Temporal RBAC Security Analysis Using Logic Programming in the Presence of Administrative Policies

Sadhana Jha[1], Shamik Sural[2], Jaideep Vaidya[3], and Vijayalakshmi Atluri[3]

[1] Advanced Technology Development Centre
[2] School of Information Technology,
Indian Institute of Technology, Kharagpur, India
{sadhanajha,shamik}@sit.iitkgp.ernet.in
[3] Management Science and Information Systems Department,
Rutgers University, USA
jsvaidya@business.rutgers.edu, atluri@rutgers.edu

Abstract. Temporal Role Based Access Control (TRBAC) is an extension of the role based access control (RBAC) model in the temporal domain. It is used by organizations needing to enforce temporal constraints on enabling and disabling of roles. For any chosen access control model, decentralization of administrative authority necessitates the use of a separate administrative model. Even with the use of an administrative model, decentralization often leads to an increased concern for security. Analysis of security properties of RBAC has been extensively done using its administrative model (ARBAC97). However, TRBAC security analysis in the presence of an administrative model so far has received limited attention. This paper proposes a method for performing formal security analysis of TRBAC considering a recently proposed administrative model named AMTRAC, which includes all the relations of ARBAC97 as well as an additional set of relations (named REBA) for administering the role enabling base of a TRBAC system. All the components of TRBAC and AMTRAC are specified in Prolog along with the desired safety and liveness properties. Initially, these properties are verified considering the non-temporal relations only, followed by handling of the temporal relations as well. Experimental results show that the method is both effective as well as scalable.

Keywords: TRBAC, AMTRAC, Prolog, Security Analysis.

1 Introduction

Providing secure and restrictive access to its resources is one of the main concerns for any organization. Role-based access control (RBAC) [14] has emerged as an effective means for specifying and meeting security goals in organizations with diverse access control requirements. It is based on the central notion of *roles*. Roles are created to perform a job functions and are associated with a set

A. Prakash and R. Shyamasundar (Eds.): ICISS 2014, LNCS 8880, pp. 129–148, 2014.
© Springer International Publishing Switzerland 2014

of permissions. Users are assigned to roles based on their organizational responsibilities. The Temporal RBAC (TRBAC) model [1] allows temporal constraints on when the roles can be used. It restricts roles to be either in the *enabled* or the *disabled* state. Transition from the enabled to the disabled state is termed as *enabling* of role and the reverse as *disabling* of role.

Administration of a large RBAC or TRBAC system is a challenging task and to address this, administrative models such as ARBAC97 [12] for RBAC and AMTRAC [16] for TRBAC have been proposed. An administrative model brings decentralization in administration by allowing a chief security officer to delegate the authority of management to other administrative officers. It incorporates certain relations that allow administrators to change the state of the system. The use of administrative models restricts the set of possible states which an access control system can transit to. However, decentralization also leads to increased possibility of intentional or unintentional violation of security, resulting in unauthorized information flow. Hence, it is imperative that a comprehensive understanding and analysis of these models be done before they are deployed in practical situations.

Access control models including RBAC and TRBAC provide a multitude of features. While this enables specifying different kinds of access control policies, analysis of the level of security provided by the access control model is tedious and error-prone, if not impossible, when attempted manually. It is more so, when security administration is distributed and state changes could be made by different administrators based on the administrative roles they are allowed to invoke. Automated security analysis is also complicated and requires appropriate tools for solving computationally hard problems. Since state transition in an RBAC or TRBAC system can take place only through the set of administrative relations defined in its administrative model, consideration of administrative models is necessary for analyzing the security properties. Till date, there is limited work on TRBAC security analysis. Additionally, none of the existing approaches consider a comprehensive administrative model.

In this paper, we present a methodology for TRBAC security analysis using Prolog in the presence of a recently proposed administrative model named AMTRAC [16]. Essentially, a security analysis problem can be considered as a searching problem in which the analyzer searches for an instance in which the desired security property does not hold. We use Prolog for security analysis since it has been shown to be able to handle such type of problems quite efficiently. Moreover, the inbuilt capability of handling lists makes it suitable for representing the temporal elements of a TRBAC system. However, effective modeling of temporal components and administrative functions so that security analysis can be done efficiently is a non-trivial task as Prolog does not directly support all such features.

Security analysis in the current context is primarily concerned with the verification of safety and liveness properties. A safety property can be stated as "*Does user u get permission p at time instant t?*" while a liveness property could be "*Is there a time instant t, when none of the roles in the system is in the enabled*

state?". Thus, safety property checks for the presence of an enabled user at a particular point of time and liveness property checks for the presence of an enabled role over the entire set of time periods defined by all the periodic events of the system under consideration. For analysis, both the TRBAC system as well as the corresponding AMTRAC relations are specified in Prolog. Security properties are also defined using Prolog syntax. These specifications are given as input to the *SWI* interpreter[1], which returns true or false depending on whether the system specifications meet the given security properties. It may be noted that, for analysis, we do not use *temporal logic* extension of Prolog, rather first order form is used along with a representation of the temporal aspects of TRBAC.

The rest of the paper is organized as follows. Section 2 contains preliminaries about RBAC, TRBAC, AMTRAC and Prolog. In Sections 3 and 4, we explain how safety and liveness analysis can be done for TRBAC under the AMTRAC administrative model using Prolog. Section 5 presents the results of experimental evaluation of the proposed approach. Section 6 discusses some of the previous work done in this field. Section 7 concludes the paper along with suggestions for prospective future work.

2 Preliminaries

In this section, we provide a brief introduction of the RBAC and TRBAC access control models (Section 2.1), their administrative models (Section 2.2), and Prolog (Section 2.3). This would help in understanding the key concepts used in the rest of the paper.

2.1 RBAC and TRBAC

RBAC [14] is an access control model in which role represent job functions within an organization. Permissions are assigned to roles instead of directly to users. Users get appropriate permissions by becoming members of corresponding roles. The basic components of RBAC include a set of users (U), a set of roles (R), a set of permissions (P), a user-role assignment (UA), a role-permission assignment (PA) and a role- role relation called role hierarchy (RH). These components collectively determine whether a particular user has permission to access a certain resource. RBAC, thus, effectively controls which users have access to which resource.

However, for many applications, which need to make access control decisions based on temporal information, RBAC is not adequate. For such applications, TRBAC, a temporal extension of RBAC, has been proposed. It includes a Role Enabling Base (REB) for defining periodic enabling and disabling of roles expressed as periodic events along with temporal dependencies among roles specified using role triggers. Periodic events (PE) are of the form {I, P, p:E}, where I represents the interval for which a periodic event is valid, P represents a periodic expression based on the notion of calendars [1] and *p:E* represents a prioritized event expression [1]. For example, (*[01/01/2014, 01/01/2020], all.years +*

[1] `http://www.swi-prolog.org/`

all.months + all.weeks + {1,2,3,4,5}.days + {10}.hours ▷ 8.hours, H: Enable r) conveys that the role r is enabled with high (H) priority for a duration of eight hours starting from the tenth hour of the first, second, third, fourth and fifth day of every week of every month of every year, during the period 01/01/2014 till 01/01/2020. A role trigger (RT) is of the form $E_1, E_2, ...E_n, C_1, C_2, ...C_m \longrightarrow$ *p:E after* $\triangle t$ where, $E_i s$ and $C_j s$ represent event expressions and role status expressions [1], respectively. *p:E* and $\triangle$t represent a prioritized event expression and delay, respectively. For example, (*Enable a, Enabled b* $\rightarrow$ *Enable c*) conveys that, enabling of role a triggers enabling of role c provided that role b is already enabled. Thus, periodic events and role triggers collectively determine which roles in a system are enabled or disabled at various points in time.

2.2 ARBAC97 and AMTRAC

An administrative model defines the set of valid state transition rules for an access control system. ARBAC97 is the first comprehensive administrative model for RBAC. It has three components, namely, URA97, PRA97 and RRA97. URA97 includes two relations, namely, *can_assign* and *can_revoke* to modify the UA of an RBAC system. These relations respectively provide authority to an administrator to assign new users to a role and to revoke existing users from a role. PRA97 includes two relations, namely, *can_assignp* and *can_revokep* to modify the PA of an RBAC system. These relations respectively provide authority to an administrator to assign new permissions to a role and revoke existing permissions from a role. RRA97 includes a relation named as *can_modify* to modify the RH of an RBAC system. It allows an administrator to insert new edges into the hierarchy and delete existing edges from the hierarchy. It also allows them to create new roles as well as delete existing roles within a role range. The components of ARBAC97, thus, can be used to change the state of an RBAC system. To change the state of a TRBAC system, along with the relations defined in ARBAC97, an additional component named as REBA (Role Enabling Base Assignment) has been introduced in AMTRAC. REBA includes a set of eighteen relations. These relations are partitioned into four different categories. While, the first and second categories of relations allow an administrator to modify an existing periodic event and role triggers, the third and the fourth category relations allow an administrator to add or delete new periodic events and role triggers to or from the REB of a TRBAC system. Thus, REBA can be used to make various possible modifications to the REB of a TRBAC system.

2.3 Introduction to Prolog

A Prolog program describes relations defined by means of *clauses*. A clause can be either a *fact* or a *rule*. A *fact* represents a predicate expression that makes a declarative statement about the problem domain. For example, consider an authorization system having *Alice*, *Charles* and *Tom* as its users. Each of the three users is associated with a password through the relation named as *username_password*. The set of facts representing this can be written as follows:

username_password(Alice, 123456)
username_password(Charles, 123abc)
username_password(Tom, a2gh45)

A *rule* is a predicate expression that uses logical implication (:-) to describe a relationship among facts. For example, for the authorization system mentioned above, a rule to check whether the combination of *username* and *password* entered by a user is valid or not can be written as follows:

valid_username_password_combination (U, P):- username_password(U, P) → write('Valid username password combination'); write('Invalid username password combination')

A program logic expressed in the syntax of Prolog is executed using an interpreter. The interpreter is provided with a query to check whether certain conditions hold or not. For instance, for the authorization system specified above, a query to check whether *Alice* and *123456* is a valid *username-password* combination can be written as:

valid_username_password_combination(Alice, 123456)

The interpreter, when provided with the given query, tries to find whether in the presence of the provided set of facts and rules, it can derive that *Alice* and *123456* form a valid *username-password* combination. If it is able to do so, then it returns *true*; else, returns *false* as output. Thus, we have seen how a Prolog interpreter can be employed to check for the presence of certain conditions in a system.

3 System Modeling in Prolog

To check whether a TRBAC state continues to remain in the safe state in presence of a set of administrative relations, the initial state of a TRBAC system as well as the set of administrative relations are provided as *facts* to the interpreter. Meaning of the security properties is defined in the form of *rules*.

3.1 Modeling TRBAC Using Prolog

In this section, we show how different components of a TRBAC system can be modeled in the form of *facts* of a Prolog program. While modeling a TRBAC system, the following assumptions are made: i) Initially, all roles are in the disabled state, ii) If an enabled role needs to be disabled, the corresponding role trigger is removed from the REB, iii) If a role r_1 triggers another role r_2, then disabling of r_1 causes automatic disabling of r_2, iv) All event expressions are of the same priority and v) Triggers are fired without any delay.

To represent *users, roles* and *permissions*, facts of the form *user*(*string*), *role*(*string*) and *per*(*string*), respectively are used. In these facts, *string* denotes

the name of a user, role or permission. To represent that u_i is a user, a fact of the form $user(u_i)$ is added to the specification, to represent that r_i is a role, a fact of the form $role(r_i)$ is added and to show that p_i is a permission, a fact of the form $per(p_i)$ is added to the program. The *UA*, *PA* and *RH* relations of a TRBAC system are represented by facts of the form *user_role*(*string*, *string*), *role_per*(*string*, *string*) and *role_per*(*string*, *string*), respectively.

A fact of the form $user\_role(u_i, r_j)$ represents that the user u_i is a member of the role r_j. A fact of the form $role\_per(r_i, p_j)$ represents that the permission p_j is assigned to the role r_i . A fact of the form $role\_H(r_i, r_j)$ represents that role r_i is senior to role r_j.

An REB is represented by adding facts corresponding to periodic events as well as role triggers. To express a periodic event, a fact named *periodic_event* is used and it is of the form:

periodic_event([Integer$_i$, Integer$_j$], [[Year], [Month], [Week], [Day], [Hour], [Duration]], role)

Integer$_i$ and *Integer$_j$* represent the *begin* and *end* of the interval component of a periodic event. Variables *Year*, *Month*, *Week*, *Day* and *Hour* represent components of the year, month, week, day and hour calendar of a periodic expression. The variable *Duration* represents the component of the duration calendar of a periodic expression. The variable *role* is used to represent the role that will get enabled through a periodic event (assumption has been made that PEs and RTs are used only for role enabling).

Even though Prolog provides sufficient flexibility to represent every form of periodic expression, for the sake of brevity, we constrain the different values that the variables of a periodic_event fact could take. These constraints are as follows:

- Integer$_i$ and Integer$_j$: 4-digit integers such that Integer$_i \leq$ Integer$_j$
- YEAR: all _
- MONTH: all _ \k, $1 \leq k \leq 12$
- WEEK: all
- DAY: all _ \k, $1 \leq k \leq 7$
- HOUR: all_ \k, $1 \leq k \leq 23$
- DURATION: $1 \leq k \leq 23$

Using the above definition of a periodic event, to represent a periodic event of the form (*[2000, 2014]*, *<all.years + all.months + {1, 2, 3}.days + 10. hours ▷ 8. hours >, Enable r1*), a tuple of the form *periodic_event([2000, 2012], [[all], [all], [_], [1, 2, 3, 4, 5], [10], [8]], r1*) needs to be added to the Prolog program.

A role trigger is represented by a relation named as *trigger*. This relation is of the form trigger($role_{i_1}$, $role_{i_2}$, $role_{i_3}$, $role_{i_4}$), where each $role_{i_k}$, $1 \leq k \leq 4$ could be either a valid role name or an anonymous variable represented as '_'(without quotes). $role_{i_1}$ represents the role present in the event expression of a role trigger, $role_{i_2}$ and $role_{i_3}$ represent the roles present in the role status expression of a role trigger and $role_{i_4}$ represents the role present in the head of

a trigger. For example, a role trigger of the form *Enable* r_1*, Enabled* r_2*, Enabled* $r_3 \rightarrow$ *Enable* r_4 could be represented by adding a fact of the from *trigger*(r_1, r_2, r_3, r_4) and a role trigger of the form *Enable* r_1*, Enabled* $r_2 \rightarrow$ *Enable* r_3 could be represented as *trigger*(r_1, r_2, _, r_3). It may be noted that, for simplicity, we restrict the form a role trigger can take, i.e., the body of the trigger can have at-most one event expression and two role status expressions, the head of a trigger can have at-most one event expression. However, Prolog itself does not impose such restrictions and could be efficiently used to represent more complex forms of role triggers.

3.2 Modeling of AMTRAC in Prolog

This sub-section gives details on modeling of the relations of AMTRAC in Prolog. We divide AMTRAC relations into three categories. The first category consists of those relations that add new elements to the components of a TRBAC system. The second category of relations removes elements from the TRBAC system components and the third category modifies the existing elements of a TRBAC system components. We refer to these categories of relations as additive relations, removal relations and modification relations, respectively.

Modeling of Additive Relations: Under this category of relations, fall *can_assign*, *can_assignp* and *insert_Edge* of ARBAC97 and also the *addRT* and *addPE* relations of REBA.

– Modeling of *can_assign*
To model can_assign, a fact of the form *canassign*(*arole*, *role*, *role*) is used, where *arole* represents an administrative role and *role* denotes a regular role. If a TRBAC system has a *canassign* relation of the form (ar_1, r_1, r_2), then this can be represented in Prolog by adding a fact of the form *canassign*(ar_1, r_1, r_2). Now the facts that an interpreter can derive from a *canassign* fact are given by the rules:

can_assign(A, *R1*, *R2*) :- *canassign*(A, *R1*, *R2*)
can_assign(A, *R1*, *R2*) :- *canassign*(A, *R3*, *R2*), *can_assign*(A, *R1*, *R3*)
assigned_user(U, R) :- *user_role*(U, R)
assigned_user(U, R) :- member_user_through_hierarchy(U, R)
user_assigned(U, R) :- *assigned_user*(U, R)
user_assigned(U, R) :- can_assign(A, *R1*, R), *assigned_user*(U, R)

The first two lines help the interpreter to find the set of *canassign* relations through which a member of *R1* can be assigned to *R2*. The third and the fourth lines define that a user U is a member of role R if either there is a tuple of the form (U, R) in *user_role* or if U gets membership of the role through hierarchy. The fourth and the fifth statements convey that a user U is assigned to role R, either directly through UA or RH, or it may get assigned due to the presence of a *canassign* fact that allows A to assign U to R.

– Modeling of *can_assignp*
To model can_assignp, a fact of the form *canassignp*(*arole*, *role*, *role*) is used, where *arole* represents an administrative role and *role* denotes a regular role. If a TRBAC system has a *canassignp* relation of the form (ar_1, r_1, r_2), then this can be represented in Prolog by adding a fact of the form *canassignp*(ar_1, r_1, r_2). Now the facts that an interpreter can derive from a *canassignp* fact are given by the rules:

can_assignp(*A, R1, R2*) :- *canassignp*(*A*, *R1*, *R2*)
can_assignp(*A, R1, R2*) :- *canassignp*(*A*, *R3*, *R2*), *can_assignp*(*A*, *R1*, *R3*)
assigned_per(*R*, *P*) :- *role_H*(*R*, *P*)
assigned_per(*R*, *P*) :- *member_per_through_hierarchy*(*R*, *P*)
per_assigned(*R*, *P*) :- *assigned_per*(*R*, *P*)
per_assigned(*R*, *P*) :- *can_assignp*(*A*, *R1*, *R*), *assigned_per*(*R*, *P*)

The first two lines help the interpreter to find the set of *canassignp* relations through which permissions of *R1* can be assigned to *R2*. The third and the fourth line define that a permission *P* is associated with a role *R*, if, either there is a tuple of the form (*R, P*) in *role_per* or if *P* is associated with some other role *R3* such that, *R3* is junior to *R*. The fourth and fifth statements convey that a role *R* gets a permission *P*, either directly through PA or RH, or it may get it through the execution of some *canassignp*, which allows *A* to assign *P* to *R*.

– Modeling of *insert_Edge*
RRA97 allows an administrator to insert new edges into the role hierarchy and also to delete existing edges from the hierarchy. To model insertion of edge, the *InsertEdge* relation is used. It is of the form *insertEdge*(*arole*, *role*, *role*), where *arole* represents an administrative role and *role* denotes a regular role. If a TRBAC system has an *insertEdge* relation of the form (ar_1, r_1, r_2), then this can be represented in Prolog by adding a fact of the form *insertEdge*(ar_1, r_1, r_2). Now the facts that an interpreter can derive from a *insertEdge* fact are given by the rules:

direct_senior(*R1*, *R2*):- *role_H*(*R1*, *R2*)
direct_senior(*R1*, *R2*):- *role_H*(*R1*, *R3*), *direct_senior*(*R3*, *R2*)
new_senior(*R1*, *R2*):- *insertEdge*(*R1*, *R2*)
new_senior(R1, R2):- *insertEdge*(*R1*, *R3*), *new_senior*(*R3*, *R2*)
senior(*R1*, *R2*) :- *direct_senior*(*R1*, *R2*)
senior(*R1*, *R2*) :- *new_senior*(*R1*, *R2*)

The first two lines help the interpreter to find the set of roles senior to a role *R2* due to initial role hierarchy. The third and the fourth lines help the interpreter to find the set of roles senior to a role *R* due to the hierarchy introduced by the *insertEdge* relation. The fourth and fifth statements convey that the role *R1* is senior to the role *R2* if either *R2* is senior due to initial hierarchical structure or due to the modified hierarchical structure.

- Modeling of *addPE* (R_{16})
  The relation R_{16} adds a new periodic event to an REB. The fact used to model R_{16} is of the form *addPE*(*perodic_ event*), where *periodic_ event* is a new periodic event such that its format satisfies all the constraints specified in Section 3.1. To add a new periodic event of the form (*[2000, 2014], all.years + 1, 2, 3.days ▷ 2.days, Enable r*), a fact of the form *addPE*([*2000, 2014*], *[[all], [_], [_], [1, 2, 3], [_], [2]], r*) is added to the prolog specification. The new facts that an interpreter can derive due to the presence of an *addPE* fact are given by:

  effective_ periodic_ event(*I, P, R*) :- *periodic_ event*(*I, P, R*)
  effective_ periodic_ event(*I, P, R*) :- *addPE*(*I, P, R*)

  The above statements convey to the interpreter that the set of effective periodic events in a system is the set of periodic events present in the initial state of a TRBAC system along with the set of periodic events added through *addPE*.
- *Modeling of addRT* (R_{17})
  The relation R_{17} adds new role trigger to a system. The fact used to model R_{17} is of the form *addRT*(*trigger*), where *trigger* is a new role trigger such that its format satisfies all the constraints specified in Section 3.1. To add a new role trigger of the form *Enable r, Enabled s → Enabled t*, a fact of the form *addRT*(*r, s, _, t*) is added to the Prolog specification. The new facts that an interpreter can derive due to the presence of an *addRT* fact are given by:

  effective_ trigger(*R1, R2, R3, R4*) :- *trigger*(*R1, R2, R3, R4*)
  effective_ trigger(*R1, R2, R3, R4*). :- *addRT*(*R1, R2, R3, R4*)

  Through these statements, the interpreter is asked to consider the facts written as *trigger* or *addRT* as the set of effective triggers in the system.

Modeling of Removal Relations: Under this category of relations, come *can_ revoke*, *can_ revokep* and *delete_ Edge* of ARBAC97 and also the *removeRT* relation of REBA.

- Modeling of *can_ revoke*, *can_ revokep* and *delete_ Edge* of ARBAC97 and *removeRT* (R_{18}) of REBA. These relations are specified as rules and are respectively of the form:
  can_ revoke(*A, R*) :- *retractall*(*user_ assigned*(*U, R*))
  can_ revokep(*A, R*) :- *retractall*(*per_ assigned*(*R, P*))
  deleteEdge(*A, R1, R2*):- *retarctall*(*role_ H*(*R1, R2*))
  removeRT(*trigger*) :- *retract*(*trigger*)

  The *can_ revoke*(*A, R*) rule asks the interpreter to remove all the assigned users *U* from the role *R*. The *can_ revokep*(*A, R*) asks the interpreter to

remove all the permissions assigned to the role R. A $deleteEdge(A, R1, R2)$ relation asks the interpreter to remove the hierarchy edge between the roles $R1$ and $R2$, and the rule $removeRT(trigger)$ asks the interpreter to remove the role trigger $trigger$ from the REB of a system.

Modeling of Modification Relations of REBA: We finally show modeling of those relations that modify an existing element of the REB. Under this category of relations, come R_1 to R_{15} of REBA. To model these relations, two rules are used: one for modifying the periodic events and the other for modifying the role triggers. Modification in a periodic event can be essentially achieved by first removing the obsolete periodic event and then adding the modified periodic event to the REB. Similar is the case for modification in a role trigger. To modify a periodic event, a rule of the following form is used.

$$modify\_periodic\_event(new\_periodic\_event, old\_periodic\_event) :- \\ retract(old\_periodic\_event), assertz(new\_periodic\_event)$$

Here, $new\_periodic\_event$ is the required new periodic event and the $old\_periodic\_event$ represents the periodic event that will get removed from the REB.

The above definition conveys to the interpreter to remove the old periodic event from the set of facts and to add $new\_periodic\_event$ to the set of facts. Consider a periodic event of the form ($[2000, 2012]$, $<all.years + all.months \triangleright 2.days>$, $Enable\ r$). If an administrator needs to modify the periodic expression to $<all.years + all.months + all.weeks \triangleright 2.days>$, then the $modify\_periodic\_event$ will be of the form:

$modify\_periodic\_event(periodic\_event([2000, 2012], <all.years + all.months \triangleright 2.days>, Enable\ r), periodic\_event([2000, 2012], <all.years + all.months + all.weeks \triangleright 2.days>, Enable\ r)$.

To modify a component of a role trigger, a rule of the following form is used:

$$modify\_trigger(old\_trigger, new\_trigger) :- \\ retract(old\_trigger), assertz(new\_trigger)$$

This definition conveys to the interpreter to remove the $old\_trigger$ from the set of facts and to add $new\_trigger$ into the REB. Consider a trigger of the form $Enable\ r1 \rightarrow Enable\ r2$. Suppose, an administrator wants to modify it to the form $Enable\ r1 \rightarrow Enable\ r3$. To model this requirement, $modify\_trigger$ will be of the form:

$modify\_trigger([r1, \_, \_, r2], [r1, \_, \_, r3])$

4 Analysis of Security Properties

In the previous section, we showed how the different relations of AMTRAC can be modeled using Prolog syntax. In this section, we show how these relations affect the security properties of a TRBAC system. We consider both safety as well as liveness analysis in this paper.

4.1 Safety Analysis

As mentioned in Section 1, a safety property for a TRBAC system could be defined as "*whether a user u gets a permission p at some time instant t.*" In Prolog, to define this property, we use a rule named as *safety*. The *safety* rule can be defined as follows:

$safety(U, P, T)$:- $user\_assigned(U, R)$, $per\_assigned(R, P)$, $enabled\_role(R, T)$

In the above rule, the predicate $user\_assigned(U, R)$ and $per\_assigned(R, P)$ respectively return the set of users and the set of permissions assigned to a role R. The predicate *enabled_role* is used to check whether the role R is enabled at some time instance T or not. A formal definition of this predicate can be written as:

$enabled\_role(R, T)$:- $pe\_enabled\_role(R, T)$; $trigger\_enabled\_role(R, T)$

The above defined predicate returns *true* if a role R is enabled at time T either through a periodic event or due to a role trigger. To check whether there is some periodic event which causes enabling of a role R at time T, the predicate *pe_enabled_role* is used. It is of the form:

pe_enabled_role(R, T) :- valid_periodic_event(X, Y, R),
element_at(IBEGIN, X ,1), element_at(EBEGIN, X ,2),
element_at(DAYLIST, Y ,4), element_at(HOURCALLIST, Y ,5),
element_at(DURCALLIST, Y ,6), element_at(HOURCAL, HOURCALLIST ,1),
element_at(DURCAL, DURCALLIST ,1), element_at(QUERYYEAR, T ,1),
element_at (QUERYDAY, T ,3), element_at(QUERYTIME, T ,4),
Z = HOURCAL + DURCAL, write('Z is '), write(Z), nl,
number_in_range(IBEGIN, QUERYYEAR, EBEGIN)→ (member(QUERYDAY,
DAYLIST)→ (number_in_range(HOURCAL,QUERYTIME, Z) → (write('Role
enabled'));(write('Not enabled'),nl));
(write('query day not in daylist')));(write('query year not in range'))

In the above definition, the predicate *valid_periodic_event* refers to the periodic events initially present in the REB as well as the new periodic events that can be added to the REB through the execution of *addPE* relations.

To check whether a role is enabled through some role trigger or not, the predicate *trigger_enabled_role* is used. It is of the form:

$trigger\_enabled(R, T)$:- $enabled(R)$, $pe\_enabled\_role(R, T)$

The above predicate conveys that a role R is enabled if both the predicates, i.e., $enabled(R)$ and $pe\_enabled\_role(R, T)$ return *true*. The predicate $enabled(R)$ checks whether all the role status expressions and event expressions specified in the role trigger expression of R are satisfied or not. It is of the form:

enabled(R):- valid_trigger(X, Y, Z, R), nl, periodic_event(I, P, X), periodic_event(First_I, First_P, Y), periodic_event(Second_I, Second_P, Z), element_at(Ibegin_ PE_Event ,I, 1), element_at (IEnd_PE_Event,I, 2), element_at(DayListPE_Event,P, 4), element_at(HourCal_PE_Event, P, 5), element_at(DurCal_PE_Event, P, 6), element _at(Ibegin_ First_ Enabled, First_I, 1), element_at(IEnd_First_Enabled,First_I, 2), element_at(Day ListPE_ FirstEnabled,First_P, 4), element_at(HourCal_First_ Enabled, First_P, 5), element_at(DurCal_First_Enabled, First_P, 6), element_at(Ibegin _Second_Enabled, Second_I, 1), element_at(IEnd_Second_Enabled,Second_I, 2), element_at(DayListPE _SecondEnabled,Second_P, 4), element_at(HourCal_Second_En- abled, Second_P, 5), element_at(DurCal_Second_Enabled, Second_P, 6), intersection (DayListPE_Event, DayListPE_FirstEnabled, L), intersection(L, DayListPE_Second- Enabled, W), find_ largest(HourCal_PE_Event, HourCal_First_Enabled, HourCal_ Second_Enabled, Max1, Max2), find_smallest(DurCal_PE_Event, DurCal_First_Enabled, DurCal_Second_ Enabled, Min1, Min2), nl, find_largest(Ibegin_PE_Event, Ibegin_First_Enabled, Ibegin_ Second_Enabled, Max_Ibegin1, Max_Ibegin2), find_smallest(IEnd_PE_Event, IEnd_First_ Enabled, IEnd_Second _Enabled, Min_Iend1, Min_Iend2), assertz(periodic_event ([Max_Ibegin2, Min_Iend2],[[all], [all], [all], W, Max2, Min2], R))

Assume there is a role trigger of the form *Enable* r_1, *Enabled* r_2, *Enabled* r_3 $\rightarrow$ *Enable* r_4, and a query is made whether r_4 is enabled at some time instance t or not. Then, *enabled*(r_4) checks whether both the roles r_2 and r_4 are enabled at time t or not.

After defining the safety rule, we next describe how analysis of a TRBAC system can be done in presence of the different administrative policies. To show the effect, we use the TRBAC system whose user policies are shown in Figure 1 and administrative policies are shown in Figure 2.

To perform safety analysis, we define certain desired safety properties for the system as shown in Table 1. To check whether a condition defined in Table 1 holds or not, individual safety queries for each of them are provided to the interpreter. The set of queries provided to the interpreter is shown in Table 2.

There are certain constraints associated with the values of T that can be used in the safety queries defined in Table 2. For the first query, the value of T should be selected in such a way that it lies in the time duration defined by the periodic expression of any periodic event. For the second query, the value of T can be any time instance between morning *0000* hrs to *1600 hrs* and for the last query, the value of T should be a time instance when, out of the two roles *Manager* and *TeamLeader*, only one is enabled.

users = {Alice, Bob, Charles, John, Tom}
roles = {Manager, Engineer, HR, TeamLeader}
permissions = {Access, Read, Edit}

UA = {(Alice, Manager), (Bob, Engineer), (Charles, Engineer), (John, HR), (Tom, TeamLeader)}
PA = {(Manager, Access), (Engineer, Read), (HR, Edit), (TeamLeader, Access)}
RH = {(Manager ≥ Engineer)}

PE1 : ([2000, 2020], all.years + all.months + all.weeks + {1, 2, 3, 4, 5}.days + 10.hours ▶ 8.hours , Enable Manager)

PE2: ([2000, 2020], all.months + all.weeks + {5, 6}.days + 10.hours ▶ 8.hours, Enable Engineer)

PE3: ([2000, 2012], all.months + all.weeks + {1, 2, 3, 4, 5}.days + 16.hours ▶ 8.hours, Enable TeamLeader)

RT1: Enable Manager, Enabled TeamLeader → Enable HR

Fig. 1. User policies for the example TRBAC system

Table 1. Safety conditions for the example TRBAC system

1. *Tom* should never get *Edit*
2. *Tom* must not get *Access* between 0000 hrs to 1600 hrs
3. *John* should be able to use *Edit* only if both the roles *Manager* and *TeamLeader* are enabled

For all the behavior defined in Table 1, a safety query is posed to the interpreter. If the condition holds, then the interpreter returns *true*; else, returns *false*. For all the defined conditions, the expected answer is *false*, and if the interpreter returns *true* for any of the queries, then the system is said to be in *unsafe* state. To show the effect of the administrative relations, initially, queries are posed in absence of the administrative relations, and then, in the presence of the administrative relations. As mentioned earlier, for the first query, any value of T representing a valid periodic time can be used. So we use the periodic time [*2012*, *1*, *3*, *10*] for executing the first query. Similarly, for the second query, we can use any periodic time not lying in the range *1600* hrs to *0000* hrs. W consider the periodic time [*2010*, *4*, *3*, *13*] as the value of T. For the third query, we use the periodic time [*2010*, *2*, *3*, *13*], since during this periodic time, only the *Manager* role is enabled. It may be noted that, all the periodic times satisfying the constraints specified earlier could also be used as the value of T.

aroles = {CSO, SSO, SO}

can_assign = {(CSO, TeamLeader, Manager), (CSO, Manager, HR)}

modify_role_trigger = (trigger(Enable Manager, Enabled TeamLeader → Enable HR), trigger(Enable Manager → Enable HR))

addRT(Enable Engineer → Enable HR)

Fig. 2. Administrative roles and Administrative relations of AMTRAC which affects safety property of a TRBAC system

Table 2. Safety conditions of the example TRBAC system

1. safety(Tom, Edit, T).
2. safety(Tom, Access, T).
3. safety(John, Edit, T).

When the interpreter is posed with all the three queries, it returns *false* for each of them, signifying that the system is in *safe* state. Next, the same set of queries is posed to the interpreter in the presence of the administrative relations defined in Figure 2. Now, the interpreter returns *true* for all the three queries. Consecutive execution of the *can_assign* relations assigns *Tom* to the roles *Manager* and *HR*, making *Edit* permission available to him. Execution of the administrative relation *addRT* causes enabling of the role *TeamLeader* whenever the role *Engineer* gets enabled. This causes *Tom* to get the *Access* permission between 1000 hrs to 1800 hrs on Saturdays and Sundays. Execution of the *modify_role_trigger* relation causes enabling of the role *HR* whenever the role *Manager* gets enabled, making permission *Edit* available to *John* even when the role *TeamLeader* is not enabled. Thus, it is seen how execution of the different administrative relations can result in undesired transition of the state of a TRBAC system and how a Prolog interpreter can be suitably used to identify such unsafe conditions.

4.2 Liveness Analysis

Liveness analysis checks for the presence of a *dead state*, i.e., it searches for a time instance when none of the roles is enabled. In Prolog, to check for the liveness of a system, we use the predicate *liveness(t)*. It is of the form:

$$liveness(T) \text{ :- } enabled\_role(R,\ T)$$

When a query is made for a certain time instance T, the interpreter tries to find a role R for which $enable\_role(R,\ T)$ returns *true*. If the interpreter is able to find such a role, then it returns *true* conveying that system is not dead at the given time instance. A liveness query for a system gets affected only if

modification is done in the periodic event. Modification in the role triggers will never bring a system to a dead state. This is because a trigger enables a role at t only if some other role gets enabled at t by a periodic event. So, even if removal of the trigger prevents enabling of the triggered role, it cannot prevent enabling of the role which triggers it. So, we consider only those administrative relations which modify the periodic event of a TRBAC system.

Effect of Modifying Periodic Event on Liveness. Modification in any of the components of a periodic event could result in a dead state for the system. Consider the REB shown in Figure 1. If a *modify_periodic_event* relation that modifies the periodic time of *PE1* to {[*2000*, *2015*], <*all.years + all.months + all.weeks + 1, 2, 3, 4, 5.days + 10.hours* ▷ *8.hours*>} is added, then this will bring the system to the dead state from the start of the year *2016*. If a liveness query of the form *liveness*([*2016, 1, 1, 1*]) (the time represented is the 1st hour of the 1st day of the 1st month of 2016), then the system returns *false* for this query, conveying that, the system is in a dead state at the given time. Similarly, it can be shown that if a *modify_periodic_event* relation of the form (*periodic_event([2000, 2020], all.years + all.months + all.weeks + 1, 2, 3, 4, 5.days + 10.hours* ▷ *8.hours, Enable Manager), periodic_event([2000, 2020], all.years + all.months + all.weeks + 1, 2, 3, 4, 5.days + 13.hours* ▷ *8.hours, Enable Manager)*) is executed, then on weekdays, the system will go to a dead state from 1000 hrs to 1300 hrs. Similarly, it can be observed that if a *modify_periodic_event* relation of the form (*periodic_event*(*[2000, 2020], all.years + all.months + all.weeks + 5, 6.days + 10.hours* ▷ *8.hours, Enable Engineer*), *periodic_event*(*[2000, 2020], all.years + all.months + all.weeks + 5, 6.days + 10.hours* ▷ *8.hours, Disable Engineer*)) is executed, then the system will come to a dead state on the sixth day of every week of every month between the years 2000 to 2020.

5 Experimental Results

To study the performance of the proposed modeling methodology, we have implemented a simulator that takes the number of roles, users, permissions, administrative roles, time intervals, periodic events and role triggers as input and generates a TRBAC system satisfying the input. The number of user-role assignments and also the number of permission-role assignments are kept at 20% of the sizes of the respective cartesian products. A uniform distribution is used to determine the user-role assignment and permission-role assignment entries that are included in the relations. A second script is written to translate the output generated by the simulator program into its corresponding Prolog specifications. For implementation, Java(7.0.1-17), on a Windows 7 system with 64-bit i5 processor @ 2.50GHz and 4GB RAM is used. SWI interpreter 6.6.1 is used for analysis.

Effect of the number of roles on the analysis time is shown in Figure 3. The data set used for the analysis consists of 2000 users, 300 permissions, 4 administrative roles, 10 time expressions, 50 periodic events and 30 role triggers. From

the figure, it can be seen that a linear increase in the number of roles causes a close to linear growth in analysis time. This is due to the fact that, a Prolog interpreter works based on the principle of backtracking. A linear increase in the number of roles causes a linear increase in the set of facts having role as one of its parameters.

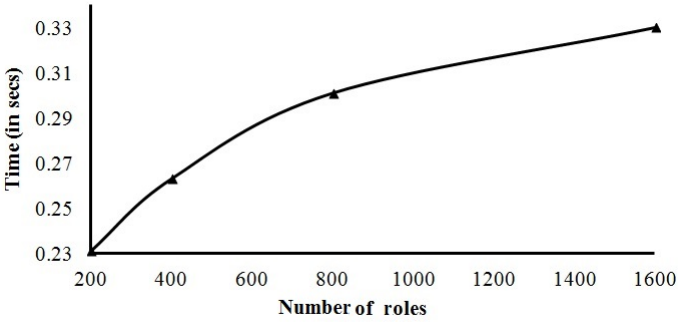

Fig. 3. Effect on analysis time due to variation in number of roles

Figure 4 shows the effect of number of users on the analysis time. The data set used for analysis consists of 300 roles, 300 permissions, 15 time expressions, 50 periodic events, 30 role triggers and 4 administrative roles. It can be observed that, even though the rate of increase in analysis time is linear with increase in the number of users, the rate of growth of time is quite low as compared to the rate of growth of time with number of roles as shown in Figure 3. This is because, when a safety query is specified, the interpreter needs to check only for the user specified in the query, thus making the result of the query independent of other users.

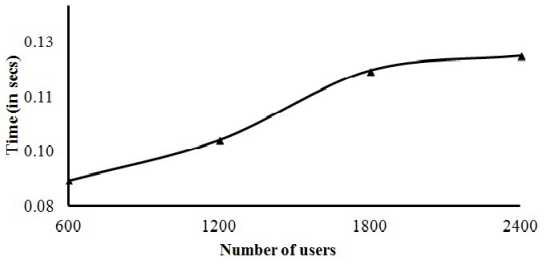

Fig. 4. Effect on analysis time due to variation in number of users

Effect of the number of periodic events and role triggers is shown in Figure 5 and Figure 6, respectively. The rate of growth is quite close to linear for these components of TRBAC as well. However, it can be observed that the analysis time itself is slightly higher as compared to the analysis time needed for the other

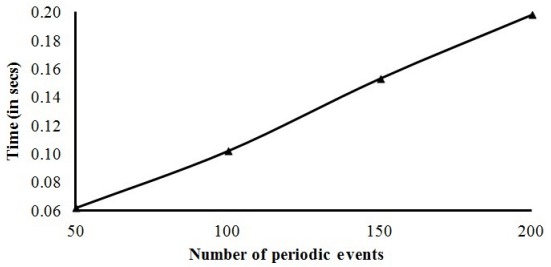

Fig. 5. Effect on analysis time due to variation in number of periodic events

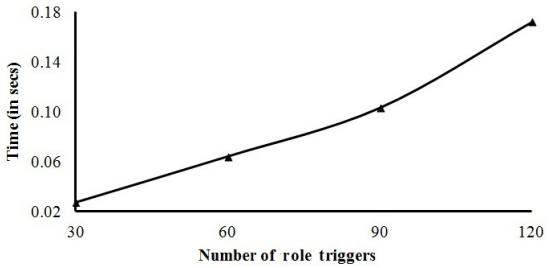

Fig. 6. Effect on analysis time due to variation in number of role triggers

components. This is due to the complex nature of the definitions of periodic events and role triggers.

We also analyzed the effect of each individual administrative relation on the analysis time. It was observed that all the relations have similar effect on the analysis time. This is due to the similarity in format of the relations. The data set used for the analysis consists of 500 users, 200 roles, 300 permissions, 2 administrative roles, 70 periodic events and 20 role triggers. For this data set, it took approximately 0.2 secs to execute a safety query. The combined effect of all the relations of AMTRAC was also studied. For a data set comprising of 1500 users, 300 roles, 300 permissions, 15 time expressions, 120 periodic events, 30 role triggers and 4 administrative roles, the interpreter took 0.232 secs in presence of the AMTRAC relations and 0.141 secs when the AMTRAC relations were not there, to answer a safety query. This is due to the increased number of facts and rules in the presence of the AMTRAC. Even then the total time required is quite encouraging and shows that our modeling and analysis methodology is quite efficient.

6 Related Work

The role-based access control (RBAC) model [14] was proposed to cater to the basic access control needs of commercial organizations. Subsequently, several ex-

tensions have been developed that incorporate context related information into the basic RBAC model. In [1], a model named TRBAC (Temporal RBAC) has been introduced for handling temporal constraint on enabling and disabling of RBAC roles. To put additional temporal restrictions on users getting a permission, GTRBAC (Generalized Temporal Role Based Access Control) model has been proposed in [4]. For considering the user location before giving access to resources, the LRBAC (Location-Aware Role Based Access control) model has been proposed in [10]. Other notable work that incorporate temporal as well as spatial information into RBAC include [2], [11].

The need for decentralization in administration has led to the development of administrative models for various access control models. In [12], ARBAC97 is proposed which includes appropriate relations for modifying user-role assignment (URA97), role-permission assignment (PRA97) and role hierarchy (RRA97). Other administrative models for RBAC are presented in [13] and [9]. While a limited number of administrative rules for making modifications to TRBAC is proposed in [18], very recently, a complete administrative model for TRBAC, named as AMTRAC has been proposed [16] . It includes REBA (Role Enabling Base Assignment), which comprises of the relations used for making changes to the role enabling base (REB) assignment of TRBAC, along with all the relations defined in ARBAC97. REBA has a set of eighteen relations for modifying the various components of REB.

Several attempts have been made to develop methods for verifying the security provided by the policies of RBAC and its variants. In [15], petri-net based modeling for the verification of RBAC policies is proposed. It first represents RBAC using a petri-net based framework and then uses it to verify the correctness of a set of underlying security policies. In [17], formal analysis of STRBAC is done using *Alloy*, which is a formal language based on first-order logic. In [7], security analysis of TRBAC using timed automata is proposed. Roles are represented using a user timed automata, while administrative commands are captured in a controller automata. Security properties are specified using Computation Tree Logic (CTL) and verified with the help of a model checking tool named *Uppaal*. In [8], a method for GTRBAC security analysis is proposed where CTL is used to specify a set of safety and liveness properties, which are then verified using the model checking approach.

Apart from performing simple security analysis of RBAC and its variants, various contributions have been made in the field of security analysis using administrative models. In [6], security analysis of RBAC in the presence of ARBAC97 has been done by reducing the security analysis instance of RBAC into a corresponding security analysis instance of RT$[\twoheadleftarrow, \bigcap]$ [5] and then further reducing the instance so obtained into *Datalog* clauses. In [3], security analysis of user-role assignment of RBAC using URA97 of ARBAC97 is performed and both model checking and logic programming approaches are compared. It has been shown that the logic programming approach outperforms the model checking approach when the number of roles increases significantly. In [18], security analysis of TRBAC is done by using certain administrative rules. However, no formal

administrative model has been considered. While security analysis of RBAC has been done based on the administrative models, use of formal administrative models for TRBAC security analysis is yet to be addressed. This is one of the factors for non-deployment of TRBAC at enterprise level even though it has the ability to support a much richer set of features than the RBAC model.

7 Conclusions and Future Work

In this work, we have introduced a methodology for performing security analysis of TRBAC using Prolog. Initially, the components of a TRBAC system and the relations of AMTRAC are modeled using Prolog syntax. To represent the initial content of the TRBAC components, facts are added and the desired security properties, i.e., safety and liveness are defined in the form of rules. Next, the effect of different components of TRBAC and AMTRAC on the analysis time is studied. It has been shown that a linear increase in the number of any of the TRBAC components causes a linear increase in the analysis time. Although each component asserts linear effect on the analysis time, impact of periodic events and role triggers is the most due to their complex nature.

Further work remains to be done to build an even more comprehensive understanding of the security properties of a TRBAC system. In this work, we have made certain simplifying assumptions for representing the administrative relations. However, in practice, some of these might not be feasible. In the future, we plan to perform analysis in the presence of unconstrained forms of administrative relations. A similar problem exists with representing temporal information. Therefore, we plan to provide a more realistic representation of time. This, in turn, will necessitate the use of alternative tools for analyzing problems with access control specifications.

References

1. Bertino, E., Bonatti, P.A., Ferrari, E.: Trbac: A temporal role-based access control model. ACM Transactions on Information and System Security, 191–233 (2001)
2. Bertino, E., Catania, B., Damiani, M.L., Perlasca, P.: Geo-rbac: A spatially aware rbac. In: Proc. of the 10th ACM Symposium on Access Control Models and Technologies, pp. 29–37. ACM (2005)
3. 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, 242–255 (2008)
4. Joshi, J.B., Bertino, E., Latif, U., Ghafoor, A.: A generalized temporal role-based access control model. IEEE Transactions on Knowledge and Data Engineering, 4–23 (2005)
5. Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust management framework. In: Proc. of the IEEE Symposium on Security and Privacy, pp. 114–130. IEEE (2002)
6. Li, N., Tripunitara, M.V.: Security analysis in role-based access control. ACM Transactions on Information and System Security, 391–420 (2006)

7. Mondal, S., Sural, S.: Security analysis of temporal-rbac using timed automata. In: Proc. of the 4th International Conference on Information Assurance and Security, pp. 37–40. IEEE (2008)
8. Mondal, S., Sural, S., Atluri, V.: Towards formal security analysis of gtrbac using timed automata. In: Symposium on Access Control Models and Technologies, pp. 33–42. ACM (2009)
9. Oh, S., Sandhu, R.: A model for role administration using organization structure. In: Proc. of the 7th ACM Symposium on Access Control Models and Technologies, pp. 155–162. ACM (2002)
10. Ray, I., Kumar, M., Yu, L.: LRBAC: A location-aware role-based access control model. In: Bagchi, A., Atluri, V. (eds.) ICISS 2006. LNCS, vol. 4332, pp. 147–161. Springer, Heidelberg (2006)
11. Ray, I., Toahchoodee, M.: A spatio-temporal role-based access control model. In: Barker, S., Ahn, G.-J. (eds.) Data and Applications Security 2007. LNCS, vol. 4602, pp. 211–226. Springer, Heidelberg (2007)
12. Sandhu, R., Bhamidipati, V., Munawer, Q.: The arbac97 model for role-based administration of roles. ACM Transactions on Information and System Security, 105–135 (1999)
13. Sandhu, R., Munawer, Q.: The arbac99 model for administration of roles. In: Proc. of the 15th Annual Conference on Computer Security Applications, pp. 229–238. IEEE (1999)
14. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer, 38–47 (1996)
15. Shafiq, B., Masood, A., Joshi, J., Ghafoor, A.: A role-based access control policy verification framework for real-time systems. In: 10th International Workshop Object-Oriented Real-Time Dependable Systems, pp. 13–20. IEEE (2005)
16. Sharma, M., Sural, S., Vaidya, J., Atluri, V.: Amtrac: An administrative model for temporal role-based access control. Computers & Security (2013)
17. Toahchoodee, M., Ray, I.: Using alloy to analyze a spatio-temporal access control model supporting delegation. IET Information Security, 75–113 (2009)
18. Uzun, E., Atluri, V., Sural, S., Vaidya, J., Parlato, G., Ferrara, A.L., Parthasarathy, M.: Analyzing temporal role-based access control models. In: Proc. of the 17th ACM Symposium on Access Control Models and Technologies, pp. 177–186. ACM (2012)