TRBAC: A Temporal Role-Based Access Control Model

ELISA BERTINO and PIERO ANDREA BONATTI
University of Milano, Italy
and
ELENA FERRARI
University of Insubria, Como, Italy

Role-based access control (RBAC) models are receiving increasing attention as a generalized approach to access control. Roles may be available to users at certain time periods, and unavailable at others. Moreover, there can be temporal dependencies among roles. To tackle such dynamic aspects, we introduce Temporal-RBAC (TRBAC), an extension of the RBAC model. TRBAC supports periodic role enabling and disabling—possibly with individual exceptions for particular users—and temporal dependencies among such actions, expressed by means of role triggers. Role trigger actions may be either immediately executed, or deferred by an explicitly specified amount of time. Enabling and disabling actions may be given a priority, which is used to solve conflicting actions. A formal semantics for the specification language is provided, and a polynomial safeness check is introduced to reject ambiguous or inconsistent specifications. Finally, a system implementing TRBAC on top of a conventional DBMS is presented.

Categories and Subject Descriptors: D.4.6 [**Security and Protection**]: Access controls; H.2.7 [**Database Administration**]: Security, integrity, and protection

General Terms: Security

Additional Key Words and Phrases: Role triggers, role-based access control, temporal constraints

1. INTRODUCTION

Role-based access control (RBAC) models are receiving increasing attention as a generalized approach to access control [Atluri 1999; Osborn 2000; Sandhu 1995, 1997, 1998a]. In an RBAC model, roles represent functions within a given organization. Authorizations are then granted to roles, rather than single users. The authorizations granted to a role are strictly related to the data objects and resources that are needed for exercising the functions associated with the

Authors' addresses: E. Bertino, Dipartimento di Scienze dell'Informazione, University of Milano, Via Comelico 39/41, 20135 Milano, Italy; email: bertino@dsi.unimi.it; P. A. Bonatti, Dipartimento di Tecnologie dell'Informazione, University of Milano, Via Bramante 65, 26013 Crema, Italy; email: bonatti@dti.unimi.it; E. Ferrari, Dipartimento di Scienze Chimiche, Fisiche e Matematiche, University of Insubria, Via Valleggio, 11, 22100 Como, Italy; email: elena.ferrari@uninsubria.it.

role. Users are thus simply authorized to "play" the appropriate roles, thereby acquiring the roles' authorizations. When users log in, they can activate a subset of the roles they are authorized to play. We call a role that a user can activate during a session, an *enabled role*.

Roles have several well-recognized advantages. Since roles represent organizational functions, a role-based model can directly support an organization's security policy. Authorization administration is also greatly simplified. If a user moves to a new function within the organization, there is no need to revoke the authorizations he or she had in the previous function and grant the authorizations needed for the new function. The Security Officer (SO) simply needs to revoke and grant the appropriate role membership. Last, but not least, RBAC models have been shown to be policy-neutral [Sandhu 1996; Osborn et al. 2000]; in particular, by appropriately configuring a role system, one can support different policies, including mandatory and discretionary policies. Such flexibility is extremely important in supporting the organization security policies.

Because of its relevance, RBAC has been widely investigated [Atluri 1999; Osborn 2000; Sandhu 1995, 1997, 1998a]. However, even though RBAC has reached a good maturity level, there are still significant application requirements not addressed by current RBAC models. One such requirement is related to the roles' temporal dimension. In many organizations, functions may have limited or periodic temporal duration. Consider, for instance, the case of part-time staff in a company, and assume that part-time staff is authorized to work within the given organization only on working days, between 9 AM and 1 PM. If part-time staff is represented by a role, then the above requirement entails that this role should be enabled only during the aforementioned temporal intervals. Similar requirements can be supported by specifying—for each role—the time periods in which they can be activated. We call *role enabling* the transition of a role from the nonenabled status to the enabled status, and *role disabling* the transition of a role from the enabled status to the nonenabled status. In a framework where roles are not always enabled, it is important to introduce ways to specify dependencies among the enabling and disabling of different roles. Suppose, for example, that the doctor-on-night-duty role is enabled during the night. Since doctors may need the assistance of a nurse, one should make sure that the corresponding role—say, nurse-on-night-duty—is enabled whenever doctor-on-night-duty is.

To cope with these requirements, we propose Temporal-RBAC (TRBAC), an extension of RBAC models that supports temporal constraints on the enabling/disabling of roles. TRBAC supports periodic role enabling and disabling, and temporal dependencies among such actions. Such dependencies expressed by means of *role triggers* (active rules that are automatically executed when the specified actions occur) can also be used to constrain the set of roles that a particular user can activate at a given time instant. The firing of a trigger may cause a role to be enabled/disabled either immediately, or after an explicitly specified amount of time. Enabling/disabling actions may be given a priority that may help in solving conflicts, such as the simultaneous enabling and disabling of a role. As expected, the action with the highest priority is executed.

To make the SO able to react to emergency situations, we allow him or her to dynamically change the status of a role and the set of users entitled to activate that particular role by issuing *run-time* requests, that is, requests that are not conditioned to the occurrence of other events and/or the verification of some conditions. For instance, by a run-time request it is possible to temporarily prevent a user from activating a role. This feature can be useful when a user under a particular role performs an action that could be dangerous for the system. The SO can immediately react by issuing a run-time request causing a temporary denial for that user to activate that particular role. Run-time requests—like triggers—may be executed immediately, or after a specified temporal delay.

To the best of our knowledge, the role-based access control model we are presenting is the first one proposing features such as role triggers and periodic enabling/disabling of roles.

Unfortunately, because of the expressive power provided by TRBAC, some specifications may be ambiguous; that is, they may lead to states where there is no unique way of deciding which roles are enabled. In this article, we introduce a notion of *safeness*, and prove that it guarantees the absence of ambiguities and inconsistencies in the specification.[1] Moreover, we define a polynomial algorithm for testing the safety of the specifications.

An additional contribution of this article is the development of a system supporting TRBAC. The system directly supports role triggers and run-time requests on top of a conventional RDBMS. In particular, role triggers are supported by translating them into DBMS triggers.

The remainder of the article is organized as follows. Section 2 describes the role-based access control model on which our work relies and the formalism we use to represent periodic time. Section 3 formally presents TRBAC. Section 4 introduces the notion of *safe* specification and gives a mechanism to check whether a specification is safe. Section 5 extends the specification language of TRBAC to support individual exceptions on role enabling and disabling. Section 6 presents a system implementing TRBAC on top of a commercial DBMS, whereas Section 7 surveys related work. Section 8 concludes the article and outlines future research directions. Finally, formal proofs are reported in Appendix A.

2. PRELIMINARIES

In this section we first present the RBAC model we refer to in the article. We then describe the formalism we use to represent periodic time.

2.1 RBAC Model

The RBAC model we use throughout the article is basically the one proposed by Sandhu et al. [1996; Sandhu 1998b]. The model consists of four basic compo-

[1]The reader is warned that “safeness,” in this article, refers to rule bases whose behavior is unambiguous. In other security contexts, the word “safe” would refer to system states with the property that no security violation occurred.

nents: a set of users `Users`, a set of roles `Roles`, a set of permissions `Permissions`, and a set of sessions `Sessions`. A user is a human being or an autonomous agent, a role is a collection of permissions needed to perform a certain job function within an organization, a permission is an access mode that can be exercised on objects in the system, and a session relates a user to possibly many roles. When a user logs in the system he establishes a session and, during this session, he can request activation of some of the roles he is authorized to play. An activation request is granted only if the corresponding role is enabled at the time of the request and the user requesting the activation is entitled to activate the role at the time of the request. If an activation request is satisfied, the user issuing the request obtains all the permissions associated with the role he has requested to activate. On the sets `Users`, `Roles`, `Permissions`, and `Sessions`, several functions are defined. The *user assignment* (*UA*) and the *permission assignment* (*PA*) functions model the assignment of users to roles and the assignment of permissions to roles, respectively. A user can be authorized to play many roles, and many users can be authorized to play the same role. Moreover, a role can have many permissions, and the same permissions can be associated with many roles. The *user* function maps each session to a single user, whereas the function *role* establishes a mapping between a session and a set of roles (i.e., the roles that are activated by the corresponding user in that session). On `Roles`, a hierarchy is defined, denoted by $\geq$. If $r_i \geq r_j$, $r_i, r_j \in \texttt{Roles}$, then role r_i inherits the permissions of role r_j.

The following definition (adapted from Sandhu [1998]) formally defines the RBAC model on which TRBAC is based.

Definition 2.1 (*RBAC Model*). The RBAC model consists of the following components.

—Sets `Users`, `Roles`, `Permissions`, and `Sessions`, representing the set of users, roles, permissions, and sessions, respectively;

—*PA*: $\texttt{Roles} \rightarrow \texttt{Permissions}$ the permission assignment function, that assigns to roles the permissions needed to complete their jobs;

—*UA*: $\texttt{Users} \rightarrow \texttt{Roles}$ the user assignment function, that assigns users to roles;

—user: $\texttt{Sessions} \rightarrow \texttt{Users}$, that assigns each session to a single user;

—role: $\texttt{Sessions} \rightarrow 2^{\text{Roles}}$, that assigns each session to a set of roles; and

—*RH* $\subseteq \texttt{Roles} \times \texttt{Roles}$, a partially ordered role hierarchy (written $\geq$).

2.2 Periodic Expressions

Periodic time is represented by means of a symbolic user-friendly formalism [Bertino et al. 1998], as a pair $\langle$`[begin,end]`, `P`$\rangle$, where `P` is a *periodic expression* denoting an infinite set of periodic time instants, and `[begin,end]` is a time interval denoting the lower and upper bounds that are imposed on instants in `P`.

The formalism for periodic expressions is based on the one proposed in Niezette and Stevenne [1992], and relies on the notion of *calendars*. A calendar is defined as a countable set of contiguous intervals,[2] numbered by integers called *indexes* of the intervals.

A subcalendar relationship can be established between calendars. Given two calendars C_1 and C_2, we say that C_1 is a subcalendar of C_2 (written $\mathrm{C}_1 \sqsubseteq \mathrm{C}_2$), if each interval of C_2 is exactly covered by a finite number of intervals of C_1, that is, if for each interval I of C_2 there exists a subset of the intervals of C_1 such that I is included in the interval resulting from their union. New calendars can be dynamically generated from the existing ones, by means of a function *generate*() (cf. Bertino et al. [1998] for a formal definition), a *reference time instant*, and a *basic calendar* (the tick of the system), denoted by τ. In the following, we assume the existence of a set of calendars containing the calendars *Hours*, *Days*, *Weeks*, *Months*, and *Years*, where *Hours* is the calendar with the finest granularity (i.e., the basic calendar).

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

Definition 2.2 (*Periodic Expression*). [Bertino et al. 1998] Given calendars $C_d, C_1, \ldots, C_n$, a *periodic expression* P is defined as

$$\mathtt{P} = \sum_{i=1}^{n} O_i \cdot C_i \rhd r \cdot C_d,$$

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

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

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

Definition 2.3 (*Function* $\Pi()$). [Bertino et al. 1998] Let $\mathtt{P} = \sum_{i=1}^{n} O_i \cdot C_i \rhd r \cdot C_d$ be a periodic expression; then $\Pi(\mathtt{P})$ is a set of time intervals whose common duration is $r \cdot C_d$, and whose set S of starting points is computed as follows.

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

For simplicity, in this article the bounds `begin` and `end` constraining a periodic expression are denoted by a pair of *date expressions* of the form `mm/dd/yyyy:hh`,

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

with the obvious intended meaning; end can also be ∞. For instance, [1/1/2000, 12/31/2000] denotes all the instants in 2000. The set of time instants denoted by $\langle$[begin,end], P$\rangle$ is defined through function *Sol*(), formally defined as follows.

Definition 2.4 (*Function Sol*()). Let t be a time instant, P a periodic expression, and begin and end two date expressions. $t \in Sol(\langle[\texttt{begin}, \texttt{end}], \texttt{P}\rangle)$ if and only if there exists $\tau \in \Pi(\texttt{P})$ such that $t \in \tau$ and $\texttt{t}_b \leq t \leq \texttt{t}_e$, where $\texttt{t}_b$ and $\texttt{t}_e$ are the instants denoted by begin and end, respectively.

3. TRBAC

In this section we present TRBAC, by introducing first its syntax and then its semantics.

3.1 Syntax

Let (Prios, $\preceq$) be a totally ordered set of priorities. We assume that Prios contains at least two distinct members $\top$ and $\bot$ such that, for all $x \in \texttt{Prios}$, $\bot \preceq x \preceq \top$. As usual, we write $x \prec y$ if $x \preceq y$ and $x \neq y$.

Roles and Prios induce the following classes of expressions.

Definition 3.1 (*Event Expressions, Role Status Expressions*).

1. (Simple) event expressions have the form enable R or disable R, where $R \in$ Roles.
2. Prioritized event expressions have the form $p{:}E$, where $p \in \texttt{Prios}$ and E is an event expression.
3. Role status expressions have the form enabled R or $\neg$enabled R, where $R \in$ Roles.

We next introduce the notion of conflicting events which plays a crucial role in defining the semantics of TRBAC.

Definition 3.2 (*Conflicting Events*). We say that two event expressions enable R and disable R' are conflicting if $R = R'$; in symbols, we write:

$$\texttt{conf(enable}\ R) \overset{\text{def}}{=} \texttt{disable}\ R\,,$$

$$\texttt{conf(disable}\ R) \overset{\text{def}}{=} \texttt{enable}\ R\,.$$

Event expressions and role status expressions are the basic building blocks of the *Role Enabling Base*, which contains temporal constraints on the enabling of roles. A Role Enabling Base is formally defined as follows.

Definition 3.3 (*Role Enabling Base, Periodic Events, Role Triggers*). A Role Enabling Base (REB) is a set of elements of the following kinds.

1. Periodic events of the form $(I, P, p{:}E)$, where
   (a) I is a time interval;
   (b) P is a periodic expression;
   (c) $p{:}E$ is a prioritized event expression with $p \prec \top$.

($\mathtt{PE}_1$) . ([1/1/2000,∞], Night-time, VH:enable doctor-on-night-duty)
($\mathtt{PE}_2$) . ([1/1/2000,∞], Day-time,VH:disable doctor-on-night-duty)
($\mathtt{PE}_3$) . ([1/1/2000,∞], Day-time,VH:enable doctor-on-day-duty)
($\mathtt{PE}_4$) . ([1/1/2000,∞], Night-time,VH:disable doctor-on-day-duty)
($\mathtt{RT}_1$) . enable doctor-on-night-duty $\rightarrow$ H:enable nurse-on-night-duty
($\mathtt{RT}_2$) . disable doctor-on-night-duty $\rightarrow$ H:disable nurse-on-night-duty
($\mathtt{RT}_3$) . enable doctor-on-day-duty $\rightarrow$ H:enable nurse-on-day-duty
($\mathtt{RT}_4$) . disable doctor-on-day-duty $\rightarrow$ H:disable nurse-on-day-duty
($\mathtt{RT}_5$) . enable nurse-on-day-duty $\rightarrow$ H:enable nurse-on-training after 2
($\mathtt{RT}_6$) . disable nurse-on-day-duty $\rightarrow$ VH:disable nurse-on-training

Fig. 1. An example of Role Enabling Base.

2. Role triggers of the form:

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

where the E_is are simple event expressions, the C_is are role status expressions, $p{:}E$ is a prioritized event expression with $p \prec \top$, and Δt is a duration expression.

Priorities and delay expressions (`after` Δt) may be omitted. In that case, by default, $p = \bot$ and $\Delta t = 0$.

Example 3.1 An example of REB for a medical domain is illustrated in Figure 1. In the figure we use intuitive names for periodic expressions, whereas `VH` (Very High) and `H` (High) denote priorities with $\mathtt{H} \prec \mathtt{VH}$. The periodic events and role triggers in the REB state that the `doctor-on-night-duty` role must be enabled during the night (such constraint is imposed by periodic events $\mathtt{PE}_1$ and $\mathtt{PE}_2$), whereas the role `doctor-on-day-duty` must be enabled during the day (periodic events $\mathtt{PE}_3$ and $\mathtt{PE}_4$). Moreover, role triggers $\mathtt{RT}_1$ and $\mathtt{RT}_2$ state that the role `nurse-on-night-duty` must be enabled whenever the role `doctor-on-night-duty` is. Role triggers $\mathtt{RT}_3$ and $\mathtt{RT}_4$ impose the same constraint for `doctor-on-day-duty` and `nurse-on-day-duty`, respectively. Finally, role triggers $\mathtt{RT}_5$ and $\mathtt{RT}_6$ specify that the role `nurse-on-training` must be enabled only during the daytime when the role `nurse-on-day-duty` is enabled. Moreover, role `nurse-on-training` must be enabled two hours after role `nurse-on-day-duty` is enabled (for instance, because within the first two hours nurses must perform urgent activities and they cannot take care of nurses on training).

SOs may enable and disable roles dynamically, at run-time, by means of the following expressions.

Definition 3.4 (*Run-time Request Expression*). A run-time request expression has the form

$$p{:}E \ \mathtt{after}\ \Delta t,$$

where $p{:}E$ is a prioritized event expression, and Δt is a duration expression. Priorities and delay expressions (`after` Δt) may be omitted. In that case, by default, $p = \top$ and $\Delta t = 0$.

Note that—unlike REB events—run-time requests are given top priority by default. Note also that the priority of periodic events and triggers must be strictly smaller than $\top$, so that the SO can always override at run-time the actions generated by any periodic event or trigger.

Example 3.2. Consider the REB in Figure 1. The following are examples of run-time requests.

—`disable nurse-on-training`

—`enable emergency-doctor`.

The first request has the effect of disabling role `nurse-on-training`, whereas the second is a request to enable role `emergency-doctor`.

Roughly speaking, in the formalization, the sequence of the SO run-time requests is modeled as a stream RQ of run-time request expressions. Each set $RQ(t)$ in the sequence models the requests submitted at time t. Time points are expressed as integers, starting from 0 (time expressions are converted to integer notation, based on the finest-grained calendar adopted).

Definition 3.5 (*Request Stream*). A request stream is a sequence $RQ = \langle RQ(0), RQ(1), \ldots, RQ(t), \ldots \rangle$, where each $RQ(t)$ is a (possibly empty) set of run-time request expressions.

3.2 Semantics

In order to simplify the main definitions, we first introduce an auxiliary notion to identify the prioritized events with maximal priority, and those that are overridden (or *blocked*), according to the ordering $\preceq$ and the disabling-takes-precedence principle. Such a principle establishes that, when conflicts among events (i.e., enabling/disabling of roles) cannot be solved by priorities, role disabling is considered as prevailing with respect to role enabling. Such a principle is an extension to the RBAC framework of the well-known denial-takes-precedence principle used by the majority of access control models supporting both positive and negative authorizations [Bertino 1998]. Under this principle, negative authorizations (representing denials) always have higher priority than positive authorizations (representing permissions). The denial-takes-precedence principle is widely adopted since it represents the most conservative approach with respect to security.

Definition 3.6 (*Blocked Event*, `Nonblocked`). Let S be a set of prioritized event expressions. We say that $p{:}E \in S$ *is* blocked by S if there exists $q \in \texttt{Prios}$ such that $(q{:}\texttt{conf}(E)) \in S$ and either

1. $E = \texttt{enable}\ R$ and $p \preceq q$, or
2. $E = \texttt{disable}\ R$ and $p \prec q$.

The set of all members of S that are not blocked by S is denoted by `Nonblocked(S)`.

Example 3.3 Let $S = \{$`H:enable` R_0,`H:disable` R_0,`VH:enable` R_1, `H:disable` $R_1\}$. Thus, `Nonblocked`$(S) = \{$`H:disable` R_0, `VH:enable` $R_1\}$, since `H:enable` R_0 is blocked by `H:disable` R_0, by the first condition of Definition 3.6 (which specifies the disabling-takes-precedence principle), whereas `H:disable` R_1 is blocked by `VH:enable` R_1, by the second condition of Definition 3.6.

The dynamics of event occurrences and role enabling and disabling is depicted as a sequence of snapshots. Each snapshot models the current set of prioritized events and the status of each role. For notational convenience, events and role status are modeled by two distinct sequences *EV* and *ST*, respectively.

Definition 3.7 (*System Trace, Canonical Trace*). A system trace—or simply a trace—consists of a pair of infinite sequences *EV* and *ST*, such that for all integers $t \geq 0$:

—the tth element of *EV*, denoted by $EV(t)$, is a set of prioritized event expressions; intuitively, this is the set of events that occur at time t;
—the tth element of *ST*, denoted by $ST(t)$, is a set of role names; intuitively, these are the enabled roles at time t.

Furthermore, traces should satisfy the following constraint, for all $t \geq 0$.

$$\begin{aligned} ST(t+1) = (&ST(t) \cup \\ &\{R \mid \texttt{enable}\ R \in \texttt{Nonblocked(EV(t))}\}) \setminus \\ &\{R \mid \texttt{disable}\ R \in \texttt{Nonblocked(EV(t))}\}. \end{aligned}$$

Finally, we say that a trace is canonical if $ST(0) = \emptyset$.

Essentially, the above constraint enforces the intended semantics of events. Those with maximal priority (i.e., the members of `Nonblocked(EV(t))`) determine role enabling and disabling as expected. Note that the above constraint determines a unique state, given the previous state and an event sequence, so the following proposition holds.

PROPOSITION 3.1 *For all event sequences EV and all initial role status* S_0, *there exists a unique trace* $\langle EV, ST \rangle$ *with* $ST(0) = S_0$.

We are left to specify which events must be in *EV*, given a REB $\mathcal{R}$ and a request stream $\boldsymbol{RQ}$. Intuitively, each event should be *caused* by some element of $\mathcal{R}$ or $\boldsymbol{RQ}$. When a prioritized event is caused by a trigger, the event expressions in the body of the trigger must not be blocked. These intuitions are formalized by the next definition.

Definition 3.8 (*Caused Events*). The set of *caused prioritized events* at time t (with respect to given trace $\langle EV, ST \rangle$, REB $\mathcal{R}$, and request stream $\boldsymbol{RQ}$) is the least set `Caused`$(t, EV, ST, \mathcal{R}, \boldsymbol{RQ})$ satisfying the following conditions.

1. If $(I, P, p{:}E) \in \mathcal{R}$ and $t \in Sol(I, P)$, then $p{:}E \in$ `Caused`$(t, EV, ST, \mathcal{R}, \boldsymbol{RQ})$.
2. If $(p{:}E\ \texttt{after}\ \Delta t) \in \boldsymbol{RQ}(t - \Delta t)$ $(\Delta t \leq t)$, then $p{:}E \in$ `Caused`$(t, EV, ST, \mathcal{R}, \boldsymbol{RQ})$.
3. If $[E_1, \ldots, E_n, C_1, \ldots, C_k \rightarrow p{:}E\ \texttt{after}\ \Delta t] \in \mathcal{R}$ and

3a. $\Delta t \leq t$,
3b. for all role status expressions $C_i = \texttt{enabled}\, R\ (1 \leq i \leq k), R \in ST(t-\Delta t)$,
3c. for all role status expressions $C_i = \neg\texttt{enabled}\, R\ (1 \leq i \leq k)$, $R \notin ST(t - \Delta t)$, and
3d. for all event expressions $E_i\ (1 \leq i \leq n)$, there exists $p{:}E_i \in \texttt{Caused}(t - \Delta t, EV, ST, \mathcal{R}, RQ)$ not blocked by $EV(t - \Delta t)$,

then $p{:}E \in \texttt{Caused}(t, EV, ST, \mathcal{R}, RQ)$.

In other words, by (1) and (2), all events scheduled via a periodic event or an explicit request are caused. By (3), all events scheduled by a trigger are caused, provided that the role status expressions in the body are satisfied (3(b) and 3(c)) and that the event expressions $E_i s$ are in turn caused at time $t - \Delta t$. Moreover, such events must not be blocked by any concurrent event (3(d)).

Example 3.4 Let $\mathcal{R}$ consist of the following role triggers.

1. $\texttt{enable}\, R_0 \rightarrow \texttt{enable}\, R_1$
2. $\texttt{enable}\, R_0 \rightarrow \texttt{disable}\, R_2$
3. $\texttt{enable}\, R_1 \rightarrow \texttt{enable}\, R_2$
4. $\texttt{enable}\, R_2 \rightarrow \texttt{enable}\, R_3$.

Suppose that $RQ(0) = \{\texttt{enable}\, R_0\ \texttt{after}\ 1\}$, and that for all $t > 0$, $RQ(t) = \emptyset$. Finally, let $\langle EV, ST \rangle$ be the (unique) canonical trace such that:

$$\begin{aligned} EV(0) &= \{\perp{:}\texttt{enable}\, R_0\} \\ EV(1) &= \{\perp{:}\texttt{enable}\, R_0, \perp{:}\texttt{enable}\, R_1, \\ &\quad \perp{:}\texttt{enable}\, R_2, \perp{:}\texttt{disable}\, R_2, \\ &\quad \perp{:}\texttt{enable}\, R_3\} \\ EV(t) &= \emptyset \quad (t > 1). \end{aligned}$$

Then, we have:

— $\texttt{Caused}(0, EV, ST, \mathcal{R}, RQ) = \emptyset$. Indeed, at $t = 0$ we have no periodic event nor any immediate request, and in turn no trigger body is caused.

— $\texttt{Caused}(1, EV, ST, \mathcal{R}, RQ) = EV(1) \setminus \{\perp{:}\texttt{enable}\, R_3\}$. The reason is that $\perp{:}\texttt{enable}\ R_0$ is caused at $t = 1$ by the request in $RQ(0)$. This event causes the preconditions of triggers (1) and (2). In turn, (1) causes the precondition of (3), which causes the precondition of (4). However, this latter precondition is blocked by the event $\perp{:}\texttt{disable}\, R_2$ in $EV(1)$. Summarizing, all triggers but (4) provide a cause for their head.

— For all $t > 1$, $\texttt{Caused}(t, EV, ST, \mathcal{R}, RQ) = \emptyset$. Same explanation as for $t = 0$.

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

Definition 3.9 (*Execution Model*). A trace $\langle EV, ST \rangle$ is an execution model of a REB $\mathcal{R}$ and a request stream RQ, if for all $t \geq 0$,

$$EV(t) = \texttt{Caused}(t, EV, ST, \mathcal{R}, RQ).$$

We say that an execution model is canonical if $ST(0) = \emptyset$.

Example 3.5. Given $\mathcal{R}$ and RQ as specified in Example 3.4, it is easy to see that the unique canonical trace with:

$$\begin{aligned} EV(0) &= \emptyset \\ EV(1) &= \{\bot\text{:}\texttt{enable}\ R_0, \bot\text{:}\texttt{enable}\ R_1, \\ &\quad\ \bot\text{:}\texttt{enable}\ R_2, \bot\text{:}\texttt{disable}\ R_2\} \\ EV(t) &= \emptyset \quad (t > 1) \end{aligned}$$

is an execution model of $\mathcal{R}$ and RQ. The corresponding role states are $ST(0) = ST(1) = \emptyset$ and for all $t > 1$, $ST(t) = \{R_0, R_1\}$.

Unfortunately, some specifications may yield no execution model, while some ambiguous specifications may admit two or more models, as the following example shows.

Example 3.6. Let $\mathcal{R} = \{\texttt{enable}\ R \rightarrow \texttt{disable}\ R\}$. Upon a request $RQ(t) = \{\bot\text{:}\texttt{enable}\ R\}$, there are the possibilities:

1. The trigger in $\mathcal{R}$ fires. But then its conclusion would immediately block the precondition `enable` R, so the trigger should not fire.
2. The trigger does not fire. But then its precondition, `enable` R, is not blocked, so the trigger should fire.

In other words, it is impossible to satisfy both the requirement that triggers should not fire if their body is blocked by a simultaneous event, and the requirement that all applicable triggers must fire. Formally, this is captured by the fact: $\bot\text{:}\texttt{disable}\ R \in EV(t)$ if and only if $\bot\text{:}\texttt{disable}\ R \notin \texttt{Caused}(t, EV, ST, \mathcal{R}, RQ)$. Clearly, this implies that the equation in Definition 3.9 cannot be satisfied, and hence there is no execution model of $\mathcal{R}$ and RQ (equivalently, $\mathcal{R}$ and RQ are mutually inconsistent).

Example 3.7. Let $\mathcal{R} = \{\texttt{enable}\ R \rightarrow \texttt{disable}\ S,\ \ \texttt{enable}\ S \rightarrow \texttt{disable}\ R\}$. Upon a combined request $RQ(t) = \{\bot\text{:}\texttt{enable}\ R,\ \bot\text{:}\texttt{enable}\ S\}$, there exist the symmetric possibilities:

1. Fire the first trigger; this yields: $EV(t) = \{\bot\text{:}\texttt{enable}\ R, \bot\text{:}\texttt{enable}\ S, \bot\text{:}\texttt{disable}\ S\}$. In this case `enable` R is not blocked, while `enable` S is. Intuitively, the first trigger blocks the second one.
2. Fire the second trigger; this yields: $EV(t) = \{\bot\text{:}\texttt{enable}\ R,\ \bot\text{:}\texttt{disable}\ R,\ \bot\text{:}\texttt{enable}\ S\}$. In this case, `enable` R is blocked, while `enable` S is not. Intuitively, the second trigger blocks the first one.

The third possibility—that is, firing both triggers in parallel—is not considered in this example, because it violates a principle enforced by our semantics; namely, the effects of fired triggers should not invalidate (block) any of the fired triggers' bodies. In this example, simultaneous firing would invalidate the bodies of both triggers. This does not mean that our execution model is incompatible with parallel trigger execution. Any parallel execution model satisfying the above principle can be adopted.

$\langle$H:enable nurse-day$, +,$ H:enable nurse-training$\rangle$
$\langle$H:disable nurse-day$, +,$ VH:disable nurse-training$\rangle$
$\langle$H:disable nurse-day$, -,$ H:enable nurse-training$\rangle$
$\langle$H:enable nurse-day$, -,$ VH:disable nurse-training$\rangle$

Fig. 2. Edges of the dependency graph of the REB in Figure 1.

Fortunately, there are many interesting cases in which the specifications yield exactly one model, for all possible run-time requests. There are simple syntactic conditions that prevent any pathological interplay between conflicting events. Such syntactic conditions, called *safeness*, are introduced in the next section.

4. SAFE REBS

In this section we introduce a syntactic condition, called *safety*, that can be verified in polynomial time and guarantees that a given REB has one and exactly one execution model.

Definition 4.1 (*Labeled Dependency Graph*). Each REB $\mathcal{R}$ is associated with a (directed) *labeled dependency graph* $DG_{\mathrm{R}} = \langle \mathcal{N}, \mathcal{E} \rangle$ where:

—$\mathcal{N}$(the set of nodes) coincides with the set of all prioritized event expressions $p{:}E$ that occur in the head of some trigger $[B \to p{:}E] \in \mathcal{R}$;

—$\mathcal{E}$(the set of edges) consists of the following triples,[3] for all triggers $[B \to p{:}E] \in \mathcal{R}$, for all events E' in the body B, and for all nodes $q{:}E' \in \mathcal{N}$:
 — $\langle q{:}E', +, p{:}E \rangle$;
 — $\langle r{:}\texttt{conf}(E'), -, p{:}E \rangle$, for all $[r{:}\texttt{conf}(E')] \in \mathcal{N}$ such that $q \preceq r$.

Definition 4.2 (*Safeness*). A REB $\mathcal{R}$ is *safe* if its dependency graph $DG_{\mathcal{R}}$ contains no cycles in which some edge is labeled "$-$".

Example 4.1. It is easy to verify that the REB in Figure 1 is safe. The corresponding dependency graph consists of the edges reported in Figure 2 (for brevity, in the figure, -on and -duty have been dropped from role names).

On the contrary, the dependency graph for the REB of Example 3.6 contains a cycle consisting of the single negative edge $\langle \texttt{disable}\ R, -, \texttt{disable}\ R \rangle$, while in Example 3.7 there exists a cycle consisting of the two edges $\langle \texttt{disable}\ R, -, \texttt{disable}\ S \rangle$ and $\langle \texttt{disable}\ S, -, \texttt{disable}\ R \rangle$.

The following theorem shows that if a REB $\mathcal{R}$ is safe, then the system's behavior is unambiguously determined by $\mathcal{R}$, RQ, and the initial status of the roles. Proofs are given in Appendix A.

THEOREM 4.1. *If a REB $\mathcal{R}$ is safe, then for all request streams RQ and for all $S \subseteq$ Roles, there exists exactly one execution model $\langle EV, ST \rangle$ of $\mathcal{R}$ and RQ such that $ST(0) = S$.*

[3]Each triple $\langle N_1, \ell, N_2 \rangle$ represents an edge from node N_1 to N_2, labeled by ℓ.

```
Input: a REB $\mathcal{R}$
Output: true if $\mathcal{R}$ is safe, false otherwise

begin
/* construction of the dependency graph */
$\mathcal{N}$:= 0;
$\mathcal{E}$:= 0;
for all $[B \to p{:}E] \in \mathcal{R}$do
  $\mathcal{N}$:= $\mathcal{N} \cup \{p{:}E\}$;
for all $[B \to p{:}E] \in\mathcal{R}$do
  for all $E' \in B$ such that $\exists q,\ q{:}E' \in\mathcal{N}$do
    $\mathcal{E}$:= $\mathcal{E} \cup \{\ \langle q{:}E', +, p{:}E\rangle\ \}$;
    for all $r : \mathtt{conf}(E') \in\mathcal{N}$such that $q \preceq r$ do
      $\mathcal{E}$:= $\mathcal{E} \cup \{\ \langle r{:}\mathtt{conf}(E'), -, p{:}E\rangle\ \}$;

/* cycle generation and checking */
SCC := strongly connected components of $\langle\mathcal{N}, \mathcal{E}\rangle$;
for all $\langle\mathcal{N}', \mathcal{E}'\rangle\in$ SCC do
  for all $\langle X, \ell, Y\rangle\in\mathcal{E}$' do
    if $\ell$ = '$-$' then return false;
return true
end
```

Fig. 3. Algorithm for safeness verification.

Corollary 4.1. *If a REB $\mathcal{R}$ is safe, then for all request streams RQ, there exists exactly one canonical execution model of $\mathcal{R}$ and RQ.*

This proves that safeness is a sufficient condition for good system behavior. Necessary conditions are much harder to find and of little practical help, because syntactic properties (such as graph-based ones) fail to recognize that ill-formed portions of the graph may be harmless because they can never be activated. In general, checking model existence and model uniqueness are NP-hard problems. This can be proved via techniques developed for the stable model semantics of logic programs; the details lie beyond the scope of this article.

A fast algorithm for safeness verification is illustrated in Figure 3. The first part of the algorithm builds the dependency graph associated with $\mathcal{R}$, whereas the second part checks for cycles with a negative edge. The correctness of the former is obvious, as it closely mimics the formal definition of the graph. The second part of the algorithm checks whether a cycle with a negative edge exists by computing the strongly connected components of the dependency graph, and then checking whether any of them contains a negative edge. This approach is correct, as stated by the following proposition.

Proposition 4.1. *A labeled dependency graph contains a cycle with a negative edge if and only if one of its strongly connected components contains a negative edge.*

PROOF. Note that each cycle is entirely contained in some strongly connected component. Therefore, if a cycle contains a negative edge, then some strongly connected component does. Conversely, if $\langle X, -, Y \rangle$ is an edge of some strongly connected component $\mathcal{C}$, then there exists a path π from Y to X in $\mathcal{C}$ (by definition of strongly connected component). By extending π with $\langle X, -, Y \rangle$ we obtain a cycle with a negative edge, entirely contained in $\mathcal{C}$. □

The algorithm runs in quadratic time.

THEOREM 4.2. *The safeness algorithm runs in time* $O(|\mathcal{R}|^2)$.

PROOF. (Sketch) If $\mathcal{N}$ is represented as an ordered vector (that can be sorted and cleaned up from duplicates at the end of the first cycle, in time $O(|\mathcal{N}| \cdot \log|\mathcal{N}| + |\mathcal{N}|)$), then the membership test in the second cycle becomes logarithmic in $|\mathcal{N}|$, and the overall cost of the second cycle can be reduced to $O(|\mathcal{R}| \cdot (\log|\mathcal{N}| + |\mathcal{N}|))$. All these terms are dominated by $O(|\mathcal{R}| \cdot |\mathcal{N}|)$, so this is the cost of the first phase.

The strongly connected components of the graph can be generated in time $O(|\mathcal{N}| + |\mathcal{E}|)$ (cf. Cormen et al. [1990]), and the global number of all their edges is bounded by $|\mathcal{E}|$, so the global cost of the second phase of the algorithm is dominated by $O(|\mathcal{N}| + |\mathcal{E}|)$.

Since each node must occur in some trigger's head, we have $|\mathcal{N}| \leq |\mathcal{R}|$, and $|\mathcal{E}|$ is $O(|\mathcal{R}|^2)$. So the costs of both phases are dominated by $O(|\mathcal{R}|^2)$. □

Note that for a fixed set of priorities, the number of iterations of the innermost loop of the graph construction phase is bounded by a constant (i.e., |Prios|). It follows that for fixed sets of priorities, the cost of the safeness verification check is reduced to $O(|\mathcal{R}| \log |\mathcal{R}|)$.

Note that the efficiency of safeness verification can be further improved by adopting incremental graph construction methods, and by caching the set of strongly connected components. Whenever a trigger is inserted or updated, only a few edges have to be inserted or deleted, and only the old strongly connected components involving the events in the new/updated trigger have to be considered, unless new nodes are created.

5. INDIVIDUAL EXCEPTIONS

The temporal RBAC model described so far allows the SO to specify the role trigger requesting the enabling and/or disabling of roles on the basis of the occurrence of specific events (i.e., enabling/disabling of other roles) and/or the verification of some conditions (i.e., the fact that a role is enabled or not). However, the specification language we have introduced so far does not allow the specification of *individual exceptions*. This means that it is not possible to selectively enable/disable a role only for specific users. However, there are many situations in which such, a finer granularity level would be useful, as the following example illustrates.

Example 5.1. Let us consider once again our medical example of Figure 1 and suppose that a nurse on training, say Mary, performs some potentially dangerous actions. What is needed in this case is a mechanism that allows the

SO to disable the role `nurse-on-training` *but only* for `Mary` and not for all the other nurses who are actually authorized to activate this role.

In order to formulate individual exceptions to role enabling and disabling we extend the specification language as follows. First we extend Definition 3.1 (event expressions) with *individual disabling and reenabling* of the form:

$$\texttt{disable}\, R\ \texttt{for}\ U, \text{ and } \texttt{re\_enable}\ R\ \texttt{for}\ U,$$

where U is either a user name or a variable ranging over user names. The two new forms of events are conflicting, and Definition 3.2 is extended accordingly. The new events may occur wherever the other events could, in REBs and request streams, so the remaining syntactic definitions need no change. At the semantic level, the notion of blocked events (Definition 3.6) can be easily extended as follows. An individual enabling/reenabling of a role $p{:}E \in S$ is blocked by S if there exists $q{:}\texttt{conf}(E) \in S$ such that either:

1. $E = \texttt{disable}\, R\ \texttt{for}\ U$ and $p \prec q$, or
2. $E = \texttt{re\_enable}\ R\ \texttt{for}\ U$ and $p \preceq q$.

Note that when $p = q$, individual disabling overrides corresponding reenabling.

Example 5.2. The run-time request:
`disable nurse-on-training for Mary`
has the effect of disabling role `nurse-on-training` for user `Mary`. Such a role can subsequently be reenabled for `Mary`, for example, by issuing the following run-time request:
`re_enable nurse-on-training for Mary after 1`
which has the effect of allowing `Mary` to activate role `nurse-on-training` one hour after the run-time request was issued.

Also traces are adapted in the natural way. Event sequences (EV) may contain individual disabling and reenabling. Role status sequences (ST) may contain both role names (as before) and pairs $\langle R, U\rangle$, where R is a role name and U a user name. A user U can activate role R at time t if

1. $R \in ST(t)$ (R is enabled at time t), and
2. $\langle R, U\rangle \notin ST(t)$ (there is no individual exception for U).

Definition 3.7 (**System Traces**). can be adapted by requiring:

$$\begin{aligned}
ST(t+1) = (&ST(t) \cup \\
&\{\langle R, U\rangle \mid (\texttt{disable}\, R\ \texttt{for}\ U) \in \texttt{Nonblocked}(EV(t))\} \cup \\
&\{R \mid \texttt{enable}\, R \in \texttt{Nonblocked}(EV(t))\}) \\
&\setminus \\
&(\{\langle R, U\rangle \mid (\texttt{re\_enable}\, R\ \texttt{for}\ U) \in \texttt{Nonblocked}(EV(t))\} \cup \\
&\{R \mid \texttt{disable}\, R \in \texttt{Nonblocked}(EV(t))\}).
\end{aligned}$$

In other words, an individual disabling inserts suitable pairs $\langle R, U\rangle$ in $ST(t+1)$, while a reenabling removes such pairs. The notions of caused events and execution models do not need to be changed. Similarly, the definitions of

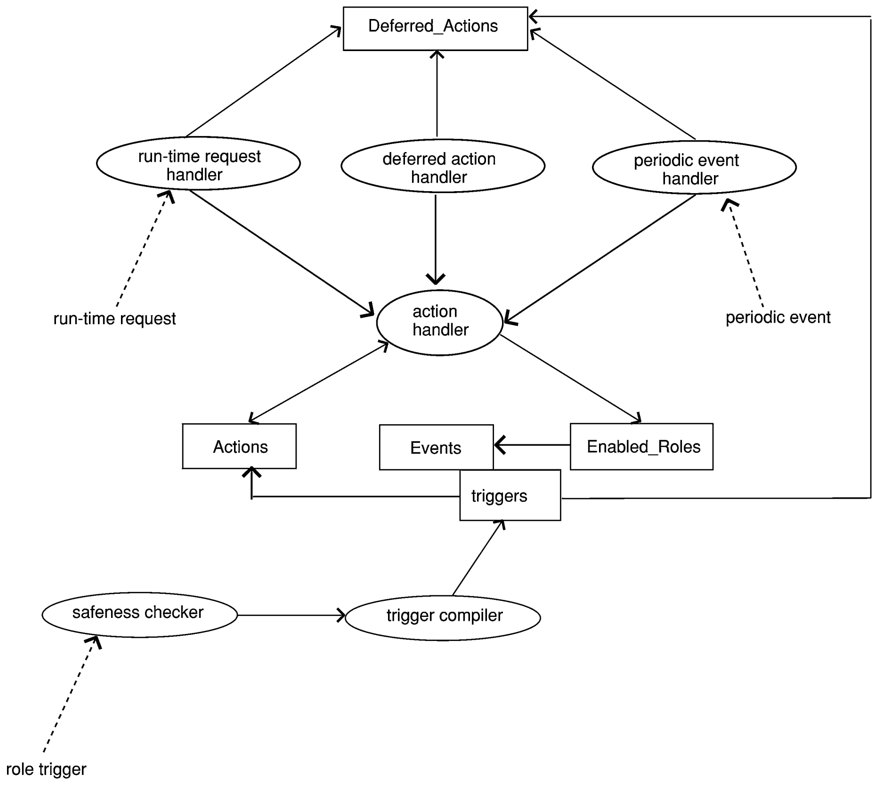

Fig. 4. System architecture.

dependency graph and safe REBs, as well as related results, do not require any modification.

6. SYSTEM ARCHITECTURE

In this section we present a system implementing TRBAC on top of an Oracle DBMS. At any given time, the system must be able to determine the roles that a user can activate, according to the periodic events and triggers contained in the REB, and the run-time requests issued until that point. A request by a user to activate a role is authorized if the user has the authorization to play the role, the role is enabled at the time of the activation request, and no individual exceptions have been specified for the user for that particular role. In designing an architecture supporting role enabling in TRBAC, key aspects that must be taken into account are that roles' enabling/disabling can be either immediately executed, or can be deferred by a fixed time interval, and that run-time requests, periodic events, and role triggers have an associated priority, to be considered when dealing with conflicting actions. Figure 4 shows the system architecture. The figure shows both data structures (rectangles) and functional components (ovals). Interactions among the various entities are represented by arrows. In the following sections we illustrate all the components of the system architecture, starting with data structures.

Role	Action	Users	Priority	Temporal Constraint
doctor-on-night-duty	enable	all	VH	[1/1/2000,∞],Night Time
doctor-on-night-duty	disable	all	VH	[1/1/2000,∞],Day Time
doctor-on-day-duty	enable	all	VH	[1/1/2000,∞],Day Time
doctor-on-day-duty	disable	all	VH	[1/1/2000,∞],Night Time

Fig. 5. Table *Deferred_Actions* for the REB in Figure 1.

6.1 Data Structures

Information on the status of a role, periodic events, and run-time requests are maintained in Oracle tables. More precisely, the data maintained by the system are:

1. *Enabled_Roles*: a table containing an entry for each enabled role. The entry corresponding to a role R also contains the set of users authorized to activate R according to the REB content and the run-time requests issued up to that point. If the REB does not contain any individual exception for role R, then the set of users associated with R in table *Enabled_Roles* coincides with the set of users authorized to play role R. In this case, the keyword `all` is used to denote all the users authorized to play a role.
2. *Deferred_Actions*: a table that contains an entry for each deferred action. The entry contains the time instant at which the action execution is scheduled and its priority. If the action is periodic, the table contains the first instant at which it is scheduled along with the temporal interval that must occur between two consecutive executions of the action (i.e., the periodicity).
3. *Actions*: a table that records the actions to be potentially executed on table *Enabled_Roles*. Such actions can be caused by the firing of a role trigger, by a run-time request, or by a periodic event. Table *Actions* is needed because not all the required actions have to be executed, in that some of them may be invalidated by conflicting actions. Intuitively, *Actions* corresponds to set `Caused` introduced in Definition 3.8.
4. *Events*. This is a global event base that records the enabling/disabling of roles as well as individual exceptions.

Note that the system does not include any data structure for role triggers, since role triggers are implemented by exploiting the trigger mechanism provided by Oracle. Each time a new role trigger is specified it is translated into an equivalent[4] Oracle trigger attached to table *Events*. By using this strategy, no mechanisms have to be developed for role trigger activations, since the automatic activations of role triggers are ensured by the DBMS services.

Example 6.1. Consider the REB in Figure 1. The corresponding table *Deferred_Actions* is shown in Figure 5. In the figure, for simplicity, we use the symbolic formalism to denote periodic time. Actually, the system uses a formalism for internal representation that translates each symbolic expression into a periodicity constraint expressed in the basic calendar. For instance, since *Hours*

[4]By equivalent, we mean a trigger having the same effects as the specified role trigger.

is the basic calendar and assuming that hour 1 is the first hour of 1/1/2000, then the periodic expression `Night Time` is translated into the periodicity constraint $t \equiv_{24} y \ \forall y = 1, \ldots, 12$, where 24 is the number of hours in a day (the periodicity of `Night Time`), and 12 is the number of hours within each night (the granularity of `Night Time`). $t \equiv_{24} y \ \forall y = 1, \ldots, 12$ is a compact notation for the disjunction of simple periodicity constraints: $t \equiv_{24} 1 \vee t \equiv_{24} 2 \vee \ldots \vee t \equiv_{24} 12$, where $t \equiv_k c$ denotes the set of integers of the form $c + kn$, ranging from $-\infty$ to $+\infty$ in Z.

The first instant of the night of `1/1/2000` tuple `(doctor-on-night-duty,all)` is inserted into table *Enabled_Roles* and this causes the insertion of the tuple `(enable,doctor-on-night-duty)` into table *Events*. This latter insertion causes the evaluation of the triggers attached to table *Events*,[5] that correspond to the role triggers in Figure 1. Trigger RT_1 fires and this causes the insertion of the tuple `(nurse-on-night-duty,enable,all,VH)` into table *Actions*. Since no conflicting actions exist, the tuple `(nurse-on-night-duty,all)` is inserted into table *Enabled_Roles* and this causes the insertion of the tuple `(enable,nurse-on-night-duty)` into table *Events*.

6.2 Functional Modules

The system consists of six functional modules (cf. Figure 4), namely, the *Safeness Checker*, the *Trigger Compiler*, the *Periodic Event Handler*, the *Run-Time Request Handler*, the *Deferred Action Handler*, and the *Action Handler*. In what follows we illustrate each of the above modules in turn.

The *Safeness Checker* is activated upon a role trigger insertion or modification, and verifies whether such operation preserves the REB's safeness. To perform this check the *Safeness Checker* uses an incremental version of the algorithm illustrated in Section 4 that does not rebuild the dependency graph from scratch each time a role trigger is inserted or updated but modifies the existing graph by changing all and only those portions that are really affected by the insertion/update. If the check fails, then the requested insertion/modification is not executed. Note that trigger deletions do not affect safeness and thus do not require any safeness check. However, to avoid the dependency graph becoming too large, an algorithm is periodically executed that removes from the graph all the nodes and arcs corresponding to triggers that have been removed from the system. If the *Safeness Checker* authorizes the insertion/modification of a role trigger, the role trigger is passed to the *Trigger Compiler* that translates it into an equivalent Oracle trigger attached to table *Events*. The translation depends on the number and types of expressions appearing in the trigger body and on the type of action that the firing of the trigger should cause. If the action caused by the trigger has to be immediately executed, the firing of the corresponding Oracle trigger inserts the action into table *Actions*, along with the action priority. By contrast, if the action has to be deferred, the firing of the corresponding Oracle trigger inserts the action into table *Deferred_Actions* along with its priority and the instant(s) at which the action has to be executed.

[5]A methodology for trigger generation is presented in Section 6.3.

The *Periodic Event Handler* is in charge of managing periodic events. When a periodic event is inserted into the REB, the *Periodic Event Handler* inserts a corresponding entry into table *Deferred_Actions*. The entry contains the action corresponding to the periodic event, its priority, and the associated periodicity constraint (i.e., the constraint denoting the set of instants at which the action has to be executed). When the deletion of a periodic event is requested, the corresponding entry is removed from *Deferred_Actions*. Moreover, the *Periodic Event Handler* sends a message to the *Action Handler* which verifies whether table *Enabled_Roles* has to be modified because of the periodic event deletion.

The *Run-Time Request Handler* is activated each time a run-time request is issued. The handler first verifies whether the request is for an immediate action or for a deferred one. In the former case, it returns the action and its priority to the *Action Handler*. In the latter case, it inserts the action into table *Deferred_Actions* along with the time of its occurrence and the action priority.

The *Deferred Action Handler* is in charge of monitoring table *Deferred_Actions* to execute the actions it contains at the associated time. Such a module is implemented as a daemon which maintains a list of instants at which it has to wake up. For periodic actions, the *Deferred Action Handler* maintains the first instant at which the action has to be executed and the time period between any two consecutive executions of the action. When the daemon wakes up it selects from *Deferred_Actions* the actions that have to be executed. If conflicting actions exist, it selects the action with the highest priority (in case of equal priority, disabling-takes-precedence). Then, it returns the action to the *Action Handler* along with the action priority.

The *Action Handler* represents the core of the architecture and is in charge of updating table *Enabled_Roles*, according to the requests by the other modules or caused by the firing of the triggers associated with table *Events*. Since conflicts may arise among the requested actions, such actions are collected into table *Actions* before their execution on table *Enabled_Roles*. If conflicts arise, the *Action Handler* solves them before updating table *Enabled_Roles*. In particular, for each action of the form `enable` R with priority p in table *Actions*, the *Action Handler* verifies whether table *Actions* contains an event `disable` R' with priority q such that $p \preceq q$ and $R = R'$. If this event exists, the *Action Handler* does not insert `enable` R into table *Enabled_Roles*. A similar check is done for action of the form `disable` R and for actions related to individual exceptions.

6.3 Trigger Generation

In the following we illustrate how the *Trigger Compiler* translates role triggers into Oracle triggers. In defining the translation we have to take into account that in TRBAC events implied by role triggers are prioritized. Moreover, even if we consider triggers all with the same priority, the safeness condition induces an evaluation order for triggers, reflecting their semantics, that must be obeyed during their evaluation, as shown by the following example.

Example 6.2. Consider a REB consisting of the role triggers:

1. `enable` $R_1 \rightarrow$ `enable` R_2
2. `enable` $R_0 \rightarrow$ `disable` R_1.

Such REB satisfies the safeness condition given in Section 4. Suppose now that at time t the run-time requests {`enable` R_1, `enable` R_0} are issued. Since the REB is safe, it admits exactly one execution model according to which only role R_0 must be enabled. Such correct behavior can be guaranteed only if trigger 2 is evaluated before trigger 1. By contrast, if trigger 1 is evaluated before trigger 2, its firing generates `enable` R_2, and this is not correct since the subsequent evaluation of trigger 2 generates `disable` R_1 which should have blocked the firing of trigger 1.

By the above considerations, it is clear that the correct evaluation of role triggers depends on the enforcement of an evaluation order which is based on the dependencies among the events appearing in the triggers. Unfortunately, Oracle does not provide any explicit mechanism for prioritizing triggers. To cope with this problem, we impose some restrictions on the specification language supported by our current prototype that ensure the correct evaluation of triggers even in the absence of a priority mechanism. Intuitively, such restrictions do not allow the specification of triggers whose results depend on the order in which they are evaluated. The first restriction we impose is that actions caused by the firing of a trigger must all have the same priority (that by default is assumed to be $\perp$). By contrast, different priorities can be associated with run-time requests. Additionally, to ensure that the evaluation of triggers with the same priority always enacts the correct semantics, we impose a further restriction to triggers causing instantaneous actions (i.e., those with $\Delta t = 0$, also called *instantaneous triggers*): if an event `enable` R appears in the head of an instantaneous trigger, then the body must contain $\neg$`enabled` R, and if the event `disable` R appears in the head of an instantaneous trigger, then the body must contain `enabled` R. Analogous restrictions apply to triggers causing individual exceptions. Such syntactic restrictions impose that instantaneous triggers can be used only to change a role status (i.e., from enabled to not enabled and vice versa) and thus ensure that two triggers which allow the derivation of conflicting events cannot be simultaneously activated. This implies that the events generated by a set of triggers do not depend on the trigger evaluation order, and thus allow their correct evaluation in the Oracle DBMS. It is important to remark that the current limitation of our implementation is due to the fact that we wish to implement TRBAC on top of a conventional DBMS by making use of its trigger support. At the current stage, commercial DBMSs do not support trigger priority (e.g., Oracle) or they only support a limited form of priority by which the evaluation order of triggers that are simultaneously fired is given by the order in which they have been specified (e.g., DB2 and the recent SQL99 standard [Gulutzan and Pelzer 1999]). However, using this latter feature would require knowing at the beginning all the triggers that need to be specified and thus does not allow subsequent insertions or modifications of role triggers, or it makes these operations cumbersome,

```
INPUT:    1. A role trigger $RT$: $E_1,\ldots,E_n,C_1,\ldots,C_k \rightarrow E$ after $\Delta t$, where each $E_i$ has the
          form enable $R$ or disable $R$, and each $C_j$ has the form enabled $R$ or $\neg$enabled $R$
OUTPUT:   The Oracle trigger equivalent to $RT$
METHOD:
  (1) Add create trigger $RT$ to the Oracle Trigger
  (2) Add before insert on Events to the Oracle Trigger
  (3) Add for each row to the Oracle Trigger
  (4) Let $A(E_i)$ be the action in the simple event expression $E_i$
  (5) Let $R(E_i)$ be the role in the simple event expression $E_i$
  (6) For each $E_i$ in $RT$: $Cond_i$ := new.action = '$A(E_i)$' AND new.role = '$R(E_i)$'
  (7) Add when ($Cond_1$ AND ... AND $Cond_n$) to the Oracle Trigger
  (8) Add declare $X_1$, ...,$X_k$, $Y_1$, ..., $Y_n$ number; to the Oracle Trigger
  (9) Add begin to the Oracle Trigger
  (10) For each role status expression $C_i$ in $RT$:
          (a) Let $R(C_i)$ be the role in $C_i$
          (b) Add to the Oracle trigger:
                select count(*) into $X_i$
                from Enabled_Roles
                where role = '$R(C_i)$';
          (c) If $C_i = \neg$enabled $R$: Add if $X_i = 0$ to the Oracle trigger
          (d) If $C_i =$ enabled $R$: Add if $X_i > 0$ to the Oracle trigger
          (e) Add then to the Oracle trigger
       endfor
  (11) For each simple event expression $E_i$ in $RT$:
          (a) Add to the Oracle trigger:
                select count(*) into $Y_i$
                from Actions
          (b) If $E_i =$ disable $R$:
                Cond := (role = '$R(E_i)$' AND action = 'enable' AND priority $>$ 0)
          (c) If $E_i =$ enable $R$:
                Cond := (role = '$R(E_i)$' AND action = 'disable' AND priority $\geq$ 0)
          (d) Add where $Cond$; to the Oracle trigger
          (e) Add if $Y_i = 0$ to the Oracle trigger
          (f) Add then to the Oracle Trigger
       endfor
  (12) If $\Delta t = 0$:
          Add insert into Actions values('$R(E)$','$A(E)$',all,0); to the Oracle Trigger
       else:
          Add insert into Deferred_Actions values('$R(E)$','$A(E)$',all,0,$\Delta t$);
          to the Oracle Trigger
  (13) For $i = 1$ to $n + k$: Add endif to the Oracle trigger
  (14) Add end; to the Oracle trigger
```

Fig. 6. Trigger generation.

since they may require the deletion and respecification of a large number of triggers.

The strategy used by the *Trigger Compiler* to translate a role trigger into an Oracle trigger is illustrated in Figure 6. For simplicity, we consider only role triggers that do not contain individual exceptions. The extension with individual exceptions is straightforward. The *Trigger Compiler* receives as input a role trigger, specified according to the TRBAC syntax, and generates a

```
create trigger Trigger_Name
{before|after} Events
on Table
[referencing References]
[for each row]
when [Conditions]
PL/SQL block;
```

Fig. 7. Oracle trigger syntax.

corresponding Oracle trigger attached to table *Events*. Oracle triggers are specified according to the syntax shown in Figure 7. In Figure 6, we make use of the function "Add" to create the Oracle trigger step by step.

Intuitively, the strategy presented in Figure 6 generates for each role trigger received as input, an Oracle trigger which is evaluated before each insertion in table *Events*. The firing condition is the conjunction of the simple event expressions (i.e., `enable` R/`disable` R) appearing in the body of the role trigger. Then, Step 10 considers each role status condition C_i (i.e., `enabled` R/¬`enabled` R) appearing in the body of the role trigger and inserts into the trigger the PL/SQL code necessary for its testing (such test is done by querying table *Enabled_Roles*). Step 11 considers each simple event expression E_i appearing in the body of the role trigger, and for each of them generates the PL/SQL code for testing that no simultaneous blocking events have taken place (such test is done by querying table *Actions*). Finally (Step 12), if the role trigger is instantaneous, then a tuple corresponding to the trigger action is inserted into table *Actions*. Such a tuple contains the requested action, the role involved in such action, the action priority, and the set of users to which the action applies. By contrast, if the trigger is not instantaneous, an analogous tuple is inserted into table *Deferred_Actions*. In such a case, the tuple also contains the temporal displacement for the requested action.

Example 6.3. As an example of trigger generation, consider trigger $\texttt{RT}_2$ in Figure 1. By the syntactic restriction we have imposed this trigger is rewritten as: `enabled nurse-on-night-duty`, `disable doctor-on-night-duty` → `disable nurse-on-night-duty`, and is thus translated by the *Trigger Compiler* into the following Oracle trigger.

```
create trigger RT2
  before insert on Events
  for each row
  when (new.action = 'disable' AND new.role = 'doctor-on-night-duty')
  declare
    X,Y number;
  begin
    select count(*) into X
    from Enabled_Roles
    where role = 'nurse-on-night-duty';
    if X > 0
    then
       select count(*) into Y
       from Actions
       where (role = ‘doctor-on-night-duty’ AND action = ‘enable’
              AND priority > 0);
       if Y = 0
       then
          insert into Actions values(‘nurse-on-night-duty’,‘disable’,all,0);
       endif
    endif
 end;
```

Such a trigger first verifies whether the role `nurse-on-night-duty` is enabled. If this check succeeds, then it verifies that the event in the body of the trigger (i.e., `disable doctor-on-night-duty`) is not blocked by conflicting actions. If even this further check succeeds, the Oracle trigger inserts into table *Actions* the tuple `(nurse-on-night-duty,disable,all,0)`, which corresponds to an immediate request for disabling the role `nurse-on-night-duty` with the lowest priority. The keyword `all` is used to denote that the action is not requested for a particular user (or set of users) but for all the users authorized to play role `nurse-on-night-duty`.

A similar translation is applied to trigger $\mathtt{RT}_5$ in Figure 1. In this case, no initial rewriting is needed since $\mathtt{RT}_5$ causes a deferred action. The corresponding Oracle trigger is thus as follows,

```
create trigger RT_5
   before insert on Events
   for each row
   when (new.action = ‘enable’ AND new.role = ‘nurse-on-day-duty’)
   declare
      Y number;
   begin
      select count(*) into Y
      from Actions
      where (role = ‘nurse-on-day-duty’ AND action = ‘disable’
             AND priority ≥ 0);
      if Y = 0
      then
          insert into Deferred_Actions values(‘nurse-on-training’,‘enable’,
                                                  all,0,2);
      endif;
   end;
```

where the major difference, with respect to the previous trigger, is that, since trigger $\mathtt{RT}_5$ is not instantaneous, the firing of the corresponding Oracle trigger causes an insertion into table *Deferred_Actions*.

7. RELATED WORK

Role-based access control has received increasing attention in the past years [Atluri 1999; Osborn 2000; Sandhu 1995, 1997, 1998a] and several role-based access control models have been proposed (see, for instance, Ferraiolo et al. [1999], Jonscher et al. [1994], Nyanchama and Osborn [1999], Sandhu et al. [1996] and Sandhu [1998b]). For example, the work by Jonscher et al. [1994] describes a role-based access control scheme for object-oriented data models, whereas Sandhu et al. [1996] present a framework of four role-based access control models.

Most of the RBAC models of recent years recognized the need of supporting constraints on both users and roles and provided some mechanisms for their support. However, most of the attention has been so far devoted to *separation of duties* constraints [Ahn and Sandhu 1999; Ferraiolo et al. 1999; Kuhn 1997; Jonscher et al. 1994; Nyanchama and Osborn 1999; Sandhu 1991, 1998; Sandhu et al. 1996; Simon and Zurko 1997; Tidswell and Jaeger 2000]. The principle of separation of duties aims at reducing the risk of fraud by not allowing any individual to have sufficient authority within the system to perpetrate a fraud on his or her own. Separation of duties is a principle often applied in everyday life; for example, opening a safe requires two keys held by different individuals, approval of a business trip requires approval of the department manager as well as an accountant, and a paper submitted to a conference is usually required to be reviewed by three referees who must be different from the author(s) of the paper.

Several authors have discussed and categorized different forms of separation of duties (e.g., static, dynamic, object-based, operational, etc.) and taxonomies have been proposed [Nyanchama and Osborn 1999; Simon and Zurko 1997] for separation of duties constraints.

The RBAC model proposed by Ferraiolo et al. [1999] provides support for an additional class of constraints, that is, cardinality constraints. Such a class of constraints imposes an upper bound on the number of users that can play a role at a given time. An example of a role cardinality constraint is the one imposing that only one individual at a time may assume the role of the department head, although such a role can be assumed by different individuals in different time periods.

Other related work has been carried on in the context of Workflow Management Systems. Bertino et al. [1999] have proposed a logic language for expressing constraints on both user and role assignments to workflow tasks, able to express both static and dynamic separation of duties. The constraint model proposed in Bertino et al. [1999] has been further extended in Kandala and Sandhu [1999] to express weighted voting constraints, that is, constraints imposing lower bounds on the number of users that must execute a task under a particular role. An example of a weighted voted constraint is the one imposing that at least three different supervisors must approve a check. However, although the specification and enforcement of constraints in RBAC models have been deeply investigated before, none of the previous work attempts to specify temporal constraints and dependencies on the enabled roles. We believe that this class of constraints is very useful for a number of application domains.

As such we believe that temporal constraints on the enabled roles cannot be neglected by RBAC models. Our work is thus a first proposal towards the development of an RBAC model supporting such a class of constraints.

Another work related to our proposal is the access control model presented by Bertino et al. [1998], which supports temporal authorizations and derivation rules, able to express temporal dependencies among authorizations. From this work we borrow the formalism for periodic time representation. However, our work substantially differs from the work in Bertino et al. [1998]. In that work access authorizations are granted to both users and roles, and temporal constraints are used to limit the time of validity of authorizations. Temporal dependencies among authorizations are expressed by means of a logic language. By contrast, in the current article, we use active rules to express temporal constraints on role enabling/disabling. Moreover, the notions of trigger priority, deferred actions, individual exceptions, and run-time requests are new with respect to the model proposed in Bertino et al. [1998]. Furthermore, we have developed a system supporting TRBAC on top of a conventional DBMS.

A further authorization model that allows temporal specification has been recently proposed in Gal and Atluri [2000]. The model, called *Temporal Data Authorization Model* (*TDAM*), is able to express access control policies based on the temporal characteristics of data, such as valid and transaction time, and thus can be seen as complementary to the model proposed in Bertino et al. [1998] with respect to the specification of temporal constraints on user authorizations. However, TDAM does not support temporal constraints on the enabled roles and thus temporal constraints that can be expressed in TDAM substantially differ from those provided by TRBAC.

Finally, the problem of modeling the semantics of active rules has been tackled in several papers (e.g., Lobo and Rachid [1994] and Lobo and Baral [1996]). Only some of them consider priorities. Our approach differs as priorities are associated with individual events, rather than entire rules. Moreover, unlike other approaches, we need to enforce unambiguous behavior, given the delicate nature of access control. These features—together with the enforced principle that if a trigger fires then its body should not be blocked—result in a sophisticated notion of a dependency graph, where each condition in a trigger's body may produce a multiplicity of edges with different labels. Our safeness condition then refers to a richer underlying semantic structure.

8. CONCLUSIONS

In this article we have presented Temporal-RBAC (TRBAC), a temporal extension of the RBAC model. The innovative features of TRBAC are the support for periodic enabling/disabling of roles, individual exceptions, and the possibility of specifying temporal dependencies among such actions, expressed by means of role triggers.

We have formally defined TRBAC, by specifying its syntax and semantics. We have provided a polynomial algorithm to verify whether specifications are free from ambiguities. Finally, we have presented a system supporting TRBAC on top of a commercial DBMS.

Further work includes the development of an SQL-like language for TRBAC specification, a graphical interface supporting administrative operations, and the extension of our prototype to be able to support the full specification language provided by TRBAC. It is important to point out that, if a DBMS supported a predefined trigger evaluation order, then it would be possible to implement the full specification language. Indeed, given a safe REB, a correct evaluation order for instantaneous triggers can be statically defined, in polynomial time, by visiting the dependency graph of the REB. This opens the door to unrestricted implementations, in view of the emerging standard encompassing controlled trigger evaluation order [Gulutzan and Pelzer 1999].

Other extensions regard the enhancement of the language provided by TRBAC to take into account other relevant events in trigger specification, besides the ones considered by TRBAC, such as the number of users actually activating a role.

Another research direction is the use of TRBAC for enforcing access control on the execution of downloaded content along the lines discussed by Jaeger et al. [1999].

APPENDIX

A.1 Proofs

This section is devoted to the proof of Theorem 4.1. The proof is based on a correspondence between execution models and the stable models of a suitable family of normal logic programs. After recalling the basic notions related to logic programming, suitable translation functions from events and triggers into logic programs are introduced, followed by the proof that the translation preserves the original semantics. This main lemma is then used to prove the main result.

Preliminaries on Logic Programming

We assume the reader to be familiar with the basic notions about first-order logic and logic programs (see, e.g., Lloyd [1984]).

A *ground normal logic program* (hereafter simply called *program*) is a set of rules of the form

$$A_0 \leftarrow A_1, \ldots, A_n, \neg A_{n+1}, \ldots, \neg A_m,$$

where the A_i ($1 \leq i \leq m$) are ground (i.e., variable-free) logical atoms. An *interpretation* for such a program is a set of ground logical atoms. A program is *positive* if it has no occurrences of "$\neg$". Each positive program P has a unique minimal model denoted by $lm(P)$.

The *Gelfond–Lifschitz transformation* [Gelfond and Lifschitz 1988] of a program P with respect to an interpretation I, denoted by P^I, is the positive program obtained from P by removing all rules containing a literal $\neg A_i$ such that $A_i \in I$, and by removing all negative literals from the remaining rules.

An interpretation I is a *stable model* [Gelfond and Lifschitz 1988] of P if and only if $I = lm(P^I)$.

The *dependency graph* of a program P consists of a set of nodes that coincide with the logical atoms in P, plus the set of all labeled edges $\langle A_i, \ell, A_0\rangle$, where

A_i occurs in the body of a rule with A_0 in its head; ℓ is "$-$" if A_i occurs in the scope of negation, and "+" otherwise. A program is *stratifiable* if its dependency graph contains no cycles comprising a *negative edge* (i.e., an edge labeled "$-$").

Translation and Auxiliary Lemmata

We start by introducing a translation from prioritized events into logical atoms.

Definition A.1. For all predicate symbols r and prioritized events p:`enable` R, p:`disable` R let

$$\begin{aligned}
\mathtt{tr(r, p{:}enable\ R)} &\overset{\text{def}}{=} \mathtt{r(p, enable\ , R)}\\
\mathtt{tr(r, p{:}disable\ R)} &\overset{\text{def}}{=} \mathtt{r(p, disable\ , R)}\\
\mathtt{tr(r, enable\ R)} &\overset{\text{def}}{=} \mathtt{r(enable\ , R)}\\
\mathtt{tr(r, disable\ R)} &\overset{\text{def}}{=} \mathtt{r(disable\ , R)}.
\end{aligned}$$

This translation function is extended to sets of prioritized events in the natural way; for instance,

$$\mathtt{tr(caused, EV(t))} = \{\mathtt{tr(caused, p{:}E)} \mid \mathtt{p{:}E} \in \mathtt{EV(t)}\}\,.$$

Next we introduce logic programs that capture the property of being caused at time t.

Definition A.2. The program $P(t, EV, ST, \mathcal{R}, RQ)$ consists of the following facts and rules.

1. All facts $\mathtt{tr(caused, p{:}E)}$ such that $(I, P, p{:}E) \in \mathcal{R}$ and $t \in Sol(I, P)$.
2. All facts $\mathtt{tr(caused, p{:}E)}$ such that $[p{:}E\ \mathtt{after}\ \Delta t] \in RQ(t - \Delta t)$, where $\Delta t \leq t$.
3. All facts $\mathtt{tr(caused, p{:}E)}$ such that
   $[E_1, \ldots, E_n, C_1, \ldots, C_k \to p{:}E\ \mathtt{after}\ \Delta t] \in \mathcal{R}$, and
   a. $0 < \Delta t \leq t$
   b. for all $C_i = \mathtt{enabled}\ R$, $R \in ST(t - \Delta t)$,
   c. for all $C_i = \neg\mathtt{enabled}\ R$, $R \notin ST(t - \Delta t)$, and
   d. for all $1 \leq i \leq n$, $E_i \in \mathtt{Nonblocked(EV(t - \Delta t))}$.
4. All rules
   $$\mathtt{tr(caused, p{:}E)} \leftarrow \mathtt{tr(valid, E_1)}, \ldots, \mathtt{tr(valid, E_n)}$$
   such that $[E_1, \ldots, E_n, C_1, \ldots, C_k \to p{:}E] \in \mathcal{R}$, and
   b′. for all $C_i = \mathtt{enabled}\ R$, $R \in ST(t)$,
   c′. for all $C_i = \neg\mathtt{enabled}\ R$, $R \notin ST(t)$.
5. All rules
   $$\mathtt{valid(A, R)} \leftarrow \mathtt{caused(p, A, R)}, \mathtt{not\_blocked(p, A, R)}$$
   such that $p \in \mathtt{Prios}$, $A \in \{\mathtt{enable}\ , \mathtt{disable}\ \}$, and $R \in \mathtt{Roles}$.

6. All rules

$$\begin{aligned}\texttt{not\_blocked(p, enable , R)} \leftarrow\ & \neg\texttt{caused(p, disable , R)},\\ & \neg\texttt{caused(q}_1\texttt{, disable , R)},\\ & \ldots,\\ & \neg\texttt{caused(q}_n\texttt{, disable , R)}\end{aligned}$$

$$\begin{aligned}\texttt{not\_blocked(p, disable , R)} \leftarrow\ & \neg\texttt{caused(q}_1\texttt{, enable , R)},\\ & \ldots,\\ & \neg\texttt{caused(q}_n\texttt{, enable , R)}\end{aligned}$$

where $\{q_1, \ldots, q_n\} = \{q \mid p \prec q\}$.

LEMMA A.1. *Suppose that for all $t' < t$, $\texttt{Caused}(t', EV, ST, \mathcal{R}, RQ) = EV(t')$. Then the following are equivalent.*

1. *Conditions 1 to 3 of Definition 3.8 hold when $\texttt{Caused}(t, EV, ST, \mathcal{R}, RQ)$ is replaced by $EV(t)$.*
2. *$P(t, EV, ST, \mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}$ has a model I such that $p{:}E \in EV(t)$ if and only if $\texttt{tr(caused, p:E)} \in I$.*

PROOF. (1 ⇒ 2) Assume 1 holds and define

$$\begin{aligned} I =\ & \texttt{tr(caused, EV(t))}\\ & \cup\ \{\texttt{tr(valid, E)} \mid \exists \texttt{p:E} \in \texttt{Nonblocked(EV(t))}\}\\ & \cup\ \{\texttt{tr(not\_blocked, p:E)} \mid \texttt{p:E} \text{ is not blocked by } \texttt{EV(t)}\}\,. \end{aligned}$$

We have to prove that 2 holds. Note that by construction $p{:}E \in EV(t)$ if and only if $\texttt{tr(caused, p:E)} \in \texttt{I}$. Therefore, we are left to show that I is a model of each fact or rule ρ in $P(t, EV, ST, \mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}$. The proof is by cases, based on Definition A.2.

a. Suppose $\rho = \texttt{caused(p, a, R)} \in \texttt{P(t, EV, ST,}\mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}$ holds by Definition A.2, point 1. Then clearly, by Condition 1 of Definition 3.8, $p{:}aR \in EV(t)$ and hence $\texttt{tr(caused, p:a R)} = \texttt{caused(p, a, R)} \in I$. Therefore I is a model of ρ.
b. Similarly, if $\rho = \texttt{caused(p, a, R)} \in \texttt{P(t, EV, ST,}\mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}$ holds by Definition A.2, point 2, then Condition 2 of Definition 3.8 implies $p{:}a\ R \in EV(t)$ and hence $\texttt{tr(caused, p:a R)} = \texttt{caused(p, a, R)} \in I$. Therefore I is a model of ρ.
c. If $\rho = \texttt{caused(p, a, R)} \in \texttt{P(t, EV, ST,}\mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}$ holds by Definition A.2, point 3, then Subconditions a to c imply the corresponding Conditions 3a to 3c of Definition 3.8. Moreover, Subcondition d implies that for $1 \le i \le n$, $E_i \in \texttt{Nonblocked(EV(t} - \Delta\texttt{t))}$. It follows by hypothesis that $E_i \in \texttt{Caused}(t - \Delta t, EV, ST, \mathcal{R}, RQ)$ and hence also 3d of Definition 3.8 holds. Then by 3 of Definition 3.8, $p{:}a\,R \in EV(t)$ and hence by analogy with the previous cases, I is a model of ρ.

d. Next assume

$$\rho = [\texttt{tr(caused, p:E)} \leftarrow \texttt{tr(valid, E}_1\texttt{)}, \ldots, \texttt{tr(valid, E}_\texttt{n}\texttt{)}]$$

is in the program by Definition A.2, Point 4. Then Conditions 3a to 3c are clearly satisfied by the E_is and $EV(t)$. Moreover, when $EV(t)$ is substituted for $\texttt{Caused}(t, EV, ST, \mathcal{R}, RQ)$, 3d collapses to $E_i \in \texttt{Nonblocked(EV(t))}$ for all $1 \leq i \leq n$. Now if the body of ρ is satisfied by I, then by definition of I we have $E_i \in \texttt{Nonblocked(EV(t))}$ for all $1 \leq i \leq n$. Then 3a to 3d hold and hence $p{:}E \in EV(t)$. By analogy with the previous case, it follows that $\texttt{tr(caused, p:E)} \in I$ and hence I is a model of ρ.

e. Let $\rho = [\texttt{valid(A, R)} \leftarrow \texttt{caused(p, A, R), not\_blocked(p, A, R)}]$. If I satisfies the body of ρ then, by definition of I, $p{:}A\ R \in \texttt{Nonblocked(EV(t))}$, and hence $\texttt{valid(A, R)} \in \texttt{I}$. This proves that I satisfies ρ.

f. $\rho = \texttt{not\_blocked(p, enable , R)} \in \texttt{P(t, EV, ST,}\mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}$ is obtained from Definition A.2, point 6 by removing all negative literals. Such literals must be true in $\texttt{tr(caused, EV(t))}$ (by definition of the Gelfond–Lifschitz transformation). It follows that for all q such that $p \preceq q$, $q{:}\texttt{disable}\ R \notin EV(t)$, and hence $p{:}\texttt{enable}\ R$ is not blocked by $EV(t)$. Then, by definition of I,

$$\texttt{not\_blocked(p, enable , R)} \in I.$$

g. $\rho = \texttt{not\_blocked(p, disable , R)} \in \texttt{P(t, EV, ST,}\mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}$ is obtained from Definition A.2, Point 6 by removing all negative literals. This case is similar to the previous one.

(2 $\Rightarrow$ 1) Conversely, suppose 2 holds. To prove 1, we show that Conditions 1 to 3 of Definition 3.8 hold when $\texttt{Caused}(t, \texttt{EV}, \texttt{ST}, \mathcal{R}, RQ)$ is replaced by $EV(t)$.

a. Let $(I, P, p{:}E) \in \mathcal{R}$ such that $t \in Sol(\texttt{I, P})$. Then by Definition A.2, Point 1, $\texttt{tr(caused,}\ p{:}E) \in P(t, EV, ST, \mathcal{R}, RQ)$ and hence $\texttt{tr(caused, p:E)} \in I$. It follows by hypothesis that $p{:}E \in EV(t)$, so $EV(t)$ satisfies Condition 1 of Definition 3.8.

b. Similarly, if the premises of Condition 2 are satisfied, then by Definition A.2, Point 2, $\texttt{tr(caused, p:E)} \in \texttt{P(t, EV, ST,}\mathcal{R}, RQ)$. By analogy with the previous case, $p{:}E \in EV(t)$, so $EV(t)$ satisfies Condition 2 of Definition 3.8.

c. Similarly, if 3a to 3d are satisfied and $\Delta t > 0$, then by Definition A.2, Point 3, $\texttt{tr(caused,}\ p{:}E) \in P(t, EV, ST, \mathcal{R}, RQ)$. By analogy with the previous case, $p{:}E \in EV(t)$, so $EV(t)$ satisfies Condition 3 of Definition 3.8 when $\Delta t > 0$.

d. Finally, suppose 3a to 3d are satisfied with $\Delta t = 0$. Then b' and c' in Definition A.2, Point 4 hold, and hence

$$\rho = [\texttt{tr(caused, p:E)} \leftarrow \texttt{tr(valid, E}_1\texttt{)}, \ldots, \texttt{tr(valid, E}_\texttt{n}\texttt{)}] \qquad (1)$$

is in $P(t, EV, ST, \mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}$. Moreover, by substituting $EV(t)$ for $\texttt{Caused}(t, \texttt{EV}, \texttt{ST}, \mathcal{R}, RQ)$ in 3d, we have that there exists p_i such that $p_i{:}E_i \in \texttt{Nonblocked(EV(t))}\ (1 \leq i \leq n)$. By hypothesis, it follows that $\texttt{tr(caused, p}_\texttt{i}\texttt{:E}_\texttt{i}\texttt{)} \in I$. Moreover, from Definition A.2, Point 6 and the hypothesis, it follows that $\texttt{tr(not\_blocked, p}_\texttt{i}\texttt{:E}_\texttt{i}) \in \texttt{P(t, EV, ST,}\mathcal{R}, RQ)$, and hence $\texttt{tr(not\_blocked, p}_\texttt{i}\texttt{:E}_\texttt{i}) \in \texttt{I}$. We conclude that

$$\{\texttt{tr(caused, p}_\texttt{i}\texttt{:E}_\texttt{i}),\ \texttt{tr(not\_blocked, p}_\texttt{i}\texttt{:E}_\texttt{i}) \mid 1 \le \texttt{i} \le \texttt{n}\} \subseteq I\,.$$

Since I is a model of the rules in Definition A.2, Point 5, it follows that I must contain

$$\texttt{tr(valid, E}_1), \ldots, \texttt{tr(valid, E}_\texttt{n})\,.$$

In order to satisfy 1, I must contain $\texttt{tr(caused, p:E)}$. By hypothesis, it follows that $p{:}E \in EV(t)$. This proves that $EV(t)$ satisfies Condition 3 of Definition 3.8, when $\Delta t = 0$. □

LEMMA A.2. *Suppose that for all* $t' < t$, $\texttt{Caused}(t', \texttt{EV}, \texttt{ST}, \mathcal{R}, RQ) = EV(t')$. *Then, the following are equivalent.*

1. $\texttt{Caused}(t, EV, ST, \mathcal{R}, RQ) = EV(t)$.
2. $P(t, EV, ST, \mathcal{R}, RQ)$ *has a stable model* I *such that* $p{:}E \in EV(t)$ *if and only if* $\texttt{tr(caused, p:E)} \in \texttt{I}$.

PROOF. First assume 1 holds. Then $EV(t)$ is the least set satisfying Conditions 1 to 3 of Definition 3.8. Note that the translation $\texttt{tr}$ and its inverse are monotonic. It follows easily from Lemma A.1 that the least model

$$I = lm\left(P(t, EV, ST, \mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}\right)$$

is such that $p{:}E \in EV(t)$ if and only if $\texttt{tr(caused, p:E)} \in \texttt{I}$.

As a consequence, $\texttt{tr(caused, p:E)} \in \texttt{I}$ if and only if $\texttt{tr(caused, p:E)} \in \texttt{tr(caused, EV(t))}$. Moreover, note that the Gelfond–Lifschitz transformation of $P(t, EV, ST, \mathcal{R}, RQ)$ with respect to I depends only on the members of I of the form $\texttt{tr(caused, p:E)}$ (given the form of negative literals in $P(t, EV, ST, \mathcal{R}, RQ)$). Consequently,

$$P(t, EV, ST, \mathcal{R}, RQ)^I = P(t, EV, ST, \mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}.$$

It follows that

$$I = lm(P(t, EV, ST, \mathcal{R}, RQ)^I);$$

that is, I is a stable model of $P(t, EV, ST, \mathcal{R}, RQ)$. This proves that 2 holds.

Conversely, assume 2 holds. Then

$$I = lm(P(t, EV, ST, \mathcal{R}, RQ)^I)$$

and $p{:}E \in EV(t)$ if and only if $\texttt{tr(caused, p:E)} \in \texttt{I}$. By analogy with the previous case, we get

$$P(t, EV, ST, \mathcal{R}, RQ)^I = P(t, EV, ST, \mathcal{R}, RQ)^{\texttt{tr(caused,EV(t))}}.$$

It follows via Lemma A.1 that $EV(t)$ is the least set satisfying Conditions 1 to 3 of Definition 3.8, and hence

$$\texttt{Caused}(t, EV, ST, \mathcal{R}, RQ) = EV(t).$$

This proves that 1 holds. □

LEMMA A.3. *If $\mathcal{R}$ is safe then $P(t, EV, ST, \mathcal{R}, RQ)$ is stratifiable.*

PROOF. Suppose not; that is, $\mathcal{R}$ is safe but the dependency graph of $P(t, EV, ST, \mathcal{R}, RQ)$ contains a cycle containing a negative edge.

Note that by the structure of $P(t, EV, ST, \mathcal{R}, RQ)$, each cycle must go through some node tr(caused, p:E), so there are two possibilities. For some edge e in the cycle, either

$$e = \langle \mathtt{tr(caused, p{:}E)}, +, \mathtt{tr(valid, p_1{:}E_1)} \rangle, \text{ or} \tag{2}$$

$$e = \langle \mathtt{tr(caused, p{:}E)}, -, \mathtt{tr(not\_blocked, p_1{:}E_1)} \rangle. \tag{3}$$

In Case (2), the edge following e in the cycle must necessarily have the form

$$e_1 = \langle \mathtt{tr(valid, p_1{:}E_1)}, +, \mathtt{tr(caused, p'{:}E')} \rangle \tag{4}$$

while in Case (3), e must be necessarily followed by two edges of the form

$$e_2 = \langle \mathtt{tr(not\_blocked, p_1{:}E_1)}, +, \mathtt{tr(valid, p_2{:}E_2)} \rangle \tag{5}$$

$$e_3 = \langle \mathtt{tr(valid, p_2{:}E_2)}, +, \mathtt{tr(caused, p'{:}E')} \rangle. \tag{6}$$

Note that (by Definition A.2) in the former case $DG_{\mathcal{R}}$ must contain a corresponding edge $\langle \mathtt{tr(caused, p{:}E)}, +, \mathtt{tr(caused, p'{:}E')} \rangle$ while in the latter $DG_{\mathcal{R}}$ must contain $\langle \mathtt{tr(caused, p{:}E)}, -, \mathtt{tr(caused, p'{:}E')} \rangle$. By repeating the same reasoning along the cycle, we obtain a corresponding cycle in $DG_{\mathcal{R}}$. Moreover, since the original cycle contains a negative edge, situation (3) must occur at least once, therefore the cycle in $DG_{\mathcal{R}}$ contains a negative edge, too. But then $\mathcal{R}$ cannot be safe, a contradiction. □

Main Results

THEOREM 4.1. *If a REB $\mathcal{R}$ is safe, then for all request streams RQ and for all $S \subseteq$ Roles, there exists exactly one execution model $\langle EV, ST \rangle$ of $\mathcal{R}$ and RQ such that $ST(0) = S$.*

PROOF. First, we prove that such an execution model exists. Let $\langle EV, ST \rangle$ be the unique trace such that

$$ST(0) = S$$
$$EV(t) = \{p{:}E \mid \mathtt{tr(caused, p{:}E)} \in I(t)\},$$

where $t \geq 0$ and $I(t)$ is the unique stable model of the stratifiable program $P(t, EV, ST, \mathcal{R}, RQ)$[6] (cf. Lemma A.3). By a straightforward induction based on Lemma A.2, we have that for all $t \geq 0$,

$$\mathtt{Caused}(t, EV, ST, \mathcal{R}, RQ) = EV(t).$$

This proves that $\langle EV, ST \rangle$ is an execution model of $\mathcal{R}$ and RQ.

To prove uniqueness, assume there exists a distinct execution model $\langle EV', ST' \rangle$. We reach a contradiction. Let t be the least index such that

[6]Note that the definition is not circular in $EV(t)$, as $P(t, EV, ST, \mathcal{R}, RQ)$ depends only on $EV(t')$ such that $t' < t$.

$EV(t) \neq EV'(t)$. Note that the programs defined in Definition A.2 depend only on the events at time $t' < t$, so by minimality of t we have

$$P(t, EV, ST, \mathcal{R}, RQ) = P(t, EV', ST', \mathcal{R}, RQ), \tag{7}$$

and $I(t)$ is also the unique stable model of $P(t, EV', ST', \mathcal{R}, RQ)$. Now, by definition of the execution model,

$$\begin{aligned} \texttt{Caused}(t, EV, ST, \mathcal{R}, RQ) &= EV(t), \\ \texttt{Caused}(t, EV', ST', \mathcal{R}, RQ) &= EV'(t). \end{aligned}$$

It follows easily, by Lemma A.2, (7), and the uniqueness of $I(t)$, that

$$\begin{aligned} p{:}E \in EV(t) \quad &\text{if and only if} \quad \texttt{tr(caused, p:E)} \in \texttt{I(t)} \\ p{:}E \in EV'(t) \quad &\text{if and only if} \quad \texttt{tr(caused, p:E)} \in \texttt{I(t)}. \end{aligned}$$

But then $EV(t) = EV'(t)$, a contradiction. □

REFERENCES

AHN, G. J. AND SANDHU, R. 1999. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control* (Fairfax, Va.), 43–54.

ATLURI, V. (ED.) 1999. *Proceedings of the Fourth ACM Workshop on Role-Based Access Control* (Fairfax, Va.).

BERTINO, E. 1998. Data security. *Data Knowl. Eng. 25*, 1–2, 199–216.

BERTINO, E., BETTINI, C., FERRARI, E., AND SAMARATI, P. 1998. An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst. 23*, 3, 231–285.

BERTINO, E., BETTINI, C., FERRARI, E., AND SAMARATI, P. 1999. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Sec. 2*, 1, 65–104.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, Mass.

FERRAIOLO, D., BARKLEY, J. F., AND KUHN, D. R. 1999. A role-based access control model and reference implementation within a corporate intranet. *ACM Trans. Inf. Syst. Sec. 2*, 1, 34–64.

GAL, A. AND ATLURI, V. 2000. An authorization model for temporal data. In *Proceedings of the Seventh ACM Conference on Computer and Communication Security* (Athens, Greece), 144–153.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth ICLP Conference*, MIT Press, Cambridge, Mass., 1070–1080.

GULUTZAN, P. AND PELZER, T. 1999. *SQL99 Complete, Really*. Miller Freeman, Kansas.

JAEGER, T., PRAKASH, A., LIEDTKE, J., AND ISLAM, N. 1999. Flexible control of downloaded executable content. *ACM Trans. Inf. Syst. Sec. 2*, 2, 177–228.

JONSCHER, D., MOFFET, J., AND DITTRICH, K. 1994. Complex subjects or: The striving for complexity is ruling our world. *Database Security VII: Status and Prospects*, North Holland, Amsterdam, the Netherlands, 19–37.

KANDALA, S. AND SANDHU, R. 1999. Extending the BFA workflow authorization model to express weighted voting. In *Research Advances in Database and Information Systems Security*, Kluwer Academic, Aingham, Mass., 145–159 .

KUHN, D. R. 1997. Mutual exclusion of roles as a means of implementing separation of duties in a role-based access control system. In *Proceedings of the Second ACM Workshop on Role-Based Access Control* (Fairfax, Va.), 23–30.

LLOYD, J. W. 1984. *Foundations of Logic Programming*, Springer-Verlag, New York.

LOBO, J. AND BARAL, C. 1996. Formal characterization of active databases. In *Logic In Databases '96*, Lecture Notes in Computer Science, vol. 1154, Springer-Verlag, New York, 175–195.

LOBO, J. AND RACHID, L. 1994. A semantics for a class of non-deterministic and causal production system programs. *J. Autom. Reason. 12*, 308–349.

NIEZETTE, M. AND STEVENNE, J. 1992. An efficient symbolic representation of periodic time. In *Proceedings of the First International Conference on Information and Knowledge Management*.

NYANCHAMA, M. AND OSBORN, S. 1999. The role graph model and conflict of interest. *ACM Trans. Inf. Syst. Sec. 2*, 1, 3–33.

OSBORN, S. (ED.) 2000. *Proceedings of the Fifth ACM Workshop on Role-Based Access Control* (Berlin).

OSBORN, S., SANDHU, R., AND MUNAWER, Q. 2000. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Sec. 3*, 2, 85–106.

SANDHU, R. 1991. Separation of duties in computerized information systems. In *Database Security IV*: *Status and Prospects*, North Holland, Amsterdam, the Netherlands, 179–189.

SANDHU, R. (ED.) 1995. *Proceedings of the First ACM Workshop on Role-Based Access Control* (Fairfax, Va.).

SANDHU, R. 1996. Role hierarchies and constraints for lattice-based access controls. In *Computer Security—Esorics '96* (Rome), E. Bertino, H. Kurth, G. Martella, and E. Montolivo, Eds., Lecture Notes in Computer Science, vol. 1146, Springer-Verlag, New York.

SANDHU, R. (ED.) 1997. *Proceedings of the Second ACM Workshop on Role-Based Access Control* (Fairfax, Va.).

SANDHU, R. (ED.) 1998a. *Proceedings of the Third ACM Workshop on Role-Based Access Control* (Fairfax, Va.).

SANDHU, R. 1998b. Role-based access control. *Advances in Computers*, *46*, Academic Press.

SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. 1996. Role-based access control models. *IEEE Comput. 29*, 2, 38–47.

SIMON, R. AND ZURKO, M. E. 1997. Separation of duty in role-based environments. In *Proceedings of the Tenth IEEE Computer Security Foundations Workshop*.

TIDSWELL, J. E. AND JAEGER, T. 2000. Integrated constraints and inheritance in DTAC. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control* (Berlin).

Received October 2000; revised July 2001; accepted July 2001