $n>1$, *and* $O_n = \{n_1, .. , n_k\}$, *then S contains the starting points of the* n_1^{th},..., n_k^{th} *intervals* (*all intervals if* $O_n = \mathtt{all}$) *of calendar* C_n *included in each interval of* $\prod(\sum_{i=1}^{n} O_i.C_i \ \rhd 1.C_{n-1})$. □

For simplicity, in this paper the bounds `begin` and `end` constraining a periodic expression will be denoted by a pair of *date expressions* of the form `mm/dd/yyyy:hh`, with the obvious intended meaning; `end` can also be $\propto$. For instance, [`1/1/2001,12/31/2001`] denotes all the instants in 2001. The set of time instants denoted by $\langle$[`begin,end`], `P`$\rangle$ is defined through the use of function *Sol*(), formally defined as follows.

Definition 2.2.3 (**Function** *Sol*())[3]: *Let t be a time instant,* `P` *a periodic expression, and* `begin` *and* `end` *two date expressions*. *Define* $t \in$ *Sol*([`begin, end`], P) *iff there exists* $\tau \in \prod$ (`P`) *such that* $t \in \tau$, $t_b \leq t \leq t_e$, *where* t_b *and* t_e *are the instants denoted by* `begin` *and* `end`, *respectively*. □

3 Temporal constraints and Role States

In this section, we discuss various types of temporal constraints relevant to role-based systems. In particular, we show how temporal constraints can be meaningfully applied to various components of an RBAC system by means of examples from various real world applications. Our temporal constraint model is quite articulated and provides both duration and periodicity constraints as well as other forms of specialized cardinality constraints.

A key aspect of our RBAC model, which is a natural consequence of the introduction of temporal constraints, is the distinction between the notions of *role enabling* and *role activation*. Such a distinction leads in turn to the notion of *states of a role*. In our model, a role can be in one of the three states at any time as shown in Figure 1. A *Disabled* state of a role indicates that the role cannot be used in any user session, i.e., a user cannot acquire the permissions associated with the role. A role in the *Disabled* state can be enabled. The *Enabled* state indicates that users who are entitled to use the role at the time of the request may activate the role, but no one has yet done so. If a user now activates the role, the state of the role becomes *Active.* When a role is in *Active* state, it indicates that there is at least one user who has activated the role. Once in *Active* state, upon any subsequent activation by the same or other users, the role remains in the *Active* state. When the role is in *Active* state, then upon deactivation, the role goes into the *Disabled* state if there is only one session in which it is active, otherwise it remains in the *Active* state. A role in *Enabled* or *Active* state goes into the *Disabled* state if a disabling event arises.

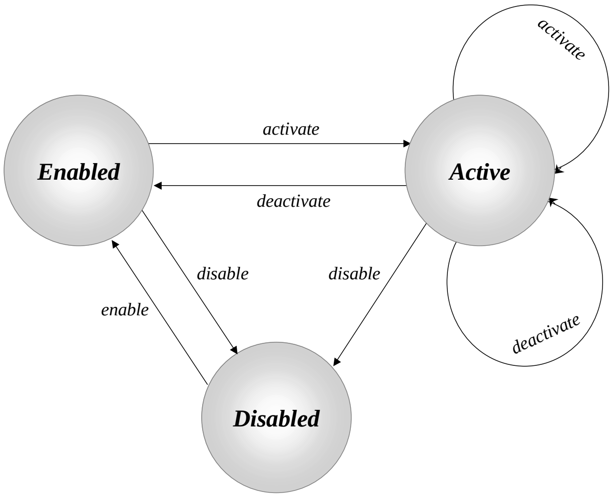


Figure 1. States of a role

In the remainder of this section, we first discuss the two forms of temporal constraints provided by our model and illustrate the use of such constraints. We then introduce the specialized activation cardinality constraints, provided by our model, whose purpose is to support fine-grained control on role activation/deactivation.

3.2 Periodicity/Duration constraints

An important feature of our model is that periodicity/duration constraints can be applied to various components of RBAC. Specifically, they can be applied to roles themselves, by constraining the times when roles are enabled or active, and to user-role and role-permission assignments. User-role assignments refer to the permissions given to users to play roles, whereas role-permission assignments refer to the grants given to roles for accessing the protected resources. Role enablings as well as assignments can be restricted to particular intervals or for a specified duration depending on requirements. Similarly, role activations can be restricted to duration constraints. Since a role activation is the result of granting a user request to activate the role, we do not allow periodicity constraint on role activations, as such a request is made at the discretion of the user. Periodic expressions can be used to specify start and end times between which a role can be enabled, or an assignment granted. Duration constraints, however, only specify the length of time that a role can be enabled or activated, or an assignment granted, and do not specify start and end times, and hence such constraints come to effect only if some other events or conditions trigger them.

In our model, we further refine duration constraints on role activations into two types: total active duration constraints and maximum duration per activation constraints.

1. **Total active duration constraint:** The total active duration of any role may need to be restricted in some cases. This notion is similar to that of the total time associated with a calling card. After the users have utilized the specified total active duration for a role, it cannot be activated again, even though it may still be enabled. Furthermore, the total active duration specified may span a number of periods of role enabling and disabling. We further classify total active duration constraint into:

   Per role: This restricts the total active duration for a role, irrespective of who the users are. Once the sum of all activations of the role reaches the maximum allowed value, no further user activation of the role is allowed.

   Per user-role: This restricts the total active duration for a role by a particular user. After the specified user uses the total active duration of the role specified for him/her, only s/he is not allowed to activate the role further.

2. **Maximum duration constraint per activation:** This constraint restricts the total maximum duration allowed for each activation of a role. This notion is similar to limiting of car parking to a fixed number of hours at one time. Once the maximum active duration allowed expires for a user, the role activation for that user is removed. However, there may still be other activations of the same role in the system, including one by the same user in some other session. This constraint can also be per role or per user role.

   Per role: This restricts the maximum active duration for each activation of a role for any user, i.e., any user assigned to the role can acquire the permissions of the role for the specified maximum duration in one activation, unless there is a per user-role constraint specified for the user.

   Per user-role: This restricts the maximum active duration allowed for each activation of a role by a particular user. Whenever the specified user activates the role, he is allowed only the specified maximum duration.

We now present various examples illustrating the use of such constraints in the various components of an RBAC model.

Examples of Periodicity/Duration constraints on user-role assignment

Some roles are often more static than others in that they can be enabled most of the time, whereas users come and go. In such cases, the users assigned to the role may need to be constrained by restricting how long or in which intervals they can use the role, as illustrated by the following example.

> Example 3.1 (**Automated Hospital**): Consider a fully automated hospital where access to doors, computer terminals in various wards, elevators, TV/Movie programs, etc. are controlled by using GTRBAC system. For example, to enter a corridor, a patient needs to activate his role at the door. We consider the following three roles a patient may be assigned to at the time he is admitted: Delivery_Patient, Cardiac_Patient and Virus_Related_Patient. Using these roles, permissions to open certain doors and elevators can be restricted so that a Delivery_Patient do not inadvertently run over a Virus_Related_Patient. Similarly a Cardiac_Patient can be restricted to watch only certain TV channels or movies so that the programs do not exert unnecessary/unhealthy stress on him. These roles are always in the *enabled* state to allow patients to be admitted at any time so long as there are rooms or beds available. When a patient is admitted, s/he is assigned to one of these roles for a projected period of stay. Some other roles in the system include NurseOnDayDuty, NurseInTraining, etc.

In the automated hospital example, we can identify the following two constraints on the user-role assignments:

1. **Periodicity constraint**: The patient assignment may informally state that '*Mary is assigned to* Delivery_Patient *role in interval* [12/1/2001, 12/20/2001]'. Within the specified interval, Mary can use Delivery_Patient role to do all permitted accesses inside the hospital.

2. **Duration constraint**: A part-time nurse, say Mary, may be assigned to the NurseOnDayDuty role for 3 hours on each workday after a particular nurse, say Cathy, has activated the NurseOnDayDuty role that day. Here, once Cathy activates the NurseOnDayDuty role, then Mary can assume the same role for three hours, although it is not mandatory.

Examples of Periodicity/Duration constraint on role enabling

Some meaningful examples of constraints from the automated hospital are as follows.

1. **Periodicity constraint**: Role NurseOnDayDuty must be enabled only from 9am to 9pm. A user who is assigned to this role can activate it at any time within the specified period provided s/he is not constrained by other temporal constraint.

2. **Duration constraint**: Role NurseInTraining may need to be enabled for 3 hours after a particular nurse, say Cathy, has activated the role NurseOnDayDuty. Anyone assigned to the NurseInTraining role may then activate it.

Examples of Periodicity/Duration constraint on role-permission assignment

Consider the following example of an online course.

Example 3.2 (**Online Course**): A professor offers an online course on '*Computer Security*'. He creates a role called CSRegistrant as shown in Figure 2. He divides the online course material into three sections –

1. **Lectures**: There are *n* lectures as depicted in Figure 2 - Lecture 1, Lecture 2 .., Lecture n. The permission set related to the *i*th lecture is represented as PL_i.

2. **Home works**: There are *n* homework assignments (shown as HW1,..,HWn) corresponding to *n* lectures. The permission set associated with the *i*th homework assignment is PHW_i.

3. **Homework solutions**: There are *n* homework solutions (shown as HWSol1,…,HWSoln) corresponding to the *n* homework assignments. The permission set associated with the *i*th homework solution is $PHW_{i.}$

The course starts on the date *startdate* and ends on *enddate*. The course duration is (*n*+2) weeks – *n* lectures for *n* weeks and 2 remaining weeks for reviews and exams. The professor uses the following three rules to control access to the different sections:

Rule 1*: The i*th *lecture is made available to students on the start of the i*th *week of the course (say Monday) and is made accessible till the end of the course.*

In the graphical representation, the arcs between the role and permission-sets are labeled as (*x*, *y*), where *x* denotes the date on which the associated permission set is assigned to the role, and *y* denotes the date on which the permissions are de-assigned from the role. For example, based on Rule 1, the permission set PL_1 is associated with the arc label (*startdate, enddate*), indicating that the permissions in PL_1 are assigned to CSRegistrant role on *startdate* and removed from it on *enddate*. Similarly, permissions in PL_2 are assigned to CSRegistrant role in the interval (*startdate + 1 week, enddate*), which says that the permissions are assigned to the role one week after the course starts, i.e., in the beginning of the 2nd week. We note that for each lecture, once the permissions are assigned to the role, they are not de-assigned until the end of the course. For homework assignments, he uses the following rules:

Rule 2a: Three days after the start of a weekly lecture, the corresponding homework assignment will be made accessible to the students.

Rule 2b: One week after the end of the ith lecture, the ith homework assignments will be removed (i.e., is made inaccessible).

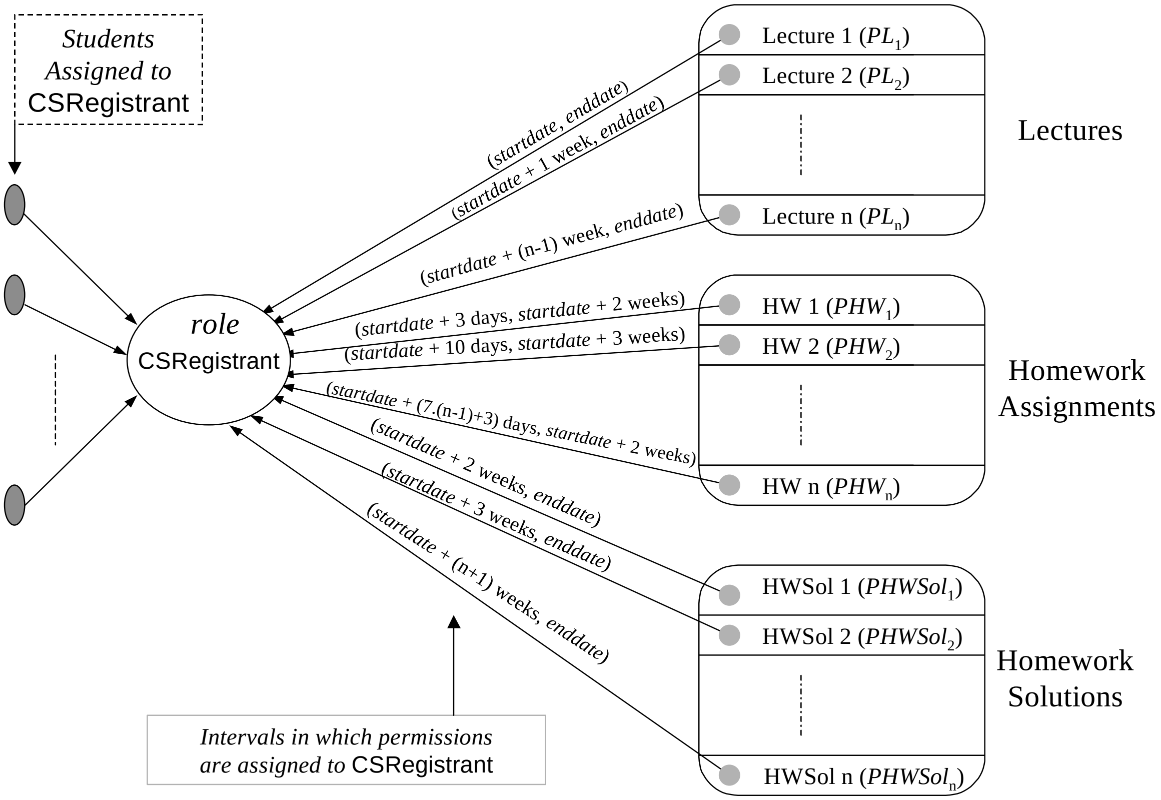

Students Assigned to CSRegistrant
role
CSRegistrant
(startdate, enddate)
(startdate + 1 week, enddate)
(startdate + (n-1) week, enddate)
(startdate + 3 days, startdate + 2 weeks)
(startdate + 10 days, startdate + 3 weeks)
(startdate + (7.(n-1)+3) days, startdate + 2 weeks)
(startdate + 2 weeks, enddate)
(startdate + 3 weeks, enddate)
(startdate + (n+1) weeks, enddate)
Intervals in which permissions are assigned to CSRegistrant
Lecture 1 (PL_1)
Lecture 2 (PL_2)
Lecture n (PL_n)
Lectures
HW 1 (PHW_1)
HW 2 (PHW_2)
HW n (PHW_n)
Homework Assignments
HWSol 1 ($PHWSol_1$)
HWSol 2 ($PHWSol_2$)
HWSol n ($PHWSol_n$)
Homework Solutions

Figure 2. Online Course Example (Role-Permission assignment)

Whereas Rule 2a seems reasonable, Rule 2b might appear unnecessary. Here, we assume that with each assignment there is a particular submission procedure to be followed, which involves executing various programs. We can assume that the *PHW1* contains permissions related to executing these required programs for the submission process too. Now consider the arc for *PHW1* , which is (*startdate* + 3 *days*, *startdate* + 2 *weeks*). As the first lecture begins on *startdate*, *PHW1* should be accessible to the students on (*startdate* + 3 *days*) as per Rule 2a. Again, the first lecture ends on (*startdate* + 1 *week*) and hence the permissions in *PHW1* are removed on *startdate* + 2 *weeks,* i.e. 1 week after the end of the corresponding lecture.

The rule corresponding to the homework solutions is:

> **Rule 3***: when a homework assignment is removed as in rule 2b above, then the corresponding solution is posted and it is made available till the end of the course.*

For example, consider the arc for *PHWSol1*, i.e., (*startdate*+2 *weeks*, *enddate*). As the first homework assignment is removed on (*startdate* + 2 *weeks*), its solution *PHWSol1* is assigned to CSRegistrant role on (*startdate* + 2 *weeks*). As the arc shows, *PHWSol1* is de-assigned from the role on *enddate*.

The example illustrates an efficient use of temporal-constraint on role-permission assignment to enforce the access policy defined by the three rules. We see that the course access policy can be expressed by using periodicity constraints on role-permission assignments. Some of the periodicity constraints can also be specified as duration constraints. For example, we can have a duration constraint on the assignment of *PHW1* to the CSRegistrant role as – "*PHW1* is assigned to role for a duration (2 weeks – 2)". Then we can trigger this assignment on the third day, after *PL1* is assigned to the role.

Examples of Duration constraint on role activation

Consider the following example of a video library.

Example 3.3 (**Video Library**): A user, say John, subscribes to a video library (VL) so that he is allowed 6 hours of movie time per week. The VL administrator assigns John to a role MovieViewer. The MovieViewer role is assigned a total of 600 hrs per week of total activation time and each user is allowed a maximum of 6 hrs/week out of that. Another user, say Mary, wants 10 hrs per week of time. In that case the VL administrator can assign Mary to the MovieViewer role and further specify that she be restricted to a maximum of 10 hrs/week of active time. For finer control over resource use, the VL administrator further employs a constraint that says each user can activate the MovieViewer role for at most 2 hours in each activation. Further, upon John's request the VL administrator adds yet another restriction that John be allowed to activate the role for maximum of 3 hours in each activation.

The example illustrates the following types of temporal constraints on the activation period of a role within a specified interval:

1. **Total active duration constraint**: The 600 hour-duration per week allowed for the MovieViewer role is a constraint on the total active duration of the role, whereas the total of 6 hrs per week allowed to individual users is a restriction per user. In particular, note that no matter who uses the MovieViewer role, once the sum total of durations of all its activations exceeds 600 hours, it can no longer be used by any user for the remaining part of the week. When the new week begins, it will again have 600 hours as total activation time. The 10 hrs/week restriction for Mary is a per-user-role constraint.

2. **Maximum duration constraint per activation**: The additional restriction that each user can activate the MovieViewer role for 2 hours in each activation of the role is a per-role maximum duration constraint. Similarly, the restriction of maximum of 3 hours activation time for John each time he activates the role is a per-user-role constraint. Since, John has 6 hrs/week of active time, he needs to have at least two activations of MovieViewer role each week in order to fully use the 6 hrs/week time allowed to him. Similarly, since Mary has total of 10 hours per week, she has to activate the MovieViewer role atleast 5 times (2 hours per activation) in order to utilize all 10 hours of time.

3.3 Activation Cardinality Constraints

The number of activations of any role may need to be restricted to control access to critical objects or resources in some applications. For example, depending on the capability of the video servers, the MovieViewer role in `Example 3.3` may need to be limited to a specified maximum number of concurrent activations at any time. Furthermore, we may want to constrain the number of concurrent activations of the MovieViewer role by each user to prevent one person accessing all the resources while others are denied access, thus preventing denial of service. In our model, we further classify activation cardinality constraints into:

3. **Total *n* activations constraint**: In a given period of time, a role may be limited to *n* total activations. This includes overlapping or non-overlapping activations. This can be of two types:
   *Per role***:** This constraint allows at most *n* activations of a role in a given period of time. The activations of a role may be associated with the same or different users. The role may be enabled and disabled a number of times before the total number of activations of the role allowed is reached, after which the role cannot be activated.

Per user-role: This constraint restricts a total of *n* activations of a role by a particular user. Each user may be specified to have a different cardinality restriction.

4. **Maximum *n* concurrent activations constraint**: A role may be restricted to *n* concurrent activations at any time. This may also be of two types:

   Per role: This restricts the number of concurrent activations of a role to a maximum number. The activation of these roles may be associated with the same or a set of different users.

   Per user-role: This restricts the total of concurrent activations of a role by a particular user to a given number. Different users may be allowed different number of concurrent activations of the same role.

4 Generalized TRBAC

In this section, we present the formal framework for a GTRBAC that can handle the constraints discussed in Section 3. We present the syntax and semantics of the expressions we use to specify various GTRBAC constraints.

4.1 Syntax

In this subsection, we introduce the syntax of our constraint language, so to formally define all components of constraints. We also introduce the notion of run-time requests. Run-time requests are necessary in order to model activation of roles by a valid user, which is done at his/her discretion. Furthermore, a security administrator may need to enable or disable roles and assign/de-assign users or permissions to roles at run-time. Such events are modeled as run-time requests that may or may not be granted depending upon the existing constraints. Before introducing the syntax, we introduce some basic event expressions used by the constraint language.

We use ($\texttt{Prios}$, $\preccurlyeq$) as a totally ordered set of priorities and assume that $\texttt{Prios}$ contains two distinct elements $\perp$ and $\top$ such that, for all $x \in \texttt{Prios}$, $\perp \preccurlyeq x \preccurlyeq \top$. We write $x \prec y$, if $x \preccurlyeq y$ and $x \neq y$.

Definition 4.1.1 (**Event Expressions, Role Status Expressions and Assignment Status Expressions**)

1. *A* simple event expression *has one of the following forms:*

   a. $\texttt{enable}\ r$ *or* $\texttt{disable}\ r$ *where* $r \in \texttt{Roles}$, *where* $r \in \texttt{Roles}$.

   b. $\texttt{assign}_{\texttt{U}}\ r\ \texttt{to}\ u$ *or* $\texttt{de-assign}_{\texttt{U}}\ r\ \texttt{to}\ u$, *where* $r \in \texttt{Roles}$ *and* $u \in \texttt{Users}$.

c. assign$_P$ p to r or de-assign$_P$ p to r, where p ∈ Permissions and r ∈ Roles.

d. enable c *or* disable c, *where c is a duration constraint expressed as* (*D*, *D_x*, *pr:E*) *or activation constraints of form* (*C*) *and* (*D*, *C*). *These constraints are elaborated in* Table 1 *and discussed below*.

2. Prioritized event expressions *have the form pr:E, where pr* ∈ Prios *and E is a* simple event expression;
3. Role status expressions *have one of the following forms:*

a. enabled *r or* ¬enabled *r* (*or* disabled *r*), *where r* ∈ Roles.

b. active *r* for *u or* ¬active *r* for *u* , *where r* ∈ Roles *and u* ∈ Users.

4. Assignment status expressions *have the following forms*

a. assigned *r* to *u* or ¬assigned *r* to *u*, *where r* ∈ Roles *and u* ∈ Users.

b. assigned *p* to *r* or ¬assigned *p* to *r*, *where r* ∈ Roles *and p* ∈ Permissions. □

Users can activate an enabled role if they are entitled to do so by valid assignments. A user makes a request for activation of a role at run-time within a user session. S/he may activate the same role in several user sessions concurrently if permitted by activation constraints. The system must ensure that activation constraints are satisfied before granting a user request for activating a role and for all instants at which the role activation is associated with a user. An administrator may also need to activate an event at run-time. We define run-time requests that model these two types of events as follows.

Definition 4.1.2 (Run-time request): *A run-time request expression has one of the following forms:*

1. *a user's run-time request expression to activate a role has the form*:

   s: activate *r* for *u* after Δ*t*, or

   s: deactivate *r* for *u* after Δ*t*

   where r ∈ Roles *and u* ∈ Users, *s* is the session attached to the request, and Δ*t is the duration*. *The priority of this event is assumed to be* ⊥.

2. *an administrator's run-time request expression has the form:*

pr:E after Δ*t*

where pr:E is a prioritized event expression and Δt is the duration expression. The priority and the delay expressions can be omitted, in which case, by default $pr = \top$ and $\Delta t = 0$. □

A relevant requirement in many application domains is represented by the need of automatically executing certain actions, such as the enabling or disabling of a role, upon occurrence of an event. Certain events are thus result of occurrences of other events. In our model, we provide the notion of trigger, defined below, in order to support such event dependencies.

Definition 4.1.3 (**Triggers**): *A trigger expression has the form*

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

where E_is are simple event expressions or run time requests, C_is are role status expressions or assignment status expressions, pr:E is a prioritized event expression with $pr \prec \top$, E is a simple expression such that $E \notin \{s\texttt{:activate } r \texttt{ for } u\}$, and Δt is a duration expression. □

We note that, because an activation request is made at a user's discretion, we do not allow event expression "s`:activate` r `for` u" to appear in the head of a trigger. However, such an event can trigger other events and hence can be a part of the body of a trigger. We also note that the event "s`:de-activate` r `for` u" is allowed to appear in the head of a trigger as it can be used to enforce system controls.

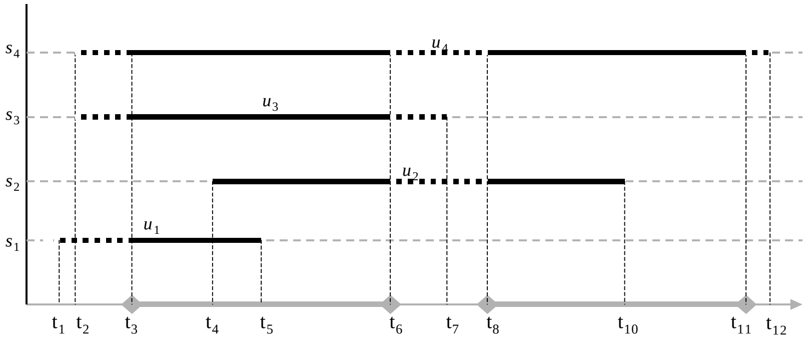
Based on the definitions of prioritized events, run-time requests and trigger expressions introduced above, we obtain the complete set of constraint expressions as shown in `Table 1`. The description of the syntax of these expressions follows next.

The periodicity constraint expressions have the general form (I, P, pr:E). The pair (I,P) specifies the intervals in which the event E takes place. E can be a role enabling event "`enable/disable` r"or either of the assignment events "$\texttt{assign}_\texttt{U}\texttt{/deassign}_\texttt{U}$ p `to` r" and "$\texttt{assign}_\texttt{P}\texttt{/deassign}_\texttt{P}$ p `to` r".

Figure 3 shows periodicity constraints on user-role assignments. The two grayed thick lines at the bottom represent the intervals (t_3, t_6) and (t_8, t_{11}) in which role r is enabled. The thick dark lines are intervals in which users are assigned to role r. The dotted portions of the thick dark lines indicate the valid user assignment periods that are ineffective because the role is disabled at those intervals. For example, user u_1 is assigned to role r in interval (t_1, t_5). However, user u_1 can only activate role r in the interval (t_3, t_5), as the role is disabled in the remaining part of interval (t_1, t_5).

Table 1. Constraint Expressions

Constraint categories	Constraints		Expression	Set/Type
Periodicity Constraint	User-role assignment		(I, P, pr:assign$_U$/deassign$_U$ *r* to *u*)	C_{URp}
	Role enabling		(I, P, pr:enable/disable *r*)	C_{Rp}
	Role-permission assignment		(I, P, pr:assign$_P$/deassign$_P$ *p* to *r*)	C_{PRp}
Duration Constraints	User-role assignment		([(I, P)| D], D_U, pr:assign$_U$/deassign$_U$ *r* to *u*)	C_{Urd}
	Role enabling		([(I, P)| D], D_R, pr:enable/disable *r*)	C_{Rd}
	Role-permission assignment		([(I, P)| D], D_P, pr:assign$_P$/deassign$_P$ *p* to *r*)	C_{PRd}
Duration Constraints on Role Activation	Total active role duration	Per-role	([(I, P)| D], D_{active}, [$D_{default}$], active$_{R\_total}$ *r*)	$C^{a}{}_{dr}$
		Per-user-role	([(I, P)| D], $D_{uactive}$, *u*, active$_{UR\_total}$ *r*)	$C^{a}{}_{dur}$
	Max role duration per activation	Per-role	([(I, P)| D], D_{max}, active$_{R\_max}$ *r*)	$C^{a}{}_{mr}$
		Per-user-role	([(I, P)| D], D_{umax}, *u*, active$_{UR\_max}$ *r*)	$C^{a}{}_{mur}$
Cardinality Constraint on Role Activation	Total no. of activations	Per-role	([(I, P)| D], N_{active}, [$N_{default}$], active$_{R\_n}$ *r*)	$C^{a}{}_{nr}$
		Per-user-role	([(I, P)| D], $N_{uactive}$, *u*, active$_{UR\_n}$ *r*)	$C^{a}{}_{nur}$
	Max. no. of concurrent activations	Per-role	([(I, P)| D], N_{max}, [$N_{default}$], active$_{R\_con}$ *r*)	$C^{a}{}_{nnr}$
		Per-user-role	([(I, P)| D], N_{umax}, *u*, active$_{UR\_con}$ *r*)	$C^{a}{}_{nmur}$
Trigger	E_1 ,…, E_n , C_1 ,…, C_k → *pr:E* after Δt			C_{tr}
Constraint Enabling	pr:enable/disable *c* where $c \in \{(D, D_x, pr{:}E), (C) , (D, C)\}$			C_c
Run-time Requests	*Users' activation request*		(s:(de)activate *r* for *u* after Δt))	C_u
	Administrator's run-time request		(pr:assign$_U$/de-assign$_U$ *r* to *u* after Δt)	C_{admin}
			(pr:enable/disable *r* after Δt)	C_{admin}
			(pr:assign$_P$/de-assign$_P$ *p* to *r* after Δt)	C_{admin}
			(pr:enable/disable *c* after Δt)	C_{admin}



Role *r* is enabled in intervals (t_3, t_6) and (t_8, t_{11}).
User u_1 is assigned to *r* in interval (t_1, t_5) but can use *r* only in interval (t_3, t_5).
User u_2 is assigned to *r* in interval (t_4, t_{10}) but can use *r* only in intervals (t_4, t_6) and (t_8, t_{10}).
User u_3 is assigned to *r* in interval (t_2, t_7) but can use *r* only in interval (t_3, t_6).
User u_4 is assigned to *r* in interval (t_2, t_{12}) and can use *r* whenever it is enabled.

Figure 3. Periodicity constraint on user-role assignment

The general form for the duration constraint expressions for role enabling and assignments is ([(*I*, *P*,)|*D*], D_x, *pr*:*E*), where *x* is R,U or P, corresponding to the three types of possible events: "enable/disable *r*", "assign$_U$/deassign$_U$ *p* to *r*" and "assign$_P$/deassign$_P$ *p* `to` *r*". The symbol "|" between (I, P) and D indicates that either (I, P) or D is specified. The square bracket in [(I, P,)|D] implies that it is an optional parameter. Hence, we have the following three forms of duration constraints: (I, P, D_x, pr:E), (D, D_x, pr:E) and (D_x, pr:E).

The form (I, P, D_x, pr:E) indicates that the event E is valid for the duration D_x within each valid periodic interval specified by (I, P). The form (D_x, pr:E) implies that the constraint is valid at all times. Thus, at any time, if event E is caused then it is restricted by duration D_x. The constraint c = (D, D_x, pr:E) implies that there is a valid duration D within which the duration restriction D_x applies on event E. In other words, the constraint c needs to be enabled for duration D. To support enabling of such constraints (and activation constraints discussed later) we include the constraint enabling expressions as shown in Table 1. The constraint enabling/disabling event has the expression "`enable/disable` *c*", where c is a constraint expression (D, D_x, pr:E). A constraint enabling event may be a run-time request or a triggered event.

Activation constraints have the general form ([(I, P)| D], C), where C represents the restriction applied to a role activation; for example, C = ($\mathrm{D_{active}}$, [$\mathrm{D_{default}}$], $\mathtt{active}_{\mathtt{R\_total}}$ r) indicates the *total active role duration per-role* constraint. [(I, P)| D] is an optional parameter and has the same meaning as in duration constraints. Thus, an activation constraint may be in one of the three forms (I, P, C), (D, C) or (C). The first two forms are similar to those for duration constraints. The form (C) implies that the activation restriction specified by C applies within each enabling of the associated role. If C is a *per-role* constraint, it has an optional default parameter that allows specifying the default value for the *per-user-role* restriction. For example, if C = ($\mathrm{D_{active}}$, [$\mathrm{D_{default}}$], $\mathtt{active}_{\mathtt{R\_total}}$ r) then $\mathrm{D_{default}}$ indicates the default *per-user-role* active duration value applied to all the users assigned to the role, e.g., the 6 hrs/week limit in `Example 3.3`. If $\mathrm{D_{default}}$ is not specified then we assume that $\mathrm{D_{active}}$ = $\mathrm{D_{default}}$, i.e, a single user can use up the entire duration $\mathrm{D_{active}}$. As shown in the table, parameters of other activation constraints can be similarly interpreted.

Figure 4 illustrates the three different forms of an activation constraint C, where C is a cardinality constraint. In Figure 4(a), we have constraint c = (D, C). A trigger or a run-time request at time t_2 enables this constraint. c is valid for duration D, which for this case is the interval (t_2, t_5) indicated by the dark thick line. However, within interval (t_2, t_5), there is a subinterval (t_3, t_4) in which role r is not enabled. The cardinality constraint, however, implies that the total number of activations of role r in the intervals (t_2, t_3) and (t_4, t_5) combined should not exceed $\mathrm{N_{active}}$. In Figure 4(b), the working of an activation constraint of form c = (I, P, C) is illustrated. Here, (t_2,

t_3) and (t_6, t_7) are actual intervals in (I, P) and hence, in each of these intervals the total number of activations of role r is restricted to N_{active}. In Figure 4(c), the constraint of form c = (C) is shown. Here, in each enabling period of role r, the constraint is valid. For example, role r is enabled by a periodicity constraint in the intervals (t_1, t_2), (t_3, t_4) and (t_7, t_8). In each of these intervals, at the most N_{active} activations of role r are allowed. Furthermore, role r is also enabled in the interval (t_5, t_6) by a duration constraint. The activation constraint c also applies to this interval and within this interval also only N_{active} activations of role r are allowed.

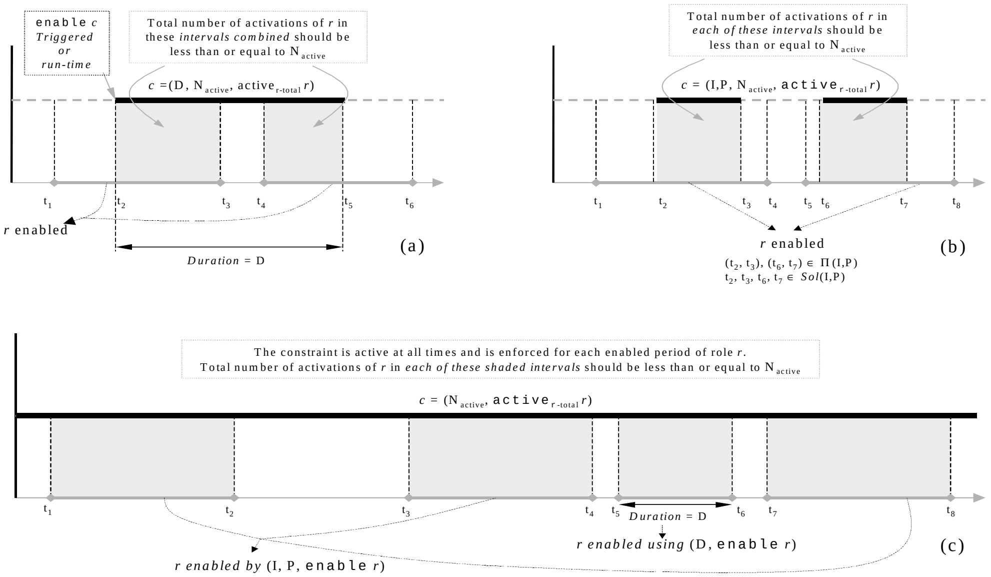
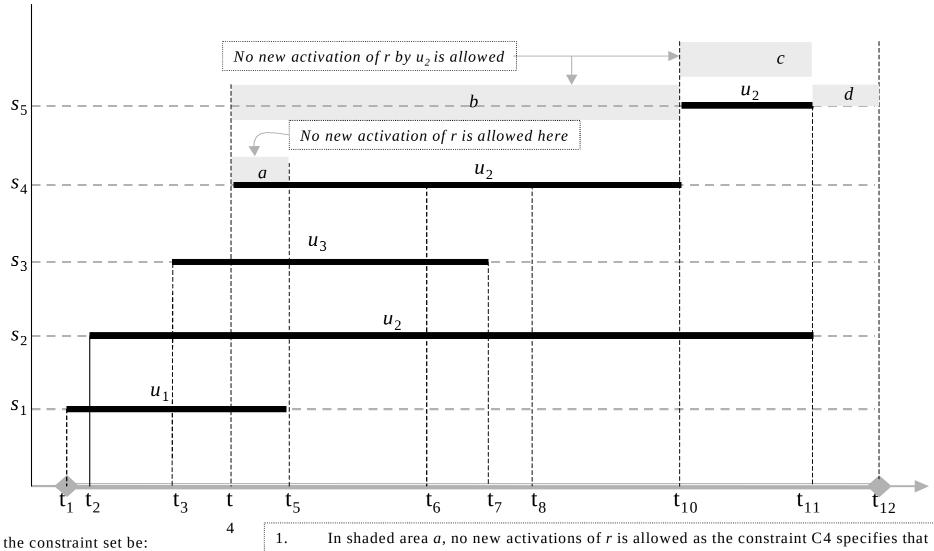

enable c Triggered or run-time
Total number of activations of r in these intervals combined should be less than or equal to N active
c =(D, N active, active r-total r)
t1
t2
t3
t4
t5
t6
r enabled
Duration = D
(a)
Total number of activations of r in each of these intervals should be less than or equal to N active
c = (I,P, N active, active r-total r)
t1
t2
t3
t4
t5
t6
t7
t8
r enabled
(t2, t3), (t6, t7) ∈ Π(I,P)
t2, t3, t6, t7 ∈ Sol(I,P)
(b)
The constraint is active at all times and is enforced for each enabled period of role r.
Total number of activations of r in each of these shaded intervals should be less than or equal to N active
c = (N active, active r-total r)
t1
t2
t3
t4
t5
Duration = D
t6
t7
t8
r enabled using (D, enable r)
r enabled by (I, P, enable r)
(c)

Figure 4. Constraint enabled (a) for a specified duration (b) in specified intervals (c) at all times.


No new activation of r by u2 is allowed
c
s5
b
u2
d
No new activation of r is allowed here
s4
a
u2
s3
u3
s2
u2
s1
u1
t1
t2
t3
t 4
t5
t6
t7
t8
t10
t11
t12

Let the constraint set be:
C1. (7, $\mathtt{active}_{R\_n}$, r),
C2. (4, $\mathtt{active}_{R\_con}$, r),
C3. (2, u_2, $\mathtt{active}_{UR\_n}$, r),
C4. (x, u_2, $\mathtt{active}_{UR\_total}$, r) and
x = sum {(t_2, t_{11}), (t_4, t_{10}), (t_{10}, t_{11})}

1. In shaded area a, no new activations of r is allowed as the constraint C4 specifies that the number of concurrent activations is 4.
2. In shaded portions b and c , there are two concurrent activations of r by u_2- in session pairs (s_2, s_4) and (s_2, s_5). And hence, C3 does not allow any new activation of r by u_2.
3. In d, two activations of r can be allowed, as per C1, but, these activations cannot be by u_2 because at t_{11}, u_2 uses its allocated total activation time as stated in constraint C4.

Figure 5. Activation constraint

An example consisting of a set of activation constraints is depicted in Figure 5. Remaining constraint expressions in the table include run-time requests as per Definition 4.1.2 and triggers as per Definition 4.1.3. The last column indicates the names of the sets that contain corresponding constraint types, e.g., C_{URp} is the set of all periodicity constraints on user-role assignments. We will also use them to represent constraint of certain types, e.g., constraint of type C_{URp} refers to the periodicity constraints on user-role assignments.

The notion of conflicting events plays a crucial role in the semantics of GTRBAC and is formalized by the following definition, which refers to Table 2. The table enlists all the conflicting pairs of events possible in GTRBAC under columns $\boldsymbol{E_1}$ and $\boldsymbol{E_2}$. A pair of events E_1 and E_2 in a row conflict if the corresponding *condition C* holds. Conflicting events cannot happen simultaneously. For example, (*a*) and (*b*) indicate that events "disable *r*" and "enable *r*" conflict with each other.

Table 2. Conflicting events

	E_1	E_2 = **Conf** (E_1)	***Condition* (*C*)**
a.	enable r	disable r'	($r = r'$)
b.	disable r	enable r'	
c.	assign$_U$ r to u	de-assign$_U$ r' to u'	($r = r'$ and $u = u'$)
d.	de-assign$_U$ r to u	assign$_U$ r' to u'	
e.	assign$_P$ p to r	de-assign$_P$ p' to r'	($r = r'$ and $p = p'$)
f.	de-assign$_P$ p to r	assign$_P$ p' to r'	
g.	s:deactivate r for u	s':activate r' for u'	($s = s'$, $r = r'$ and $u = u'$)
h.	s:activate r for u	s':deactivate r' for u'	($s = s'$, $r = r'$ and $u = u'$)
i.		disable r'	($r = r'$)
j.		de-assign r' to u'	($r = r'$ and $u = u'$)
k.	enable c	disable c'	($c = c'$)
l.	disable c	enable c'	

Definition 4.1.4 (**Conflicting Events**): *Let E_1, E_2 be two event expressions. We say that E_2 is a conflicting event of E_1, written as E_2 =* Conf (E_1)*, as shown in* Table 2*, if the corresponding condition C holds true.*

In Table 2, (*h*), (*i*) and (*j*) indicate that there are three events that conflict with an activation event. We note that the conflicts in (*i*) and (*j*) involve different categories of events – activation and disabling or de-assignment events. If there is a request for an activation of a role *r* and at the same time, if there is another event attempting to disable *r*, a conflict results, as both cannot be satisfied simultaneously. Similarly, if there is a user *u* requesting an activation of role *r* and at the same time there is an event that de-assigns the user *u* from role *r*, then both the events cannot occur simultaneously. We use priorities and precedence rules to eliminate conflicts. We also note that events “`enable` *r* ” and “*s*`:deactivate` *r* `for` *u*” do not conflict as both can occur simultaneously. In this paper, unless otherwise specified explicitly, when we refer to a conflicting event of an activation event, we will refer to the conflicting pair (*h*).

We refer to the set of all the event expressions, constraints and triggers in a GTRBAC system as Temporal Constraint and Activation Base (TCAB). We define a TCAB as follows.

Definition 4.1.5 (**Temporal Constraint and Activation Base**): *A Temporal Constraint and Activation Base* (TCAB) *T consists of the following components* :

1. A set of role enabling and assignment constraints *of form* ($I, P, pr{:}E_s$) or ([$I,P,$|$D,$], D_x, $pr{:}E_s$), *where*
   a. *I is a time interval*;
   b. *P is a periodic expression*;
   c. *D_x is the duration restriction on event E_s*;
   d. *D is the duration in which the duration restriction specified by D_x applies to E_s*;
   e. *pr:E is a prioritized event expression with $p \prec \top$.*
2. A set of activation constraints *of the forms* ([$I,P,$|$D,$] C), i.e., (I, P, C), (D, C) or (C),
3. A set of triggers *as per* Definition 4.1.3. □

In the formalization, the user’s run-time requests and the administrator’s run-time requests are modeled as a stream *RQ* of run-time request expressions. *RQ*(*t*) represents the set of run-time requests at time *t*. Here, time points are expressed as integers, starting from 0.

Definition 4.1.5 (**Request Stream**) *A request stream is a sequence RQ* = ⟨*RQ*(0), *RQ*(1),…, *RQ*(t), …⟩, *where each RQ*(*t*) *is a* (possibly empty) *set of run-time request expressions*. □

4.2 Semantics

GTRBAC, like TRBAC, uses the notion of blocked events to resolve conflicts. When priorities cannot resolve conflicts, the model uses *negative*-takes-precedence principle. By this principle, for example, disabling of a role takes precedence over enabling of the role and deactivation of a role takes precedence over the activation of the role.

Definition 4.2.1 (**Blocked Event**, **Nonblocked**) *Let S be a set of prioritized event expressions. Let pr:E be a prioritized event expression. We say that pr:E is blocked by S if there exists* $q \in$ Prios *such that* (q:Conf(E)) $\in$ *S and either*

1. $E \in$ {enable r, assign$_U$ r *to* u, assign$_P$ p *to* r, s:activate r for u, enable c} and $p \preccurlyeq q$, *or*
2. $E \in$ {disable r, de-assign$_U$ r *to* u, de-assign$_P$ p *to* r, s:deactivate r' for u, disable c } and $p \prec q$;

The set of all members of S that is not blocked by S will be denoted by Nonblocked(S). □

We note that the conflict between "disable r" (or de-assign$_U$ r *to* u) and "s:activate r for u" does not create any problem as role activation is a low priority event and will always be blocked by a role disabling event (or deassignment event). However, as the definition of the caused events later shows, care must be taken to ensure that disabling event is not blocked, if it is to block an activation event. The following example illustrates the notion of blocked events.

Example 4.1 Let S = {H:enable r_0,H:disable r_0,VH:enable r_1, H:disable r_1}. Thus, Nonblocked(S) = {H:disable r_0, VH:enable r_1}, since H:enable r_0 is blocked by H:disable r_0, by the first condition of Definition 4.2.1 (which specifies the disable-takes-precedence principle), whereas H:disable r_1 is blocked by VH:enable r_1, by the second condition of Definition 4.2.1

The dynamics of event occurrences, and various states of role enablings and activations in GTRBAC is represented as a sequence of snapshots. Each snapshot models the current set of prioritized events and the status of role, user-role and role-permission assignments as well as that of the activation constraints. To efficiently represent status information within these snapshots, we first define the two structures, called respectively as *u-snapshot* and *r-snapshot.*

Definition 4.2.2 (**u-snapshot**): *We define a u-snapshot for user u with respect to a role r as the tuple* (u, r, d_{ua}, n_{ua}, d_m, n_m, S_u,, D_u)*, where*:

- $r \in$ Roles, $u \in$ Users *such that u is assigned to r,*
- d_{ua} *is the remaining total duration for which u can activate r,*
- n_{ua} *is the remaining number of times that u can activate r,*
- d_m *is the maximum duration for which u can activate r at one time,*
- n_m *is the maximum number of concurrent activations of r that u can have,*

- $S_u = (s_1, s_2, \ldots, s_k)$ *is the list of sessions in which u is currently using r and* $D_u = (d_1, d_2, \ldots, d_k)$ *is the list of durations of activations of r by u in each of these sessions*.

Definition 4.2.3 (**r-snapshot**): *We define an r-snapshot for a role r as the tuple* (r, d_{ra}, n_{ra}, d_{rm}, n_{rm}, *status*, P_r, U_r) *where*:

- $r \in$ `Roles`,
- d_{ra} *is the remaining total active duration for r,*
- n_{ra} *is the remaining total number of activations of r,*
- d_{rm} *is the remaining total active duration for r,*
- n_{rm} *is the remaining total number of activations of r,*
- *status* ∈ {`enabled, disabled`} *is the current status of r.*
- P_r *is the set of permissions that are assigned to r.*
- U_r *is the set of u-snapshot such that, for all* $ut \in U_r$, $ut.r$[2] $= r$. □

By using the above structures, we model events, various role and assignment status, and status of constraints, by three distinct sequences *EV*, *ST* and *CT*, respectively. We note that activation constraints of forms (*D, C*) and (*C*) can be active at certain instants of time. To capture this semantics, we use *CT* to maintain such a list of valid activation constraints.

Definition 4.2.4 (**System Trace**) *A* system trace - *or simply a* trace – *consists of infinite sequences of EV, ST and CT, such that for all integers t* ≥ 0:

- *the t^{th} element of EV, denoted as EV(t), is a set of prioritized event expressions; intuitively, this is the set of events which occur at time t*;
- *the t^{th} element of ST, denoted as ST(t), is a set of r-snapshots corresponding to existing roles at time t. Algorithm* `ComputeST` *in* Figure 6 *is used to compute ST(t) for each t; and*
- *the t^{th} element of CT, denoted as CT (t), is a set of activation constraints of forms* (*C*) *and* (*D, C*) *valid at time t. Algorithm* `ComputeCT` *in* Figure 8 *is used to compute CT*(*t*).

We assume that a system starts at an initial state where all the roles are disabled and there are no user-role assignments, role-permission assignments or valid activation constraints. Such a state exists at time *t* = 0. As the time progresses the events listed in `Table 2` take place changing various role and assignment status and valid activation constraints. The notion of a GTRBAC trace with such an initial state is formalized by a canonical trace defined as follows.

Definition 4.2.5 (**Canonical Trace**) *We say that a* trace *is* canonical *if*

[2] We use *ut.r* to refer to the element *r* of the *u-snapshot ut*.

- *ST*(0) = *set of r-snapshots of the form* (r, ∝, ∝, ∝, ∝, `disabled`, ∅, ∅) *for all roles r in the system, i.e., all r-snapshots are initialized to* (r, ∝, ∝, ∝, ∝, `disabled`, ∅, ∅), *and*
- *CT*(0) = ∅*, i.e., CT is initially empty.*

The above trace definitions enforce the intended semantics of events. The set `Nonblocked`(*EV*(*t*)) contains the maximal priority events that actually happen. We note that the constraints determine a unique state. We can see that the status information contained in *ST*(*t*) concerning the active state of roles depends on the valid activation constraints in *CT*(*t*). Given the previous state, event set and the valid activation constraint set, the following proposition holds.

Proposition 4.2.1 *For all event sequences EV, the initial status* S_0*, and an initial set of valid activation constraints* C_0*, there exists a unique trace* (*EV, ST,CT*) *with* *ST*(0)=S_0 *and* *CT*(0)=C_0.

We use algorithm `ComputeST` shown in Figure 6 to update the status information based on the caused events that are nonblocked. It works as follows. All events in `Nonblocked`(*EV*(*t*)) happen at time *t*. The status information *ST*(*t*) contains effect of the events in `Nonblocked`(*EV*(*t*)) on *ST*(*t*-1). First, all de-assignment events are handled. De-assigning of users and permissions simply involves removing associated *u-snapshots*, and specified permissions associated with an *r-snapshot*.

In Step 3, the disabling of a role is handled. First, the status is changed to `disabled` and. If a *per--role* constraint of form (*C*) is present, the values for d_{ra}, n_{ra} , d_{rm} and n_{rm} are adjusted. $d_{\mathrm{ra}}= \infty$ indicates that in the next role enabling there is possibly no total activation constraint. Next, using the FOR loop, all active user sessions corresponding to the disabled role are removed from the *r-snapshot*. For constraints of forms (*D, C*) and (*I, P, C*), we do not need to make any update, as we simply use value 0 of a constraint variable as implying that the associated constraint cannot be satisfied anymore. In Step 4, *per-role* constraints of type (*I, P, C*) or (*D, C*) that are present in *CT*(*t*-1) but not in *CT*(*t*) are considered. These are constraints that just became inactive. The corresponding constraint variables are reset to ∞.

```
Algorithm ComputeST
Input    : $t, EV, ST, CT$;
Output   : $ST(t)$;
/ * Initially $ST(0) = (r, \propto, \propto, \propto, \propto,$ disabled, $\emptyset, \emptyset)$. For each pair $(r, u)$ we use the associated snapshots $rt = (r, d_{ra}, n_{ra}, d_{rm}, n_{rm}, status, P_r, U_r)$ ,  and $ut = (u, r, d_{ua}, n_{ua}, d_m, n_m, S_u, D_u)$, where $ut \in U_r$. */
Let  $C1 = D_{active}, [D_{default}], active_{R\_total}\ r$;      $C2 = D_{uactive}, u, active_{UR\_total}\ r$;
     $C3 = D_{max}, active_{R\_max}\ r$;                  $C4 = D_{umax}, u, active_{UR\_max}\ r$;
     $C5 = N_{active}, [N_{default}], active_{R\_n}\ r$;      $C6 = N_{uactive}, u, active_{UR\_n}\ r$;
     $C7 = N_{max}, [N_{default}], active_{R\_total}\ r$;     $C8 = N_{umax}, u, active_{UR\_con}\ r$;
STEP 1: FOR each (de-assign $r$ to $u$) $\in$ Nonblocked($EV(t)$) DO
            $U_r = U_r - \{ut\}$;
STEP 2: FOR each (de-assign $p$ to $r$) $\in$ Nonblocked($EV(t)$) DO
            $P_r = P_r - \{p\}$;
STEP 3: FOR each (disable $r$) $\in$ Nonblocked($EV(t)$) DO
            $rt.status$ = disabled;
            IF ((C1)$\in CT(t)$) THEN  $d_{ra} = \infty$;          IF ((C3)$\in CT(t)$) THEN  $d_{rm} = \infty$;
            IF ((C5)$\in CT(t)$) THEN  $n_{ra} = \infty$;          IF ((C7)$\in CT(t)$) THEN  $n_{rm} = \infty$;
            FOR each $ut \in U_r$ DO
                Set $(S_u, D_u)$ to $(\emptyset, \emptyset)$;
                IF (C2 $\in CT(t)$ OR C1$\in CT(t)$) THEN $d_{ua} = \infty$;  IF (C4 $\in CT(t)$ OR C3$\in CT(t)$) THEN $d_m = \infty$;
                IF (C6 $\in CT(t)$ OR C5$\in CT(t)$) THEN $n_{ua} = \infty$;  IF (C8 $\in CT(t)$ OR C7$\in CT(t)$) THEN $n_m = \infty$;
STEP 4 FOR each ((X, C) $\in CT(t-1)$ and (X, C) $\notin CT(t)$) where X $\in \{(I, P), D\}$ and C is a per-role activation constraint DO
            IF C  = C1 THEN $d_{ra} = \infty$;           IF C  = C3 THEN $d_{rm} = \infty$;
            IF C  = C5 THEN $n_{ra} = \infty$;           IF C  = C7 THEN $n_{rm} = \infty$;
STEP 5: FOR each (enable $r$) $\in$ Nonblocked($EV(t)$) DO
            $rt.status$ = enabled;
            IF ([$(I, P)|D$], C1) $\in CT(t)$) THEN $d_{ra} = \min(D_{active}, d_{ra})$;   IF ([$(I, P)|D$], C3) $\in CT(t)$) THEN $d_{rm} = \min(D_{max}, d_{rm})$;
            IF ([$(I, P)|D$], C5) $\in CT(t)$) THEN $n_{ra} = \min(N_{active}, n_{ra})$;   IF ([$(I, P)|D$], C7) $\in CT(t)$) THEN $n_{rm} = \min(N_{max}, n_{rm})$;
STEP 6: FOR each (assign $p$ to $r$ ) $\in$ Nonblocked($EV(t)$) DO
            $P_r = P_r \cup \{p\}$;
STEP 7: FOR each (assign $r$ to $u$ ) $\in$ Nonblocked($EV(t)$) DO
            $U_r = U_r \cup \{(u, \infty, \infty, \infty, \infty, \emptyset, \emptyset)\}$;
STEP 8: FOR each (s:deactivate  $r$ for $u$) $\in$ Nonblocked($EV(t)$) DO
            remove($s, S_u, D_u$);
STEP 9: FOR each  (s:activate  $r$ for $u$) $\in$ Nonblocked($EV(t)$) DO (assume $rt$ for $r$ and $ut$ for $u$ in $rt$)
            $rt.n_{ra} = rt.n_{ra} - 1$;  $ut.n_{ua} = ut.n_{ua} - 1$;
            IF (([$(I, P)|D$], C2) $\in CT(t)$) THEN $d_{ua} = D_{uactive}$;
            ELSEIF (([$(I, P)|D$], C1) $\in CT(t)$) THEN $d_{ua} = D_{default}$;
            IF (([$(I, P)|D$], C4) $\in CT(t)$) THEN $d_m = D_{umax}$;
            ELSEIF (([$(I, P)|D$], C3) $\in CT(t)$) THEN $d_m = D_{default}$;
            IF ([$(I, P)|D$], C6) $\in CT(t)$ THEN $n_{ua} = N_{uactive}$;
            ELSEIF ([$(I, P)|D$], C5) $\in CT(t)$ THEN $n_{ua} = N_{default}$;
            IF ([$(I, P)|D$], C8) $\in CT(t)$) THEN $n_m = N_{default}$;
            ELSEIF ([$(I, P)|D$], C5) $\in CT(t)$ THEN $n_m = N_{default}$;
            $d = \min(d_{ua}, d_m)$;
            add($s, d, S_u, D_u$);
STEP 10: FOR each $r$-snapshot DO
            IF $status$ = enabled
            THEN decrement (durations($r$));       $d_{ra} = d_{ra} - |$sessions$(r)|$;
            ELSE
                $d_{ra} = d_{ra} - 1$;               // for $(I, D, C)$ and $(D, C)$ constraints
                FOR each user assigned to $r$ DO
                    $d_{ua} = d_{ua} - 1$;       // for $(I, D, C)$ and $(D, C)$ constraints
```

We define the following supporting functions for use in the algorithm.

remove(s, S, D), where, s is a session id, $S = \{s_1, s_2, \ldots , s_k\}$ and $D = \{d_1, d_2, \ldots , d_k\}$ is a procedure that computes (S, D) such that $S = S - \{s\}$ and $D = D - \{d\}$, where d corresponds to s.

add(s, d, S, D), where, s is a session id, d is the duration of activation related to s, $S = \{s_1, s_2, \ldots , s_k\}$ and $D = \{d_1, d_2, \ldots , d_k\}$; after processing, we get $S = S \cup \{s\}$ and $D = D \cup \{d\}$.

decrement(D), where $D = \{d_1, d_2, \ldots , d_k\}$ is a set of integers; after processing we get $D = \{d_1-1, d_2-1, \ldots , d_k-1 \}$.

sessions (r) returns a set of sessions $\{s_1, s_2, \ldots , s_k \}$in which role r is currently activated. We can see that **durations**(r) returns a set of active durations $\{d_1, d_2, \ldots , d_k\}$that corresponds to the sessions in sessions(r).

Figure 6. Algorithm computeST

In Step 5, all events that enable roles are considered. The constraints corresponding to the enabled roles are checked and the constraint variables are set. We note that if the constraint variables are equal to ∞, then the values are set according to the current constraint, otherwise it indicated that there is either (*I, P, C*) or (*D, C*) constraint governing the variables and the constraint variables are still not equal to 0. So, each activation should not last more than the total remaining activation duration of the role. Hence the minimum of the remaining role duration and the duration specified by the currently active constraint is taken.

Step 6 just adds the permissions assigned to roles at time *t*. Step 7 adds a new user who is assigned to a role. Note that the variables are initialized to ∞ and the constraint variables for a user are set when actual activation takes place. In Step 8, deactivation of a role by a user is handled by simply removing all the active sessions from the associated *u-snapshot*. Note that Step 8 does not necessarily delete a user session, as the same session may exist in other *r-snapshot*. In Step 9, activation of a role by a particular user is handled. First, the cardinality variables *per-role* and *per-user-role* are decremented to indicate the remaining number of activations allowed after this activation request has been granted. Next, users' constraint variables are initialized and session information is entered in to session list. In step 10, each user's remaining active duration is decremented. The total role duration is also adjusted accordingly. For the roles that are disabled, the duration constraint variables (for both roles and users assigned to them) are simply decremented. This takes care of any activation constraint that is valid at the time the associated role is disabled.

We are left to specify which events must be in *EV*, given a *TCAB T* and a request stream *RQ*. Intuitively, each event should be caused by some element of *T* or *RQ*. When a trigger causes a prioritized event, the event expressions in the body of the trigger must not be blocked. These are formalized by the next definition, which captures the events that are caused at each time instant. An event can be caused by scheduled periodic events, triggers and by the enforcement of activation constraints.

Definition 4.2.6 (**Caused Events**) *The set of caused prioritized events at time t (w.r.t. a given trace, a given TCAB T and request stream RQ), is the least set* `Caused`(*t, EV, ST, CT, T, RQ*) *computed by Algorithm* `ComputeCausedSet` *reported in Figure 7.* □

Algorithm `ComputCausedSet` shown in Figure 7 works as follows. In step 1, all events scheduled via a periodic event are added into set `Caused`(*T, EV, ST, CT, T, RQ*). In step 2, all run-time events, but for the user-activation requests, are handled. The activation requests are handled later because before granting such a request one must ensure that all activation constraints are enforced including the ones that become active at time *t*. In step 3, all events scheduled by a trigger are caused provided the status conditions are satisfied and all event expressions E_is are caused at time (t-Δt). The status information includes the role and assignment as well as the activation status information. Role status can be determined by checking *status* of the corresponding *r-snapshot*. Assignment status can be determined by checking P_r and U_r of the associated *r-snapshot*. A status expression C_i holds if it is true – for example if C_i is '*r active for u*' then it is true if *rt* is the *r-snapshot* corresponding to role *r* and there is a *u-snapshot* $ut \in rt.U_r$ such that $u = ut.u$ *and* $|ut.S_u|>0$. To check if status (`assigned` *r* `to` *u*) holds we check if there exists a *u-snapshot* in $rt.U_r$ such that the *u-snapshot* is associated with user *u*. To determine if status (`assigned` *p* `to` *r*) holds we check if *p* is in $rt.P_r$. In step 4, duration constraints of form $c = (I, P, C)$ are handled. If *t* is in *Sol*(*I*,*P*) then *c* is valid. Additionally if there is a trigger or a run-time request for and event *E* for which *C* specifies the duration restriction, then based on the remaining duration, *E* is caused. Step 5 handles the duration constraint of form c = (*D*, *C*); here, *c* itself must have been enabled by a trigger or a run-time event.

In step 6, the deactivation events that are caused because of activation constraints and/or the presence of a role-disabling or deassignment event are handled. In 6a (6b), we check to see if event "`disable` *r*" ("`de-assign`$_U$ *r* `to` *u*") is already in the caused set. If it is, then it causes *deactivate* events for all active sessions of *r* (activations of *r* by *u*) Note that if the disabling event is to happen (it should not be blocked) then the corresponding deactivations must also happen, which involves removing all active sessions associated with the role. In 6c, we check to see if users meet restrictions on active durations. The sessions of users who have used all the allowed active durations need to be deactivated – hence, the deactivation events are caused by the expiry of the duration limits. In 6d, we check to see if all currently active sessions can be retained till the next time instant based on the remaining total active duration of the role. If not, then some *selection criteria* is used to select sessions that are to be removed by causing deactivation events – such a selection may be based on priorities on user role assignment or on elapsed durations.

```
Algorithm ComputeCausedSet
Input     : $t, EV, ST, CT, T$ and $RQ$ ;
Output    : Caused ($t, EV, ST, CT, T, RQ$)
            /* We will use CausedSet($t$) = Caused ($t, EV, ST, CT, T, RQ$) ;*/
            CausedSet($t$) = = $\emptyset$;
            // Handle periodic events //
Step 1.     FOR each ($I, P, pr{:}E$) $\in T$ DO
                IF $t \in Sol(I, P)$ THEN CausedSet($t$) = CausedSet($t$) $\cup$ {$pr{:}E$};
            // Handle run-time non-activation request //
Step 2.     FOR each ($pr{:}E$ after $\Delta t$) $\in RQ(t-\Delta t)$ DO
                IF $E \neq s$: activate $r$ for $u$ and $0 \leq \Delta t \leq t$ THEN CausedSet($t$) = CausedSet($t$) $\cup$ {$pr{:}E$};
            // Handle triggers //
Step 3.     FOR each trigger [$E_1, \ldots, E_n, C_1, \ldots, C_k \rightarrow p{:}E$ after $\Delta t$ ] $\in T$ DO
            IF ((0$\leq \Delta t \leq t$) & ($\forall C_i$, ($1 \leq i \leq k$) $C_i$ holds at time ($t$ - $\Delta t$)) & ($\forall E_i$ ($1 \leq i \leq n$), $pr{:}E_i \in$ Nonblocked($EV(t - \Delta t)$)))
            THEN CausedSet ($t$) = CausedSet ($t$) $\cup$ {$pr{:}\ E_i$};
            // Handle valid Duration constraints with (I, P)//
Step 4.     FOR each $c = (I, P, D_x, pr{:}E) \in T$ where $x \in$ {U, R, P} and $t \in Sol(I, P)$ DO
                IF ($\exists\ t_1$ such that
                        ($t_1 \in Sol(I, P)$ & $0 \leq \Delta t = (t - t_1) \leq D_x$ &
                         ($\exists$ [$B \rightarrow pr{:}E$ after $\Delta t$] $\in T$ OR $pr{:}E \in RQ(t - t_1)$, by which $pr{:}E \in$ Nonblocked(CausedSet($t - t_1$)))))
                THEN CausedSet ($t$) = CausedSet ($t$) $\cup$ {$pr{:}\ E$};

            // Handle valid Duration constraints with D //
Step 5.     FOR each $c = (D, D_x, pr{:}E) \in T$ where $x \in$ {U, R, P} DO
                IF $\exists\ t_1, t_2$ such that
                        ($t_1 \leq t_2$ & $0 \leq \Delta t_1 = (t - t_1) \leq D$ & $0 \leq \Delta t_2 = (t - t_2) \leq D_x$ &
                         ($\exists$ [$B \rightarrow pr{:}$enable $c$ after $\Delta t_1$] $\in T$ OR $pr{:}$enable $c \in RQ(t - t_1)$
                              because of which enable $c \in$ Nonblocked(CausedSet($t - t_1$))) &
                         ($\exists$ [$B \rightarrow pr{:}E$ after $\Delta t_2$] $\in T$ OR $pr{:}E \in RQ(t - t_2)$, by which $pr{:}E \in$ Nonblocked(CausedSet($t - t_2$)))
                THEN CausedSet ($t$) = CausedSet ($t$) $\cup$ {$pr{:}\ E$};
Step 6.     FOR each r-snapshot $rt$ associated with role $r$ DO
                // Handle the effect of disable r – deactivates all active instances of r //
    6a.         IF ( (disable $r$) $\in$ Nonblocked(CausedSet($t$)) & $rt.status$ = enabled, where rt is r's r-snapshot) ) THEN
                    FOR each $s_i \in ut.S_u$ such that $ut \in rt.U_r$ DO
                        CausedSet ($t$) = CausedSet($t$) $\cup$ {$s_i$: de-activate $r$ for $ut.u$};
                // Handle the effect of the de-assignment events //
    6b.         IF ( ( (de-assign $r$ to $u$) $\in$ Nonblocked(CausedSet($t$)) & (assigned $r$ to $u$) ) THEN
                    FOR each $s_i \in ut.S_u$ such that ($ut \in rt.U_r$ & $ut.\ u = u$) DO
                    CausedSet ($t$) = CausedSet($t$) $\cup$ {$s_i$: de-activate $r$ for $.u$};
                // Remove sessions that expire //
    6c.         FOR each $ut = rt.U_r$., such that $\exists\ s, d$ ($s \in ut.S_u$, and $d \in ut.D_u$ and $d = 0$) DO
                    CausedSet ($t$) = CausedSet ($t$) $\cup$ {$s_i$: de-activate $r$ for $ut.u$);
                // Cardinality control //
    6d.         IF ( ((disable $r$) $\notin$ Nonblocked(CausedSet($t$))& ($rt.status$ = enabled) ) OR
                      ((enable $r$) $\in$ Nonblocked(CausedSet ($t$)) )) THEN
                    $DA$ = |{$e$| $e$ is of type ($s$: de-activate $r$ for $u$') and $e \in$ Nonblocked(CausedSet ($t$))}|;
                    $x = d_{ra}$ - | sessions($r$) - $DA$|;
                    IF $x < 0$ THEN
                        Select a set $U_d$ = {$u_{\pi 1}$, .. , $u_{\pi|x|}$ }using some predefined selection criteria
                        For all $u_i \in U_d$ do CausedSet ($t$) =CausedSet($t$) $\cup$ {$s_i$: de-activate $r$ for $u_i$);
Step 7          // Handle user's activation requests //
                $i = 1$;
                FOR each ($s$: activate $r$ for $u$ after $\Delta t$) $\in RQ(t-\Delta t)$ such that $\Delta t \leq t$ DO
                    IF (( ((disable $r$) $\notin$ Nonblocked(CausedSet($t$))& $rt.status$ = enabled) OR
                          ((enable $r$) $\in$ Nonblocked(CausedSet($t$)) ) & (assigned $r$ to $u$) )
                    THEN
                        Let the associated $rt \in ST(t$-1) be ($r, d_{ra}, n_{ra}, status, P_r, U_r$);
                        $DA$=|{$e$| $e$ is of type ($s$: de-activate $r$ for $u$') & $e \in$ Nonblocked(CausedSet($t$))}|;
                        IF (($d_{ra}$ - $i$ - | sessions($r$)- $DA$| > 0) & ($n_{ra}$ - $i$ > 0) & ($d_{ua}$ - |$S_u$| - 1 > 0) & (|$S_u$| < $n_m$) THEN
                            CausedSet ($t$)=CausedSet ($t$) $\cup$ {$s_i$: activate $r$ for $u_i$);
                            $i = i + 1$;
```

Figure 7. Algorithm computeCausedSet(t)

In step 7, a run-time activation request is accepted as a caused event if the request for activation can be granted. This is possible if the role is enabled and the activation of the event does not violate any activation constraints. The conditions ensure that (a) the role is enabled, (b) the acceptance of the i^{th} activation request will not violate the constraint in the current as well as next time instant. For example if d_{ra} - i - |sessions (r) + DA| = 0, it means that when d_{ra} is updated in the next time instant the total activation time allowed for the role will be used up exactly; in that case only i requests can be granted, rest of the activation events are not caused.

Now we are ready to define the system behavior induced by specific TCABs and request streams. Intuitively, we require each EV (t) to contain all and only those events that have a specific cause.

Definition 4.2.7 (**Execution Model**) *A trace (EV, ST, CT) is an execution model of a TCAB T and a request stream RQ, if for all t* ≥ 0,

$$EV\,(\mathrm{t}) = \texttt{Caused}(t, EV, ST, CT, T, RQ)\ . \qquad \square$$

Unfortunately, some specifications may yield no execution model, whereas some ambiguous specifications may admit two or more such models. However, there are many interesting cases in which the specifications yield exactly one model, for all possible run-time requests. There are simple syntactic conditions that prevent any pathological interplay between conflicting events. Such syntactic conditions - called *safeness* - will be introduced in the next section. Before we introduce the notion of safety, we establish the correctness of the two algorithms discussed above with the following theorems.

For all practical purposes we can assume that the number of users, roles and sessions allowed are finite. Let n_R = |`Roles`| be the number of roles, n_P = |`Permissions`| be the number of permissions, n_U = |`Users`| be the number of users and n_{Sm} = $|\texttt{Sessions}|_m$ be the maximum number of sessions allowed in the system.

Theorem 4.1 (Correctness and Complexity of `ComputeCausedSet`): *Given EV*(t)*, CT*(t)*, ST*(t-1)*, a TCAB T with a finite number of constraints, and RQ*(t) *with a finite number of run-time requests, the following holds true for Algorithm* `ComputeCausedSet`:

1. *it outputs* `Caused`(*t, EV, ST,CT, T, RQ*) *which contains an event E iff at least one of the following holds true:*
   a. *E is caused by a periodicity constraint,*
   b. *E is caused by a run-time request,*

c. E is caused by a trigger,

d. E is caused by a duration constraint,

e. E is a deactivation event that is caused by disabling of a role, de-assignment of a role to a user or an active constraint, or

f. E is an activation event that is not blocked and can be allowed by valid activation constraints at time t.

2. it terminates, and has complexity $O(n_T + n_{RQ} + n_R(n_{Sm} + n_U))$, *where* n_T *is the number of constraint in T at time t,* n_{RQ} *is the number of run-time requests considered at time t.* □

Theorem 4.2 (**Correctness and complexity of** `ComputeST`): *Given EV(t), CT(t), ST(t-1) and TCAB T, the following holds true for algorithm* `ComputeST`:

1. *it produces ST*(t) *such that the updated status of r-snapshots and u-snapshots in ST*(t) *satisfies all the constraints in T and the valid activation constraints in CT for interval* (t, t+1).
2. *it terminates, and has complexity* $O(n_R (n_U + n_P + n_{Sm}))$. □

For practical purposes, we can expect n_{RQ} to be smaller compared to other factors contributing to the complexity. Provided that there is at least one constraint for each role enabling/disabling and user-role or role-permission assignment, we can see that the worst case for the number of use-role and role-permission assignments is $O(n_R . n_U + n_R . n_P))$. Similarly, the worst-case scenario for the number of activation constraints can be expressed as $O(n_R + n_R .n_U)$, considering all the possible *per-role* and *per-user-role* constraints. Thus considering that n_{RQ} and number of triggers in n_T are small compared to the other factors, the complexity expression $O(n_T + n_{RQ} + n_R . n_{Sm} + n_R . n_U)$ of algorithm `ComputeCausedSet` has three major complexity factors, viz, $n_R . n_U$ (i.e., all roles assigned to all users), $n_R . n_P$ (i.e., all permissions assigned to all the roles) and $n_R . n_{Sm}$. (i.e. all the roles are activated in all user sessions). However, this worst-case scenario is also present in any RBAC systems. Generally, in practice, we expect that users will not activate many sessions. Similarly, in practice, not all roles are assigned to every user, and not all permissions are assigned to every role. While handling of triggers in Step 3 may be costly if there is a huge set of triggers, we expect it to be a small set in practice.

The worst case for n_{RQ} can be given as $O(n_R . (n_U + n_P + n_{Sm}))$ considering that all user-role, and role-permission assignments and activation requests are in *RQ*(t). However, this does not introduce any new complexity factor we have not considered above.

We also see that the complexity of `ComputeST` also has the same three key complexity factors and hence discussions above also apply here.

The algorithm `computeCT` shown in Figure 8 is used to compute the active constraints after the caused event set has been computed by algorithm `ComputeCausedSet`.

```
Algorithm computeCT
Input: Caused set, $CT(t\text{-}1)$
Output $CT(t)$
$CT(t) = CT(t\text{-}1)$;
For each (enable $c$) $\in$ Nonblocked(CausedSet($t$))
        $CT(t) = CT(t) \cup \{c\}$; // Note that c = (C) or (D, C)
For each (disable $c$) $\in$ Nonblocked(CausedSet($t$))
        $CT(t) = CT(t) - \{c\}$; // Note that c = (C) or (D, C)
For each c = (C) $\in T$ do
        $CT(t) = CT(t) \cup \{c\}$;
For each  $(I, P, C) \in T$ do
                If $t \in Sol(I, P)$ then   $CT(t) = CT(t) \cup \{c\}$;
```

Figure 8. Algorithm `computeCT`

4.3 Safe TCABs

Next, we introduce a syntactic condition that can be verified in polynomial time and guarantees that a given TCAB has one and exactly one execution model. The notion of dependency graph is essential for this purpose.

Definition 4.3.1 (**Labeled Dependency Graph**) [5]: *Each TCAB T is associated with a* (directed) labeled dependency graph $DG_R = \langle N, E\rangle$ *where*:

- *N* (*set of nodes*) *coincides with the set of all prioritized event expressions pr:E that occur in the head of the trigger* $[B \rightarrow pr{:}E] \in T$;
- *E* (*the set of edges*) *consists of the following triples*[3], *for all triggers* $[B \rightarrow pr{:}E] \in T$, *for all events E' in the body B, and for all nodes* $q{:}E' \in N$
    - $\langle q{:}E', +, pr{:}E\rangle$
    - $\langle r{:}\mathrm{Conf}(E'), -, pr{:}E\rangle$, *for all* $[r{:}\mathrm{Conf}(E')] \in N$ such that $q \preccurlyeq r$. □

Definition 4.3.2 (**Safeness**): *A TCAB T is safe if its dependency graph* DG_R *contains no cycles in which some edge is labeled '-'.*

It can be shown using the results from [5] that if a TCAB *T* is safe, then the system's behavior is unambiguously determined by *T*, *RQ* and the initial status of the roles and assignments.

[3] Each triple (N_1, l, N_2) represents an edge from node N_1 to N_2, labeled by *l*.

We note that safeness is a sufficient condition for good system behavior. Further, it is difficult to find necessary conditions and even if they are found, they offer little practical help, because such syntactic properties (such as graph-based ones) fail to recognize that ill-formed portions of the program may be harmless because they can never be activated [5]. Checking model existence and model uniqueness are, in general, NP-hard problems.

```
Algorithm SafetyCheck
Input: : a TCAB  $T$
Output: true if $T$ is safe, false otherwise
begin
/* construction of the dependency graph */
$N$ := 0;  $E$:= 0;
for all $[B \rightarrow pr{:}E] \in T$ do
        if ($E$ = s:activate $r$ for $u$) then return false;
        $N := N \cup \{pr{:}E\}$;
for all $[B \rightarrow pr{:}E] \in T$ do
        for all $E' \in B$ such that $\exists q; q{:}E' \in N$ do
                $E := E \cup \{\langle q{:}E', +, pr{:}E_i\rangle\}$;
                for all $r$:conf($E'$) $\in N$ such that $q \preccurlyeq r$ do
                        $E := E \cup \{\langle r{:}\mathrm{conf}(E'), -, pr{:}E_i\rangle\}$
/* cycle generation and checking */
SCC := strongly connected components of $\langle N, E\rangle$
for all $\langle N', E'\rangle \in$ SCC do
    for all $\langle X, l, Y\rangle \in E'$ do
                if l = '-' then return false;
return true
end
```

Figure 9: Algorithm for safeness verification

Algorithm `SafetyCheck` illustrated in Figure 8 is used for the safeness verification of a TCAB. The first part of the algorithm builds the dependency graph associated with T, and the second part checks for cycles with a negative edge. The correctness of the algorithm can be simply proved from the results reported in [5]. We note the following with respect to the labeled dependency graph:

- Dependency graph construction takes polynomial time. Such complexity can be reduced to $O(|T|.|N|)$ by representing the graph as an ordered vector, which can be sorted in time $O(|N|.\log|N|)$.
- The strongly connected components of the graph can be determined in $O(|N| + |E|)$ time (cf. [7]). As the total number of edges is bounded by $|E|$, the second phase of the algorithm has cost $O(|N| + |E|)$.
- As each node must occur in some trigger's head, $|N| = |T|$ and $|E|$ is in $O(|T|^2)$.

From this, we see that the algorithm's complexity is in $O(|T|^2)$. We note that the number of iterations of the innermost loop of the graph construction phase is bounded by a constant (i.e., |`Prios`|) for a fixed set of priorities. Hence, for a given set of priorities, the cost of the safeness verification check drops down to O(|*T*|. log|*T*|).

Further improvements to the costs of safeness verification can be achieved by adopting incremental graph construction methods, and by caching the set of strongly connected components [6]. Insertion or update of a trigger simply involves insertion/deletion of only a few edges, and the consideration of only the old strongly connected components containing the events in the new/updated trigger if no new nodes are created.

5 Implementation Architecture

In this section, we present implementation architecture of a system supporting the proposed GTRBAC model. The architecture extends the one proposed in [5] by including required functionalities to handle the new temporal constraints.

A critical design issue is to determine at each time the set of roles that a user can activate and/or the duration for which the user can activate, according to various constraints and triggers contained in the TCAB, and the run-time requests issued till that point.

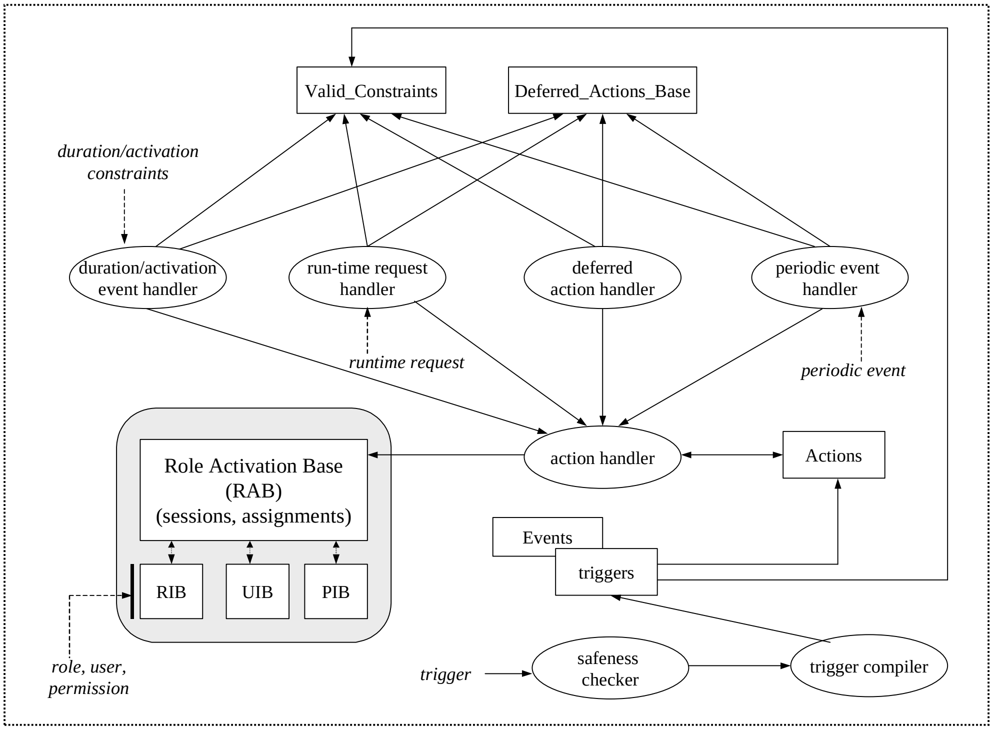


Figure 10. System architecture

A request by a user to activate a role is granted if the user is assigned to the role at that particular time, and there are no activation constraints enabled at that time, preventing the activation of the role by the user. User-role and role-permission assignments, role enabling, constraint enabling as well as the activations/deactivations of roles by users can be either immediately executed, or deferred by a fixed time interval. Moreover, priorities are associated with each event, and such priorities must be considered when dealing with conflicting actions. Figure 10 shows the architecture of the system we have developed for supporting GTRBAC constraint on top of a commercial DBMS. The figure shows data structures as rectangles and functional components as ovals. Arrows represent interactions among the various entities. Various components are described in the following section.

5.1 Data structures

Information on role activations and deactivations, periodic events and run-time requests are maintained in the database tables. More precisely, the data maintained by the system are:

1. *Role_Activation_Base*: This is a set of tables that actually implements the *r-snapshot* representation of each role in the system. The tables contain information about valid assignments, currently active user-role sessions and their corresponding durations as well as various constraint parameters related to activation constraints. A sample set of tables for RAB is shown in Figure 11.

   As shown in *Table 1* of Figure 11, the values of the constraint parameters change as different per-role constraints become active. Similarly *Table 2* shows user-role assignments. The parameters refer to the *per-user-role* constraints. *Table 3* lists the current sessions and corresponding duration for each activation of a role and *Table 4* shows the valid permission assignments. In Figure 10, role information base (RIB), users information base (UIB) and permission information base (PIB) indicate static reservoir of roles, users and permissions along with supporting description. Although they are not strictly needed, such a separation from RAB can be advantageous particularly because all tables in RAB are essentially dynamic and are affected by constraints and triggers in TCAB, and run-time requests.

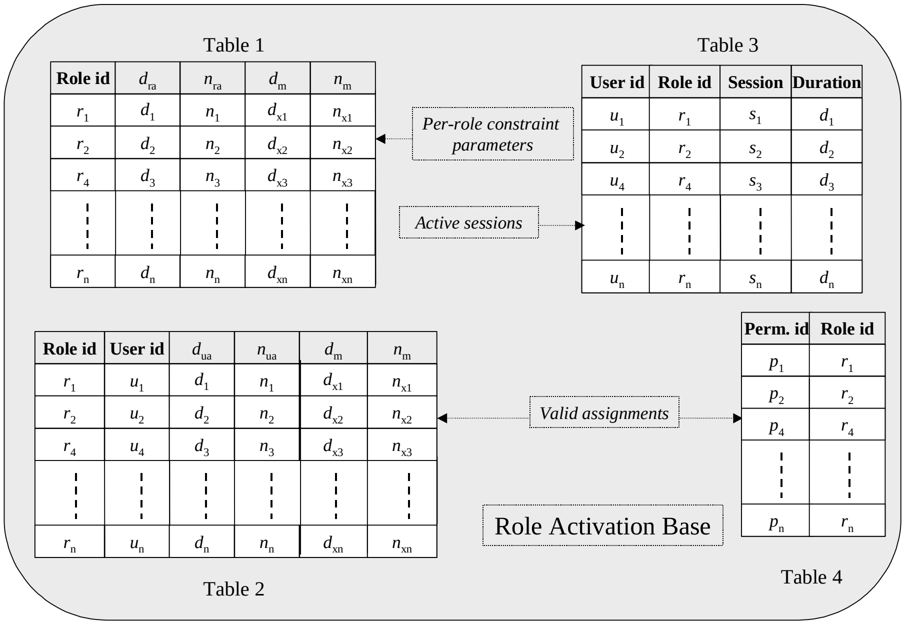


Table 1

Role id	d_{ra}	n_{ra}	d_m	n_m
r_1	d_1	n_1	d_{x1}	n_{x1}
r_2	d_2	n_2	d_{x2}	n_{x2}
r_4	d_3	n_3	d_{x3}	n_{x3}
⋮	⋮	⋮	⋮	⋮
r_n	d_n	n_n	d_{xn}	n_{xn}

Table 3

User id	Role id	Session	Duration
u_1	r_1	s_1	d_1
u_2	r_2	s_2	d_2
u_4	r_4	s_3	d_3
⋮	⋮	⋮	⋮
u_n	r_n	s_n	d_n

Role id	User id	d_{ua}	n_{ua}	d_m	n_m
r_1	u_1	d_1	n_1	d_{x1}	n_{x1}
r_2	u_2	d_2	n_2	d_{x2}	n_{x2}
r_4	u_4	d_3	n_3	d_{x3}	n_{x3}
⋮	⋮	⋮	⋮	⋮	⋮
r_n	u_n	d_n	n_n	d_{xn}	n_{xn}

Table 2

Perm. id	Role id
p_1	r_1
p_2	r_2
p_4	r_4
⋮	⋮
p_n	r_n

Table 4

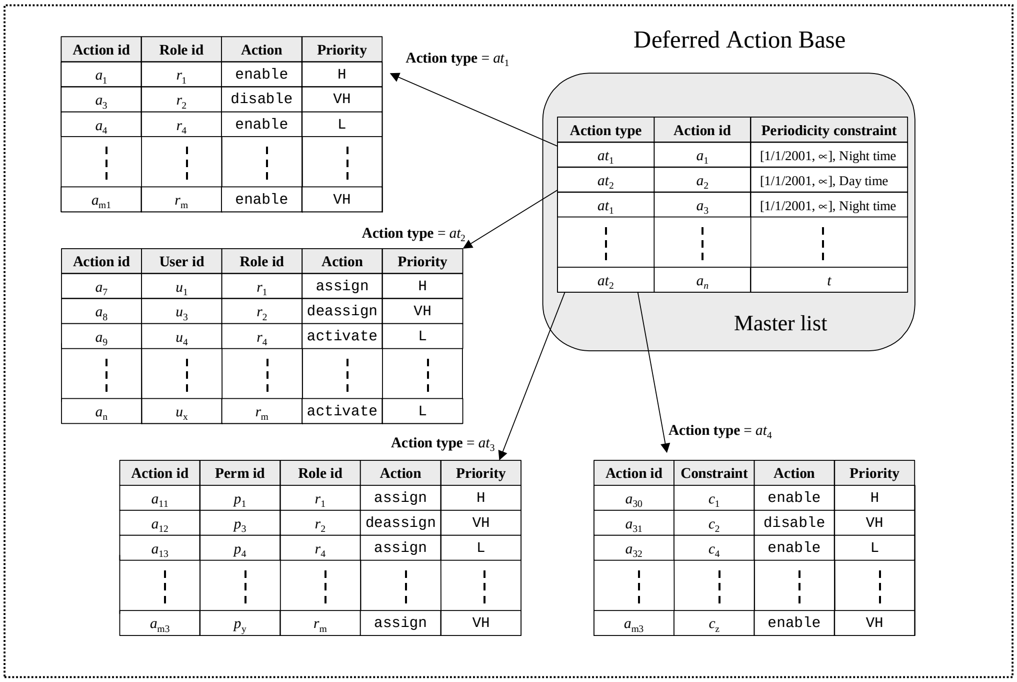
Figure 11. Role activation tables

2. *Deferred_Actions_Base*: This is a set of tables that store actions that need to be executed after a certain amount (or a period) of time. For each action, the time instant (or the periodic set of instants) at which it has to be executed is recorded along with the priority of the action. To efficiently store different kinds of event actions, more than one table may be desirable. Figure 12 shows a sample set of DAB tables. The main table is essentially the master list of all actions with the temporal constraint information. The detailed action information is stored in different tables according to the type of action. For example, **Action Type** = at_1 represents the enabling and disabling of roles and the table corresponding to this store information about the role, action to perform (enabling or disabling) and the priority associated with the action. In Figure 12, the table that corresponds to **Action Type** = at_2 is for both user-role assignment and role activation by users (they can be separate if necessary). We also note that an `activate` request can only come from a run-time user request and hence the associated time of the request is stored in the master list (for example the last entry *t*). The remaining two tables are associated with the role-permission assignment and constraint enabling. If a deferred action is caused by a trigger, the time of firing of the trigger is stored in the master list. We also note that a duration constraint *c* of type (*D*, *C*) can be enabled only by a run-time request or a trigger. Hence, whenever such a constraint is enabled, it can be stored in the corresponding DAB table along with the time at which such a run-time request is made or a trigger fired.

3. *Valid_Constraint*: It is a table that stores valid constraint at a particular time. A constraint can be valid based on the periodic expression that specifies the intervals in which it is valid or when it is enabled through a run-time request or a trigger.

4. *Actions*: It is a table that records all the actions to be potentially executed and corresponds to the `Caused` defined earlier. Such actions can be caused by a trigger, a run-time request, or a periodic/duration event. Note that all events caused at a particular time need be non-blocked. This table is necessary in cases where conflicting events are caused. In such a scenario, some of the events are invalidated.

5. *Events*. This is a global event base, which records all the events of the system.



Action type	Action id	Periodicity constraint
at_1	a_1	[1/1/2001, ∝], Night time
at_2	a_2	[1/1/2001, ∝], Day time
at_1	a_3	[1/1/2001, ∝], Night time
⋮	⋮	⋮
at_2	a_n	t

Action id	Role id	Action	Priority
a_1	r_1	enable	H
a_3	r_2	disable	VH
a_4	r_4	enable	L
⋮	⋮	⋮	⋮
a_{m1}	r_m	enable	VH

Action id	User id	Role id	Action	Priority
a_7	u_1	r_1	assign	H
a_8	u_3	r_2	deassign	VH
a_9	u_4	r_4	activate	L
⋮	⋮	⋮	⋮	⋮
a_n	u_x	r_m	activate	L

Action id	Perm id	Role id	Action	Priority
a_{11}	p_1	r_1	assign	H
a_{12}	p_3	r_2	deassign	VH
a_{13}	p_4	r_4	assign	L
⋮	⋮	⋮	⋮	⋮
a_{m3}	p_y	r_m	assign	VH

Action id	Constraint	Action	Priority
a_{30}	c_1	enable	H
a_{31}	c_2	disable	VH
a_{32}	c_4	enable	L
⋮	⋮	⋮	⋮
a_{m3}	c_z	enable	VH

Figure 12. Deferred action tables

5.2 Functional modules

The system consists of seven main modules (Figure 10), namely the *Safeness Checker*, *Trigger Compiler*, the *Periodic Event Handler*, the *Duration/activation Event Handler*, the *Run-time Request Handler*, the *Deferred Action Handler*, and the *Action Handler*. Next, we describe each of the modules.

Safeness checker: Whenever a trigger is inserted or modified, the *Safeness Checker* is activated. It determines whether such an operation can be allowed. In case the operation makes the TCAB unsafe then the operation is rejected by the system. An incremental version of the algorithm in Figure 9 that does not rebuild the dependency graph whenever such a trigger is added can be used. If the insertion/modification of a trigger does not make the TCAB unsafe, the *Safeness Checker* passes the inserted trigger to the *Trigger Compiler*. Deletion of a trigger does not make the TCAB unsafe and hence requires no safeness checks.

Trigger Compiler: *Trigger Compiler* is used to translate the inserted trigger into an equivalent underlying database trigger attached to table *Events*. The translation depends on the number and types of expressions appearing in the trigger body and on the type of action that the firing of the trigger causes. If the action caused by the trigger has to be immediately executed, the corresponding database trigger inserts the action into table *Actions*, along with its action priority. However, if the action has to be deferred, the firing of the corresponding database trigger will cause the insertion of the action into *Deferred_Actions_Base* along with its priority and the information about the time instant at which the action has to be executed. Note that trigger can enable some constraints. Such enabled constraints are inserted into the *Valid_Constraints* table. The *Trigger compiler* is an extended version of the *Trigger Compiler* described in [5]. The extension is aimed at incorporating all types of events possible in GTRBAC. For detailed description of trigger generation for the GTRBAC in Oracle Object-database systems, we refer the readers to [5].

Periodic Event Handler: Whenever a periodic event is inserted into TCAB, the *Periodic Event Handler* inserts a corresponding entry into *Deferred_Actions_Base*. The entry contains the action requested by the periodic event, its priority, and the associated periodic constraint. Note that some duration constraints have periodic expressions that determine when the constraints are valid. Corresponding entries for these are also inserted into the *Deferred_Actions_Base*. A deletion is handled simply by removing the corresponding entry from the *Deferred_Actions_Base*.

Duration/activation Event Handler: When a duration constraint for role enabling/assignment or an activation constraint is inserted, it is first processed by the *Duration/activation Event Handler.* If the constraint is of form (*C*), it means the constraint is immediately enabled (not restricted by (*I,P*) or (*D*)) and hence it is put into *Valid_Constraint*. If the constraint is of the form (*I, P, C*), then it checks to see if it is enabled at that time. If it is then the constraint (*C*) is inserted into *Valid_Constraints* and (*I*, *P*, *C*) is inserted into *Deferred_Action_Base*, otherwise only (*I*, *P*, *C*) needs to be entered in *Deferred_Action_Base*.

Run-time Request Handler: Run-time requests are processed by the *Run-time Request Handler*. If the request needs to be immediately processed, i.e., if $\Delta t = 0$, it sends the action to the *Action Handler*, otherwise it puts an entry into the *Deferred_Action_Base* along with the time of request and its priority.

Deferred Action Handler: Execution of the deferred periodic actions that the firing of a trigger, the issuing of a run-time request, or a periodic/duration event can cause, is handled by the *Deferred Action Handler*. It essentially monitors the *Deferred_Action_Base* to execute the actions it contains at appropriate times. Such a module can be implemented as a daemon that maintains a list of instants at which it has to wake up. With regards to the periodic actions, the *Deferred Action Handler* maintains the first instants at which the action has to be executed and the time period between any two consecutive executions of the action. Whenever a new entry is inserted into *Deferred_Action_Base* these time instants are updated. When the daemon wakes up, the actions that have to be carried out at that time instant are selected. It selects the action with the highest priority if If there are conflicting actions (in case of equal priority *negatives-take* precedence rule is used). Then, it returns the action along with its priority to the *Action Handler*.

Action Handler: The *Action Handler* is the core of the system, as it is in charge of updating *Role_Acivation_Base* according to the actions requested by the other modules or caused by the firing of the triggers associated with the *Events* table. As conflicts may arise among actions, the *Action Handler* collects all the required actions into table *Actions*. It resolves any conflicts before updating the *Role_Acivation_Base*. Note that the *Action handler* essentially extracts the non-blocked events caused at the particular time and implements algorithm `computeST` to update the *Role_Acivation_Base*. The remaining functional modules each implements a part of the algorithm `comaputeCausedSet`. However, each is augmented with functionalities to determine the deferred events and inserting it into the *Deferred_Action_Base* tables.

6 Related Work

Need for supporting constraints in an RBAC model has been addressed by many researchers. In particular, the attention has been in supporting *separation of duties* (SoD) constraints [1, 4, 9, 10, 12, 13, 15, 16, 17, 21]. SoD constraints are mainly aimed at reducing the risk of a fraud by not allowing any individual to have sufficient rights to perpetrate such frauds. Ferrariolo *et. al.* [8] propose an RBAC model that supports the cardinality constraints. In [1], Ahn *et. al.* propose *RCL2000* – a role based constraint specification language. Bertino *et. al.* have proposed a logic based constraint specification language that can be used to specify constraint on roles and users and their assignments to workflow tasks [4]. In [9], the workflow model of [4] has been extended. However, none of these include temporal constraints in their specification models.

The TRBAC model proposed by Bertino *et. al*. [5], is the first known model that addresses the temporal constraints. It, however, provides constraints only on role enabling and triggers, considerably limiting its use in many diverse real world application requirements. The work presented here is a generalization of the TRBAC model and constitutes a substantial enhancement over it. Another related work is the access control model presented by Bertino *et. al.* in [3] that supports temporal authorization and derivation rules. Formalism for periodic time used in this paper has been borrowed from [3, 11].

7 Conclusions and future work

We have proposed a generalized temporal role based access control model that can handle a comprehensive set of temporal constraints. The model allows temporal constraints on role enablings and role activations. Various temporal restrictions can be specified on user-role and role-permission assignments. We have also illustrated through examples the applicability of the GTRBAC temporal constraints.

We plan to extend the present work in various directions. The first direction is an extensive investigation on how various temporal constraints on roles affect the inheritance semantics of a role hierarchy. Role hierarchy is a highly beneficial feature of an RBAC model and hence it is very essential to have a proper notion of temporal hierarchy in a GTRBAC framework.

Another key issue that needs to be extensively investigated is the issue of whether such a huge set of temporal constraint is beneficial from a practical as well as theoretical perspective. We plan to establish through theoretical analysis that having the current set of constraints is beneficial from the perspective of user-convenience and better constraint representation, even though we believe that the model may not be minimal. We plan to investigate and identify such a minimal model that has the least set of temporal constraint with the same expressive power as that of the current set of GTRBAC constraints. We believe that the current set of GTRBAC constraints provide a flexible, more intuitive, convenient and computationally better representation of temporal constraints than such a minimal model, which we believe exists. Finally, we also plan to develop an SQL-like language for specifying the GTRBAC temporal.

References

[1] G. Ahn, R. Sandhu. Role-Based Authorization Constraints Specification. *ACM Transactions on Information and System Security*, Vol. 3, No. 4, November 2000.

[2] V. Atluri editor. *Proc. of the Fourth ACM Workshop on Role-Based Access Control*, Fairfax (VA), 1999.

[3] E. Bertino, C. Bettini, E. Ferrari, P. Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transactions on Database Systems*, 23(3):231-285, September 1998.

[4] 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):65-104, 1999.

[5] E. Bertino, P. A. Bonatti, E. Ferrari. TRBAC: A Temporal Role-based Access Control Model. *ACM Transactions on Information and System Security* (TISSEC) 4(3), August 2001 (in print).

[6] T.H.Cormen, C.E.Leiserson, R.L.Rivest. Introduction to Algorithms. MIT Press, 1990.

[7] D. F. Ferraiolo, D. M. Gilbert, and N Lynch. An examination of Federal and commercial access control policy needs. In *Proceedings of NISTNCSC National Computer Security Conference*, pages 107--116, Baltimore, MD, September 20-23 1993.

[8] D. Ferrariolo, J.F. Barkley, and D.R. Kuhn. A Role-based Access Control Model and Reference Implementation within a Corporate Intranet. *ACM Transactions on Information and System Security*, 2(1):34-64, 1999

[9] S. Kandala and R. Sandhu. Extending the BFA Workflow Authorization Model to Express Weighted Voting. In *Research Advances in Database and Information Systems Security*, pages 145-159, Kluwer Academic Publishers, 1999.

[10] D.R. Kuhn. Mutual Exclusion of Roles as a Means of Implementing Separation of Duties in a Role-based Access Control System. *ACM Transactions on Information and System Security*, 2(2):177-228, 1999.

[11] M. Niezette and J. Stevenne. An efficient symbolic representation of periodic time. In *Proc.First International Conference on Information and Knowledge Management*, 1992.

[12] M. Nyanchama and S. Osborn. The Role Graph Model and Conflict of Interest. *ACM Transactions on Information and System Security*, 2(1):3-33, 1999.

[13] S. Osborn editor. *Proc. of the Fifth ACM Workshop on Role-Based Access Control*, Berlin, Germany, July 2000.

[14] S. L. 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*, Vol. 3, No. 2, February 2000.

[15] R. Sandhu. Separation of Duties in Computerized Information Systems. In *Database Security IV: Status and Prospects*, pages 179-189. North Holland, 1991.

[16] R. Sandhu editor. *Proc. of the First ACM Workshop on Role-Based Access Control,* Fairfax (VA),1995.

[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, pages 65-79.

[18] R. Sandhu editor. *Proc. of the Second ACM Workshop on Role-Based Access Control*, Fairfax (VA), 1997.

[19] R. Sandhu editor. *Proc. of the Third ACM Workshop on Role-Based Access Control*, Fairfax (VA), 1998.

[20] R. Sandhu. Role-based Access Control. *Advances in Computers*, vol. 46, Academic Press, 1998.

[21] R. Simon and M.E. Zurko. Separation of Duty in Role-based Environments. In *Proc. 10th IEEE Computer Security Foundations Workshop*, June 1997.

Appendix

Proof of Theorem 4.1 - **Proof of part (1)**

(⇒) Assume that $E \in$ `Caused`(t, EV, ST, CT, T, RQ). We have to show that E is caused by one of the constraints in (a)-(f). We show this by cases.

Case 1. Let E = `enable`(`disable`) r

First we note that event E can be added to `Caused`(t, EV, ST, CT, T, RQ) at five different places, the THEN parts of the statements in steps 1-5, through the execution of the statement

$$\texttt{Caused}(t, EV, ST, T, RQ) = \texttt{Caused}(t, EV, ST, T, RQ) \cup \{pr{:}E\}. \qquad \text{(s1)}$$

First, assume that $E \in$ `Caused`(t, EV, ST, CT, T, RQ) because of the execution of statement (s1) in Step 1. This means that the condition specified in FOR and IF parts of Step 1 are true. However, the FOR and IF conditions imply that there is a periodicity constraint $(I, P, pr{:}E) \in T$ for which $t \in Sol(I,P)$ hold true. Hence, E is caused by a periodicity constraint.

Assume that $E \in$ `Caused`(t, EV, ST, CT, T, RQ) because of the execution of statement (s1) in Step 2. This means that the conditions specified in FOR and IF parts of Step 2 are true. But these conditions imply that there is a runtime request ($pr{:}E$ `after` Δt) made at time (t -Δt). Hence, E is caused by a runtime request.

Suppose $E \in$ `Caused`(t, EV, ST, CT, T, RQ) because of the execution of statement (s1) in Step 3. This means that the conditions specified in FOR and IF parts of Step 3 are true. This means there is a trigger [E_1 ,…, E_n , C_1 ,…, $C_k \rightarrow$ $p{:}E$ `after` Δt] in T at time (t -Δt). Furthermore, the conditions imply that each of the events E_i is non-blocked at time (t -Δt) and each of the status expressions C_is holds true at time (t -Δt). Hence, by the semantics of triggers, [E_1 ,…, E_n , C_1 ,…, $C_k \rightarrow$ $p{:}E$ `after` Δt] is the cause for event E..

Similarly, assume that $E \in$ `Caused`(t, EV, ST, CT, T, RQ) because of the execution of statement (s1) in Step 4. Then, as the conditions specified in FOR part of Step 4 must be true, it means that there is a duration constraint $c = (I, P, D_U, pr{:}E)$ in T valid at time t as indicated by the FOR condition. Similarly, the IF condition implies that, the event E is caused at time t_1, which is a valid time instant of (I, P) (as indicated by the condition $t_1 \in Sol(I,P)$). Since $(t\text{-}t_1) \leq D_U$ and the event E is triggered or requested at run-time, it implies that event E can be allowed under this duration constraint. Hence, E is caused (or allowed) by the duration constraint c.

Finally, assume that $E \in$ `Caused`(t, EV, ST, CT, T, RQ) because of the execution of statement (s1) in Step 5. Then, the conditions specified in FOR part of Step 4 imply that there is a duration constraint $c = (D, D_R, pr{:}E)$ in T valid at time t. Furthermore, truth of the IF condition implies that constraint c has been enabled by a trigger or a run-time request at time t_1 and as $(t - t_1) \leq D$, it means that the valid duration D of the constraint c hasn't expired. The remaining part of the IF condition implies that E is triggered or requested at run-time at t_2 and because $(t - t_2) \leq D_R$ and $t_1 \leq t_2$, it follows that c allows E to be caused. Hence, E is caused (or allowed) by a duration constraint.

As the remaining steps do not put E = `enable(disable)` r into `Caused`(t, EV, ST, CT, T, RQ), they cannot cause E. The cases for E = "`assign`$_{UR}$(`deassign`$_{UR}$) u `to` r", E = "`assign`$_{PR}$(`deassign`$_{PR}$) p `to` r" are similar to the case for E = "`enable(disable)` r". Hence, the proof for them is similar to that given above. The case for '`enable(disable)` c' is the same as for `enable(disable)` r, except that it is not caused by a periodicity or duration constraint. Hence, for '`enable(disable)` c', we need to focus only on steps 2 and 3, the arguments for which are, again, the same as that for '`enable(disable)` r'.

Case 2: $E = s$:`deactivate` r `for` u

In this case E can be in `Caused`(t, EV, ST, T, RQ) only through the execution of the following statement in steps 6(a) - 6(d):

$$\texttt{Caused}(t, EV, ST, T, RQ) = \texttt{Caused}(t, EV, ST, T, RQ) \cup \{\ s{:}\texttt{deactivate}\ r\ \texttt{for}\ u\ \} \qquad \text{(s2)}$$

We note that steps 6(a), - 6(d) repeat for each role as indicated by the FOR loop at Step 6**.** Now, assume that $E \in$ `Caused`(t, EV, ST, CT, T, RQ), because of the execution of statement (s2) in Step 6(a). This means the IF condition of Step 6(a) is true, which implies that '`disable` r' is a non-blocked event at that time and user u has an active session of role r at t. Since '`disable` r' is non-blocked, r will be disabled at t, and there cannot be any active user session for role r henceforth. Therefore, all currently active sessions need to be deactivated. This causes the event E to be added to `Caused`(t, EV, ST, T, RQ). Hence, the disabling of a role causes E. The argument is similar for 6(b), except that here the non-blocked de-assignment of a role to a user is considered and only the active sessions of the user are removed.

Next, assume that $E \in$ `Caused`(t, EV, ST, CT, T, RQ) because of the execution of statement (s2) in Step 6(c). This means the FOR condition of Step 6(Cc) must be true, which implies that the activation of role r by u cannot be allowed further, as the total active duration (because of an activation constraint) allowed for the user has expired (d = 0). Hence, E is caused by an active per-user-role constraint, i.e., by (e).

Finally, assume that $E \in E \in$ `Caused`(t, EV, ST, CT, T, RQ) because of the execution of statement (s2) in Step 6(d), so both the inner and the outer IF conditions are true. The outer IF condition ensures that role r will be non-blocked at time t. As the inner IF condition is also true, E is selected by the pre-defined *selection criteria*. The condition x < 0, indicates that the total role active duration allowed will be exceeded if the activation of role r by u is not cancelled by deactivating it. Thus, in this case, E is caused by the total role active duration.

Case 3: E = s: `activation` r `for` u

In this case E can be in $E \in$ `Caused`(t, EV, ST, CT, T, RQ) only through the execution of the following statement in Step 7:

`Caused`(t, EV, ST, CT, T, RQ) = `Caused`(t, EV, ST, CT, T, RQ) $\cup$\{ s:`activate` r `for` u \}(s3)

We note that Step 7 repeats for each role as indicated by the FOR loop at Step 6**.** Since $E \in$ `Caused`(t, EV, ST, CT, T, RQ) can only be because of the execution of statement (s3) in Step 7, it follows that all conditions in FOR and IF statements in Step 7 must be true. The FOR condition implies that there is a runtime request for activating role r for user u, as s: `activate` r `for` u `after` $\Delta t \in RQ(t\text{-}\Delta t)$. The outer IF condition ensures that the role r will be enabled at time t and there is a valid user-role assignment. Let E be the i^{th} activation event caused (or allowed). The condition (d_{ra} - i - | `sessions`(r)- DA| > 0) implies that by causing E, the system will not violate the total role active duration constraint. Similarly, since (n_{ra} - i > 0) is true it implies that the cardinality restriction for the role is not violated either. The condition d_{ua} - $|S_{\text{u}}|$ - 1 > 0 is true implies that *per-user-role* constraint for total active duration is not violated by granting the user activation request. And lastly, ($|S_{\text{u}}| < n_{\text{m}}$) ensures that the cardinality restriction specified for the user u as to how many concurrent activations of role r s/he can have, is not violated. Thus, E is caused by the combination of a run-time request and the activation constraint, i.e., by (b) and (f).

($\Leftarrow$) Assume that one of (a)-(f) causes event E . We need to show that $E \in$ `Caused`(t, EV, ST, CT, T, RQ).

Case 1. *E is caused by* (*a*), *i.e. periodicity constraint*.

Since E is caused by a periodicity constraint, it means there is a constraint (I, P, $pr{:}E$) $\in T$ such that $t \in Sol(I, P)$. All such constraints are handled in Step 1, as FOR loop repeats for all such constraints. The IF condition is satisfied as $t \in Sol(I, P)$ and therefore, the THEN part of the IF statement is executed adding event E into `Caused`(t, EV, ST, T, RQ). Thus, after the execution of the IF statement (Step 1), $E \in$ `Caused`(t, EV, ST, CT, T, RQ).

Case 2. $E \neq$ `activate` r `for` u, *is caused by* (*b*), *i.e. by a run-time request other than the activation request*:

Since E is caused by a run-time request, the event E must be in RQ at the time when the request is made. The request made is in the form "$pr{:}E$ after Δt". Therefore a request "$pr{:}E$ after Δt" must be in $RQ(t\text{-}\Delta t)$ for E to be caused at time t by it. This condition is precisely the one used in the FOR loop for picking up all such events that need to be caused at time t in Step 2. The IF condition further checks if the request was made at or before time t and is satisfied. Therefore, the THEN part of the statement adds the event E in `Caused`(t, EV, ST, T, RQ). Hence, $E \in$ `Caused`(t, EV, ST, CT, T, RQ) after Step 2.

Case 3. *E is caused by* (*c*), i.e. *by a trigger*.

Since E is caused by a trigger, a trigger of form "[$B \rightarrow pr{:}E$ `after` Δt]" must be in $RQ(t\text{-}\Delta t)$ for $0 \leq \Delta t \leq t$. Furthermore, all the events in B must have been caused and non-blocked at time (t-Δt), and all the conditions in B must hold true at time (t-Δt). The conditions for the IF statement in Step 3 checks for these and hence are all satisfied. Therefore, the THEN part of the IF statement executes adding event E in `Caused`(t, EV, ST, CT, T, RQ), i.e., $E \in$ `Caused`(t, EV, ST, CT, T, RQ) after Step 3.

Case 4. *E is caused by* (*d*), i.e. *by a duration constraint*

Since GTRBAC has two forms of duration constraint, we will take each one separately. First assume that E is caused by a duration constraint of the form $c = (I, P, D, pr{:}E)$. This means $t \in Sol(I, P)$, because the constraint c must be valid at time t, and (I, P) precisely defines those instants at which c is valid. Furthermore, for E to be caused by (or allowed under) constraint c, there must be a non-blocked event E, which is either triggered or present in RQ at time t or earlier, but at a time instant that is in $Sol(I, P)$. The IF condition of Step 4 precisely checks for such a time instant t_1; furthermore the duration should not be expired, i.e. $t\text{-}\ t_1 \leq D$. hence, the IF condition is satisfied. It is possible that both the enabling of the constraint and the triggering or run-time request of the event *E* can also take place at time *t*. The algorithm will handle this correctly because Step 1 through Step 3 captures both these types of events. Thus, the THEN part of the statement in Step 4 will be executed adding the event *E* into `Caused`(*t, EV, ST, CT, T, RQ*) after esuring that $t - t_1 \leq D$, and hence, $E \in$ `Caused`(*t, EV, ST, CT, T, RQ*) after Step 4.

Now assume that *E* is caused by (or allowed under) a duration constraint of form $c = (D, D_x, pr{:}E)$. Since the constraint *c* itself is valid for only a particular duration *D*, there must be an event "`enable` *c*" that has been caused as the result of which *c* is valid at time *t*. A trigger or a run-time request must have caused such an "`enable` c" event. Hence, one of the condition that checks for such event is satisfied in the IF statement of Step 5. Furthermore, *D* and D_x, should not be expired implying that we must have ($t- t_1 \leq D$) and ($t- t_2 \leq D_x$), where t_1 is the time at which *c* is enabled and t_2 is the time at which *E* is caused. Furthermore, $t_1 \leq t_2$ ensures that D_x is contained in *D*. This implies that the conditions in the IF statement are satisfied. Hence, the THEN part of the IF statement is executed, adding the event *E* in `Caused`(*t, EV, ST, T, RQ*). We note that if the event *E* is triggered at time *t* for the first time since *c* became active, then it is added in `Caused`(*t, EV, ST, CT, T, RQ*) two times, once in Step 2 or Step 3 and another in step 5; but since `Caused`(*t, EV, ST, CT, T, RQ*) is a set, it is represented only once. Hence, $E \in$ `Caused`(*t, EV, ST, CT, T, RQ*) after step 5.

Case 5. *E is caused by* (*e*), i.e. *E is a deactivation event that is caused by disabling of a role or active constraints,*

First, we consider *E* caused by the disabling of a role. That is, *E* is an event "*s*:`deactivate` *r* `for` *u*" associated with role *r*. Since it is caused by disabling of a role, "`disable` *r*" must be a non-blocked event in $E \in$ `Caused`(*t, EV, ST, CT, T, RQ*). Furthermore, as deactivation involves removing an active session, the role must have been in enabled state in interval (*t*-1, *t*). Thus the IF condition of Step (6a) is true. Hence, the THEN part of Step (6a) executes, adding event *E* into `Caused`(*t, EV, ST, CT, T, RQ*). Therefore, $E \in$ `Caused`(*t, EV, ST, T, RQ*) after Step (6a). The deactivation events produced by a deassignment statement in Step 6(b) can be similarly outlined.

Next, assume that deactivation event *E* is caused because of activation constraints. Then, *E* must have been caused to remove active user sessions that have expired or that cannot be retained because of some duration or cardinality constraint on role.

If E is caused because the session has expired then the remaining duration corresponding to the expired session is 0. Thus, the condition that is used by the FOR loop is satisfied for E in Step (6c). As a result, the statement inside the FOR loop is executed, $E \in$ `Caused`(t, EV, ST, T, RQ).

If E is caused in order to ensure that the duration constraint on the associated role is not violated in interval (t, t+1), then E must have been selected by the *selection criteria* used to remove active user sessions so that the sessions allowed altogether do no violate the activation constraint. The fact that E has been caused means that the corresponding role r will remain enabled at t and after t. Thus, '`disable` r' should not be a non-blocked event at time t and afterwards, or r should be enabled at time t. The first IF condition of Step (6d) checks for this and hence is true. The *selection criteria* is used only if the remaining total number of sessions so far (given by |`session`(r) –DA|) cannot be allowed to be active for one more time unit. Thus, the inner IF condition is satisfied too. Hence, the event E is added in `Caused`(t, EV, ST, CT, T, RQ) by the THEN part. Therefore, $E \in$ `Caused`(t, EV, ST, CT, T, RQ) after Step 6(d).

Case 6. *E is caused by (f), i.e. E is an activation event that is not blocked and can be allowed by valid activation constraints at time t.*

Since E is caused at time t, there must be a run-time request at time (t-Δt), i.e, "E `after` Δt" is in RQ(t-Δt) such that $0 \leq t \leq \Delta t$. Thus, the condition attached to the FOR loop of Step 7 is satisfied. Since E is caused, it must be true that the associated role r is enabled at t and afterwards, which means that the "`disable` r" is not a non-blocked event and r has already been in enabled state, or, alternatively, there is a non-blocked "`enable` r" event caused at time t. In addition, there must be a valid assignment of roles to users for granting an activation request. The first IF condition checks precisely for these; hence, it is satisfied. Since E is caused, it implies that the activation constraints can be satisfied. Hence, the inner IF condition is satisfied as it checks to see if the addition of the new activation request can be granted. Since all the conditions are satisfied, the THEN part of the inner IF statement is executed, adding E to `Caused`(t, EV, ST, T, RQ). Hence, $E \in$ `Caused`(t, EV, ST, T, RQ).

Proof of Theorem 4.1 - <u>Proof of part (2) and (3)</u>

Steps 1-5 each look for a particular type of constraint in T and and events in RQ. Let n_{dRQ} and n_{aRQ} be the number of activation and deactivation run-time requests scheduled for time t, and n_{admin} be the number of administrator's runtime events scheduled for t, then we have $n_{RQ} = n_{admin} + n_{aRQ} + n_{dRQ}$. Steps 1-5 check for all constraints, triggers and the runtime request other than the activation requests. And hence, total time taken for steps 1-5 is (n_T + n_{admin} + n_{dRQ}), considering n_T includes the cost for checks involving triggers in Step 3.

FOR loops of steps 6(a) - 6(c) each repeat at most n_{Sm} times, as each looks for all applicable sessions for each user who has activated role *r* and for whom the conditions are satisfied. The FOR loop of 6(d) is also bounded from above by n_U (we can consider *selection criteria* algorithm to be linear in the number of users). Hence, for steps 6(a) - 6(d), the maximum complexity is n_R (n_{Sm}+ n_U) as these steps are repeated for each *r*. Step 7 considers all the activation constraints, hence giving a complexity of n_{aRQ} (note that the fact that step 7 is inside the FOR loop of step 6 does not matter). Hence, the complexity is (n_T + n_{admin} + n_{dRQ}) + n_R (n_{Sm}+ n_U) + n_{aRQ}. That is, the complexity is $O(n_T + n_{RQ} + n_R\ (n_{Sm}+ n_U))$. Since each of the above terms is finite, the algorithm terminates.

Proof of Theorem 4.2: **Proof of part 1**

By theorem 4.1, the `Caused`(*t, EV, ST, CT, T, RQ*) produced by `ComputeCausedSet`(*t*) contains only those events that are caused at time *t* and satisfy all the constraints in *T*. Hence, to prove that the status update done by `computeST`(*t*) satisfies all the constraints in *T*, we need to prove that the update done is with respect to each of the non-blocked events in the `Caused`(*t, EV, ST, CT, T, RQ*). This is because the non-blocked events of `Caused`(*t, EV, ST, CT, T, RQ*) are the only events that actually happen at time *t*. Since by definition 4.2.7, *EV*(*t*) = `Caused`(*t, EV, ST, CT, T, RQ*), we can proceed by showing that each of the events of `nonblocked`(*EV*(*t*) is considered by the algorithm. In addition, the algorithm needs to ensure that all the valid activation constraints in *CT* are also satisfied by the updated information, which is to say that the effect of all such constraints are considered by the algorithm. We will proceed step by step.

Step 1 and 2: In these steps, all the non-blocked deassignment events of *EV*(*t*) are considered. Since the presence of a *u-snapshot* associated with user *u* in U_r corresponding to the role *r* indicates that the role *r* has been assigned to user *u*, the removal of the *u-snapshot* corresponding to the user to whom the role *r* is deassigned correctly updates the effect of the deassignment event. Since *ut* contains all the activation status of a particular user associated with a role, the removal of the *u-snapshot ut* completely removes all such information. Similarly, in step 2, permissions that are deassigned from a role are removed from the permission list P_r associated with the role. The next earliest change to these new values of U_r and P_r occur at time *t*+1. We also note that no activation constraints affect deassignments. Hence steps 1 and 2 are in accord to condition (1).

Step 3: In this step, all non-blocked events that disable roles are considered. Since a role *r* is disabled at *t*, its status is changed to `disabled`, which remains so until it is changed later, as there are no statements below step 3 that changes *rt.status*; the only change that can occur to this parameter afterwards is at time (*t*+1), which occurs if "`enable` *r*" event is non-blocked at time (*t*+1). As a role is disabled, all valid *per-role* activation constraints whose validity is not restricted by (*I*, *P*) or a duration *D* (and hence they are valid for each enabled duration of the role), must be reset to default values. This is done by each of the next set of IF statements. Furthermore, each *per-user-role* constraints must also be considered and corresponding associated parameter values reset. This is done by the FOR loop which considers each of the users assigned to the role being disabled. We note that, for *per-role* activation constraints of type (*I, P, C*) and (*D, C*), the updating of the corresponding parameter values is dictated by (*I*, *P*) and *D* respectively as is done in step 4 and are not affected by the disabling of a role. Hence, step 3 does the required update in accord with condition (1).

Step 4: The fact that a constraint of form (*I*, *P*, *C*) or (*D*, *C*) which was in *CT*(*t*-1) but is not in *CT*(*t*) implies that these constraints are not in effect anymore (disabled). This step considers all such *per-role* activation constraints and reset the values of the corresponding parameters such as d_{ra}, n_{ra}, etc., to ∞. These parameter values can only be altered at *t*+1 or later; hence, the update is in accord with condition (1).

Step 5: This step handles the effect of the enabling of a role *r*. First the status is updated to `enabled`. The first IF condition checks if a constraint of type *C1* in one of forms (*I*, *P*, *C*), (*D*, *C*), or (*C*), is present. If it is, then d_{ra} is updated to the minimum of D_{active} in *C1* or the current value of d_{ra}. If $d_{ra} < \infty$ then it implies that some valid constraint is restricting the role activation time and is left unchanged. If $d_{ra} = \infty$, then it means no constraint of type *C1* has been active earlier. Thus the update essentially conforms to the semantics and hence is in accord with condition (1). Similar arguments apply to the remaining IF statements of step 5.

Step 6: This step simply adds the permission *p* to the set P_r associated with the role *r* if "`assign` *p* `to` *r*" is a non-blocked element in *EV*(*t*). The presence of *p* in set P_r associated with the role *r* indicates that *p* is assigned to *r* and hence this update correctly establishes the status of the assignment until it is changed by a deassignment at time *t*+1 or after.

Step 7: This step simply adds a *u-snapshot*, say *ut*, associated with user *u* to the set U_r that is associated with the role *r* if "`assign` *r* `to` *u*" is a non-blocked element in *EV*(*t*). The presence of *ut* in set U_r associated with the role *r* indicates that *u* is assigned to *r* and hence this update correctly establishes the status of the assignment. Furthermore, the set of sessions and their durations are currently empty. The remaining parameters are set to initial default values of ∞. Thus, the updates reflect that an assignment event occurs and is in accord with condition (1).

Step 8: This step simply updates the effect of deactivation events. For each deactivation event, there is a session and the duration associated. Thus for a (*s*:`deactivate` *r* `for` *u*), the associated session *s* and its associated duration *d* are removed from the *u-snapshot* of *u* associated with role *r*. Note that session *s* may still be present in which other roles are still active. In such a case, the session *s* will be present in the *u-snapshot* of *u* associated with the other roles. Thus removal of (*s, d*) is in accord with condition (1).

Step 9: In this step, all non-blocked activation events are considered. Since a new activation of a role is being added, if there is a *per-role* cardinality constraint active at this time ($rt.n_{ra}$<∝), then $rt.n_{ra}$ must be decremented, as required. The new value will be checked to control role activation events in the next run of `computeCauseSet` at (*t*+1). Note that we do not need to check if $rt.n_{ra}$=0, because the reason the activation event is in `nonblocked`(*EV*(*t*)) is because no cardinality constraint has blocked the activation event, as is done in step 7 of `computeCauseSet`. Hence, the simple decrement operation is adequate. Similar argument applies the *per-user-role* cardinality constraint (the associated parameter is $ut.n_{ua}$).

Each of the IF statements that follows updates the user parameter based on the type of *per-user-role* or *per-role* constraint. The first IF statement checks to see if a *per-user-role* total activation constraint is active at time *t*. If it is active, then the corresponding user parameter d_{ua} is updated as specified. If the constraint form is (*C1*) then this value will be restricted for the duration d_{ua} or till the time the corresponding role *r* is disabled (in which case, step 3 will reset this value when it is updated next). The ELSE part considers the *per-role* constraint that applies to the activation of *r* by *u*. In that case, the default value specified is assigned to d_{ua}. We note that this default value, if not explicitly specified, is equal to the value specified for the role, for example, D_{active} in this case). Hence, the IF statement updates the user parameter as required. Similar arguments apply to the rest of the IF statements. The remaining duration for the activation of *r* by *u* is then added to sets S_u and D_u, so that they can be decremented at each time instant until they become 0 (in which case the `computeCausedSet()` will remove it in step 6(c)). Hence, step 8 updates the required parameters in accord with condition (1).

Step 9: In this step, for each enabled role, the duration of each session is decremented. We note that the decrement will not make any individual duration negative because such a possibility is eliminated by Step 6(c) in `computeCausedSet`. Similarly, the total active duration of the role is also adjusted. The decrement value represents the total value that will be decremented at the end of the interval (t, t+1). The else part simply decrements the value of d_{ra} by one. This is necessary because there may be a *per–role* constraint of type (I, P, C) or (D, C) on the role which is valid even when the role is disabled. Similarly, each user values are also decremented. Thus, step 9 updates all the duration values as required.

Hence, it follows that the condition of (1) is satisfied by $ST(t)$ produced by the algorithm.

<u>Proof of part 2 and 3</u>

We look at the complexity of each step and sum them up to get the overall complexity. The FOR loop of step 1 repeats for each of the deassignment events that is non-blocked. At worst, it repeats ($n_R.n_U$) times. Similarly, the worst case for step 2 is ($n_R.n_P$). Step 3 handles role-disabling events. At worst all roles need to be disabled. The inner FOR loop repeats for all users assigned to each role. Thus, again the worst case for step 3 is ($n_R.n_U$). Step 4 checks for four different types of *per-role* activation constraints active at the time. Since, there are n_R roles, the maximum number of such constraints is 4n_R. The FOR loop of step 5 repeats at most n_R times. Similar to step 2, step 6 has worst case of ($n_R.n_P$). Similar to step 1, step 7 has worst case of ($n_R.n_U$). Each of steps 8 and 9 is bounded by the maximum number of sessions allowed in the system ($n_R.n_{Sm}$). Step 10 repeats for each role, and the worst case occurs when the else part is executed, giving the worst case of ($n_R.n_U$). Because n_R, n_U, n_P and n_{Sm} are each finite, we see that each step terminates. Hence the algorithm `computeST` terminates. Furthermore, the overall complexity of the algorithm is:

$$= \quad n_R.n_U + n_R.n_P + n_R.n_U + 4n_R + n_R + n_R.n_P + n_R.n_U + 2\, n_R.n_{Sm} + n_R.n_U$$

Thus the complexity of the algorithm can be expressed as:

$$O(n_R.(n_U + n_P + n_{Sm})).$$