available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

# Security analysis of GTRBAC and its variants using model checking

*Samrat Mondal*[a,*], *Shamik Sural*[b], *Vijayalakshmi Atluri*[c]

[a] *Information and Communication Technology, DAIICT Gandhinagar, India*
[b] *School of Information Technology, IIT, Kharagpur, India*
[c] *MSIS Department and CIMIC, Rutgers University, USA*

ARTICLE INFO

*Article history:*
Received 1 April 2010
Received in revised form
20 August 2010
Accepted 10 September 2010

*Keywords:*
Computation tree logic
GTRBAC
Timed automata
Verification
Safety property
Liveness property

ABSTRACT

Security analysis is a formal verification technique to ascertain certain desirable guarantees on the access control policy specification. Given a set of access control policies, a general safety requirement in such a system is to determine whether a desirable property is satisfied in all the reachable states. Such an analysis calls for the use of formal verification techniques. While formal analysis on traditional Role Based Access Control (RBAC) has been done to some extent, recent extensions to RBAC lack such an analysis. In this paper, we consider the temporal RBAC extensions and propose a formal technique using timed automata to perform security analysis by analyzing both safety and liveness properties. Using safety properties one ensures that something bad never happens while liveness properties show that some good state is also achieved. GTRBAC is a well accepted generalized temporal RBAC model which can handle a wide range of temporal constraints while specifying different access control policies. Analysis of such a model involves a process of mapping a GTRBAC based system into a state transition system. Different reduction rules are proposed to simplify the modeling process depending upon the constraints supported by the system. The effect of different constraints on the modeling process is also studied.

## 1. Introduction

In a multiuser environment, access control is required for controlled sharing and protection of resources. Over the past few decades, many access control models have been proposed to specify the different access control policies. These primarily include Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC). Among these, RBAC has gained considerable attention due to its flexibility, ease of administration and intuitiveness. RBAC can be viewed as a state transition system where state changes occur via administrative operations.

With the proliferation of wireless technology and mobile networking several new access control requirements have come up. The traditional RBAC model is not capable of handling such requirements. Today, for instance, it is often required that access to a particular resource is given on the basis of the current time or location of the subject making the request. Due to such growing requirements, the RBAC model has been extended in various dimensions. These extensions offer flexibility in specifying a variety of security policies. In the temporal domain two significant access control models have been proposed. Bertino et al. (2001) introduced TRBAC, which is an extension of RBAC in which a user may activate a role only during certain time intervals. In TRBAC, only temporal constraints on role activation and temporal dependencies among roles are considered. As a result, only a limited number of temporal policies can be specified using this model.

* *Corresponding author.*
E-mail addresses: samrat_mondal@daiict.ac.in (S. Mondal), shamik@cse.iitkgp.ernet.in (S. Sural), atluri@rutgers.edu (V. Atluri).
0167-4048/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2010.09.002

Joshi et al. (2005a) later proposed a more general model known as the Generalized Temporal RBAC (GTRBAC).

In this model it is possible to specify a wider range of constraints, which include temporal constraints on role enabling, role activation, role hierarchy, Separation of Duty (SoD) and role trigger. Bhatti et al. (2005a,b) augment the GTRBAC model with XML to allow for supporting the policy enforcement in a heterogeneous, distributed environment. Their model is known as XGTRBAC. Another model on top of GTRBAC is XGTRBAC Admin (Bhatti et al., 2005a,b). It allows decentralization of policy administration tasks.

Mere development of RBAC models does not ensure that a given system satisfies all the desirable security properties under different conditions. The extensions of RBAC lack such analysis. The fundamental question of safety is to answer the following: "Given the current authorization state and the policy specification, will a user (subject) ever gain access to a specific resource (object)? " Thus, the basic aim of security analysis is to check whether the system maintains desirable security properties in all the states. The security properties are formulated from a set of access control policies. A formal verification technique then can be used to ensure the consistency of an access control specification. Security analysis provides answers to the queries whether an undesirable state is reachable or all the reachable states satisfy the desirable properties. In the context of an RBAC model, a typical safety query would be "Whether a role remains enabled forever" or "Whenever a role is enabled, is another role also enabled?"These queries can be classified primarily into two types, namely safety and liveness. With safety property, we try to ensure that "something bad should never happen". With liveness property we try to capture that "finally something good should happen". When the analysis techniques consider only safety properties, then we call it as safety analysis. Security analysis generalizes the notion of safety analysis in the sense that it considers not only safety properties but also other types of properties.

Recently, some work has been initiated on formal analysis of temporal and spatiotemporal extensions of access control models. In this paper, we propose a generalized approach which can be used to model and analyze several temporal features in addition to the standard role based access control features. For this purpose we have explored the characteristics of timed automata. The approach maps a given temporal RBAC system to a timed automata based system and various properties are analyzed for the resulting system using model checking verification mechanism.

The rest of the paper is organized as follows. Section 2 comprises of the preliminaries of temporal extensions of RBAC, its categorization and a brief overview of timed automaton. In Section 3 the structure framework for proposed approach and the detailed mapping mechanism is discussed. The algorithms required for mapping are presented in Section 4. Section 5 illustrates the completeness of the mapping process. The reduction rules are proposed in Section 6. A general form of safety and liveness query is given in Section 7. Complexity analysis is made in Section 8. Experimentations and analysis are carried out in Section 9. Section 10 summarizes prior work done in this area. Section 11 concludes this paper and provides future direction for work.

## 2. Preliminaries

### 2.1. *Temporal RBAC extensions and their categorization*

The first temporal RBAC extension, the TRBAC model has been proposed by Bertino et al. (2001). Later Joshi et al. (2005a) have introduced a comprehensive set of temporal constraints by proposing the Generalized TRBAC (GTRBAC) model. Since this is the most general temporal RBAC extension proposed, we consider all the features supported by this model as a base line. In GTRBAC model the distinction is made between role enabling and role activation. A role is in the *enabled* state if it is ready for user assignment and the role is in the *disabled* state if the users cannot activate the role. A role is said to be *active* if there is at least one user who has assumed that role. Once a role is in active state, reactivation of the role does not change the state of the role. When all the users deactivate the role then the role moves to the *enabled* state. Several important aspects that can be handled in the TRBAC models are given below.

- Temporal constraints on
  1. *Role Enabling*: Specify valid time instances during which a role is enabled and disabled.
  2. *Role Activation*: Allow when a role will be activated or deactivated.
  3. *User to Role Assignment*: Control when a user may be assigned to a role.
  4. *Permission to Role Assignment*: Determine when a permission is assigned to a role.
- *Run time events*: A run time event can dynamically initiate an action. It is initiated by a user. Role activation is such an example because this is done at the user's discretion.
- *Triggers*: Express dependency among different events. For example, enabling of a role may force to enable of another role.

In addition to this, the TRBAC extensions also support the following constraints as per the NIST RBAC standard (Ferraiolo et al., 2001).

- *Role Hierarchy* (*RH* ): A user of a senior role can acquire the permissions associated with a junior role.
- *Separation of Duty* (*SoD*): Two or more conflicting roles cannot be activated by a user at the same time.

### 2.2. *Categorization of the temporal extensions of RBAC*

In order to study the impact of different constraints on the temporal extensions of the TRBAC models, we categorize them into the following based on the properties they support. The categorization is driven by the fact that the cost of analysis is different for each of these categories.

1. $GTRBAC_{-SoD}$: Does not support the Separation of Duty constraints. However, all other features of GTRBAC are included in this model.
2. $GTRBAC_{-RH}$: Does not include role hierarchy constraint.
3. $GTRBAC_{-PA\_time}$: Does not support the temporal constraints on permission-role assignment relation.

In the next section, the details of mapping from GTRBAC to timed automata based model are presented.

### 2.3. Timed automata

In this section we give a brief overview of timed automaton and some of the fundamental ideas related to it. These are deemed essential for understanding the modeling mechanism and the analysis process presented in this paper. Alur and Dill (1994) introduce the idea of timed automaton which is a finite state machine equipped with a set of clocks. All the clocks are synchronized, i.e., they advance at the same pace. The clocks can take any non-negative real number in $\mathbb{R}$. If $C$ is a finite set of clock variables, then a clock valuation $v$ over $C$ is a function $v : C \rightarrow \mathbb{R}$ which associates with every clock $c$, its value $v(c) \in \mathbb{R}$.

**Definition 1**. *Timed Automaton: It is a 6-tuple ($L$, $l_0$, $C$, $A$, $E$, $I$) where:*

- $L$ is a finite set of control states, also called locations.
- $l_0 \in L$ is the initial location.
- $C$ is a finite set of clocks.
- $A$ is a set of actions.
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a finite set of transitions. An element $e \in E$ can be expressed as $e = (l_1, a, g, r, l_2)$ which represents a transition from $l_1$ to $l_2$, $g$ is a guard which is a conjunction of boolean expressions involving clocks or some other variables, $r$ is the set of clocks that is reset by $e$, and $a$ is the action of $e$. The transition can also be represented by $(l_1, a, g, r, l_2)$.
- $I{:}l \rightarrow B(C)$ assigns invariants to locations.

In the context of timed automaton, another important definition is *Deterministic Timed Automaton*. It is represented by a deterministic timed transition table.

**Definition 2**. *Deterministic Timed Automaton: A timed automaton is deterministic if*

- There is only one initial location, i.e., $|L_0| = 1$,
- No edges labeled with $\epsilon$, i.e., some input conditions must be required for any transition and
- Two edges with same source and same label must have disjoint guards.

At every point in time the future behavior of a timed automaton is determined by its state and the values of all its clocks. This introduces the idea of region automaton.

**Definition 3**. *Region Automaton: Given a timed automaton $A = (L, l_0, C, A, E, I)$, the corresponding region automaton $R(A)$ is defined as*

- The states of $R(A)$ are of the form $\langle l, \alpha \rangle$ where $l \in L$ and $\alpha$ is a clock region,
- The initial states are of the form $\langle l_0, [v_0] \rangle$ where $l_0 \in L_0$ and $v_0(x) = 0$ for all $x \in C$,
- *$R(A)$ has an edge $\langle \langle l, \rangle, \langle l', \alpha' \rangle, a \rangle$ if there is an edge $\langle l, a, \delta, \lambda, l' \rangle \in E$ a region $\alpha''$ such that (1) $\alpha''$ is a time successor of $\alpha$, (2) $\alpha''$ satisfies $\delta$, and (3) $\alpha' = [\lambda \mapsto 0]\alpha''$.*

Each state of the region automaton can be represented in space polynomial in the description of the input automata. A *Timed Automaton* and its corresponding *Region Automaton* are shown in Fig. 1.

A system can be described as a composition of several automata. Alur and Dill (1994) have shown that it is possible to construct a timed automaton, termed as *implementation automaton $A_I$* which can represent the composite process. The size of $A_I$ is exponential in the size of each component automaton. The specification of a property can also be represented by a *deterministic timed automaton*, denoted by $A_S$. The specification is represented by S which is a $\omega -$ language over the alphabet $\Sigma$. The $\omega -$ language is a subset of $\Sigma^\omega$. If we want to represent a GTRBAC safety property – "whenever time is between 10:00 and 17:00 a role is always in *enabled* state" then the corresponding deterministic timed automaton is shown in Fig. 2.

The implementation meets the specification iff $L(A_I) \subseteq L(A_S)$.

**Lemma 1**. *For a timed automaton $A_1$ and a deterministic timed automaton $A_2$, the problem of deciding whether $L(A_1)$ is contained in $L(A_2)$ is PSPACE-complete.*

*Proof*

$L(A_1) \subseteq L(A_2)$ holds iff the intersection of $L(A_1)$ with the complement of $L(A_2)$ is empty. If $A$ be a timed automaton from the product of $A_1$ and $A_2$ then size of $A$ is proportional to the product of the sizes of $A_i$ where $i = 1, 2$. A region automaton, $R(A)$ is constructed from A. $L(A_1) \subseteq L(A_2)$ iff R(A) has a cycle which is accessible from its start state, meets the progressive requirement, the acceptance criterion for $A_1$, and the complement of the acceptance criterion for $A_2$. The existence of such a cycle can be checked in space polynomial in the size of A Alur and Dill (1994).

## 3. Mapping of GTRBAC to timed automata based model

### 3.1. Proposed framework

Our approach is based on a structured framework as depicted in Fig. 3. To perform the security analysis, at first the given set of policies is considered. The policies are next evaluated to check for conflict and ambiguity. Next, a possible set of components and constraints of the given GTRBAC system is identified. Also, relations among the different components are checked. Next, two tasks are performed in parallel. First is a mapping where the components and constraints are mapped to the timed automata based model. This is explained in detail in the next subsection. Secondly, a desirable set of security properties is generated from the given set of policies. This is discussed in Section 7. In the proposed approach, the properties are classified as safety and liveness properties. We have expressed the security properties using Computation Tree Logic (CTL). In the next stage, the properties are verified against the timed automata based model using the model checking mechanism. In the verification process, if there exists some desirable properties which are not satisfied by the model then the whole procedure is repeated again starting

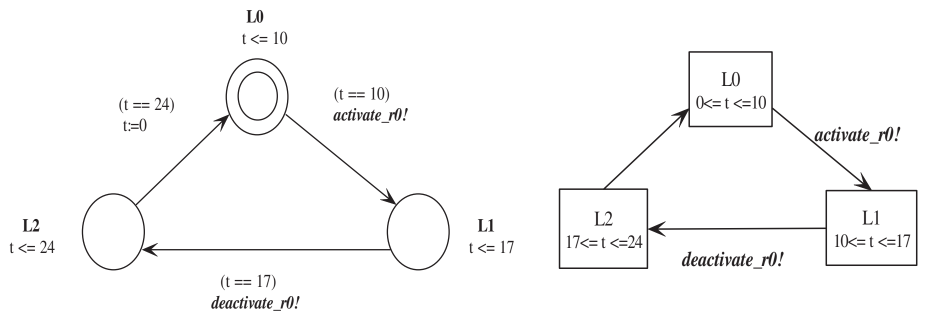

Fig. 1 – A timed automaton and its region automaton.

from the *Evaluate the policies* process. The verification result of the properties is used to detect for further conflicts in the given set of policies. Based on this new evaluation, the timed automata based model is reconstructed and the properties are generated again. Thus if any inconsistency is detected by the verification process then in the next iteration it is taken care of by the appropriate processes used in the framework.

## 3.2. *Mapping scheme*

For mapping GTRBAC to timed automata, the primary target is to identify the different states of each GTRBAC component and represent them by one or more control states or locations of an automaton. As mentioned before, in GTRBAC, a role can be in disabled, enabled or active state. For representing enabling and disabling behavior, two control states – labeled as *Disabled* and *Enabled* – are created in the automaton where *Disabled* is considered to be the initial state. Transitions from *Disabled* to *Enabled* and *Enabled* to *Disabled* are annotated using guards or some synchronization actions. Thus, with these two locations, the role enabling and disabling behavior can be represented as shown in Fig. 4.

Next, the active state of a role is considered. An enabled role becomes active when a user assumes that role. The user can acquire the permissions associated with the role only when it is in the active state. So, with the first user assignment, an enabled role goes to the active state. A user assignment is performed by an activation operation invoked by the user. Any subsequent activation operation does not change the state of the role. When all the users are unassigned from the role then the role goes back to the enabled state. From enabled or active state, a role can go back to the disabled state. To capture this behavior in the timed automaton, a control location labeled as *Active* is added. The transition from *Enabled* to *Active* location passes through some special locations which are labeled as $A_{p0}$, $A_{p1}$, $\cdots$, etc. Such special location is termed as committed location. This is defined as follows.

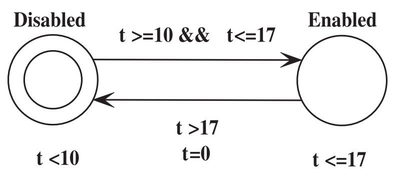

Fig. 2 – A deterministic timed automaton representing a safety property.

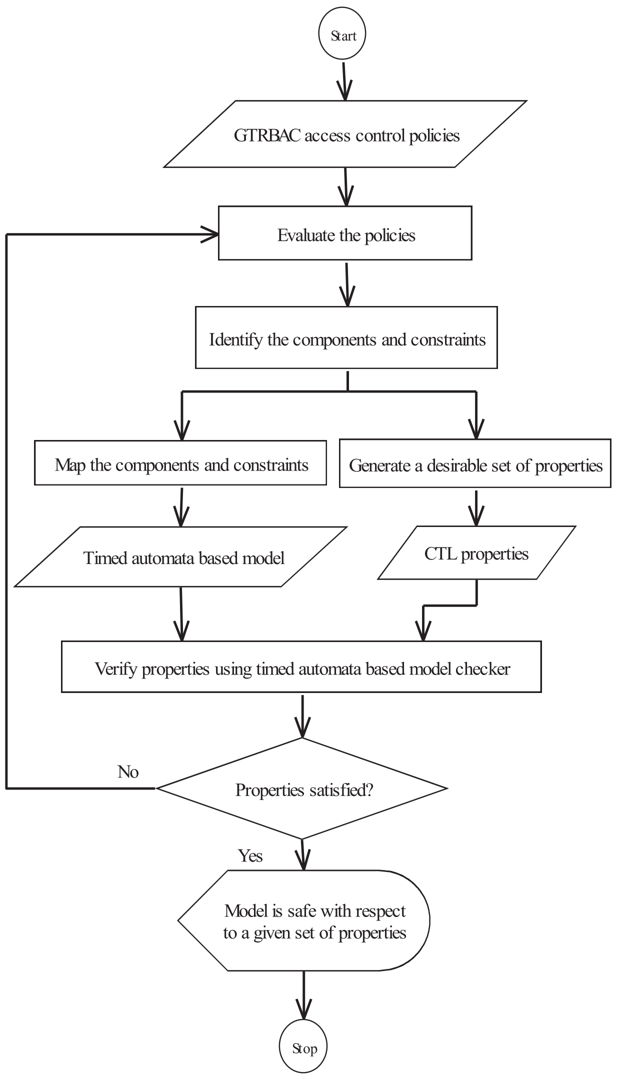
Start
GTRBAC access control policies
Evaluate the policies
Identify the components and constraints
Map the components and constraints
Generate a desirable set of properties
Timed automata based model
CTL properties
Verify properties using timed automata based model checker
No
Properties satisfied?
Yes
Model is safe with respect to a given set of properties
Stop

Fig. 3 – Proposed framework for performing security analysis.

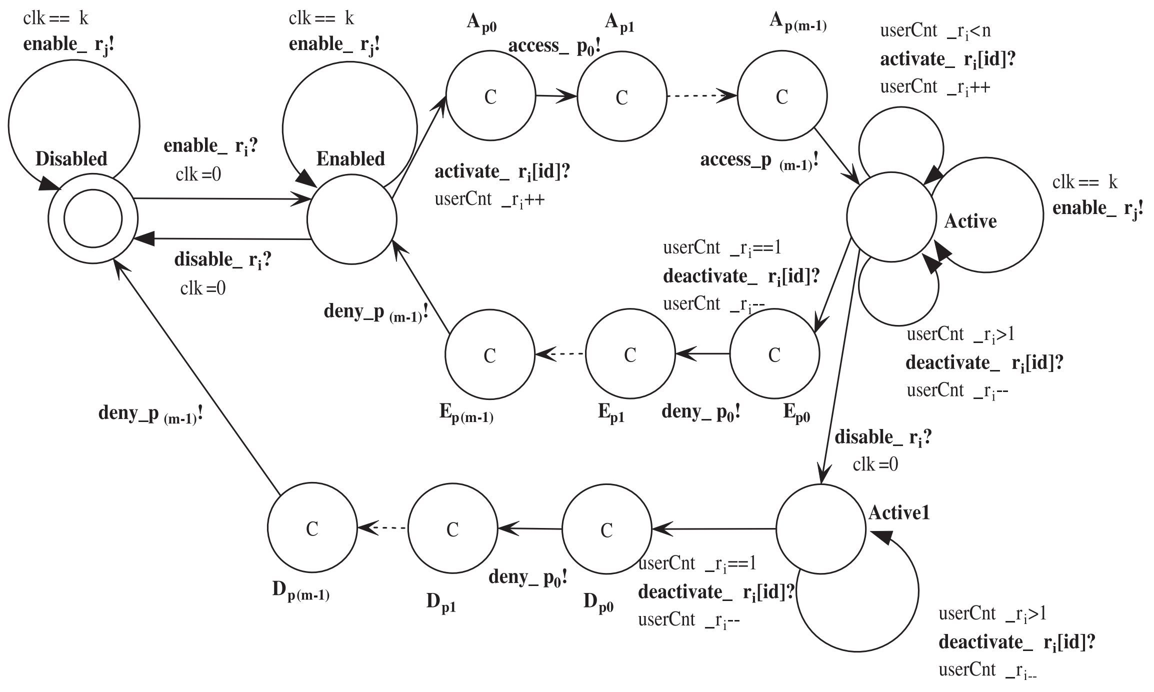

**Fig. 4 – GTRBAC role timed automaton.**

**Definition 4**. Committed Location: *A committed location in a timed automaton is a location which freezes time. Thus in a committed location, the values of the clock variables do not progress.*

The number of such locations depends on the number of permissions associated with the role for which the automaton is constructed. The transition from *Enabled* to $A_{p0}$ is labeled by a role activation action which is synchronized when a corresponding action is fired from a user automaton as discussed later. This action is represented as "activate_$r_i$[id]" where $r_i$ is the role under consideration and *id* is an integer variable representing the user who is trying to activate the role. To keep track of the number of users assigned to a role, a counter *userCnt_$r_i$* is used. The value of the counter is adjusted with each activation and deactivation operation. The outgoing transitions from each committed location is labeled with an action of the form "access_$p_i$" which is synchronized with the corresponding actions in the permission time automaton. This action signifies that an access request is made from a role but it does not guarantee that the permission would be available for access. A self-loop labeled with activation action is required at the *Active* location to reflect the fact that when a role is in active state, it remains in that state with further such activation operations.

Similar to role activation, two transitions – one from *Active* to *Active* and another from *Active* to *Enabled* via the committed locations $E_{p0}$, $E_{p1}$, ⋯, etc. – are used for role deactivation. Action labeled as "deactivate_$r_i$[id]" is used for this purpose. The outgoing transitions from these committed locations are labeled as "deny_$p_i$". The corresponding action in the permission timed automaton prevents the users to gain access to the permission. When a role is in the *Active* location, it can also reach the *Disabled* location via an intermediate location, named *Active1*. The path through this location avoids *Enabled* location. So the role does not go into the enabled state. But the role cannot directly go to the disabled state unless all the users are deactivated from the role. The self loop at *Active1* and the transition from *Active1* to *Disabled* through committed locations $D_{p0}$, $D_{p1}$, ⋯, etc., perform all the required deactivation operations. A complete role timed automaton is shown in Fig. 4.

The GTRBAC role timed automaton also needs to support triggering events. If a role $r_i$ triggers another role $r_j$ after *k* time units then it can be done by adding three self loops at locations *Disable*, *Enabled* and *Active* of the $r_i$ role timed automaton. In this case, a local clock variable, named as *clk*, is used to trigger the $r_j$ role to be enabled or disabled after *k* time units.

In GTRBAC, temporal constraints are also applicable on the permission assignment relation. When a role is active, the permissions associated with it are made available on satisfying the given temporal constraints. With the first role activation operation, the temporal condition associated with a permission is checked. Once the condition is satisfied, then the permission is made available to the user who has activated the role.

A permission timed automaton as shown in Fig. 5 is created using locations – *Inactive*, *loc1*, *loc2* and *Active*. Transition from *Inactive* to *loc1* is labeled with the action "access_$p_i$". The transition from location *loc1* to *Active* is annotated with a temporal condition. Upon satisfying the temporal condition, the automaton reaches the *Active* location, which indicates that the permission is now available for access. Another temporal condition may lead the automaton from *Active* to *loc2* representing the fact that permission becomes unavailable to the users. When the automaton is at location *Active* or *loc1* or *loc2* then a transition can be made to the *Inactive* location if there is a corresponding synchronization action labeled "deny_$p_i$" fired from the role timed automaton.

It can be observed that the basic structure of role timed automaton is same for all roles except for the action names.

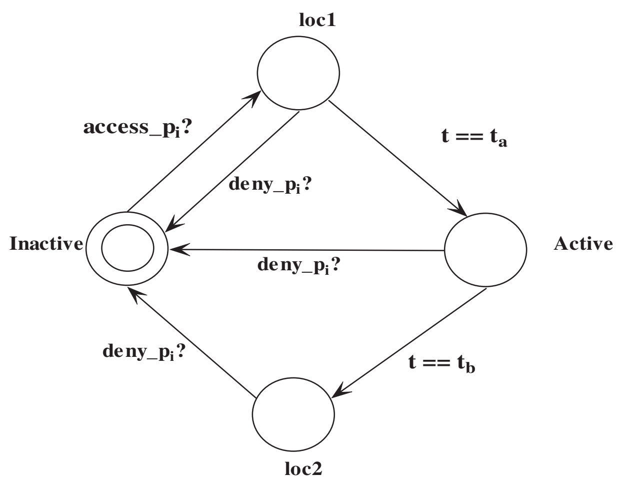

**Fig. 5 – GTRBAC permission timed automaton.**

This is also true for permission timed automata. But to represent various types of user behaviors, the structure of the user timed automaton would not remain the same. For example, one user may be allowed to activate only a single role, whereas another user may be allowed to activate multiple roles at the same time, or multiple roles but not at the same time. So, depending upon such behavioral characteristics, four different types of user interaction are identified. They are termed as Type-1, Type-2, Type-3 and Type-4 interactions.

- *Type-1 interaction*: A user participating in this type of interaction can activate only a single role at a time. To represent it using a timed automaton, two locations – *Idle* and *Active* are needed. The transition from *Idle* to *Active* is labeled with a temporal constraint and an action for activating the role. Similarly, another transition from *Active* to *Idle* is needed. In Fig. 6, a user timed automaton for Type-1 interaction is shown. Here the user sends an activation action to $r_i$ when $t = t1$, and at $t = t2$, the user sends a deactivation action. The user can remain in the *Active* location till $t \leq t2$, so an invariant $t \leq t2$ is used for the *Active* location.
- *Type-2 interaction*: In this type, a user can activate multiple roles at different time instants. With one activation request, the user goes to the *Active* state and it remains in that state if it wants to activate any other role. When the user has sent all the deactivation requests, it goes back to the *Idle* state. The basic structure of user automaton for Type-2 interaction is same as that of Type-1, but in this case, the *Active* location has an extra pair of self loops for each additional role activation and deactivation operation, which are labeled with appropriate temporal constraints. This is illustrated in Fig. 7.
- *Type-3 interaction*: In this type of interaction, a user can activate multiple roles at a particular time instant. This can be done using a set of committed locations. So to activate $n$ roles at a single time instant, committed locations $A_0, A_1, \ldots, A_{n-2}$ are used in the automaton as shown in Fig. 8. Using the first activation operation, the automaton goes to the $A_0$ location. Due to the property of committed location, time does not proceed further as long as the transitions pass through the committed locations. The outgoing edge of each committed location is labeled with the corresponding activation actions. Thus, when the automaton reaches *Active* location, a total of $n$ activation requests have been made by the user without change of time. Similarly, the user can make multiple deactivation requests at a particular time instant using a separate set of committed locations denoted as $I_0, I_1, \ldots, I_{n-2}$.
- *Type-4 interaction*: This type of interaction restricts a user to activate multiple roles at the same time. A user of this type can activate multiple roles but not at the same time. To implement such type of interaction, parallel transitions are needed between locations. Both the transitions are annotated with a temporal constraint in addition to an activation or a deactivation action. At a given time, either of the two transitions is selected to move from one location to the other. So if there are two conflicting roles, then it requires two parallel transitions from *Idle* to *Active* for activation operations and two more parallel transitions from *Active* to *Idle* for deactivation operation. In Fig. 9, a user timed automaton having Type-4 interaction is shown with two conflicting roles $r_i$ and $r_j$ which are activated at $t1$ and $t3$ and deactivated at $t2$ and $t4$ respectively. So the user can stay at *Active* location at most max($t2$, $t4$) time units which is used as an invariant to *Active* location.

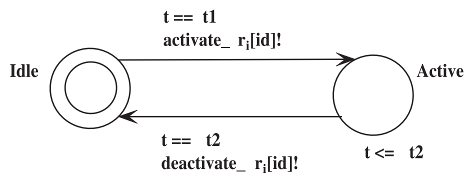

**Fig. 6 – GTRBAC user timed automaton for Type-1 interaction.**

To model a complete GTRBAC system, several timed automata are required. In a timed automaton, temporal constraints are specified using clock variables which are to be used carefully since they directly affect the verification process. More the number of clock variables, higher is the complexity of the model checking process (Baier and Katoen, 2008; Bhatti et al., 2005a,b). To express the temporal constraints, a global clock variable can be used by all the timed automata. Although these automata can use the global clock, they are not allowed to update the clock value. So a separate automaton is created to keep control over the global clock variable. It is responsible for enabling and disabling of roles and updating of the clock variable. This automaton is termed as the *controller automaton*.

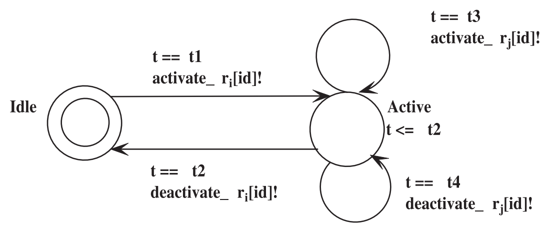

**Fig. 7 – GTRBAC user timed automaton for Type-2 interaction.**

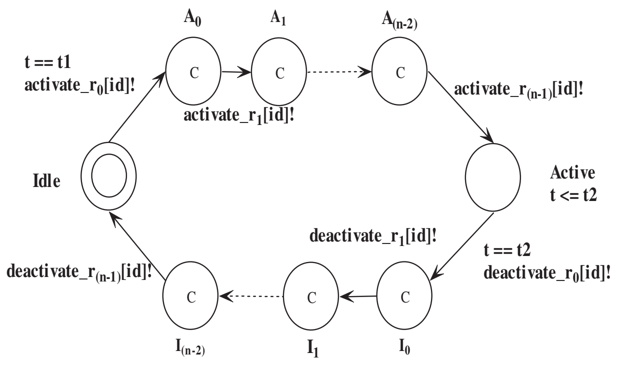

**Fig. 8 – GTRBAC user timed automaton for Type-3 interaction.**

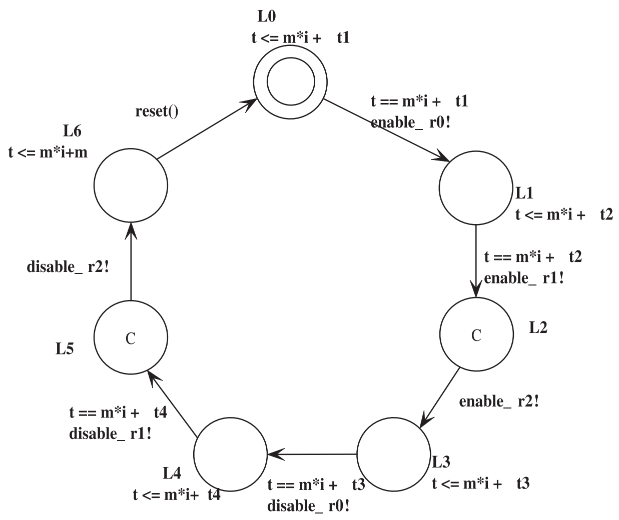

**Fig. 10 – GTRBAC controller timed automaton.**

The role enabling/disabling temporal constraints are specified in the controller automaton, as shown in Fig. 10. A single clock variable is used to specify various temporal constraints. The transition of the automaton corresponds to a temporal condition either for enabling a role or disabling a role. So, when such a temporal condition is satisfied in the controller automaton, it allows the transition to fire an action. The action is synchronized in the corresponding role timed automaton. With this action the role timed automaton moves to either *Enabled* or *Disabled* location indicating that the role is either enabled or disabled. If two roles are to be enabled or disabled at the same time instant, then committed location can be used for this purpose. In Fig. 10, roles *r1* and *r2* enable at the same time instant and also both are disabled at another time instant. Once role *r1* is enabled, it reaches location *loc2* which is a committed location. So time does not progress in *loc2*. The outgoing transition from *loc2* takes place immediately without changing time. Thus, role *r2* is enabled at the same time as *r1*. Similarly, committed location *loc5* allows role *r2* to disable at the same time as *r1*. The clock variable is reset in the final transition of the automaton.

In GTRBAC, time is represented by a periodic expression (Joshi et al., 2005a). A periodic time instant can be expressed as a tuple ⟨[begin, end], *P*⟩ where *P* is a periodic expression denoting an infinite set of time instants and [begin, end] is a time interval denoted by a lower and an upper bound that are imposed on instants in *P*. Next we describe how periodicity is incorporated in a controller automaton.

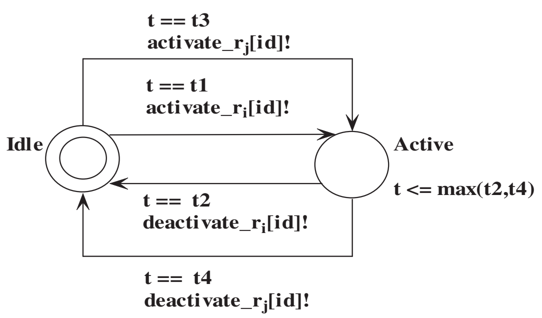

**Fig. 9 – GTRBAC user timed automaton for Type-4 interaction.**

The periodicity part of a time instant can be represented in the timed automaton as $m \times i + t_c$, where *m* is a constant and *i* is an integer variable. If *hour* is taken as the lowest time unit, then to represent each day's behavior for a week, *m* can be assumed to be 24 and the value of *i* may range from 0, 1, ..., 6. Durability constraint can also be applied here. Suppose a role remains in the *enabled* state for *n* hours with one transition, say $t == 24 \times i + t1$, the role gets enabled and with another transition, say $t == 24 \times i + t2$, the role is disabled. Here *t2* should be defined in such a way that $t2 = t1 + n$. Thus, duration is the difference between the time instants of outgoing and incoming transitions of a location. Also, if *day* is considered to be the lowest time unit, then *m* becomes 7 to represent weekly behavior of the role and *i* can range from 0, 1, ..., 3 to represent the four weeks. In the controller automaton, the last transition resets the clock. In Fig. 10, a controller timed automaton incorporating the periodic time instants is shown.

## 4. Algorithms for construction of timed automata

The algorithms required to map a GTRBAC system to the corresponding set of timed automata are discussed in this section. The different GTRBAC components are used as input to the algorithms. The inputs are shown by the required field to the algorithms. Algorithm 1 describes the steps required for the construction of role timed automata. For each role an automaton is constructed. Lines 2–10 compute the number of users and permissions associated with the role under consideration. The variable *n* stores the number of users and *m* stores the number of permissions. Next the actual automaton is constructed. The different tuples of a timed automaton are defined in lines 12–25. The trigger function is used to check whether any dependent role exists or not. This is checked in line 17. If the trigger set is not empty then a local clock is defined. The clock is used to enable and disable the dependent role. Lines 18–22 build the automaton to include the trigger constraint. If the trigger set is empty then the automaton is constructed using lines 24 and 25.

**Algorithm 1**. Construction of GTRBAC Role Timed Automaton

```
Require R, U, P, UA, PA
1: for each $r \in R$ do
2:    $i = 0; j = 0;$
3:    for each ($<u, r'> \in UA$) AND ($r' == r$) do
4:       $i$++;
5:    end for
6:    $n = i$; {/*$n$ users can activate role $r$*/}
7:    for each ($<r', p> \in PA$) AND ($r' == r$)
8:       $j$++;
9:    end for
10:   $m = j$;{/*$r$ is associated with $m$ no. of permissions */}
11:   construct a timed automaton $TA_r = <L, l_0, C, A, E, I>$ such that
12:   $L$ = {Disabled, Enabled, Active, $A_{p0}, \ldots, A_{p(m-1)}, E_{p0}, \ldots, E_{p(m-1)}, D_{p0}, \ldots, D_{p(m-1)}$};
13:   commit({$A_{p0}, \ldots, A_{p(m-1)}, E_{p0}, \ldots, E_{p(m-1)}, D_{p0}, \ldots, D_{p(m-1)}$}); {/* commit() function marks a set of locations as "Committed" */}
14:   $l_0$ = Disabled;$I = \emptyset$;
15:   $A$ = {enable_r, disable_r, activate_r[n], deactivate_r[n],
         access_$p_0$, …, access_$p_{m-1}$, deny_$p_0$, …, deny_$p_{m-1}$};
16:   $E$ = {(Enabled, activate_r[id]?, Ø, Ø, $A_{p0}$), ($A_{p0}$, access_p0!, Ø, Ø, $A_{p1}$), …, ($A_{p(m-1)}$, access_p(m – 1)!,Ø, Ø, Active), (Active, activate_r[id]?,userCnt_r < n,Ø, Active), (Active, deactivate_r[id]?, userCnt_r > 1, Ø, Active), (Active, deactivate_r?>?,user Cnt_r == 1, Ø, $E_{p0}$), ($E_{p0}$, deny_p0!, Ø, Ø, $E_{p1}$), …, ($E_{p(m-1)}$, deny_p (m – 1)!,Ø, Ø, Enabled ), (Active1, deactivate_r[id]?, userCnt_r > 1, Ø, Active1), (Active1, deactivate_r[id]?, userCnt_r == 1, Ø, $D_{p0}$),($D_{p0}$, deny_p0!, Ø, Ø, $D_{p1}$), …, ($D_{p(m-1)}$, deny_p(m – 1)!,Ø, Ø, Disabled )};
17:   If trigger(r) ≠ Ø then
18:      $r' \leftarrow$ roleTrigger(r); {$r'$ is the role triggered by role $r$}
19:      $k \leftarrow$ timeTrigger(r,r'); {$k$ is the triggering time}
20:      $C$ = clk;
21:      $A = A \cup$ {enable$_{r'}$, disable$_{r'}$};
22:      $E = E \cup$ {(Disabled, enable_r?, Ø,clk,Enabled), (Enabled, disable_r?, Ø,clk, Disabled), (Enabled, enable$_{r'}$!, clk == k, Ø, Enabled ), (Disabled, disable$_{r'}$!, clk == k, Ø, Disabled ), (Active, enable$_{r'}$!, clk == k, Ø, Active), (Active, disable_r?,Ø,clk == 0, Active1)};
23:   else
24:      $C = \emptyset$;
25:      $E = E \cup$ {(Disabled, enable_r?, Ø, Ø, Enabled ),
         (Enabled, disable_r?,Ø, Ø, Disabled ),
         (Active, disable_r?,Ø, Ø, Active1)};
26:   end if
27:   end for
```

For each role, the algorithm will run only once. In each run, a timed automaton is constructed. Each timed automaton can be thought of as a directed graph. The time complexity of the construction of a timed automaton is $O(|L| + |E|)$ where $|L|$ denotes the number of locations or vertices and $|E|$ represents the number of edges in the automaton. In a GTRBAC role timed automaton, $|L|$ depends on the number of permissions associated with a particular role. Let $|P|$ denote the number of permissions. So $|L|$ becomes $4 + 3 \times |P|$ as four fixed locations are *Disabled*, *Enabled*, *Active* and *Active1* and the rest of the locations depends on the number of permissions. Again for each role, $|E|$ may vary from $9 + 3 \times |P|$ when there is no trigger to $12 + 3 \times |P|$ when the trigger modeled. Thus the time complexity of Algorithm 1 can be computed as $O(|R| \times (|L| + |E|))$ which can be reduced to $O(|R| \times |P|)$. So it depends both on the number of roles and the number of permissions.

Algorithm 2 is used for construction of the controller automaton. It requires the set of role R as input. To express the temporal constraints, a single global clock variable, named as t, is used in this automaton. It is responsible for enabling and disabling of roles and updating of the clock variable. The following two functions are used by the algorithm.

- *enable(r,k)* is *true* if role *r* is enabled at time instant *k*; otherwise, it is *false*.
- *disable(r,k)* is *true* if role *r* is disabled at time instant *k*; otherwise, it is *false*.

**Algorithm 2**. Construction of GTRBAC Controller Timed Automaton

```
Require R
1: $i = 0$; {/*i is used to keep track of no. of constraints in the system*/}
2: maxTime = 24; {/*To check the hourly behavior of a day*/}
3: for each ($r \in R$) do
4:    if enable(r, k) then
5:       list[i++] = $\langle r, k, 0\rangle$; {/* 0 is used for role enabling action */}
6:    end if
7:    if disable(r, k) then
8:       list[i++] = $\langle r, k, 1\rangle$; {/* 1 is used for role disabling action */}
9:    end if
10: end for
11: $n = i$;
12: sortAsc(List,time); {sort the list elements in ascending order of time}
13: construct a timed automaton $TA_c = \langle L, l_0, C, A, E, I\rangle$ such that
14: $L$ = {$loc_0$};
15: $l_0 = loc_0$;
16: $A = \emptyset$;
17: $E = \emptyset$;
18: $C$ = {t};
19: prev_time = 0;
20: for $i = 0$ to $(n - 1)$ do
21:    $L = L \cup \{loc_{i+1}\}$;
22:    curr_time = list[i].time;
23:    if (curr_time = prev_time) then
24:       commit({$L_i$}); {/* commit() marks a set of locations as "Committed"*/}
25:    else
26:       B(C ) ← curr_time; {constructing the temporal constraint}
27:       I ← inv($loc_i$,B(C )); {assigning invariant at $loc_i$}
28:    end if
29:    action ← list[i].type; {it determines whether the action is enabling or disabling}
30:    $A = A \cup$ {action};
```

```
31:     $t_i \leftarrow list[i].time$; {/*constructs the temporal expression*/}
32:     $E = E \cup \{(loc_i, action!, t == t_i, \emptyset, loc_{i+1})\}$;
33:     $prev\_time = curr\_time$;
34: end for
35: $B(C) \leftarrow maxTime$;
36: $I \leftarrow inv(loc_i, B(C))$; {assigning invariant at $loc_i$}
37: $E = E \cup \{(loc_{i+1}, \emptyset, t == maxTime, t, loc_0)\}$;
```

The algorithm uses an array named *list[]*. Each element of *list []* is a structure data type, having three elements – *Role* which specifies a role id, *Time* indicates the time instant when the role is enabled or disabled and *Type* which is used to identify whether the time is used for role enabling or disabling. It can either have value 0 for role enabling event and 1 for role disabling event. As an example, if two roles *r1* and *r2* are enabled at 10 and 11 respectively and disabled at 17 and 20 respectively, then *list*[] = {(*r*1, 10, 0), (*r*1, 17, 1), (*r*2, 11, 0), (*r*2, 20, 1)}.

In the algorithm, lines 3–10 are used to add the elements in the *list[]* array, which is sorted in ascending order of time. This is done by the sorting function *sortAsc()* at line 12. After sorting, the items of the array become *list*[] = {(*r*1, 10, 0), (*r*2, 11, 0), (*r*1, 17, 1), (*r*2, 20, 1)}.

Initially, the automaton has a single location *loc0*. Then for each temporal condition, a location is added to the automaton. The algorithm uses two variables *prev_time* and *curr_time* to check whether multiple events occur at the same time. Using the committed location two transitions can fire two actions at the same time. This is done using lines 23–24. Lines 25–28 construct the temporal constraint which is assigned to the previously created location. Lines 29–32 define the action, guard and the transition.

For each role there may be one role enabling and one role disabling constraint. The *for* loop in line 20 executes *n* times and *n* can be at most $2 \times |R|$. Each execution adds a location and an edge. In the controller automaton the number of locations is $|L| = n + 1 = 2 \times |R| + 1$ and the number of edges is $|E| = n + 1 = 2 \times |R| + 1$. Now the time complexity of the construction of the automaton is $O(|L| + |E|)$ which can be derived as $O(|R|)$.

**Algorithm 3**. Construction of GTRBAC Permission Timed Automaton

```
Require P, list_perm[]
1: for each $p \in P$ do
2:     $t_a = getTime(list\_perm, p, 0)$; {/*Get the permission availability time*/}
3:     $t_b = getTime(list\_perm, p, 1)$; {/*Get the permission unavailability time*/}
4:     construct a timed automaton $TA_p = \langle L, l_0, C, A, E, I\rangle$ such that
5:     $L = \{Inactive, Active, loc1, loc2\}$;
6:     $l_0 = Inactive$; $C = t$;
7:     $A = \{access\_p, deny\_p\}$;
8:     $E = \{(Inactive, access\_p?, \emptyset, \emptyset, loc1), (loc1, deny\_p?, \emptyset, \emptyset, Inactive), (loc1, \emptyset, t == t_a, \emptyset, Active), (Active, \emptyset, t == t_b, \emptyset, loc2), (loc2, deny\_p?, \emptyset, \emptyset, Inactive), (Active, deny\_p?, \emptyset, \emptyset, Inactive)\}$;
9:     $B(C) = t_b$; $I \leftarrow inv(Active, B(C))$; {/* assigning invariant to Active*/}
10: end for
```

Here, the time complexity of Algorithm 2 is also $O(|R|)$ which indicates that the complexity linearly depends on the total number of roles in the system.

Algorithm 3 is used to construct the permission timed automaton. The algorithm requires an array named *list_perm*[] and set of permission *P* as inputs. Each element of it is a structure data type, having three elements – *Permission* which specifies a permission id, *Time* indicates the time instant when the permission would be available or unavailable for access and *Type* which is used to represent whether the permission is available or not. The value 0 indicates the time instant from when the permission would be available and 1 signifies the time instant from when the permission would be unavailable. For an instance, if a permission *p* is available from 12:00 and unavailable from 15:00 and again it is available at 18:00 and unavailable at 20:00 then *list_perm []* = {( *p*, 12, 0), ( *p*, 15, 1), ( *p*, 18, 0), ( *p*, 20, 1)}

For each permission there is a corresponding timed automaton. The different tuples are defined in lines 5–9. The clock set *C* uses the global clock variable *t*. This is used to specify the temporal constraint which determines whether the activated users can access the permission or not. In the permission timed automaton $|L|$ is 4 and $|E|$ is 6. These values are fixed for any GTRBAC permission automaton. Thus the time complexity of Algorithm 3 can be computed as $O(|P| \times (|L| + |E|))$ which can be simplified to $= O(|P|)$. Therefore, the complexity linearly depends on the number of permissions only.

**Algorithm 4**. Construction of GTRBAC User Timed Automaton of Type-1 Interaction

```
Require $U1 \in U$, list_user[] {U1 represents the set of users who can activate a single role}
1: for each $u \in U1$ do
2:     $u\_id \leftarrow getID(u)$;
3:     construct a timed automaton $TA_u = \langle L, l_0, C, A, E, I\rangle$ such that
4:     $L = \{Idle, Active\}$;
5:     $l_0 = Idle$; $C = t$;
6:     $A = \{activate\_r[u\_id], deactivate\_r[u\_id]\}$; where $\langle u, r\rangle \in UA$
7:     $t1 = getTime(list\_user, u, r, 0)$; {/*Get the user activation time*/}
8:     $t2 = getTime(list\_user, u, r, 1)$; {/*Get the user deactivation time*/}
9:     $E = \{(Idle, activate\_r[u\_id]!, t == t1, \emptyset, Active), (Active, deactivate\_r[u\_id]!, t == t2, \emptyset, Idle)\}$;
10:    $B(C) = t2$; $I \leftarrow inv(Active, B(C))$; {/* assigning invariant to Active*/}
11: end for
```

Algorithms 4–7 are used to construct the different types of user automata. The algorithms require a specific set of users and an array termed *list_user[]* as inputs. Each element of the array is a structure data type, having four elements – *User* is used to specify a user id, *Role* stands for a role id which a user wants to activate or deactivate, *Time* indicates the time instant when the user wants to activate or deactivate the role and *Type* is a boolean data type. The value 0 indicates that the user wants to activate the role and 1 signifies that the user wants to deactivate the role. For an instance, if a user *u* wants to activate a role $r$ at 10:00 and deactivate the role at 15:00 and another role $r'$ at 16:00 and 19:00 respectively then *list_user[]* = {($u$, $r$, 10, 0), ($u$, $r$, 15, 1), ($u$, $r'$, 16, 0), ($u$, $r'$, 19, 1)}

Each automaton captures some specific characteristics of the user. Algorithm 4 constructs Type-1 interaction to represent the users who can activate only a single role in a particular session. Along with *list_user[]* the algorithm requires a set of users of Type-1 interaction $U1$ as inputs. Each user is uniquely identified by a *u_id* value. This is obtained by a function named as *getID(user)*. The tuples of the automaton are defined through lines 4–10. The global clock variable is used to specify the temporal constraints when the user wants to activate and deactivate a role. If $|U1|$ indicates the total number of users of Type-1 interaction then the time complexity of the algorithm becomes $O(|U1| \times (|L| + |E|))$ which can be reduced to $O(|U1|)$. Now the value of $|U1|$ can be at most $|U|$. So the time complexity of Algorithm 4 is $O(|U|)$.

**Algorithm 5.** Construction of GTRBAC User Timed Automaton of Type-2 Interaction

```
Require $U2 \subseteq U$, list_user[] {U2 represents the set of users who can activate multiple roles at different times}
1: for each $u \in U2$ do
2:   $u\_id \leftarrow getID(u)$;
3:   construct a timed automaton $TA_u = \langle L, l_0, C, A, E, I\rangle$ such that
4:   $L = \{Idle, Active\}$; $l_0 = Idle$; $A = \emptyset$;
5:   $loc1 = Idle$;
6:   $loc2 = Active$;
7:   for each $(r \in R) \wedge (\langle u, r\rangle \in UA)$ do
8:     $A = A \cup \{activate\_r[u\_id], deactivate\_r[u\_id]\}$;
9:     $t1 = getTime(list\_user, u, r, 0)$; {/*Get the user activation time*/}
10:    $t2 = getTime(list\_user, u, r, 1)$; {/*Get the user deactivation time*/}
11:    $E = \{(loc1, activate\_r[u\_id]!, t == t1, \emptyset, loc2), (loc2, deactivate\_r[u\_id]!, t == t2, \emptyset, loc1)\}$;
12:    $loc1 = loc2 = Active$;
13:  end for
14:  $B(C) = t2$; $I \leftarrow inv(loc2, B(C))$; {/* assigning invariant to Active*/}
15: end for
```

Algorithm 5 constructs Type-2 interaction to represent the users who can activate multiple roles at different times. The algorithm requires *list_user[]* a set of users of Type-2 interaction or $U2$ as inputs. The tuples of the automaton are defined through lines 4–14. The user of this type can activate the set of roles computed using the *for* loop in lines 7–13. Here we assume that the role which is activated first will be deactivated last. The other roles can be activated and deactivated in between these two time instants. For Type-2 interaction, the construction of the automaton is dependent on the number of roles involved with this type of users. If a user activates multiple number of roles then for each such role two transitions along with temporal constraints are to be added in the automaton. So the number of transitions or edges in the automaton would be dependent on the number of roles a user activates. However, the number of locations in the automaton would not change with the number of roles. Now the time complexity of the algorithm can be computed as $O(|U2| \times (|L| + |E|))$ and which can be simplified to $O(|U2| \times |R|)$. Now the value of $|U2|$ can be at most $|U|$. Thus the complexity of Algorithm 5 becomes $O(|U| \times |R|)$. So unlike the Type-1 interaction, this algorithm depends on both the number of users and number of roles.

**Algorithm 6.** Construction of GTRBAC User Timed Automaton of Type-3 Interaction

```
Require $U3 \in U$, list_user[] {/*U3 represents the set of users who can activate multiple roles at same time*/} do
1: for each $(u \in U3)$ do
2:   $i = 0$;
3:   $u\_id \leftarrow getID(u)$;
4:   construct a timed automaton $TA_u = \langle L, l_0, C, A, E, I\rangle$ such that
5:   $A = \emptyset$; $E = \emptyset$;
6:   $L = \{Idle, Active\}$; $l_0 = Idle$;
7:   $loc1 = Idle$; $loc2 = Active$;
8:   for each $((r \in R) \wedge (<u, r> \in UA))$ do
9:     $A = A \cup \{activate\_r[u\_id], deactivate\_r[u\_id]\}$;
10:    $t1 = getTime(list\_user, u, r, 0)$; {/*Get the user activation time*/}
11:    $t2 = getTime(list\_user, u, r, 1)$; {/*Get the user deactivation time*/}
12:    for each $((r' \in R) \cup (r > r'))$ do
13:      $A = A \cup \{activate'_r[u\_id], deactivate_r'[u\_id]\}$;
14:      $L = L \cup \{A_i, I_i\}$;
15:      $commit(\{A_i, I_i\})$; {commit() function marks a set of locations as "Committed"}
16:      $loc3 = A_i$; $loc4 = I_i$;
17:      if $(isCommit(loc1) \vee isCommit(loc2))$
18:        $E = E \cup \{(loc1, activate\_r[u\_id]!, \emptyset, \emptyset, loc3), (loc2, deactivate\_r[u\_id]!, \emptyset, \emptyset, loc4)\}$;
19:      else
20:        $E = E \cup \{(loc1, activate\_r[u\_id]!, t == t1, \emptyset, loc3), (loc2, deactivate\_r[u\_id]!, t == t2, \emptyset, loc4)\}$;
21:      end if
22:      $loc1 = loc3$; $loc2 = loc4$;
23:      $i{+}{+}$;
24:    end for
25:    $loc3 = Active$; $loc4 = Idle$;
26:    $E = E \cup \{(loc1, activate\_r[u\_id]!, \emptyset, \emptyset, loc3), (loc2, deactivate\_r[u\_id]!, \emptyset, \emptyset, loc4)\}$;
27:  end for
28: end for
```

Algorithm 6 constructs Type-3 interaction to represent the users who can activate multiple roles at the same time. The algorithm requires an array termed *list_user[]* and a set of users of Type-3 interaction $U3$ as inputs. The functionality of the algorithm is similar to activating a senior role which in turn activates the junior roles. So we can use this interaction to represent a role hierarchy relation. Initially, a particular role is considered which can be activated by the user. Then it is required to check whether any junior role exists for that role. If such roles exist, then those roles are to be activated at the same time. The role hierarchy relation is considered in the *for* loop of line 12. Here $r \succ r'$ indicates that role $r$ has a junior role $r'$. For each junior role two locations and two edges are added to the location and edge sets of the automaton. The locations are marked as committed type so that all the junior roles are activated or deactivated at the same time as the senior role. The automaton construction is influenced by two *for* loops at lines 8 and 12. The number of iterations of both the loops is dependent on $|R|$. So the time complexity of Algorithm 6 can be derived as follows $O(|U3| \times (|R| \times (|L| + |E|)))$. This can be shown to be equivalent to $O(|U3| \times |R|^2)$. Again $|U3|$ can be at most $|U|$. So in this case, the complexity becomes $O(|U| \times |R|^2)$ which is more sensitive to number of roles. This is caused due to the role hierarchy relation.

**Algorithm 7**. Construction of GTRBAC User Timed Automaton of Type-4 Interaction

```
Require U4 ⊆ U, list_user[] {U4 represents the set of users who can activate one of the conflicting role at a time}
1: for each $u \in U4$
2:   $u\_id \leftarrow getID(u)$;
3:   construct a timed automaton $TA_u = \langle L, l_0, C, A, E, I \rangle$ such that
4:   $i = 0; C = t$;
5:   $L = \{Idle, Active\}; l_0 = Idle; A = \emptyset; E = \emptyset$
6:   for each $((r \in R) \wedge (\langle u, r \rangle \in UA)$ do
7:     $A = A \cup \{activate\_r[u\_id], deactivate\_r[u\_id]\}$;
8:     $maxTime = 0$;
9:     $t1 = getTime(list\_user, u, r, 0)$; {/*Get the user activation time*/}
10:    $t2 = getTime(list\_user, u, r, 1)$; {/*Get the user deactivation time*/}
11:    $E = E \cup \{(Idle, activate\_r[id]!, t = = t1, \emptyset, Active), (Active, deactivate\_r[id]!, t = = t2, \emptyset, Idle)\}$;
12:    $maxTime = t2$;
13:    if $(r' \in R) \wedge (u, r' \in UA) \wedge (r' \in conf(r))$ then
14:      $A = A \cup \{activate_{r'}[u\_id], deactivate_{r'}[u\_id]\}$;
15:      $t3 = getTime(list\_user, u, r', 0)$; {/*Get the user activation time*/}
16:      $t4 = getTime(list\_user, u, r', 1)$; {/*Get the user deactivation time*/}
17:      $E = E \cup \{(Idle, activate_{r'}[u\_id]!, t = = t3, \emptyset, Active), (Active, deactivate_{r'}[u\_id]!, t = = t4, \emptyset, Idle)\}$;
18:      $maxTime = max(maxTime, t4)$;
19:    end if
20:  end for
21:  $B(C) = maxTime; I \leftarrow inv(Active, B(C))$; {/* assigning invariant to Active*/}
22: end for
```

Algorithm 7 constructs automaton corresponding to Type-4 interaction. The algorithm requires as inputs an array termed *list_user[]* and *U4* representing the set of users who can activate only one of the conflicting roles at a time. Thus it can be used to map the SoD constraint. The algorithm uses a function called *conf(r)* which is defined as follows. $conf(r) = \{r' | (SoD(\{r, r'\}, 2))\}$ where $SoD(r, r', 2)$ indicates that 2 roles from the set $\{r, r'\}$ cannot be activated in the same session.

To activate a role $r$, the transition from *Idle* to *Active* and to deactivate the role the transition from *Active* to *Idle* is added at line 11. At line 13, the conflict condition is checked. If there is a conflicting role, say $r'$, then one transition from *Idle* to *Active* and another from *Active* to *Idle* is added using line 17. A variable *maxTime* is used to denote the maximum time the user can remain in active state. Using *maxTime* the invariant is assigned to *Active* location. It is observed that for defining the tuples of the automaton, the number of locations or $|L|$ is not dependent on $|R|$ but the number of edges is dependent on the *for* loop at line 7. So the time complexity of the algorithm is $O(|U4| \times (|L| + (|R| \times |E|)))$ and this is equivalent to $O(|U4| \times |R|)$. Now if the value of $|U4|$ becomes $|U|$ then the complexity is $O(|U| \times |R|)$. Thus, like Type-2 interaction, the complexity depends both on the number of users and the number of roles. So, the SoD constraint does not significantly affect the mapping complexity.

## 5. Completeness of the mapping process

To show the completeness of the proposed mapping scheme, we need to verify that the mapping scheme correctly models the given system. Also it is important to know whether the resultant system retains all the properties of the GTRBAC model. For that each of the component and constraints of GTRBAC is considered separately and is checked whether it is properly addressed or not. In Section 4 each algorithm is discussed in detail to show that how it is going to represent the desired features. In this section we briefly take few of such desired features.

- *Role*: Each role of the given is represented by a timed automaton. It can have three possible states as explained earlier in Section 3. The automaton is designed in such a way that it retains the properties of actual role. For instance the role is initially in Disabled state and it can not directly go to Active state. But from Active state it can directly go to Disabled state. So such properties are maintained in the role timed automaton.
- *Permission*: The timed automaton corresponding to permission component has been designed in such a way that it can be accessible depending upon satisfying some temporal constraint. But this constraint will only be evaluated if an access request is triggered from an associated role. Thus the permission maps a valid permission component.
- *User*: In our proposed scheme the user component is defined using four different interactions represented by four different automata. These are used to capture the constraints like SoD and Role Hierarchy. In GTRBAC framework, a user has to be associated with a role. Now if that role has some role hierarchy or SoD relation then the users associated with it are defined so that it would reflect SoD or Role Hierarchy relation. So although SoD and Role Hierarchy are mainly the properties of a role but these are represented in the user interaction automata to simplify the modeling process.
- *Temporal constraints on role enabling/disabling*: A role can be enabled or disabled depending upon satisfying some temporal constraints. In our proposed mapping scheme, the controller timed automaton is used for triggering the

enabling and disabling actions. In Section 3 it has been explained how the controller automaton works.

- *Role activation/deactivation constraints*: While mapping a role, we have used a variable *userCnt_r* to keep track of the number of users who have activated a role. GTRBAC allows each role to be activated by a certain number of users. In a role timed automaton, the activation transition from location *Active* to location *Active* is annotated with a condition that allows only a limited number of users to activate the role.
- *Temporal constraints on user-role assignment*: Temporal constraints on user-role assignment relation determine when a user can activate or deactivate a role. These constraints are specified in the user timed automata. The transition from *Idle* to *Active* in a user automaton is labeled with a temporal constraint. An activation action is fired during the transition. The action is synchronized in the corresponding role timed automaton. With this action the role automaton moves to location *Active* indicating that the role is in the active state. Similarly, on satisfying another temporal constraint in a user automaton, the user can deactivate the role.
- *Temporal constraints on permission-role assignment*: GTRBAC allows temporal constraints on the permission assignment relation. This is incorporated using the permission timed automata. When a user activates a role, an action is fired from the role automaton. This is synchronized in the corresponding permission automaton. With this action the permission automaton goes to *loc1* location. From this location the automaton goes to the *Active* location on satisfying some temporal constraint. When the automaton is at *Active* location, only then the permission is available for access. On satisfying another temporal constraint, the automaton moves to location *loc2* indicating that the permission is no more available to the users.
- *Role hierarchy*: Role hierarchical relation is implemented using user timed automaton with Type-3 interaction. Role hierarchy means the users of an active role (senior role) can activate another role (junior role). In Type-3 interaction shown in Fig. 8, the transition from *Idle* to *A0* activates a senior role. Then the transitions from $A_0$ to *Active* via $A_1$, …, $A_{(n-2)}$ activates the set of junior roles. The locations $A_0$, …, $A_{(n-2)}$ are committed locations. So all the junior roles are activated at the same time as that of the senior role. Similarly, along with the senior role, all the junior roles are deactivated at the same time. This is done with the help of the transitions from *Active* to *Idle* via $I_0$, …, $I_{(n-2)}$.
- *SoD*: This constraint is incorporated using Type-4 interaction. In SoD, whenever a user has activated a role, it cannot activate another conflicting role in the same session. In Type-4 interaction shown in Fig. 9, two transitions from *Idle* to *Active* are used for activation of two conflicting roles. Any one of the transitions is enabled at a time, and with this enabled transition, one of the roles is active. Unless the user has deactivated the role, it cannot activate the other role.
- *Trigger*: In GTRBAC, an event can trigger another event, which is also captured in the proposed scheme. When a role is enabled, it may enable another role. For example, when a role $r_i$ is enabled, it can enable another role, say $r_j$. This is achieved by adding three self loops at locations *Disabled*, *Enabled* and *Active* as shown in Fig. 4. This self-loop transition is labeled with a condition which allows the dependent role to enable or disable after a certain time instance. A local clock variable is used to determine the time after which the role will be triggered.

Thus it is seen that all the features of GTRBAC can be appropriately represented using the timed automata based system.

## 6. Reduction rules

In this section we study some reduction rules that can be used to obtain the GTRBAC sub models discussed in SubSection 2.2. The rules are obtained from the general modeling technique discussed in Section 3.

- *Reduction from GTRBAC to GTRBAC$_{-SoD}$*: In Section 3 it has been shown that SoD can be modeled using Type-4 User Interaction. So if we need to model GTRBAC without SoD then there is no need to execute Algorithm 7. In Section 4 we have seen that the time complexity of this algorithm is $O(|U| \times |R|)$ which is lower than the overall modeling time complexity. So, from the modeling complexity point of view, no improvement is achieved without modeling SoD constraint.
- *Reduction from GTRBAC to GTRBAC$_{-RH}$*: We use Type-3 User Interaction to model role hierarchy. As seen in Section 4, the time complexity of construction of Type-3 User Interaction, i.e., Algorithm 6, is $O(|U| \times |R|^2)$. And the complexity of this algorithm is the maximum among all the algorithms used for modeling. So if we have to map a GTRBAC system without role hierarchy, then we exclude this algorithm which, in turn, reduces the overall complexity of modeling a GTRBAC system. It becomes $O(|R| \times |P|)$ if $|P| \geq |U|$ or $O(|U| \times |R|)$ if $|P| < |U|$. Thus the decision to consider or exclude role hierarchy constraint influences the modeling complexity considerably.
- *Reduction from GTRBAC to GTRBAC$_{-PA\_time}$*: In our modeling approach, temporal constraint can also be applied in the permission automata. This represents the temporal constraint on permission-role assignment relation. However, if the temporal transition is not used in the permission automata then the location *loc1* of Fig. 5 itself is sufficient to represent the active state of a permission. In this case, whenever a role is active, the permissions associated with the role become accessible without checking any other temporal constraints. So in the construction of permission automaton only two locations and two edges would be required to represent a permission. However, the complexity of Algorithm 3 will not change. So it would not effect the mapping complexity.

## 7. Security queries on GTRBAC

In this section, we propose a desirable set of security queries that can be used for the security analysis of GTRBAC. The queries are specified using temporal logic, which expresses time as a sequence of states. Linear Temporal Logic (LTL) and

Computation Tree Logic (CTL) are two variants of temporal logic. In LTL, the future is considered to be a single path while CTL models time as a tree like structure where the *future* may consist of different paths. Two common operators used to express state formulae are G, which means "globally" or "in all states", and F, which means "finally" or "there exists some state". In our proposed scheme, we use CTL as it has the feature to express quantifiers over paths of reachable states. Two path quantifiers – A for "all the paths" and E for "there exists a path", are frequently used in specifying the queries. A set of constraints applicable to GTRBAC is listed in Table 1.

The predicate *enable*(*r*, *t*) asks whether a role *r* can be enabled at a particular time instant *t*. To ensure safety, a query can be framed whether a role remains enabled at some *unfavorable condition*.

**Definition 5**. Unfavorable Condition: *An unfavorable condition with respect to an event is defined as a time instant, or some condition like enabling or activation of a role, which must not hold for the event to be true.*

An *unfavorable condition* is represented by $\kappa$ which is a time instance (possibly a periodic time instant) or some event unfavorable for role *r*. Thus, if *r.Active* denotes a role activation event for role *r*, then an unfavorable event $\kappa$ may be defined as $t = t_k$ or $r'$*.Active* where $r'$ is a conflicting role for role *r*. So, in general, a safety query related to role enabling constraint is "A role *r* should never be enabled at some unfavorable condition". In our proposed scheme this can be expressed in CTL as follows:

$$AG(r.Enabled \rightarrow \neg\kappa)$$

Here *AG* means "in all the paths and in all the states". Similar reasoning holds for the predicate *active*(*r*, *t*). In CTL, the safety property related to role activation can be expressed as

$$AG(r.Active \rightarrow \neg\kappa).$$

Another predicate related to user-role activation constraint is *activate*(*u*, *r*, *t*), i.e., it asks whether a user *u* can activate role *r* at time *t*. Here a general safety query can be stated as "A user should never activate a role at some unfavorable condition". This can be expressed in CTL as

$$AG((u.Active \rightarrow r.Active) \rightarrow \neg\kappa).$$

**Table 1 – GTRBAC constraints and semantics.**

| Constraints | Representation | Semantics |
|---|---|---|
| Role Enabling | *Enable*(*r*, *t*) | Whether *r* is enabled at time *t* |
| User-Role Assignment | *activate*(*u*, *r*, *t*) | Whether *u* can activate *r* at time *t* |
| Permission-Role Assignment | *Available*(*r*, *p*, *t*) | Whether *p* associated with *r* is available at time *t* |
| Role Hierarchy | *r_active*(*r*1, *r*2, *t*) | Whether users of *r*1 can activate *r*2 at time *t* |
| SoD | ($\pi$, *SoD*(*r*1, *r*2 …, *rn*, *u*)) | In the interval $\pi$, *u* should not activate more than one role from *r*1, *r*2, …, *rn* |
| Trigger | *e*1 → *e*2 | Event *e*1 triggers event *e*2 |

The predicate related to permission assignment relation is *available*( *p*, *t*), i.e., it asks whether a permission *p* is available at time *t*. In this context, a safety property may be stated as "A permission should never be available at some unfavorable condition". In CTL, this can be expressed as

$$AG(p.Active \rightarrow \neg\kappa).$$

Next we consider the role hierarchy constraint. Predicate *r_active*(*r*1, *r*2, *t*) represents a role hierarchical relation. Here *r*1 is a senior role and *r*2 is a junior role. The relation indicates that the users of role *r*1 can activate role *r*2 at time instance *t*. So in this context, a safety query is "Whether a user of a senior role *r*1 activates a junior role *r*2 at some unfavorable condition". This is an example of activation-time hierarchy constraint, i.e., a user of a senior role can activate the junior role only if it has already activated the senior role. In CTL the property can be expressed as

$$AG(((u.Active \rightarrow r1.Active) \rightarrow (u.Active \rightarrow r2.Active)) \rightarrow \neg\kappa).$$

Other types of temporal hierarchies can also be addressed but for that the necessary adjustments are to be made in the role timed automaton.

SoD is an important constraint in GTRBAC. During a particular time interval if a user has activated a role then he should not be allowed to activate any other conflicting roles. Here a strong form of interval constraint on Static SoD is considered. In this context, an important security property can be stated as – "A user should never activate more than one conflicting roles within a particular time interval". In CTL, this can be expressed as

$$AG(((u.Active \rightarrow r1.Active) \rightarrow \neg(u.Active \rightarrow r2.Active)) \vee$$
$$((u.Active \rightarrow r2.Active) \rightarrow \neg(u.Active \rightarrow r1.Active))).$$

Another constraint in GTRBAC is through the use of triggers. An event *e*1 may trigger another event *e*2 which is expressed as *e*1 → *e*2. For example, when a role is enabled, it can enable another role. Here one is the cause for the other. So a safety requirement may be "An event *e*1 should always trigger another event *e*2". This can be expressed as

$$AG(e1 \rightarrow e2).$$

Along with safety queries it is also important to specify some liveness queries to ensure that the system is doing something good as well. For liveness, a property should eventually be satisfied at some reachable state. So a general liveness query related to role enabling constraint is "A role should eventually be enabled at some favorable condition".

**Definition 6**. Favorable Condition: *A favorable condition with respect to an event can be defined as a time instant, or some specific event like enabling or activation of a role, that must be satisfied when the event is true.*

A *favorable condition* is represented by $\mu$. In CTL, the liveness query mentioned above can be specified as $EF(r.Enabled \wedge \mu)$. Here *EF* means "there exists a path in which there exists a state".

Similar to the role enabling constraint, a liveness query can be applied on role activation constraint as well. For example, "a role will eventually be active at some favorable condition".

This can be expressed in CTL as $EF(r.Active \wedge \mu)$ A liveness query can also be asked on user-role activation constraint. This can be put as "a user can eventually activate a role at some desirable situation". In CTL, this is expressed as

$$EF((u.Active \rightarrow r.Active) \wedge \mu).$$

On permission assignment relation, a general liveness query can be stated as "A permission will eventually be available at some favorable condition". This can be expressed in CTL as follows:

$$EF(\ p.Active \wedge \mu)$$

Next, the role hierarchical relation is considered. Here a general liveness property can be stated as "A user of role $r1$ will eventually activate another role $r2$ at some favorable condition". The equivalent CTL expression can be written as

$$EF(((u.Active \rightarrow r1.Active) \rightarrow (u.Active \rightarrow r2.Active)) \wedge \mu).$$

On SoD constraint, we can state a liveness requirement as "A user can eventually activate one of the two conflicting roles at some favorable condition". In CTL this can be expressed as

$$EF((((u.Active \rightarrow r1.Active) \rightarrow \neg(u.Active \rightarrow r2.Active)) \vee ((u.Active \rightarrow r2.Active) \rightarrow \neg(u.Active \rightarrow r1.Active))) \wedge \mu).$$

Finally, the trigger constraint is considered. A liveness query on a triggering constraint could be – "An event $e1$ should eventually trigger another event $e2$". In CTL, we can specify the property as

$$EF(e1 \rightarrow e2).$$

## 8. Complexity of security analysis on the models

The complete modeling process involves the set of algorithms discussed in Section 4. The complexities of the algorithms are summarized in Table 2.

All these algorithms execute one after another irrespective of any inter dependency. Without loss of generality the following assumptions can be made.

1. $|R| \ll |P|$
2. $|R| \ll |U|$

Among all the modeling algorithms, the Type-3 User Interaction has the highest complexity. Thus, in general, the complexity of modeling GTRBAC to timed automata based system is $O(|U| \times |R|^2)$.

**Table 2 – Complexity of the mapping algorithms.**

| Algorithms | Complexity |
|---|---|
| Role TA | $O(\|R\| \times \|P\|)$ |
| Controller TA | $O(\|R\|)$ |
| Permission TA | $O(\|P\|)$ |
| Type-1 User Interaction | $O(\|U\|)$ |
| Type-2 User Interaction | $O(\|U\| \times \|R\|)$ |
| Type-3 User Interaction | $O(\|U\| \times \|R\|^s)$ |
| Type-4 User Interaction | $O(\|U\| \times \|R\|)$ |

For modeling $GTRBAC_{-SoD}$, the algorithm for Type-4 User Interaction is not required. The complexity of this algorithm is $O(|U| \times |R|)$ which is less than $O(|U| \times |R|^2)$. So the overall modeling complexity of $GTRBAC_{-SoD}$ is same as that of *GTRBAC*.

Now to model $GTRBAC_{-RH}$, the algorithm for Type-3 User Interaction is not required. The complexity of this algorithm is $O(|U| \times |R|^2)$. So overall complexity of modeling $GTRBAC_{-RH}$ becomes either $O(|R| \times |P|)$ or $O(|U| \times |R|)$ depending on the number of $|U|$ and number of $|P|$. If $|U| > |P|$ then resulting modeling complexity is $O(|U| \times |R|)$ else $O(|R| \times |P|)$.

For modeling $GTRBAC_{-PA\_time}$, the algorithm for permission timed automaton requires a slight modification as discussed in Section 6. But with this modification the complexity of the algorithm remains unchanged. Thus the overall modeling complexity of $GTRBAC_{-PA\_time}$ is $O(|U| \times |R|^2)$.

As seen in Section 2.3, the desirable security properties can be represented using deterministic timed automata. Verification of a property is an inclusion problem, i.e., the language corresponding to timed automata based model must be contained within the language corresponding to deterministic timed automata (Alur and Dill, 1994). So if $L(A_I)$ is the timed automata based language and $L(A_S)$ is the language of the deterministic timed automaton then $L(A_I) \subseteq L(A_S)$ refers that the model meets the specification. From Lemma 1 it can be shown that this language inclusion problem is a PSPACE-complete problem. The verification problem for real time systems has also been shown to be PSPACE-complete in Baier and Katoen (2008).

The verification process requires the transition table of the region automaton corresponding to the product of all the automata used for mapping the GTRBAC system. Each state of the region automaton can be represented in space polynomial as shown by Alur and Dill (1994). Several factors such as number of states, largest constant compared with the clocks and number of permutations over the set of all clocks are responsible for exponential blow of verification process. For a same system configuration, number of states in a $GTRBAC_{-PA\_time}$ model is much less compared to the *GTRBAC*, $GTRBAC_{-SoD}$ and $GTRBAC_{-RH}$ models. Thus if a safety property is verified in a $GTRBAC_{-PA\_time}$ model then the number of states explored will be much less compared to the other models. So when the number of system components is large then the model $GTRBAC_{-PA\_time}$ outperforms rest of the models discussed in this paper.

## 9. Experimentation and analysis

In this section we consider different configurations of a *GTRBAC* model by varying the number of roles, users and permissions and study the time and states required for verifying the properties. In addition to the main *GTRBAC* model we have also considered the other models reduced from the main model. At first we consider the following example.

**Example 1.** *There are three roles – represented by* $R = \{r0, r1, r2\}$, *twelve users – represented by* $U = \{u0, u1, \ldots, u11\}$ *and five permissions – denoted as* $P = \{p0, p1, p2, p3, p4\}$.

The temporal information associated with role and permissions are given below.

- Role $r1$ is enabled daily at 10:00 and disabled at 18:00.
- Role $r2$ is enabled after 2 time units of enabling of the role $r0$ and disabled after 2 time units of the disabling of role $r0$.
- Permission $p1$ is available at 12:00 and unavailable at 15:00.
- Permission $p2$ is available at 12:00 and unavailable at 14:00.
- Permission $p3$ is available at 13:00 and unavailable at 15:00.
- Permission $p4$ is available at 14:00 and unavailable at 17:00.

To carry out the experiments, we have used timed automata based verification tool Uppaal (Behrmann et al., 2004) with the following settings of the configurable parameters – breadth first search order, conservative state space reduction, Difference Bound Matrix (DBM) state space representation, automatic extrapolation and 16 MB hash table size. The verification process was carried out on a Linux platform with 2 GB RAM using Intel(R) Xeon 2.8 GHz processor.

With this information, a timed automata based state transition system is developed as per the algorithms given in Section 4. Next some of the safety and liveness properties are formed using the guidelines described in Section 7. Here a system configuration is characterized by the representation *user*:*role*:*permission*. So the example we have introduced above is denoted as 12:3:5. For verification of the properties, we start with this system configuration and later, vary the other parameters such as role, permission and user.

From the experimental results it has been observed that the time required and states explored of various safety properties are almost same. But this is not true for liveness properties. So we have considered only one safety property and a few liveness properties for the purpose of analysis. With the given set of access control policies a safety property could be "$r0$ is never disabled between 9:00 and 17:00". In CTL, this can be expressed as

$$P1 : AG(r0.Disabled \rightarrow \neg((t > 9) \wedge (t < 17))).$$

In this context, a general liveness property is "$r0$ is eventually enabled at time 9:00". This can be expressed in CTL as

$$P2 : EF(r0.Enabled \wedge \ t == 9)$$

Another liveness property can be stated as "$r0$ will be activated at some time". This can be expressed in CTL as follows:

$$P3 : EF(r0.Active).$$

To ensure that $u0$ can activate a role, we use the following CTL property.

$$P4 : EF(u0.Active).$$

If we want to check that permission $p0$ will be eventually available for access then we can use the liveness property given as

$$P5 : EF(\ p0.Active).$$

Now we observe the time required and states explored for verification of safety property $P1$. The property is verified against the GTRBAC and other reduced models. For a particular model the property is verified for different configurations. We start with the system configuration 12:3:5 and later vary the number of permissions, roles and users. It is obvious fact that time required for verification is proportional to the states explored. So we analyze the properties in terms of number of states explored.

## 9.1. *Impact of number of permissions on verification*

From Tables 3–6 it can be observed when we consider *GTRBAC*, $GTRBAC_{-SoD}$, $GTRBAC_{-RH}$ models then the states explored increases exponentially. But from Table 7 it can be seen that for $GTRBAC_{-PA\_time}$ the states explored increases at a steady rate. Permission automaton of $GTRBAC_{-PA\_time}$ requires less number of locations and no temporal constraints. When the temporal constraints on PA relation is specified then a permission automaton requires four locations, six edges and two temporal constraints. But if there is no temporal constraints on PA relation then we need only two locations and two edges. This significantly reduces the number of states in the resulting system which is a product of all the automata defined by the modeling process. Thus the resulting product automaton in $GTRBAC_{-PA\_time}$ has less number of states compared to other *GTRBAC* models. This gives a much better performance in terms of states required for verification of safety property. This can be illustrated using the graphs shown in Fig. 11. In the figure $x$-axis represents the number of permissions and $y$-axis represents the number of states explored.

Table 8 shows the states explored for different liveness properties in a *GTRBAC* model. We have observed that verification of the properties is not affected by the increasing number of permissions. This happens because the addition of a permission does not introduce intermediate states between the initial state and the state to be explored. Similar result is also obtained for other models also.

## 9.2. *Impact of number of roles on verification*

Now if we vary number of roles keeping other parameters constant then the verification of safety property behaves as shown in Fig. 12. In the figure $x$-axis represents the number of roles and $y$-axis represents the number of states explored. It is

**Table 3 – Time required and states explored for verification of *P1* safety property in *GTRBAC* model.**

| User:Role: Perm | Time required | States explored |
|---|---|---|
| 12:3:5 | 6.269 s | 137,528 |
| 12:3:7 | 22.389 s | 461,752 |
| 12:3:9 | 1 m 24.016 s | 1,602,378 |
| 12:5:9 | 1 m 2.657 s | 1,253,318 |
| 12:7:9 | 28.0338 s | 522,574 |
| 12:5:11 | 49.063 s | 977,823 |
| 12:7:13 | 1 m 53.315 s | 2,153,436 |
| 13:7:9 | 1 m 54.791 s | 2,217,166 |
| 15:7:9 | 8 m 37.647 s | 8,711,392 |

**Table 4 – Time required and states explored for verification of P1 safety property in $GTRBAC_{-SoD}$ model.**

| User:Role: Perm | Time required | States explored |
|---|---|---|
| 12:3:5 | 6.222 s | 141,064 |
| 12:3:7 | 20.325 s | 439,258 |
| 12:3:9 | 1 m 16.060 s | 1,526,275 |
| 12:5:9 | 52.775 s | 1,092,254 |
| 12:7:9 | 1 m 15.518 s | 1,574,316 |
| 12:5:11 | 40.475 s | 831,223 |
| 12:7:13 | 2 m 32.336 s | 2,985,729 |
| 13:7:9 | 2 m 35.892 s | 3,111,218 |
| 15:7:9 | 42 m 29.814 s | 12,919,738 |

**Table 6 – Time required and states explored for verification of P1 safety property in $GTRBAC_{-RH}$ model.**

| User:Role: Perm | Time required | States explored |
|---|---|---|
| 12:3:5 | 8.350 s | 178,599 |
| 12:3:7 | 25.933 s | 530,593 |
| 12:3:9 | 1 m 34.928 s | 1,773,018 |
| 12:5:9 | 1 m 7.191 s | 1,326,926 |
| 12:7:9 | 48.437 s | 998,398 |
| 12:5:11 | 51.080 s | 1,009,096 |
| 12:7:13 | 2 m 55.520 s | 3,240,054 |
| 13:7:9 | 2 m 59.346 s | 3,381,498 |
| 15:7:9 | 190 m 10.357 s | 17,255,613 |

observed that with the increase in number of roles the states explored for a safety property does not increase always. In some cases the states explored may reduce as well. This strange behavior is dependent on the how many permissions are assigned to a particular role.

For instance, in the system configuration 12:3:9, there were twelve users, three roles and nine permissions. In this example, one of the roles is associated with five permissions. So in that role automaton, the number of committed locations was 15. But when a role is added to the system, to keep the permission parameter constant, we remove a permission from that automaton and add it to the new role automaton. With the removal of a permission from a role, three committed locations are removed. So the number of committed locations on a single role effects the verification result. A system with less number of committed locations on a single automaton performs better.

Usually, addition of a role also requires addition of new permissions to an exiting system. So we have varied both role and permission together to observe the effect on the verification of safety and liveness properties. For safety property, we observe that the states explored for verification increases exponentially in *GTRBAC*, $GTRBAC_{-SoD}$, $GTRBAC_{-RH}$ models. This is due to the fact that the number of states in the system increases with the addition of role and permission timed automata. Also, the number of temporal constraints increases due to the addition of role enabling and disabling constraints and due to the addition of permission availability constraints as well. This causes the increase in the verification time and number of states explored for the safety property. However in $GTRBAC_{-PA\_time}$ model the states explored increases at a steady rate. This happens because of the simplified structure of the permission automaton when the temporal constraints on PA relations are not required to be specified.

Table 9 shows the states explored for verification of liveness properties. The result is not affected by a variation in the number of roles. Like the previous case, similar reasonings can be applied here also.

## 9.3. *Impact of number of users on verification*

Finally we show the effect of varying number of users on the verification of safety property for different system configurations. When the number of users is increased then the states explored increases exponentially. This can be seen in Fig. 13. In the figure *x*-axis represents the number of users and *y*-axis represents the number of states explored.

Table 3 shows that with the addition of users in the system, the states explored by the verification of the safety property increases at a much higher rate as compared to the other parameters like role and permission. Each user is accompanied by at least two temporal constraints – one for role activation and another for role deactivation. For Type-2 and Type-4 Interactions more such constraints are required. For Type-3 Interaction more number of committed locations are required. Thus adding more users means more states and more temporal constraints in the final product automaton. These directly influence the verification process. Another important observation is that the verification of safety property in $GTRBAC_{-RH}$ requires more states to be explored than $GTRBAC_{SoD}$. This is because of the non determinism involves in the parallel transitions of Type-4 Interaction.

**Table 5 – Time required and states explored for verification of P1 safety property in $GTRBAC_{-SoD}$ model.**

| User:Role: Perm | Time required | States explored |
|---|---|---|
| 12:3:5 | 6.222 s | 141,064 |
| 12:3:7 | 20.325 s | 439,258 |
| 12:3:9 | 1 m 16.060 s | 1,526,275 |
| 12:5:9 | 52.775 s | 1,092,254 |
| 12:7:9 | 1 m 15.518 s | 1,574,316 |
| 12:5:11 | 40.475 s | 831,223 |
| 12:7:13 | 2 m 32.336 s | 2,985,729 |
| 13:7:9 | 2 m 35.892 s | 3,111,218 |
| 15:7:9 | 42 m 29.814 s | 12,919,738 |

**Table 7 – Time required and states explored for verification of P1 safety property in $GTRBAC_{-PA\_time}$ model.**

| User:Role: Perm | Time required (s) | States explored |
|---|---|---|
| 12:3:5 | 0.390 | 9109 |
| 12:3:7 | 0.509 | 12,881 |
| 12:3:9 | 0.636 | 16,653 |
| 12:5:9 | 0.730 | 17,371 |
| 12:7:9 | 0.781 | 18,689 |
| 12:5:11 | 0.808 | 18,975 |
| 12:7:13 | 1.017 | 23,057 |
| 13:7:9 | 1.292 | 30,249 |
| 15:7:9 | 4.939 | 100,993 |

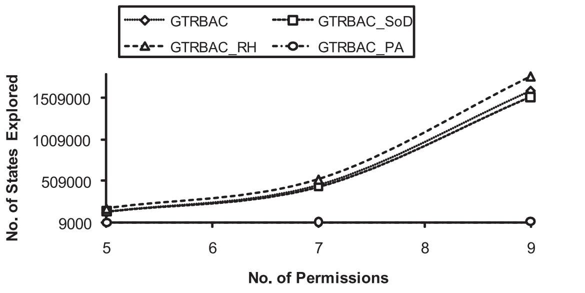

**Fig. 11 – No. of states explored for *P*1 safety property with varying number of permissions in different models.**

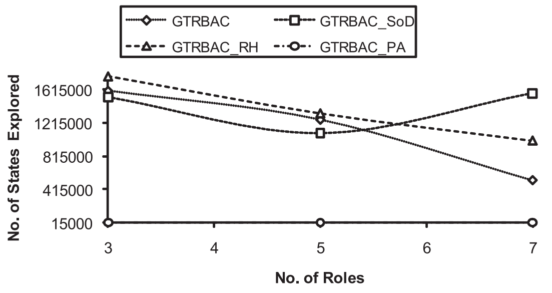

**Fig. 12 – No. of states explored for *P*1 safety property with varying number of roles in different models.**

However for liveness properties, the variation in the number of users does not substantially effect the verification. The results shown in Table 10 depict that. So it can be concluded that unlike other parameters, the safety verification process is more sensitive to a variation in the number of users. Thus to verify a safety property in a large system with many users, state space explosion may occur. But the result also clearly shows that if *GTRBAC* model is reduced to $GTRBAC_{-PA\_time}$ then there is a significant improvement in the verification result. So if the successful activation of a role is the only criteria to get access to the permissions associated with the role then there is no need to check temporal constraints on PA relation. In such situations, $GTRBAC_{-PA\_time}$ model can be effectively used for verifying the different safety properties.

The main limitation of our approach is that it suffers from state space explosion problem. For instance, we encountered state space explosion problem while verifying a safety property in a *GTRBAC* system configuration of 16:7:9. Thus scalability becomes an issue when a system with large number of components is considered for the analysis purpose.

## 10. Related work

Safety analysis in access control models has been started quite sometime back. The first significant work is done by Harrison et al. (1976). They formalize the problem of safety analysis in the context of access control matrix model (commonly known as the HRU model). The goal of this work is to determine whether a protection system could reach a state in which a particular right is leaked. It has been shown that in general, safety analysis problem for access control matrix model is undecidable. Later Jones et al. (1976) propose Take-Grant Protection Model (TGPM) which represents a system as a directed graph where vertices are either subjects or objects. The edges between the vertices are labeled with the rights that the source of the edge has over the target. Using the rules of this model one can determine whether the system reaches a state in which a particular query can be answered or not. They have also proved that it is possible to decide on the safety of a system even when the number of subjects and objects is very large, or unbounded. Thus, in a general scheme

**Table 8 – States explored for verification of liveness properties with varying number of permissions in a GTRBAC model.**

| Property | User:Role:Perm | States explored |
|---|---|---|
| P2 | 12:3:5 | 3 |
| P3 | 12:3:5 | 7 |
| P4 | 12:3:5 | 5 |
| P5 | 12:3:5 | 8 |
| P2 | 12:3:6 | 3 |
| P3 | 12:3:6 | 7 |
| P4 | 12:3:6 | 5 |
| P5 | 12:3:6 | 8 |
| P2 | 12:3:7 | 3 |
| P3 | 12:3:7 | 7 |
| P4 | 12:3:7 | 5 |
| P5 | 12:3:7 | 8 |
| P2 | 12:3:8 | 3 |
| P3 | 12:3:8 | 7 |
| P4 | 12:3:8 | 5 |
| P5 | 12:3:8 | 8 |
| P2 | 12:3:9 | 3 |
| P3 | 12:3:9 | 7 |
| P4 | 12:3:9 | 5 |
| P5 | 12:3:9 | 8 |

**Table 9 – States explored for verification of liveness properties with varying number of roles in a GTRBAC model.**

| Property | User:Role:Perm | States explored |
|---|---|---|
| P2 | 12:3:9 | 3 |
| P3 | 12:3:9 | 7 |
| P4 | 12:3:9 | 5 |
| P5 | 12:3:9 | 8 |
| P2 | 12:4:9 | 3 |
| P3 | 12:4:9 | 7 |
| P4 | 12:4:9 | 5 |
| P5 | 12:4:9 | 8 |
| P2 | 12:5:9 | 3 |
| P3 | 12:5:9 | 7 |
| P4 | 12:5:9 | 5 |
| P5 | 12:5:9 | 8 |
| P2 | 12:6:9 | 3 |
| P3 | 12:6:9 | 7 |
| P4 | 12:6:9 | 5 |
| P5 | 12:6:9 | 8 |
| P2 | 12:7:9 | 3 |
| P3 | 12:7:9 | 7 |
| P4 | 12:7:9 | 5 |
| P5 | 12:7:9 | 8 |

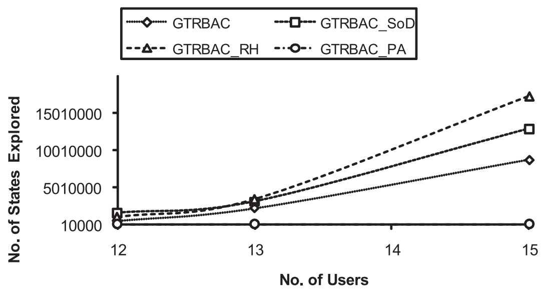

**Fig. 13 – No. of states explored for *P*1 safety property with varying number of users in different models.**

like HRU, safety analysis is undecidable but in a scheme enforced with reasonable restrictions such as TGPM, safety analysis becomes decidable.

So a balance is required between the ability to perform an efficient safety analysis and the generalization ability of a model. For this purpose, Sandhu (1988) introduce the idea of Schematic Protection Model (SPM). In SPM, entities are strongly typed, i.e., every entity is created to be of a specific type, which cannot change thereafter. It is shown that analysis is tractable for this model provided certain restrictions are imposed on subject creation. SPM provides a “high-level” structure compared to the “low-level” structure of the access matrix. This makes policy specification more convenient in SPM.

Later, Koch et al. (2002) show that with reasonable restrictions on the rules, the safety analysis problem becomes decidable. They use graph transformations as a general formalism to specify access control policies based on roles. Ahmed and Tripathi (2003) propose a model checking mechanism for verification of security requirements in role based Computer Supported Cooperative Work (CSCW) systems. Li and Tripunitara (2006) perform security analysis on two restricted versions of administrative RBAC. These are known as AATU (Assignment And Trusted Users) and AAR (Assignment And Revocation). They propose two reduction algorithms and study complexity results for various analysis problems such as safety, availability and containment. Stoller et al. (2007) consider negative preconditions and static mutually exclusive role constraints in the policy analysis of administrative RBAC. Zhang and Joshi (2008) address constraints like hybrid hierarchy and dynamic separation of duty in handling the user authorization process. Jha et al. (2008) perform a comparison between the use of model checking and first order logic programming for the security analysis of RBAC. It is concluded that model checking is a promising approach in this context. Recently, Rakkay and Boucheneb (2009) have performed a security analysis on RBAC using colored Petri-Nets and CPN tools. In this work, a security administrator can use the graphical representation and analysis framework to check why some permissions are granted and to detect whether security constraints are violated.

Formal analysis of GTRBAC needs to be done differently due to the presence of temporal constraints in the system. Joshi et al. (2005b) present an analysis of the expressiveness of the constructs provided by the GTRBAC model and show that its constraints-set is not minimal. Gal and Atluri propose a Temporal Data Authorization Model (TDAM). It is capable of expressing access control policies based on the temporal characteristics of data, such as valid and transaction time (Gal and Atluri, 2000). Bertino et al. (2001) provide a polynomial time algorithm to verify whether specifications in TRBAC are free from ambiguities. Shafiq et al. (2005) study a Petri-Net based framework for verifying the correctness of event-driven RBAC policies in real time systems. It considers only the cardinality and separation of duty constraints. But it does not handle temporal RBAC policies. A few other systems where time is considered to be a critical factor have also been analyzed. Alur et al. (1990) have explored model checking for the analysis of real time systems. Furfaro and Nigro (2003) have used timed automata for temporal verification of real time state machines. In a recent work timed automata have been used for conformance testing of TRBAC system (Masood et al., 2009). The work presents a complete fault coverage with respect to a proposed TRBAC fault model.

Recently, some work has been done for the security analysis of temporal RBAC models using timed automata. In the first work, the TRBAC framework is considered (Mondal and Sural, 2008a, 2009). Temporal constraints such as role activation and deactivation are mapped to a timed automata based system. The performance of this model is reasonably good for large number of processes representing users. A limitation with this approach is that all the users for a given role behave in the same manner. Also, multiple role activation by a single user is not allowed. In the second work, some more temporal constraints are taken into consideration (Mondal and Sural, 2008b). Temporal constraints on role enabling, activation, permission assignment, user assignment, role hierarchy and SoD are addressed during the mapping process. However, it does not handle role triggers. Also, the approach assumes that a user can activate only a single role at a particular instance of time. In this case, security analysis performance degrades with growing number of users, roles and permissions.

**Table 10 – States explored for verification of liveness properties with varying number of users in a GTRBAC model.**

| Property | User:Role:Perm | States explored |
|---|---|---|
| P2 | 12:7:9 | 3 |
| P3 | 12:7:9 | 7 |
| P4 | 12:7:9 | 5 |
| P5 | 12:7:9 | 8 |
| P2 | 13:7:9 | 3 |
| P3 | 13:7:9 | 7 |
| P4 | 13:7:9 | 5 |
| P5 | 13:7:9 | 8 |
| P2 | 14:7:9 | 3 |
| P3 | 14:7:9 | 7 |
| P4 | 14:7:9 | 5 |
| P5 | 14:7:9 | 8 |
| P2 | 15:7:9 | 3 |
| P3 | 15:7:9 | 7 |
| P4 | 15:7:9 | 5 |
| P5 | 15:7:9 | 8 |
| P2 | 16:7:9 | 3 |
| P3 | 16:7:9 | 7 |
| P4 | 16:7:9 | 5 |
| P5 | 16:7:9 | 8 |

Of late, Mondal et al. (2009) have presented initial results on security analysis of the GTRBAC model. This work covers a wide number of temporal constraints. Here a general form of safety and liveness queries are verified using the model checking mechanism. In the current paper, we have categorized the GTRBAC model and proposed the reduction rules to obtain those models which are based on different constraints. We have also made an analysis on those models and compare the results.

## 11. Conclusions

In this paper, a state transition model for GTRBAC has been proposed utilizing the expressibility of timed automaton. It has been observed that a timed automaton can express any temporal constraint using its clock variables. The proposed model maps the behavior of components such as users, roles and permissions. The diverse behavior of users is captured by creating automata of different types of interaction. A location in an automaton represents the state of the corresponding component. Reduction rules are also proposed to model the GTRBAC sub models. A desirable set of security properties is constructed from the constraints specified in the GTRBAC model. These properties are then used for verification. The effect of various GTRBAC components such as role, user and permission on the verification analysis of safety and liveness properties has been studied. From the results obtained, it is seen that the verification process is sensitive to the number of users. But if we reduce *GTRBAC* model to a $GTRBAC_{-PA\_time}$ model then the verification can be carried out for a large number of components. But in such a situation we need to compromise with the temporal constraints on *PA* relation. So a trade off decision has to be made on whether to examine a system with large number of components or with a large number of constraints.

## Acknowledgments

This work is partially supported by a research grant from the Department of Science and Technology, Government of India, under Grant No. SR/S3/EECE/082/2007 and by the National Science Foundation under Grant No. CNS-1018414.

## REFERENCES

Ahmed T, Tripathi A. Static verification of security requirements in role based CSCW systems. In: Proceedings of the 8th ACM symposium on access control models and technologies; 2003. p. 196–3.

Alur R, Courcoubetis C, Dill DL. Model checking for real time systems. In: Proceedings of the 5th symposium on logic in computer science; 1990,p. 414–25.

Alur R, Dill DL. A theory of timed automata. Theoritical Computer Science 1994;126(2):183–235.

Baier C, Katoen JP. Principles of model checking. 55 Hayward Street, Cambridge, MA: MIT Press; 2008.

Behrmann G, David A, Larsen KG. A tutorial on UPPAAL. In: 4th International school on formal methods for the design of computer, communication and software systems; 2004. p. 200–236.

Bertino E, Bonatti PA, Ferrari E. TRBAC: a temporal role based access control model. ACM Transactions on Information and System Security August 2001;4(3):191–223.

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

Bhatti R, Joshi JBD, Bertino E, Ghafoor A. X-GTRBAC admin: a decentralized administration model for enterprise wide access control. ACM Transactions on Information and System Security November 2005b;8(4):388–423.

Ferraiolo DF, Sandhu R, Gavrila S, Khun DR, Chandramouli R. Proposed NIST standard for role based access control. ACM Transactions on Information and System Security August 2001;4(3):224–74.

Furfaro A, Nigro L. Temporal verification of communicating real time state machines using uppaal. In: Proceedings of the IEEE international conference on industrial technology; 2003. p. 399–4.

Gal A, Atluri V. An authorization model for temporal data. In: Proceedings of 7th ACM conference on computer and communication security; 2000. p. 144–3.

Harrison MA, Ruzzo WL, Ullman JD. Protection in operating systems. Communications of the ACM August 1976;19(8):461–71.

Jha S, Li N, Tripunitara M, Wang Q, Winsborough W. Towards formal verification of role based access control policies. IEEE Transactions on Dependable and Secure Computing October 2008;5(4):242–55.

Jones AK, Lipton RJ, Snyder L. A linear time algorithm for deciding security. In Proceedings of the 17th annual IEEE symposium on foundations of computer science; 1976. p. 33–41.

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

Joshi JBD, Bertino E, Ghafoor A. An analysis of expressiveness and design issues for the generalized temporal role-based access control model. IEEE Transactions on Dependable and Secure Computing April 2005b;2(2):157–75.

Koch M, Mancini LV, Parisi-Presicce F. Decidability of safety in graph-based models for access control. In: Proceedings of the 7th European symposium on research in computer security; 2002. p. 229–43.

Li N, Tripunitara M. Security analysis in role based access control. ACM Transactions on Information and System Security November 2006;9(4):391–420.

Masood A, Ghafoor A, Mathur A. Conformance testing of temporal role-based access control systems. IEEE Transactions on Dependable and Secure Computing; 2009.

Mondal S, Sural S. Security analysis of Temporal RBAC using timed automata. In Proceedings of the 4th international symposium on information assurance and security; 2008a. p. 37–40.

Mondal S, Sural S. A verification framework for temporal RBAC with Role Hierarchy. In Proceedings of the 4th international conference on information and systems security; 2008b. p. 140–7.

Mondal S, Sural S. Security analysis of RBAC with temporal constraints – a model checking approach. Journal of Information Assurance and Security July 2009;4(3):319–28.

Mondal S, Sural S, Atluri V. Towards formal security analysis of GTRBAC using timed automata. In: 14th ACM symposium on access control models and technologies; 2009. p. 33–42.

Rakkay H, Boucheneb H. Security analysis of role based access control models using colored Petri Nets and CPN tools. In: Transactions on computational science, special issue on security in computing. Lecture Notes in Computer Science 5430, vol. 4. Springer; 2009. 149–176.

Sandhu RS. The schematic protection model: its definition and analysis for acyclic attenuating systems. Journal of the ACM 1988;35(2):404–32.

Shafiq B, Masood A, Joshi J, Ghafoor A. A role based access control policy verification framework for real time systems. In: Proceedings of the 10th IEEE international workshop on object oriented real time dependable systems; 2005. p. 13–20.

Stoller SD, Yang P, Ramakrishnan CR, Gofman MI. Efficient policy analysis for administrative role based access control. In: Proceedings of the 14th ACM conference on computer and communications security; 2007. p. 445–5.

Zhang Y, Joshi JBD. UAQ: a framework for user authorization query processing in RBAC extended with hybrid hierarchy and constraints. In: Proceedings of the 13th ACM symposium on access control models and technologies; 2008. p. 83–2.

**Samrat Mondal** is presently working as an Assistant Professor in Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar. He received the Ph.D. degree from Indian Institute of Technology, Kharagpur in 2010. His main research interests include information system security, database systems and data mining.

**Shamik Sural** is an associate professor at the School of Information Technology, IIT Kharagpur India. He received the Ph.D. degree from Jadavpur University in 2000. Before joining IIT, he held technical and managerial positions in a number of organizations both in India as well as in the USA. Dr. Sural is a senior member of IEEE and a recipient of the Alexander von Humboldt Fellowship for Experienced Researchers. He has published more than a hundred research papers in reputed international journals and conferences. His research interests include database security, access control and multimedia database systems.

**Vijayalakshmi Atluri** is a Professor of Computer Information Systems in the MSIS Department, and research director for the Center for Information Management, Integration and Connectivity (CIMIC) at Rutgers University. Her research interests include Information Security, Privacy, Databases, Workflow Management, Spatial Databases, Multimedia and Distributed Systems. She has published extensively in such journals and conferences as the ACM Transactions on Information Systems Security, IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on Dependable and Secure Computing, The VLDB Journal, ACM Conference on Computer and Communication Security, IEEE Symposium on Security and Privacy, IEEE Conference on Data Engineering. She currently serves as the Vice-chair for the ACM Special Interest Group on Security Audit and Control (SIGSAC), and Chair of the IFIP WG11.3 Working Conference on Database Security. She was the recipient of the National Science Foundation CAREER Award, and the Rutgers University Research Award for untenured faculty for outstanding research contributions. She is a senior member of the IEEE Computer Society and member of the ACM.