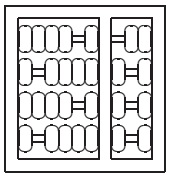

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Security Management in the presence of delegation and revocation in workflow systems

Jacques Wainer *Akhil Kumar*
Paulo Barthelmess

Technical Report - IC-01-014 - Relatório Técnico

October - 2001 - Outubro

Security Management in the presence of delegation and revocation in workflow systems

Jacques Wainer* Akhil Kumar† Paulo Barthelmess‡

2001-10

Abstract

This paper extends the RBAC model to deal with permission in a workflow management system. The extended model allows for dynamic constraints on instances of processes. We also extend the model so that delegations from an user to another user, and revocations of such delegations are dealt with. We also discuss the issues on delegations from a user to a group of users.

1 Introduction

The Role-based Access Control Model (RBAC) (for example [20]) is receiving attention as a systematic way of implementing the security policy of an organization. It groups individual users into roles that relate to their position within an organization and assigns permission to various roles according to their stature in the organization. Roles are generic terms like manager, vice-president, etc. and anybody in a role can perform certain tasks assigned to him or her.

A previous work by the authors [16] extends RBAC to the specificities of workflow system, where permissions are seen as permissions to perform tasks in a particular business procedure.

In this work we extend those ideas to include both delegation and revocation of permissions. An important characteristic of real-world systems is that often an assigned user is not available. In such a situation it is not clear what a system should do. Hence, there is an important need for delegation. An important approval should not get held up if a vice-president is on vacation or is unavailable while flying on an overseas trip. Delegation means that a user should be able to notify the system before hand about the most appropriate users to handle her tasks while she is away. Then, if the system knows that a user is unavailable, it will automatically transfer the tasks to the delegate(s). Moreover, it should also be possible to rank order the delegates based on their suitability.

*Institute of Computing, State University of Campinas, Brazil
†Database Systems Research Department Bell Laboratories, Lucent Technologies, USA
‡Department of Computer Science University of Colorado, USA

This paper is organized as follows. Section 2 describes in a somewhat succinct way the extensions the RBAC model to deal with workflow systems. Section 3 gives an architecture for a permission service and shows how it will interface with a workflow model. Then, in Section 4 we discuss delegation, in section 5 we discuss the rights to delegate, and section 6 discusses revocation of such delegations. Section 7 presents some preliminary ideas on the delegations to groups of peoples, which seems a interesting need in real life systems. In Section 8 we discuss a prototype implementation of our proposal. Section 9 discusses related works and section 10 lists the conclusions and future works.

2 Our Model

Our model is an extension of the Role Based Access Control model (RBAC). We will briefly describe the RBAC model and then state our extensions.

2.1 RBAC

The basic RBAC model can be described in terms of: 1) **entities**: *users*, *roles*, and *privileges*, 2) **relationships** between these entities, and 3) **constraints** over these relationships. A RBAC meta-model is displayed in Figure 1.

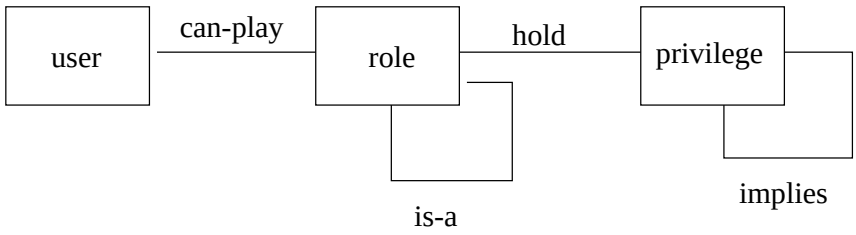

Figure 1: RBAC Meta model.

- user (U) represent individuals users.
- privilege (P) represent classes of rights to perform operations, tasks, access data and so one, possibly with explicit attributes. For example, TRAVEL-APPROVAL(US$500) represents the class of all approval of travel expenses tasks up to US$500.
- role (R) describe meaningful groupings of privileges that can be assigned to users, e.g., the role of *manager*.

The basic relationships are:

- can-play(U,R), state that the user U can play the role R. can-play(mary, manager) says that Mary plays the role of manager.

- is-a(R1,R2) states that role R1 inherits all privileges of role R2. This allows for a more economic description of roles, a *c-programmer* role inherits all privileges form a *programmer* role.
- hold(R,P), state that role R holds certain privilege P
- imply(P1,P2), state that privilege P1 is stronger, or supersedes, or includes P2. For example the right to approve travel expensed up to US$1000 implies (is stronger than) the right to approve travel expenses up to US$500.

Actual instantiations of *user*, *role* and *privilege* entities and their relationships will determine which are the rights of each user at a particular moment in time. If, e.g., Mary can-play the role *manager*, and a manager role hold the privilege to *approve travel expenses*, this will be represented by having an *user* instance describing Mary, a *role* instance describing managers, and a *privilege* instance describing approval of travel expenses. These three instances are related to each other through the can-play and hold relationships. Each instance can (and probably will) be related to other instances of the same type through subordinate and imply relationships, respectively. Furthermore, if Mary is the only user that can-play manager and manager is the only role that hold the privilege to approve travel expenses, no other user but Mary will be able to perform this action. In other words, the model specifies what is allowed. What is not represented is assumed to be forbidden.

2.2 Constraints in RBAC

The model described above can confer broad privileges on users and occasionally this may not be desirable. Hence, we need a mechanism to fine-tune the model. For example, some instantiations of the model may cause "conflicts of interest" [17]) in the form of incompatible privileges. The classic case occurs when a traveler who is claiming travel reimbursement ends up receiving the privilege to approve his/her own expense claim. This anomalous situation could result if the traveler's manager transfers to the traveler all the privileges that belong to the manager herself. In this case there is clearly a need for enforcing an exception in this special case. This can be done by treating the privilege to *request a reimbursement* and to *approve* it as *incompatible privileges*.

Thus, *constraints* allow us to impose limitations on actual instantiations in a systematic manner, according the some security policy of an organization. It should be noted clearly that the constraints *override* or *supersede* the permissions allowed by the general model. Modifications that invalidate a constraints are prevented by the security system. An attempt to include a relationship between a role, say *clerk* and two or more incompatible privileges, e.g., *request* and *approve*, will be blocked by the security system. By enforcing constraints, the security system guarantees that the model is consistent at all times.

Constraints on these objects are represented as integrity constraints on the relationships. Generic integrity constraints can capture information about the cardinality of the relationships, such as: *all roles must have at least one user*, *all users must play at least one role*, *all privileges must be held by at least one role*, etc. Also organization specific static constraints can be captured as *integrity constraints*, such as: role R3 must have exactly three users, the privilege of doing task T3 must only hold for the role VP and their superiors.

We will represent an integrity constraint as

$$\bot \leftarrow C \tag{1}$$

where C describes an invalid situation.

The constraint is expressed as a standard logic program clause, that is, a constraint is expressed as

$$\bot \leftarrow A_1, A_2, \ldots A_k \textbf{ not } B_1, \textbf{ not } B_2, \ldots \textbf{ not } B_l$$

where either k and l may be zero, but not both. A_i and B_j are atomic terms of the form $p(t_1, t_2, \ldots, t_m)$ where p called a predicate, is either one of the relations defined above, or a relation recursively defined based on those relations, m is the arity of the predicate p, and t_i, called terms, are either variables, taken to be existentially quantified, or constants that represent the instances of the concepts described above (users, roles, or privileges).

Thus the constraint that no one should be able to hold both the privileges P1 and P2 is expresses as:

$$\bot \leftarrow \mathsf{can\text{-}play}(u, r1), \mathsf{can\text{-}play}(u, r2), \mathsf{hold}(r1, \mathrm{P1}), \mathsf{hold}(r2, \mathrm{P2}) \tag{2}$$

The formula above should be read as "the following situation is invalid: there exists a user u and two roles $r1$ and $r2$ not necessarily distinct, such that the user can play both roles, and one of the roles hold privilege P1 and the other P2"

2.3 Extensions to the model

Within RBAC it is possible to describe and enforce what we will call **static constraints**. Static constraints may be too coarse to express the exact limitations. Let us go back to the travel reimbursement example. The static constraint may express that no user may have both the privilege of *request* and *approve* a travel reimbursement. But that is not exactly what is needed; if Alice has the right to approve reimbursements, she will not be able to travel and herself ask for reimbursement. What is needed in this case is a the concept of a **instance** of reimbursement: what cannot happen is that the same user have both the privileges to *request* and *approve* the same instance of a reimbursement. Thus if Alice travels she cannot both request and approve the reimbursement, but someone else must approve it.

This concept of a instance of a travel reimbursement is very natural in workflow systems, it is an instance of the process, or a case. The revised meta model is described in figure 2. Until section 5 the only privilege we are interested in the right to perform a task in a process, and thus we will use (the right to perform a) task interchangeable with privilege.

We also include to the model the relation doer(U,T,C) that state that a user (U) exercised a particular task (T) on a particular instance (C). In particular we feel that there is no need to extend the doer relation into a 4-tuple, which would involve the role the user U was playing when the exercised the privilege P on instance C.

Dynamic constraints, or history based constraints do not allow users to perform actions on instances, depending on the *history* of past execution. A user may make an

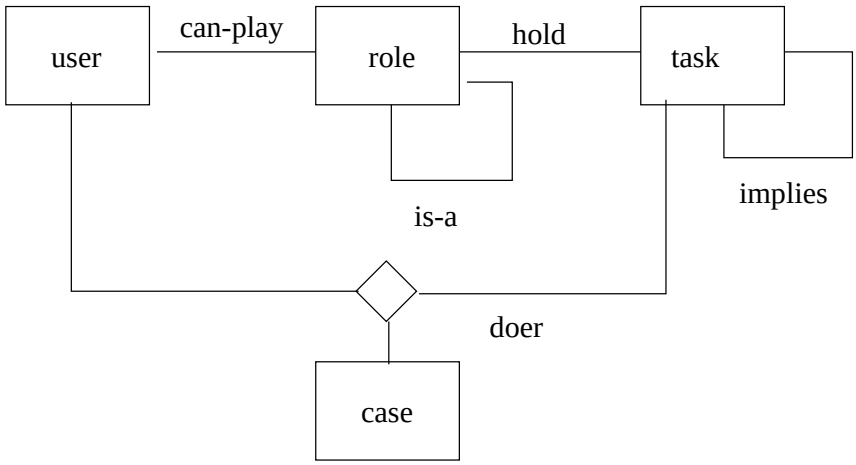

Figure 2: Meta model.

request, and she may approve a request, but she cannot make and approve the same request. Thus if she was the one that made the request she will not be allowed to approve it.

History based constraints can be either *negative* or *positive* - users and roles are either blocked from performing some actions, and/or required to perform specific actions on an instance, respectively, depending on their previous actions over that instance:

- *Dynamic separation of duties* can prevent the user who executed an action from performing another mutually exclusive one as well, for instance, an *approval*, whenever he or she performed a *request*. For example, if T2 and T4 cannot be done by the same person:

$$\bot \leftarrow \mathsf{doer}(u, \mathrm{T2}, c), \mathsf{doer}(u, \mathrm{T4}, c) \tag{3}$$

- *Binding of duties* is just the opposite - a user that performed some action is bound to execute other related actions in the future. The rationale is that by performing the first action, the user has acquired knowledge that will be required or useful while performing the related ones. For example if T2 and T3 must be performed by the same person:

$$\bot \leftarrow \mathsf{doer}(u, \mathrm{T2}, c), \mathsf{doer}(u', \mathrm{T3}, c),\ \mathbf{not}\ u' = u \tag{4}$$

History based security matches quite naturally workflow systems, that usually employ role based mechanisms independently of security reasons, as a means to distribute work. History is also naturally kept by most workflow systems as well, for auditing and recovery purposes. As a consequence, the bulk of the workflow related security is concerned with history based mechanisms, e.g., [7, 8, 18, 15].

3 A permission system integrated with a WFMS

The basic framework is of a *permission service*, attached to a workflow engine. The workflow system contains the knowledge about the processes, the ordering of activities, deadlines, and the so on. The permission service knows about the organizational structure, roles, permissions, and so on.

The workflow system communicates with the permission system through two channels. The first channel is used to inform the permission system of the history of the process instances. The workflow provide facts of the form:

- doer(U,T,C), which states that user U has done the task T for the process instance C.
- done(C) which state that the instance C is terminated.

The second channel is used by the workflow system to query the permission system. A basic query asks which users can perform a particular task for a particular instance. The workflow system sends the query who?(P,O,T,C), that is, who are the users among the ones that satisfy the predicate P that can perform the task T for instance C. The workflow receives back an ordered list of users that satisfy all constraints, ordered according to O. We will discuss the predicate P and the order O below.

3.1 Answering the queries from the workflow system

With the concepts presented so far, we can define what is the appropriate answer to the query who?(P,O,T,C) posed by the workflow. The permission system should return all users u for which P(u) is true, for which there exists a r such that can-play(u,r) and (hold(r,T), and for which doer(u,T,C) does not violate any integrity constraint. Furthermore, the permission system should order the users according to the order O to be explained shortly. First, we describe predicate P.

P is an **extended role** if P is a one-place predicate constructed by using existential quantifiers, "and," "or," and "not," (where *not* is understood as negation as failure) from the relations, predicates and constants defined for the permission system. Thus an extended role is a logic program query that specify a set of users from which that the workflow expects the subset of potential executors of the task is included.

The idea is that the workflow system may already incorporate some constraints regarding the executors of tasks. A task of approval of the process of travel reimbursement may, according to the workflow, be executed by a manager if the amount is less that $10000.00 but only by a department head if the amount is greater than that. These constraints may conflict with the information expressed in the permission system, which may specify that approvals of any reimbursement above $5000.00 are to be performed by department heads. Such conflicting information is not desirable but expected.

The solution is that both the constraints of the permission system and the workflow are taken into consideration: only users that satisfy the extended role, and thus the workflow constraints, are further selected by the permission system constraints.

The case in which the separation of duties between the workflow and the permission system is total, that is, the workflow knows about ordering of tasks and the permission system knows about rights, can still be contemplated. In this case the extended role will be a query that selects all users.

An example of extended role is $\lambda x.[\mathsf{can\text{-}play}(\mathrm{x}, VP),\ \ \mathbf{not}\ \mathrm{x} = mary]$ This predicate is true for all VPs but Mary. The λx notation is to make it explicit what is the variable that is being queried.

Elsewhere [16] we discuss the importance of ordering according to some different criteria the set of users that may perform a task. For example, the system may order such users by such that the least "important" user that is able to perform the task is given higher priority. For example, if both a group manager and a department manager can approve a reimbursement of $ 5000.00, it may be more appropriate to prefer the group manager, which is less "important" in the subordinance sense to the department manager.

The idea is that the permission system is really the knowledge base of organizational information, which include constrains, and the user/role/permissions relations, which has been discussed above, but may also include other relations, such as subordinance, and so on.

Even a general rule such as "less important users should be preferred" have different definitions, which result in different orderings. The workflow should select one of such definitions, and ask the permission system to order the possible users that can perform the task for the case according to it.

3.2 Answering the workflow system

Finally we can define what is the answers to the query $\mathsf{who?}$(P,O,T,C).

Definition 1 *The permission system's* **answer** *to the workflow query* $\mathsf{who?}(P,O,T,C)$ *is the ordered list of users* $\langle u_1, u_2, \ldots u_n \rangle$ *such that*

- $P(u_i)$ *is true*
- *there exists an* r_i *for each* u_i *such that* $\mathsf{can\text{-}play}(u_i, r_i)$ *and* $\mathsf{hold}(r_i, T)$
- $\mathsf{doer}(u_i, T, C)$ *does not violate any constraint*
- $O(u_i, u_{i+1})$, *for* $0 \leq i < n$
- *for all* y *that satisfy the items above, then there is some* u_i *in* $\langle u_1, u_2, \ldots u_n \rangle$ *such that* $y = u_i$.

4 Delegation

In general, delegation needs can be quite complex. Improperly implemented delegation mechanisms can result in heavy costs to a company. A delegation system should be both flexible and at the same time prevent incorrect delegations from being made inadvertently. We wish to address various types of delegation requirements in a common framework. In the

subsequent subsections we discuss various kinds of delegation services that are supported in our framework.

4.1 Specific delegations

We will summarize delegations into a single predicate: delegate(U1,U2,T,C) state that user U1 transfer his rights of performing task T for case C to user U2. We will call U1 the **grantor**, and U2 the **delegate**.

The semantics of delegation will be divided into two distinct moments. A delegation is **accepted** if it does not violate any static constraint, and if the grantor has the right to perform such delegation. We will discuss more of the issue of right to delegate in the next section. For the moment, it suffices to say that an accepted delegation does not violate static constraints. The second moment in the semantic of delegation is the **execution** when the delegation causes a change of attribution: if U1 delegates to U2 the task T of C then when asked who can perform T for C, the permission system (in some cases and after some checking) remove U1 from that list and include U2. The acceptance and the execution of a delegation may happen in very different moments.

4.2 Generic delegations

A **generic delegation** is a statement if the form delegate(U1,U2,T), and should be understood as a delegation of all future cases, that is

Definition 2 *delegate(U1,U2,T)* $= \forall c$ *delegate(U1,U2,T,c).*

A generic delegation is accepted if U1 has the right to perform T (and has the right to delegate T – see section below). A generic delegation is valid until it is revoked (section 6) and as soon as a new case D is created, it should be understood that a delegate(U1,U2,T,D) is entered into the accepted delegations. From there on, the behavior of a generic delegation is the same as a specific delegation.

4.3 Semantics of delegation

In order to introduce the issues on the semantics of delegation, let us explore some alternatives regarding the previous rights of both the grantor and the delegate. In general a delegation delegate(U1,U2,T,C) can be "bad/wrong" because of "static" or "dynamic" reasons. We will call a static reason as **empty** and dynamic reasons as **void**. A delegation can be:

- **grantor empty** or **g-empty** if the grantor has no right to the execute the task T, that is
$$\nexists r \; \mathsf{can\text{-}play}(U1, r) \wedge \mathsf{hold}(r, T)$$
- **grantor void** or **g-void** if the grantor has no right to execute T on C, but has the right to execute T. Formally adding $\mathsf{doer}(U1, T, C)$ to the knowledge base violates some constraint

- **delegate empty (d-empty)** if the delegate has no previous right to perform T, that is

$$\nexists r \ \mathsf{can\text{-}play}(U2, r) \wedge \mathsf{hold}(r, T)$$

- **delegate void (d-void)** if the delegate could not perform T on C because of constraint violation, that is doer(U2,T,C) would violate a constraint.

Clearly, g-empty delegations seems to convey something "wrong", if the person cannot do the task T how can she delegate it so someone else? This is an example of a delegation that cannot be accepted because it does not match to ones common sense definition of delegation. In such cases we will say that the delegation statement is **not accepted**, that is, an error message is returned to the user and the statement is not entered into the system.

G-void delegations seems also to be "wrong:" if U1 cannot perform T on C she cannot delegate others to do it. In fact since U1 is not one of the persons that can execute T on C, the delegation will never be executed. But the delegation may be accepted never the less: when the delegation was entered into the system, there was not enough information to state that the grantor would not be able to perform the task T, and thus by then the delegation must be accepted.

Both d-empty and non d-empty delegations are seem reasonable. In fact d-empty delegations are the "real" ones: the delegate received new rights by the delegation. A non d-empty delegation means that the delegate already had the right to perform the task, and thus the delegation did not change that. If the delegate, because some dynamic constraint, cannot perform the task, then the delegation will not change that (as we will see below in the discussion of g-void delegations). Thus a non d-empty delegation may only change the order in which the delegate will appear in the answer to the workflow, because one may define a ordering of solutions in which a delegate inherits the position of the grantor.

D-emty delegations closely correspond to the intuitive notion of delegation: of someone that gained rights to do something because of the delegation.

D-void delegations are similar to g-void delegations: if the delegate cannot perform the task on the case, it is not because of the delegation that she should now be able to do so. But again, at the moment the delegation was entered into the system, such information may not be available, and thus the the delegation must be accepted. Only the execution semantics of the delegation would preclude it from being executed.

4.4 Answering the workflow

In fact the execution semantics of delegations are better understood as a part of the computations needed to answer a who? query from the workflow system, and some auxiliary definitions are needed.

Definition 3 *A* **chain of delegation** *on task T for case C is a sequence of users* $\langle a_0, a_1, a_2, \ldots a_n \rangle$ *where* $n \geq 0$*, such that either* $\mathsf{delegate}(a_i, a_{i+1}, T, C)$ *or* $\mathsf{delegate}(a_i, a_{i+1}, T)$ *are accepted delegations.* a_0 *is called the* **head** *of the chain, and* a_n *the* **end***.*

A **maximal** *chain of delegation on task T for case C is a chain* $\langle a_0, a_1, \ldots, a_n \rangle$*, and there is no* a_{n+1} *such that either* $\mathsf{delegate}(a_n, a_{n+1}, T, C)$ *or* $\mathsf{delegate}(a_n, a_{n+1}, T)$ *are accepted delegations.*

We will define the answer to the query who?(P,O,T,C) in a constructive way:

begin
  $Sol := \{u_i | \mathrm{P}(u_i) \wedge \exists r \mathsf{can\text{-}play}(u_i, r) \wedge \mathsf{hold}(r, \mathrm{T})\}$
  $Chains :=$ set of all maximal chains of T on C whose head are u_i
  for $c_i = \langle a_0^i, a_1^i, \ldots, a_n^i \rangle, \in Chains$ **do**
    if $\exists a_j^i \in c_i$ such that $\mathsf{doer}(a_j^i, \mathrm{T}, \mathrm{C})$ violates some constraint
      then next
      else $Sol := Sol - \{a_0^i\} \cup \{a_n^i\}$ **fi**
  od
  $Sol :=$ sort(Sol, O)
  return Sol
end

This definition entails that:

- if any participant in a chain of delegation is void (d-void or g-void) the whole chain is disregarded (alternative: stop the chain at the right most participant that is not void).
- a head of a chain that has a void participant remains as part of the solution (alternative: the head of a chain is always not part of the solution)
- if a user is the head of many chains of delegation, all ends of these chains are part of the solution (alternative: select one chain randomly and only its end will be part of the solution).

5 Right to delegate

Trust is not transitive: if Amanda trusts Beth and Beth trusts Charlotte, that does not mean that Amanda trusts Charlotte. This non-transitivity is carried along to delegation: Amanda may trust Beth to delegate to her a task, but would not want her to delegate it further to Charlotte. Thus here must be a way of controlling chaining of delegations.

In our model delegation is a right and thus it fits as a subclass of our privilege class. The privilege delt(T) allows one to delegate the task T of a particular case to (almost) anybody else, that is, it allows d-empty delegations of task T (of C), where as dell(T) allows for non d-empty delegations. delt is after **t**otal delegation, and dell, after **l**imited delegation. And we will user del* to refer to a delegation if the difference between total aand limited delegations are not relevant.

If Amanda hold the privilege delt(T) (and the privilege T) she can delegate T for case C to Beth. But she may chose not to further delegate delt(T) to Beth. In this case Beth can perform the job but cannot further delegate it. Or Amanda may delegate not only T to Beth but also delt(T); in this case Beth can further delegate (including d-empty delegations) T to whomever she chooses.

We can now complete the definition of an accepted delegation:

- delegate(U1,U2,T,C) is accepted if:
  - can-hold(U1,T,C) and
  - doer(U1,T,C) does not cause constraint violation and
  - doer(U2,T,C) does not cause constraint violation and
  - can-hold(U1,delt(T),C) or can-hold(U1,dell(T),C)
- delegate(U1,U2,delt(T),C) is accepted if:
  - can-hold(U1,delt(T),C)
- delegate(U1,U2,dell(T),C) is accepted if:
  -
  - can-hold(U1,delt(T),C) or can-hold(U1,dell(T),C)

can-hold(U,P,C), is a derived predicate that state that user U can hold privilege P for case C, either directly or through a chain of accepted delegations. The recursive definition of can-hold(U,P,C) is:

- there exists a role r such that can-play(U,r) and hold(R,P)
- there exists a user x such that can-hold(x,P,C) and there exists an accepted delegation delegate(x,U,P,C)

6 Revocation

Revocation is the process by which a delegation that was accepted is removed. But there is an issue on what to do with the consequences, more specifically future consequences of a delegation. Let us examine the details.

The central issue in revocation, in our understanding, is that a delegation may increase the potential rights of a user (potential because it may the case that dynamic constraints would not allow the user to make use of the rights), and thus revocation must remove all potential rights that users gained, because of that delegation. For example if A delegates T and del1(T) to B, and B delegates T and del2(T) to C, and A revokes both the task and del1 delegations, then what happens to the delegation from B to C depends on whether B already has the rights to T and del1(T), or, in our terminology whether the two delegations were d-empty or not. Let us suppose that the two delegations where not d-empty. Therefore the delegation from B to C was possible because of the extra rights brought about by A's delegation; by revoking A delegation, B's delegation should also be revoked. That would be so even if B already had the right to perform T; it was because of A's delegation of del1(T) that B could further delegate it to C.

But if B already has the right to T and say somebody else had delegated del1(T) to B (say A'), then A's revocation should not cause the change in B's delegation: at the time of the revocation B had the appropriate rights to make the delegation to C.

To formally define the revocation of a delegation we must refine the definition of chain of delegation to include sequences of delegations of del* rights.

A **chain of del*-delegations** is a sequence of users $\langle a_0, a_1, a_2, \ldots a_n \rangle$ where $n \geq 0$, such that, there exists j such that $0 \leq j \leq n$, and for all $i \leq j$, there is an accepted delegation of the form delegate(a_{i-1}, a_i,delt(T),C), and for $i > j$ there is an accepted delegation of the form delegate(a_{i-1}, a_i,dell(T),C) That is, a chain of del*-delegation of del* is possible provided:

- all refer to the same task T
- all refer to the same case C
- all delt occur before all dell.

Let A the the set of accepted delegations. A particular delegation is **grounded** on the set A if it would be accepted if A is all of the delegations. A delegation may be in A but not grounded in A: to be in A it must have been accepted when it was entered, but because of revocations that happened since the delegation was accepted, if entered now, the delegation would not be accepted.

Formally, in a constructive definition of revoke(delegate(U1,U2,T,C)) is:

begin
  A is the set of all accepted delegations
  $C :=$ set of all maximal chains of T on C in which U1 is the head
  $A := A - \{$delegate(U1,U2,T,C)$\}$
  for $c_i = \langle a_0^i, a_1^i, \ldots, a_n^i \rangle, \in C$ **do**
    if $\exists a_j^i \in c_i$ such that delegate(a_j^i, a_{j+1}^i, T,C) is not d-empty $\wedge$ $\nexists z$ delegate(z, a_j^i, T,C) $\in A$
      then $A := A - \{$delegate(a_j^i, a_{j+1}^i, T,C)$\}$ **fi**
  od
end

The constructive definition of revoke(delegate(U1,U2,del*(T),C)) is:

begin
  A is the set of all accepted delegations
  $C :=$ set of all maximal chains in which U1 is the head
  $A := A - \{$delegate(U1,U2,D,C)$\}$
  for $c_i = \langle a_0^i, a_1^i, \ldots, a_n^i \rangle, \in C$ **do**
    for $a_j^i \in c_i$ sequentially **do**
      if delegate($a_j^i, a_{j+1}^i, del*(T), C$) is not grounded in A
        then $A := A - \{$delegate($a_j^i, a_{j+1}^i, del*(T), C$)$\}$ **fi** **od**
  od
  revoke(delegate(U1,U2,T,C))
end

7 Delegation to a group

In some real process situations it is common that a grantor does not delegate a task to a particular user, but to a set, or group of users. Thus for example a VP may delegate a task, or more specifically a decision, only she can perform, to two of her department head, with the proviso that they must make the decision separately and but must both agree on the outcome.

This example shows that in order to delegate to a group, it is usually necessary to specify a sub workflow on how the members of the group will execute the task.

First, we will extend the meta model to include objects that represents groups of users, as part of the class user and define a relation member between a user and a group its belongs to. We will consider three types of group definitions: groups consisting of specific individuals, groups consisting of a generic role, and groups consisting of other groups. These three kinds of groups are defined using predicates called member1, member2, and member3. Thus *g1* may represent a group and to state that *john* and *james* are members of the group, one would assert both member1(*john*,*g1*) and member1(*james*,*g1*)

In addition to group delegation to named individuals as above, it is also useful to introduce the notion of a group consisting of certain number of individuals from a role. Thus, it should be possible to specify a virtual group of (any) three VPs as follows: member2(Role, Number, Group-name). So, for instance, g2 is a group that must consist of three vice-presidents and it is defined as member2(vp,3,g2). The semantics of this definition are as follows:

$$\mathsf{member2}(R, N, G) = \forall U1, U2 \in G, \mathsf{can\text{-}play}(U1, R) \wedge \mathsf{can\text{-}play}(U2, R) \wedge U1 \neq U2 \wedge size(G) = N$$

Above, we assume that the size function returns the cardinality or size of a group. Similarly, member2(director,2,g3) defines a group of two directors, named g3.

The third type of group definition is used to form unions of groups. Thus, the member3 predicate is used to specify that a group is a subgroup of another group as follows: member3(g2,g4) means that group g2 is contained in group g4. Similarly, member3(g3,g4) adds group g3 also to g4. Thus, together these two predicates result in g4 being a union of g2 and g3.

Now, a delegation to a group may be expressed as group-delegate(U1,g1,T,C). In this case the grantor is U1, the delegates are members of group g1, and they are given the permission to perform task T on case C. For example, user Mary may group-delegate task T1 to the above group g1 as follows: group-delegate(mary, g1, T1,C). The workflow system must verify that all members of g1 have performed a task in order for it to be considered as completed. In our example, both John and James must complete the task in order for it to be considered as done. In this example, g1 was a type 1 group. Delegations to type 2 and type 3 groups may also be handled similarly. Thus, group-delegate(U1,g4,T) would delegate a task to a union of any 3 vice-presidents and 2 directors.

7.1 Subdelegation in the context of group delegation

The discussion of subdelegation in earlier sections was in the context of individual delegations. Some of the issues of subdelegation in the group context are similar. Clearly, it is necessary to ensure that long chains of subdelegation are prevented. However, there are other considerations as well. For instance, consider a case where a vice-president has delegated certain of her powers to three named directors. If each of these directors has further delegated their own powers to three managers respectively, this would create a situation where three managers would receive the approval powers of the vice-president. Therefore, in group delegation, the dilution of power can occur much faster. In this sense, subdelegation assumes greater importance in this context. There are two mechanisms to prevent unintended consequences:

- uniformly disallowing subdelegation of group tasks.
- restricting subdelegation powers of group tasks by imposing constraints.

The first option is very drastic, while the second one is more flexible. Therefore, in the above example, one may visualize a scenario where the group of three directors may subdelegate the work such that one manager and two directors could perform it. Thus, it is important to ensure that each member of a group does not perform individual subdelegations. However, a group as a whole can subdelegate tasks to another group. In this way, the complexities of interactions between individual members would not arise.

7.2 Revocation of group delegation and other issues

Revocation of delegation was discussed in a previous section, also in the context of individual delegation. The major consideration in revocation relates to subdelegation and chaining. Since, we are treating a group as an atomic unit for subdelegation purposes, the issues of revocation will be very similar. Therefore, the revoke command would be used in the same manner as before, and it would take away the delegated authority from a group and all its subdelegates.

There are also additional issues of consistency and integrity to be addressed in the group context. For instance, while two groups that receive delegated powers may overlap (e.g., Mary may delegate to a group of James and John, and another group of James and Sue), yet one group should not be a subset of another group because then the larger group is meaningless. Similarly, it would not make sense to delegate a task to a group of three directors and also to three managers at the same time. By role subsumption, the former delegation would be unnecessary, since it is implied by the latter delegation.

8 Implementation

A prototype of the system was implemented in Prolog. All relations of our extended RBAC model, the delegate relations, are stored as Prolog facts. The constraints, both static and dynamic are stored as a Prolog rule, in which the head is a special predicate `violation`, which represents the $\perp$ symbol. The translation of the constraint to Prolog syntax is direct.

To verify if an instance of a relation , for example, doer(U,T,C), violates any constraint, one asserts the fact in the Prolog database and verifies if `violation` is provable. Although, theoretically, this computation can be in the worst case, exponential in the number of facts in the Prolog database we did not find it to be time consuming. Furthermore, the optimizations can be performed so that some parts of the computations are done in advance, before the query. We have not experimented with these optimizations.

The components of the permission system that deals with delegations and revocations, implement in Prolog the algorithms described in section 4 and 6. Again we find that these computations are not time consuming.

9 Related work

The term *delegation* is usually employed in the security literature to describe transfer of rights from some user to a machine, that then acts as a surrogate for that user (as in an ATM transaction, for instance). Here, we restrict our analysis of related work to the few papers that employ the term in the same sense as we do, as user to user delegation.

Kumar [14] introduces issues of user to user delegation support in workflow systems and related security implications. Delegated powers can represent just a subset of those possessed by the grantor. Such restriction can be based on monetary amounts, document access restrictions, and partial delegation of privileges. Subdelegation and chains of delegation are also discussed. The present paper extends this work by exploring in further detail delegation and revocation in presence of dynamic constraints, and delegation to groups and corresponding revocation issues.

Barka and Sandhu [4] presents a framework with the objective of identifying interesting cases that could be used for building role-based delegation models. Some characteristics of delegation are identified and their exhaustive combination is analyzed. A reduced set of useful combinations is then determined. This work cannot be directly compared to the one presented here, since it is not a security model, but rather provides a list of features that would be addressed by a model. In terms of the nomenclature defined in [4], our model addresses temporary, non-monotonic, partial, self-acted, multi-level, group, unilateral, cascading and grant-dependent delegations. With respect to the lattice that is presented in [4], our model can handle the permanent delegation either though a temporary delegation that never expires and is never revoked or through the intervention of some security administrator. We consider that the basic premise of the delegation is that the delegate should act as a substitute for the grantor, therefore only non-monotonic delegation is considered. Our model tackles multi-step delegation, which was excluded by Barka and Sandhu in their paper. Finally, [4] is restricted to the static cases without history, i.e., the issues that are related to dynamic constraints are not considered at all, given that it is based on a conventional RBAC model, and not on a workflow related one.

In [5], Barka and Sandhu propose the RBDM0 model (role-delegation model zero). This model is an extension of RBAC (more specifically of the RBAC0 model of the RBAC96 family [19]) that includes delegation. In practical terms, being derived from RBAC0 means that the model is restricted to flat roles (no hierarchies). RBDM0 is based on one-step, total

delegation (of all rights attached to a role); revocation is either by an expiration mechanism or by any member of the same role as the grantor. Delegation is controlled by a can-delegate relation. Some extensions are discussed: grant-dependent revocation, delegatable and non-delegatable permissions, and two-step delegation, as well as delegation in hierarchical roles. Even with extensions, the model does not address the dynamic issues, the partial and group types of delegation that are addressed by our model.

Bertino et al. [7, 6] proposes a constraint based security model for workflow systems, whose language allow for more expressivity than the one presented here. Their language and model refers to the many instances of the activation of a task within a case, which was not considered here, and their work discuss in details optimizations that can be performed offline so that queries performed on line would have a good chance of being answered quickly (but the worst case is still exponential). Delegation is not addressed in any form by [7, 6].

Security in workflow systems are also considered e.g. by Castano et al. [8], Atluri and Huang [2, 3, 11], Karpalem and Hung [12, 13], Pernull and Herrman [10, 9] and the Meteor project [15, 21, 1].

Each of the models mentioned above explore different aspects of workflow related security, but none addresses delegation issues, the focus of the present paper.

10 Conclusions and future work

In this paper, we extended the RBAC model to develop our proposed model. Our work is based on the premise that the current RBAC model does not handle delegation well and has several shortcomings in this respect. Consequently, we propose two improvements. Firstly, by incorporating dynamic delegation, we give users the ability to transfer their tasks on-the-fly to an appropriate substitute. In addition to finding substitutes, we also rank order them in terms of appropriateness. The current RBAC framework does not provide for such a capability. Secondly, we extend the notion of delegation to groups. For instance, a vice-president may wish to delegate her authorities to a group of 2 directors and 2 managers. This feature requires novel ways of modeling groups and managing the delegation process. Therefore, our contributions add to the RBAC model in a substantial way.

The issues regarding delegation to groups and revocation of such delegations are not yet fully explored. In particular we did not extend the definitions of the semantics of delegations and revocations to such cases; that remains to be done.

This paper did not address any of the concerns of management of the permission system. RBAC model already addresses that in its ARBAC version. Some of those ideas can be incorporated here, but other issues such as management of the constraints, for example how to determine when constraints are mutually inconsistent has not been dealt with.

The management of delegations, at least in a first approximation, seems not to be difficult. We envision that each user will enter his/her own delegations statements into the system. Delegations that are not accepted, cannot be entered, and that includes cases in which the user has no right to make the delegations in the first place (because he does not hold the appropriate del^* permission).

References

[1] Gail-Joon Ahn, Ravi Sandhu, Myong H. Kang, and Joon S. Park. Injecting rbac to secure a web-based workflow system. In *5th ACM Workshop on Role-Based Access Control*, Berlin, Germany, July 2000. http://citeseer.nj.nec.com/ahn00injecting.html.

[2] V. Atluri and W-K. Huang. An authorization model for workflows. In *Proc. of the Fifth European Symposium on Research in Computer Security*, number 1146 in Lecture Notes in Computer Science, pages 44–64. Springer-Verlag, 1996.

[3] V. Atluri and W-K. Huang. A petri net based safety analysis of workflow authorization models. *Journal of Computer Security*, 1999.

[4] Ezedin S. Barka and Ravi Sandhu. Framework for role-based delegation models. In *16th Annual Computer Security Applications Conference*, December 2000. http://www.acsac.org/2000/abstracts/34.html.

[5] Ezedin S. Barka and Ravi Sandhu. A role-based delegation model and some extensions. In *23rd National Information Systems Security Conference*, October 2000. http://csrc.nist.gov/nissc/2000/proceedings/papers/021.pdf.

[6] E. Bertino, E. Ferrari, and V. Atluri. A flexible model supporting the specification and enforcement of role-based authorization in workflow management systems. In *Proc. of the second ACM workshop on Role-based access control*, pages 1–12, 1997.

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

[8] S. Castano, F. Casati, and M. Fugini. Managing workflow authorization constraints through active database technology. 1999.

[9] G. Herrmann. Security and integrity requirements of business processes - analysis and approach to support their realisation. In *Proc. of CAiSE*99 6th Doctoral Consortium on Advanced Information Systems Engineeering*, pages 36–47, 1999.

[10] G. Herrmann and G. Pernul. Viewing business-process security from different perspectives. *International Journal of Electronic Commerce*, 3(3):89–103, 1999.

[11] W-K. Huang and V. Atluri. Secureflow: A secure web-enabled workflow management system. In *Proc. of the fourth ACM workshop on role-based access control on Role-based access control*, pages 83–94, 1999.

[12] P.C.K. Hung, K. Karlapalem, and J.W. Gray III. A study of least privilege in capbasedams. In *Proc. of the 3rd IFCIS International Conference on Cooperative Information Systems*, pages 208 – 217, 1998.

[13] K. Karpalem and P. Hung. Security enforcement in activity management systems. In *Advances in Workflow Management Systems and Interoperability*, pages 166–194, 1997.

[14] A. Kumar. A framework for handling delegation in workflow management systems. In *Proc. of Workshop on Information Technologies*, 1999.

[15] J. A. Miller, M. Fan, S. Wu, I. B. Arpinar, A. P. Sheth, and K. J. Kochut. Security for the meteor workflow management system. Uga-cs-lsdis technical report, University of Georgia, 1999.

[16] names removed. A workflow security model incorporating delegation and controlled overriding of constraints. Submitted, 2001.

[17] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security*, 2(1):3–33, 1999.

[18] M.S. Olivier, R.P. van de Riet, and E. Gudes. Specifying application-level security in workflow systems. In *Proc. of the Ninth International Workshop on Database and Expert Systems Applications*, pages 346 – 351, 1998.

[19] R Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[20] R. Thomas and R. Sandhu. A trusted subject architecture for multilevel secure object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 1996.

[21] S. Wu. Task and role combined access control model for workflow system. Uga-lsdis, University of Georgia, 1999.