Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

Information Systems 32 (2007) 365–384

www.elsevier.com/locate/infosys

DW-RBAC: A formal security model of delegation and revocation in workflow systems

Jacques Wainer[a,*], Akhil Kumar[b], Paulo Barthelmess[c]

[a]*Institute of Computing, State University of Campinas, Campinas, 13083-970 SP, Brazil*
[b]*Smeal College of Business, Penn State University, University Park, PA 16802, USA*
[c]*Department of Computer Science and Engineering, Oregon Health & Science University, Beaverton, OR 97006, USA*

Received 31 August 2004; received in revised form 12 July 2005; accepted 2 November 2005

Recommended by M. Weske

Abstract

One reason workflow systems have been criticized as being inflexible is that they lack support for delegation. This paper shows how delegation can be introduced in a workflow system by extending the role-based access control (RBAC) model. The current RBAC model is a security mechanism to implement access control in organizations by allowing users to be assigned to roles and privileges to be associated with the roles. Thus, users can perform tasks based on the privileges possessed by their own role or roles they inherit by virtue of their organizational position. However, there is no easy way to handle delegations within this model. This paper tries to treat the issues surrounding delegation in workflow systems in a comprehensive way. We show how delegations can be incorporated into the RBAC model in a simple and straightforward manner. The new extended model is called RBAC with delegation in a workflow context (DW-RBAC). It allows for delegations to be specified from a user to another user, and later revoked when the delegation is no longer required. The implications of such specifications and their subsequent revocations are examined. Several formal definitions for assertion, acceptance, execution and revocation are provided, and proofs are given for the important properties of our delegation framework.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Workflow; Access control; Delegation; Role-based access control; Security

1. Introduction

Despite various advances in workflow technology, current workflow systems still do not handle delegation well. In its simplest form, a user who has been assigned a task and is unavailable to perform it for any reason (such as leave of absence, sickness, etc.) should be able to delegate it to another user. If such support is not provided, the task will not get done.

The role-based access control model (RBAC) (for example [1]) 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

*Corresponding author. Tel.: +55 19 37885871.
E-mail addresses: wainer@ic.unicamp.br (J. Wainer), akhilkumar@psu.edu (A. Kumar), paulo@cse.ogi.edu (P. Barthelmess).

0306-4379/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2005.11.008

terms like manager, vice-president, etc. and anybody in a role can perform certain tasks assigned to him or her.

The term *delegation* is usually employed in the security literature to describe transfer or inheritance of rights from some user to a machine, that then acts as a surrogate for that user (as in an ATM transaction, for instance). Only recently researchers are starting to recognize the importance of introducing delegation into the RBAC framework. The two significant research efforts that we are aware of in this direction are those of Barka and Sandhu [2,3] and of Yao, Moody and Bacon [4]. The work of Barka and Sandhu allows a role to delegate to another role, and also considers multi-step delegations and revocations. Yao et al. [4] introduces the notion of an appointment whereby a user can appoint another user to perform a task.

RBAC features are increasingly being supported in commercial database systems such as Informix, Sybase and Oracle [5], and the term *grant* is used to refer to the assignment of privileges to users and role. Moreover, *grant* is itself a right that can be conferred. In this way, delegation can be implemented in a database system. However, such support is still limited and does not permit very fine-grained control in a dynamic environment.

In RBAC, in addition to roles, users and privileges, there is also a notion of sessions. Thus, a user may log into different sessions with different roles that she is entitled to play. For example, in one session Mary may be logged in as a cashier and in another as an accounts manager. In workflow applications, the concept of a session is less clear. Instead workflow systems have a notion of *cases*, corresponding to the processing of a specific instance, such as Beth's expense reimbursement claim for travel to Chicago, or Carl's auto accident claim. In this context, privileges must be case-specific, i.e., a user may have permission to perform a task for a certain case but not be allowed to perform the same task for another case. For example, Beth must not be the *requester* and *approver* for *the same reimbursement*, but, of course, Beth may be the approver for Carol's request, and may herself be the requester of a different reimbursement process. Thus, Beth may be *at the same time* requesting her reimbursement and approving Carol's, but that is acceptable if these roles are being played in different *reimbursement cases even within the same session*. Previous work by the authors [6] extends RBAC to accommodate case-based privileges in the context of workflow systems. The present paper is an attempt to include delegation into such a framework. This work extends the ideas presented in [7], which discuss some of the ideas of a fine-grained delegation/revocation framework in RBAC. In this work those ideas have been extended into a workflow domain.

This paper is organized as follows. Section 2 describes briefly the motivation behind the W-RBAC model, an extension to the RBAC model to deal with workflow systems. Section 3 describes the key intuitions behind our model of delegation. Then Section 4 discusses the formal aspect of the assertion of a delegation. Section 5 discusses the intuitive and formal aspects of revocation of delegations. Section 6 proposes extensions to our framework to allow richer kinds of delegations, and Section 7 describes a proof of concept implementation. Then, Section 8 discusses related work, while Section 9 gives the conclusions of this work.

2. W-RBAC and workflow/permission system

2.1. Workflow management systems and RBAC

Workflow management systems allow for the definition and enactment of business processes. A workflow system stores definitions (or schemas) of processes in terms of their tasks definitions, users that should perform tasks, usually given in terms of roles, and a partial ordering of tasks that establishes constraints on task execution sequences. Additional constraints may also be imposed on the ordering. After a workflow process W is defined, instances of this process (also known as *cases* in the workflow literature) may be created and are managed (or *enacted* in the workflow literature) by the workflow system. At any time, zero or more instances of a workflow process might be being enacted.

The focus of the present paper is on the set of users that can perform workflow tasks. Besides timely instantiation of tasks (control flow aspect), one of the main duties of a workflow system is to determine who among the users are the most appropriate to execute each task, as well as determining an order of preference if more than one user fits the requirements for execution (resource allocation aspect). In most commercial workflow systems, once the set of potential executors is determined, either the system announces to all of them that there is work to be done, and one of the users will accept the work, or some more complex selection process that picks out the executor from the set of potential executors is used, including a random selection. In this paper we are not interested in the selection process itself, but on the set of potential executors. Elsewhere (see [8,6,9]), we and other authors discuss selection criteria and policies. In this paper, delegation is the process by which a new user is brought into the set of potential executors of a task for a case (by another potential executor).

An RBAC model is described by (1) *entities*: users, roles, and privileges; (2) *relationships* between these entities; and, (3) *constraints* over these relationships. In this paper we use the following RBAC standard relations among the entities (see the inside box in Fig. 1):

- can-play(U,R) states that user U can play the role R. For example if Alice is (among other things) a Java programmer, one would state that can-play(Alice, java-programmer).
- hold(R,P) states that role R holds the right (or privilege) P. If for example, Java programmers (a role) can use (or have the right to use) the Java optimizing compiler, one would state that hold (java-programmer,use-java-opt-compiler).
- imply(P, P′) states that the right P includes the right P′. For example if the right of using all programming development tools include the right of using the Java optimizing compiler, one would state that imply(use-development-tools,use-java-opt-compiler).

Basic familiarity with RBAC is assumed. For more details, please see [1].

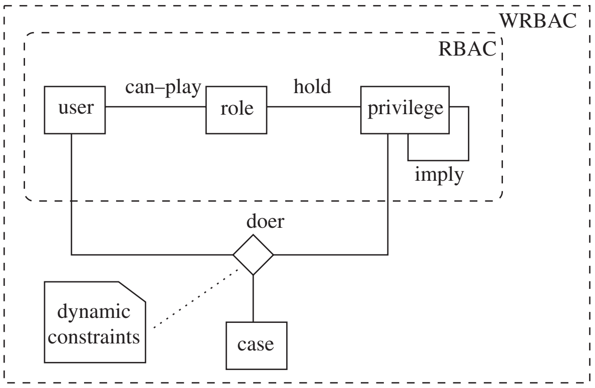

Fig. 1. A conceptual model of RBAC and W-RBAC concepts.

2.2. W-RBAC

W-RBAC [6] is a framework which employs a workflow component and an enhanced RBAC-based permission service. While the workflow component is responsible for process enactment (as defined earlier in this section), the permission service handles the selection of authorized and most appropriate users to execute each task, based on an organizational and authorization model that it manages. W-RBAC defines a protocol that regulates the interaction between these two modules, so that the enactment and permission concerns are clearly separated, i.e., permissions are encapsulated in the permission service that is solely responsible for all authorization related information.

The components of W-RBAC relevant to this paper are:

- the entity *case* which represents an instance of a workflow process,
- the relation $\mathsf{doer}(U, T, C)$ states that user U executed the task T for a particular case C
- *dynamic constraints* that limit who can execute a particular task for a particular case. For example, a dynamic constraint would forbid the same person from performing the activities *sign check* and *audit expenses* for *the same case*, but not for different cases. Dynamic constraints are represented in W-RBAC as

  $\perp \leftarrow Cn,$

  where Cn represents an invalid situation. For example, to assert that no one can be the check signer and the auditor for the same process instance or case, one would write

  $$\perp \leftarrow \mathsf{doer}(x, \text{check\_signer}, c), \\ \mathsf{doer}(y, \text{auditor}, c), x = y.$$

A conceptual model for the RBAC and W-RBAC concepts relevant to this paper is presented in Fig. 1. W-RBAC extends RBAC with the notion of a case and a 3-way relationship called doer between a user, privilege and case. It also introduces dynamic constraints as a way to handle case-specific requirements, such as those involving binding and separation of duties.

The workflow system interacts with the permission system through a relatively simple interface such that, before a task is assigned to a user, the workflow module makes a call to the permission module to determine the users that can perform that task. The permission system will query the RBAC relations to verify which users have the right to perform the task, and for each user, it will verify if adding the corresponding doer relation will violate some of the dynamic constraints. All users who do not violate the dynamic constraints will be returned to the workflow system as potential executors of the task. The workflow might then select among the potential candidates using other concerns such as the ones discussed in [9]. Moreover, after each task is completed the workflow system will notify the permission service of the user who actually performed the task so that the permission service can properly implement various dynamic constraints.

The selection process is an important aspect of the workflow system. Many concerns may arise, such as optimization of some metrics like number of late jobs [10], quality of the work [9], among others, which are beyond the permission system and W-RBAC. W-RBAC will return the set of potential executors as an ordered list of sets of equally preferred users. The workflow may choose among the most preferred ones (regarding the ordering defined), or may choose another user who belongs to the list.

3. Delegation: definitions

The essence of our delegation model is that a user (say, Alice) delegates to another user (say, Beth) the right to perform a task (say, negotiate salary) for a particular workflow case (say, Yasmin's job application). We call this a *specific delegation*. We will also discuss later a more generic form of delegation, in which one delegates just the right to execute a task, but does not specify the case.

In the example above, Alice is the *grantor* and Beth is the *delegate*. There are two stages in a delegation: the first where Alice makes the delegation, which we call *assertion*, and the second where the workflow determines that a next task has to be performed and asks the permission system for possible executors of that task. We call this second stage the *execution* of the delegation. Note that a delegation may be asserted but never be executed.

In order for Alice to be able to delegate a right to Beth, she must have the right to start with. She may have received it through some other kind of delegation (to be discussed shortly), or she may possess the right already. We say that Alice has *direct rights* to a privilege if the RBAC structure allows her to have that right, or more formally, there is a role that Alice can play, and the role holds the particular privilege. For example, Alice may have a direct right to unlock-door because that right belongs to the role of security-guard, and security guard is one of Alice's roles.

If Beth feels that Carla is better suited to perform the salary negotiation than her, she may further subdelegate the task to Carla. Such further subdelegation is called delegation *chaining*. On the other hand, Alice (the original grantor) may not want Beth to further delegate the task. Thus, there is a need to control chaining. We will discuss different ways of controlling chaining, but in general, we will consider that delegation is an action that needs the appropriate right to be executed. Therefore, to delegate the salary negotiation task to Beth, Alice not only must have the right to execute the task, but also the *right to delegate* that task. Moreover, Alice may either only delegate the right to execute the task to Beth, or she may delegate both the right to execute and right to delegate the task. In the latter case only, Beth will have the right to further delegate the task to someone else. We will discuss in Section 4.1 that Alice may also delegate the right to delegate with restrictions, and so Beth will not be totally free to delegate the task to whoever she wants. In this paper we call rights to execute a task a *task right*, and *delegation* rights are the rights that allow one to delegate. We will use the symbol T for a task right, the symbol D for a delegation right and P for any right when it is not necessary to distinguish between task and delegation rights.

The fact that a delegation is accepted does not necessarily mean that the delegate will be the one to execute the task. First, delegation is *additive*, i.e., although Alice delegated the task to Beth, Alice is still a possible executor of the task herself. Secondly, there may be dynamic constraints that prevent a delegate from executing a task. For example, if a delegate has already signed a check for a case, then she would be forbidden from auditing the same case; in this situation we say that the delegate is *blocked*. The fact that the delegate may be blocked does not prevent a delegation from being accepted; in fact, it may only be known that a delegate is blocked much after the delegation was issued and accepted.

The concepts of DW-RBAC as they relate to W-RBAC are presented in Fig. 2. In particular, DW-RBAC further extends W-RBAC by introducing two types of delegations: *specific* and *generic*,

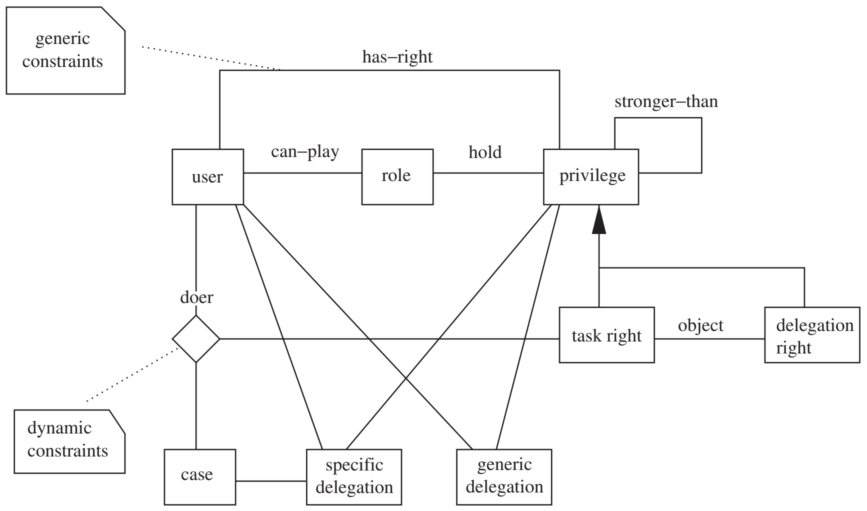

Fig. 2. Main concepts of DW-RBAC.

and a *stronger-than* relation for partially ordering privileges. These new relations are discussed at length in the next section.

All delegations and revocation of delegations are issued by means of operations. We summarize now these operations, and will discuss them in further detail in the rest of the paper.

delegate(G,D,R,C): This is the operation issued by the user G (grantor) to delegate the right R for case C to user D (delegate). The right R may be the right to execute a task, or it may be the combination of such a right and the right to further delegate it. We will denote this as $\mathsf{delegate}(G, D, T + D, C)$, where T is the right to execute the task and D a delegation right for it. (Delegation will be discussed in Section 4.)

delegate(G,D,R): This is the operation to issue a generic delegation from grantor G to the delegate D of right R for all cases (generic delegation is discussed in Section 4.8).

revoke(G,D,R,C): This revokes the delegation of right R from G to D for case C. (Revocation is discussed in Section 5.)

revoke(G,D,R): This revokes the generic delegation of right R from G to D. (Discussed in Section 5.3.)

transfer(G,D,T,C): This transfers the task T of case C, already assigned to user G, to user D. (Discussed in Section 6.1.)

4. Delegation: assertion

Some of the complexity of delegation arises because *trust*, *in general*, *is not transitive*: if Alice trusts Beth, and Beth trusts Carla, that does not mean that Alice trusts Carla. This non-transitivity is carried along to delegation: Alice may trust Beth enough to delegate a task to her (and Beth may likewise trust Carla), but Alice may not want Beth to delegate it further to Carla. Thus, there must be a way to control the chaining of delegations. In our proposal, this is achieved by a complex set of rights to delegate, and through generic constraints.

4.1. Right to delegate

In our model, delegation is a right, and thus, a kind of privilege. We will define several such delegation privileges. If T is a task, the following are delegation privileges: ud(T,n), ud*(T), cd(T,Q,n), cd*(T,Q), ud(0) and ud(T,0) where Q is a one place predicate and $n > 0$. In all these cases T is the *object* of the delegation right.

Intuitively, the privilege ud(T,1) (which stands for *unconditional delegation* of T) allows one that has the privilege of executing T, to delegate T for a particular case to anybody else, provided the delegation does not violate the generic constraints. If one holds the privilege ud(T,2), and can execute T, one can also delegate T, and delegate ud(T,1) as well. So the delegate not only can execute T but can further delegate it to someone else. In general, if a user holds ud(T,n) *and also holds the privilege of executing T*, she can start a chain of delegations of T (for a case C) at most *n* steps long.

In this paper, we will assume that one cannot normally delegate just delegation rights, but one must also delegate the task, and possibly a delegation right to the task. Thus if Alice holds ud(negotiate-salary,3) she cannot just delegate ud(negotiate-salary,2) to Beth, she must also delegate the task negotiate-salary. We will denote that Alice delegated task negotiate-salary and the right ud(negotiate-salary,2) for case c123 to Beth as

$$\text{delegate(alice,beth,negotiate-salary} + \text{ud(negotiate-salary,2),c123)}$$

The special privilege ud*(T) (*total delegation*) is in fact ud(T,∞); thus, if one holds ud*(T), one cannot only delegate T but also ud*(T). So, the delegate will have the same rights regarding T and the delegation of T as the grantor herself.

The privilege cd (for *conditional delegation*), allows one to set up further controls on the chains of delegation. If a user holds cd(T,Q), and can execute T herself, then she can also delegate T to a user U provided U satisfies the property Q, i.e., Q(U) is true. Q is the *condition* of the delegation. If the user holds cd(T,Q,n) she can start a multi-step chain of at most *n* steps (for a case), in which each delegate in the chain must satisfy the property Q. As before, cd*(T,Q) is in fact cd(T,Q,∞).

The purpose of the delegation rights ud(T,0) and ud(0) will be discussed below, but, intuitively, they are rights that do not allow for any further delegation.

4.2. Stronger relation

We will define two relations between task rights and between delegation rights, both of them called *stronger* relation and with similar intuitions. In fact, the semantics of the stronger relation is "stronger than or equal to," and we will use the $\geqslant$ symbol for it.

Definition 1. The stronger relation $\geqslant$ among the task privileges is defined as $T \geqslant T'$ iff

- $T = T'$ or
- imply(T, T'),

where imply is the RBAC relation between two rights.

Definition 2. The stronger relation $\geqslant$ among the delegation privileges is defined as follows:

- $\text{ud}^*(T) \geqslant \text{ud}(T, n) \geqslant \text{ud}(T, k)$ for $n > k$,
- $\text{ud}^*(T) \geqslant \text{cd}^*(T, Q)$, for any Q,
- $\text{ud}(T, n) \geqslant \text{cd}(T, Q, n)$, for any $n > 0$ and Q,
- $\text{cd}^*(T, Q) \geqslant \text{cd}(T, Q, n) \geqslant \text{cd}(T, Q, k)$ for $n > k$,
- If $T \geqslant T'$ then $\text{cd}(T, Q, n) \geqslant \text{cd}(T', Q, n)$ and $\text{cd}^*(T, Q) \geqslant \text{cd}^*(T', Q)$,
- If $T \geqslant T'$ then $\text{ud}(T, n) \geqslant \text{ud}(T', n)$,
- If $T \geqslant T'$ then $\text{ud}^*(T) \geqslant \text{ud}^*(T')$,
- If Q and Q′ are one place predicates, then $\text{cd}(T, Q, n) \geqslant \text{cd}(T, Q \wedge Q', n)$, for any n and T,
- If Q and Q′ are one place predicates, then $\text{cd}^*(T, Q) \geqslant \text{cd}^*(T, Q \wedge Q')$, for any T,
- $\text{cd}(T, Q, n) \geqslant \text{ud}(T, 0)$ for all Q and $n > 0$,
- $\text{ud}(T, 0) \geqslant \text{ud}(0)$ for all T.

If $P \geqslant P'$ then we will say that P' is *weaker* than P. We can also define $P > P'$ as $P \geqslant P'$ and not $P = P'$, and we call $>$ as a *strictly stronger* relation.

Intuitively, a delegation right is stronger than another one if it reaches more people, or it allows "more to be done". For example, ud(T,3) is stronger than ud(S,3) because of imply(T,S), and thus, having the privilege T allows one to "do more" than just having the privilege S. ud(T,3) is stronger than ud(T,2) because it allows chains of size 3 instead of 2; ud(T,3) is stronger than cd(T,Q,3) because the three delegates allowed by cd(T,Q,3) must satisfy condition Q, whereas the three delegates allowed by ud(T,3) are unconstrained.

The purpose of ud(T,0) and ud(0) is to define a minimum for the stronger relation; ud(T,0) is weaker than any other delegation right for the privilege T, and ud(0) is weaker than any delegation whatsoever. They will be useful when we define generic constraints below.

4.3. Generic constraints

RBAC allows specification of the privileges of a user through the role concept. We can define the can-do relation between users and privileges indirectly through the composition of can-play and hold. But delegation adds a new relation between users and privileges, in fact a relation between users, privileges and cases. If task T of case C has been delegated to U, then U gains the privilege to perform T (for case C) independent of the roles she can or cannot play. Thus, constraints that limit delegation must limit directly this new relation between user and privilege.

By their nature, generic constraints do not make reference to cases. Thus, they will limit particular instances of a relation between user and privilege, independent of the case. We will use the relation has-right to capture this new relationship between users and privileges, unmediated by role, that underlies delegation.

We express generic constraints as

$$\perp \leftarrow \text{has-right}(U, R), \gamma,$$

where γ is any conjunction of primitive predicates of RBAC (can-play between users and roles and hold between roles and rights, imply between rights, is-a between roles) or any other derived predicate that refer to users, roles, and rights. For example, we assume that relation subordinate(A,B) is a predicate that states that user A is subordinate to user B in the organizational hierarchy.

To express that no subordinate can receive a right that is stronger than one his superior has, one would define the generic constraint:

$$\perp \leftarrow \text{has-right}(user, p), \text{subordinate}(user, boss), \text{can-play}(boss, role), \text{hold}(role, q), \text{imply}(p, q) \quad (1)$$

Generic constraints cannot be used to place constraints on sets of rights. Thus, in the current proposal, one cannot create constraints that would prevent a subordinate from holding *more* rights in the aggregate than his superior. That would require to make reference to the set of rights both the superior and the subordinate have.

Finally, the two delegation rights ud(0) and ud(T,0) should be used to specify that a user should not gain any delegation right at all, or to a particular task T. The constraint

$$\perp \leftarrow \text{has-right}(u, \text{ud}(\text{unlock-door}, 0)), \text{can-play}(u, \text{security-guard}) \quad (2)$$

states that a person who can play the role of a security guard cannot receive rights to delegate the task of unlocking doors.

4.4. Definitions of delegation acceptance

A delegation assertion is accepted by the system under certain conditions. In this subsection, we define these conditions formally.

Let us call $\mathscr{A}$ the *set of accepted delegations* so far. We first define the auxiliary relation has(U,P,C).

Definition 3. has(U,P,C) is true if:

- there is a role R and can-play(U,R) and hold(R, P^+), and $P^+ \geqslant P$,
- or there is a delegation of the form delegate(x, U, P^+, C) in $\mathscr{A}$ and $P^+ \geqslant P$.

That is, has(U,P,C) is true if either by direct right or by delegation user U has the right P^+ stronger or equal to P for case C.

Now we define when a delegation can be accepted.

Definition 4. The delegation delegate(U1, U2, T + D, C) where T is a task and D is a delegation right whose object is T (or a weaker task) is accepted if delegate(U1,U2,T,C) and delegate(U1,U2,D,C) are accepted (in that order).

Definition 5. The delegation delegate(U1, U2, T, C), where T is a task right, is accepted if:

- case 1 (grantor has ud)
  - has(U1, T, C),
  - has(U1, ud(T^+, n), C) or has(U1, ud*(T^+), C) for some $T^+ \geqslant T$,
  - for all rights T^- such that $T \geqslant T^-$, adding has-right(U2, T^-) does not violate any generic constraint.
- case 2 (grantor has cd)
  - has(U1, T, C),
  - has(U1, cd(T^+, Q, n), C) or has(U1, cd*(T^+, Q), C) for some $T^+ \geqslant T$,
  - for all rights T^- such that $T \geqslant T^-$ adding has-right(U2, T^-) does not violate any generic constraint,
- Q(U2) is true.

To define acceptance of the delegation of a *delegation right*, we must first introduce the *decrement* function as

- decrement(ud*(T)) = ud*(T),
- decrement(ud(T, n)) = ud(T, n − 1) for n > 0,
- decrement(cd*(T, Q)) = cd*(T, Q),
- decrement(cd(T, Q,n)) = cd(T, Q, n − 1) for n > 0.

The decrement function is needed for the proper operation of the delegation mechanism. If Alice received the right ud(T,3), which allows her to start a chain of at most three steps, the strongest delegation right she can give to Beth is ud(T,2) which is the decrement of ud(T,3).

Definition 6. The delegation delegate(U1,U2,D,C), where D is a delegation right whose object is T, is accepted if:

- delegate(U1,U2,T,C) is accepted,
- has(U1, D^+, C) and decrement(D^+) $\geqslant$ D,

- for all rights D^- such that $P \geqslant D^-$, adding has-right$(U2, D^-)$ does not violate any generic constraint, and
- if D^+ is of the form cd(T^+, Q, n) or cd$^*(T^+, Q)$, then U2 must also satisfy Q.

Next, we turn to see how some of these ideas are operationalized.

4.5. Algorithm for acceptance of a delegation

In order to implement the definitions above as an algorithm, there are two general steps: (1) verify if the grantor has the right to delegate, and (2) verify if the delegate can receive the right. We saw above that the rights are expressed in two relations: has-right contains rights obtained directly, while has refers to rights obtained either directly or through delegation. Thus, the two verification operations involve searching the space of a possibly large number of rights: in relation has, one must look for all possible stronger rights, and in relation has-right one must verify if a weaker right contradicts a generic constraint (not to be confused with dynamic constraint). The problem is that, for some delegation rights, there may be an infinite number of stronger rights, while others may have an infinite number of weaker rights; and thus, one cannot naively test all the alternatives.

Hence, in order to verify if has(U,Q,C), that is, if user U has the right Q for case C, one should

1. verify if the user U has direct rights to some Q^+ stronger than P. This information does not depend on the case or on delegations, so it can be computed previously and kept as auxiliary information to the RBAC data as follows: for each user U, compute the maximal privileges the user can hold. A privilege Q is maximal for user U if there is a role r such that can-play(U, r) and hold(r, Q), but there is no role r', and a privilege $Q' > Q$ such that can-play(U, r') and hold(r', Q'). Then collect all maximal privileges for user U and verify if one of them is stronger than Q. This step takes time proportional to the number of maximal privileges of the user (which we assume is not too large).
2. if the user has no direct rights to Q, then collect all accepted delegations whose delegate is U1 (for case C). For each delegation of the form delegate(X, U, P', C), one has to verify if P' is stronger than P. This whole step takes time proportional to the number of accepted delegations of the form delegate$(X, U1, P', C)$.

The pseudo-code for this algorithm is as follows:

```
Algorithm **verify**(has(U,Q,C))
  **if** (there exists $Q^+ \geqslant Q$ and $r$ such that
        can-play$(U, r) \bigwedge$ hold$(r, Q^+)$) **then**
    **return** true
  **else begin**
    **for each** delegate$(X, U, R, C) \in \mathscr{A}$ **do**
      **if** $(R \geqslant Q)$ **then return** true
    **end for each**
    **return** false
  **end**
```

As for the verification of whether the delegate can receive the right, we need to make sure there is no constraint that prohibits the delegate from receiving right R or a weaker right. To do this efficiently, constraints involving right R and delegate U are rewritten to include weaker rights than R as well. Thus, for example, constraint 2 should be rewritten to cover weaker rights as well as

$$\perp \leftarrow \text{has-right}(u, X), X \geqslant \text{ud}(\text{unlock-door}, 0)), \text{can-play}(u, \text{security-guard}) \quad (3)$$

Using this format, each generic constraint rule has to be evaluated at most once for each delegation.

4.6. Multiple delegations

The definition allows for multiple delegation of a right to the same user. User A may delegate to X the right to execute task T (for case C), and also the right to further delegate it to anyone who satisfies Q, that is, if the delegation delegate$(A, X, T + \text{cd}(T, Q), C)$ is accepted. On the other hand user B may delegate to X the right to execute task T for case C and the right to further delegate it at most three times (delegate$(B, X, T + \text{ud}(T, 3), C)$). Thus, user X has received two delegations of the right to execute T, and different delegation rights to further delegate it to others.

In view of the way delegation is defined in this paper, such multiple delegations to X are not a problem. User X can further delegate T to others based on either of the two delegation rights he received. If X further delegates T and some delegation right P to Y, that delegation will be accepted if at *least one* of decrement(ud(T,3)) or decrement(cd(T,Q)) is stronger than P. Thus, X can delegate ud(T,1) to Y even if Y does not satisfy Q, because in this case the delegation right of X being used for this delegation is ud(T,3), which he received from B. Hence, in case of multiple delegations, the user may end up with what is equivalent to the union of all rights he received.

Multiple delegation may also cause a cycle. Say, A delegates to B, who further delegates to C, who in turn delegates to A. The algorithm for acceptance of a delegation does not check if the delegate already had the right before accepting it, nor does it check if the delegation would cause a cycle. In fact, we feel there is no particular problem in the existence of this cycle of delegations: delegations are individual decisions of the grantor that do not require him to know any history of previous delegations of the same task. Thus, perhaps A felt that B was the right person to execute the task T, or to further delegate it to someone else. However, B felt the same about C, who in turn felt the same about A, and thus the cycle was produced.

We will see later that the existence of a cycle will cause some problems when defining the revocation of a delegation. But we decided to place the extra processing burden required to deal with loops in the revocation algorithm. Revocations are less common than delegations, and thus, the extra processing time should be placed there from an efficiency perspective.

4.7. Properties of delegation

There is no single definition of what are *correct delegations*, and thus we cannot *prove* that the delegation mechanism described above generates only correct delegations. In fact, our delegation scheme is very unconstrained—once a user has delegation rights to a privilege X, no restriction is placed on whom she can delegate privilege X to. And if the user has the right to further subdelegate the delegation right, again the scheme places no restriction on who can be the new delegate. But, of course, organizations would want some control on who receives the right. This control is imposed through the generic constraints—the organization may create as many rules or constraints to limit delegation. As we discussed above, one can define a rather generic constraint that no subordinate can receive a right that is stronger than one that his superior holds. Or an even stronger, and likely too draconian, rule that no subordinate may receive a privilege that is not held by his superior (this is too strong because in most organizations the subordinate does already hold rights that his superior does not—the Director does not have the database privileges that the database manager does, although the Director is his superior).

The following theorem shows that indeed all generic constraints will be enforced in our delegation scheme:

Theorem 1. *No set of delegations will give a right P to a user U if that violates some generic constraints, (that is,* has-right(U,P) *violates some generic constraint).*

Proof. (by contradiction) Let us first assume that the right P is a task right, and let us examine one of the delegations that eventually attributed the right P to user U for case C. This delegation could be of one of the following forms:

1. delegate(X,U,P,C)
2. delegate(X,U,S,C), where $S \geqslant P$
3. delegate$(X, U, P + D, C)$, where D is some delegation right on P
4. delegate$(X, U, S + D, C)$, where D is some delegation right on S, and $S \geqslant P$

In the first case, the delegation would not be accepted, because by Definition 5, the assertion has-right(U,P) would have to be checked against the generic constraints. Since it directly violates some constraint, the delegation could not be accepted. In the second case, there is at least a right (in this case P), such that $S \geqslant P$ and for which has-right(U,S) violates some generic constraint. The next two cases involve delegation of a task along with delegation rights for it. By Definition 4, first the suitability of the task delegation (P or S) would be tried, and so cases 3 and 4 would be similar to 1 and 2, respectively. Since the task delegation itself would not be accepted in these cases, they will be rejected.

If P is a delegation right (say, for some task T), the two forms of delegation that would delegate it to U are

- delegate$(X, U, P + T, C)$,
- delegate$(X, U, S + T, C)$, where $S \geqslant P$.

By Definition 4, the first case would not be accepted because has-right(U,P) violates the constraint, and the second case would not be accepted because there is at least one right (in this case P) which is weaker than S and which violates the generic constraints. □

4.8. Execution of delegations, dynamic constraints, and generic delegations

Once a delegation is accepted, it confers a potential right on the delegate. Thus, if user U received a delegation of T (for case C), it means that U is among the potential executors of activity T for that case. Whether U actually executes T will usually not be left to U's decision. First, dynamic constraints may preclude U from executing T. Second, it may be that other potential users have higher priority in executing T, and thus, they may be chosen to execute T for case C. In database systems, if a user U receives a delegation of right T, she can execute T when she wants. In a workflow system it is not so.

An accepted delegation is exercised when the workflow asks for the potential executors of an activity for a case. Then, in principle, every user who has the right to execute the task (directly or through delegation) is a potential executor. Only then is the delegation exercised.

A user U is *blocked* from executing task T for case C if

- U has the right to execute T of C, but
- the inclusion of doer(U,T,C) causes a violation of a constraint.

This means that U is blocked on account of some *dynamic* constraint. For example, even though Alice received a delegation to execute the task approve-purchase for case c34, nevertheless, since she is the one who ordered the purchase, she is blocked from executing the approval by a *separation of duties* requirement.

If user U is not blocked, then the fact that she received the right to execute task T through delegation may have consequences regarding her priority to execute T. We will not discuss this issue further in this paper, but clearly the workflow component that assigns priorities to the potential executors of a task must take into consideration both the grantor and the delegate of a delegation—it seems reasonable that the grantor's priority should be reduced while the delegate should be increased.

So far, we were concerned with case specific delegations, both the acceptance and the execution of such delegations. Recall, a case specific delegation is specific to a workflow case. On the other hand, a generic delegation applies to all cases where a task, say T, may occur. A *generic delegation* is a statement in the form delegate(U1,U2,T).

The definition of acceptance of a generic delegation delegate(U1,U2,P) is very similar to the definition of specific delegations: the grantor must have the right to delegate, and the expanded rights of the delegate must not contradict the generic constraints. Thus, we define the relation has-g(U,P), in a similar way to has(U,P,C), and then adapt Definitions 4–6 to define when delegate(U1, U2, T + D), delegate(U1,U2,T), and delegate(U1,U2,D) are accepted.

From an execution point of view, an *accepted* generic delegation delegate(U1,U2,P), where P is any privilege (related to task or delegation rights) is used to create a specific delegate(U1,U2,P,C) when the case C is created. In other words, the system maintains a set of accepted generic delegations $\mathscr{A}_g$; when a new case C is created, for each generic delegation delegate(U1, U2, P) $\in \mathscr{A}_g$, a case specific delegation delegate(U1,U2,P,C) is added to $\mathscr{A}$. The specific delegation delegate(U1,U2,P,C) is called a *spawn* of the generic delegation.

5. Revocation

Revocation is the process by which a delegation that was accepted is removed or retracted. However, since delegations may be chained, a revocation can produce side effects and other consequences. This section will examine the details concerning revocation. We will first concentrate on revocation of specific delegations, and later Section 5.3 will extend the results to generic delegations.

It should also be pointed out that specific delegations have a very definite life span—once the case is terminated all specific delegations for that case are no longer relevant and can be removed from the set of accepted delegations. Thus revocations occur in cases where a certain delegation was accepted, and then the grantor changed her mind before the task was started. The central issue is that revoking a delegation not only removes the rights received by the delegate, but also undoes all further delegations that were brought about by the revoked delegation. Moreover, a delegate may receive a delegation from multiple grantors, and in such a situation, if one delegation is removed, another one may still be active. We will try to address these issues shortly, but first we introduce the concept of a chain of delegation.

Definition 7. A delegation $d_1 = \mathsf{delegate}(g_1, r_1, \mathrm{T}_1 + \mathrm{X}_1, \mathrm{C})$ *supports* another delegation $d_2 = \mathsf{delegate}(g_2, r_2, \mathrm{T}_2 + \mathrm{X}_2, \mathrm{C})$ iff

- $r_1 = g_2$ that is the grantor of d_2 is the delegate of d_1,
- T_1 and T_2 are task rights and $\mathrm{T}_1 \geqslant \mathrm{T}_2$, that is, the tasks right being delegated in d_2 is at most as strong as the task right received in delegation d_1,
- $\mathrm{decrement}(\mathrm{X}_1) \geqslant \mathrm{X}_2$, that is the delegation rights delegated in d_2 are at most as strong as $\mathrm{decrement}(\mathrm{X}_1)$.

Intuitively, d_1 supports d_2 if d_1 is one of the ways the grantor of d_2 received the rights to make that delegation.

Definition 8. A *chain of delegation* for a task T and case C is a sequence of delegations $\langle d_0, d_1, d_2, \ldots d_n \rangle$, where each d_i is of the form $\mathsf{delegate}(g_i, r_i, \mathrm{T}_i + \mathrm{X}_i, \mathrm{C})$, where g_i is the grantor of d_i and r_i the delegate (or recipient) of d_i, such that:

- d_i supports d_{i+1},
- no g_i of any delegation d_i has the direct right to X_i,
- there is no pair d_i and d_j in the chain, and such that $d_i = d_j$.

The delegation d_0 is called the *starter* of the chain. Given the chain $\langle d_0, d_1, d_2, \ldots d_{n-1}, d_n \rangle$, we will say that $\langle d_0, d_1, d_2, \ldots d_{i-1} \rangle$ is a *support* chain for d_i.

Fig. 3 gives an example of a delegation chain, where $ud(\mathrm{T}, 4)$ is an abbreviation for $\mathsf{delegate}(\mathrm{Grantor}, \mathrm{Delegate}, \mathrm{T} + \mathsf{ud}(\mathrm{T}, 4), \mathrm{C})$ for task T, and some case C, for the grantor and delegate indicated. The support chains for the delegation $\mathsf{delegate}(\mathrm{J}, \mathrm{G}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 1), \mathrm{C})$ are

- $\langle \mathsf{delegate}(\mathrm{A}, \mathrm{B}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 5), \mathrm{C}),$ $\mathsf{delegate}(\mathrm{B}, \mathrm{J}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 4), \mathrm{C}),$ $\mathsf{delegate}(\mathrm{J}, \mathrm{G}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 1), \mathrm{C}) \rangle$,
- $\langle \mathsf{delegate}(\mathrm{A}, \mathrm{B}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 5), \mathrm{C}),$ $\mathsf{delegate}(\mathrm{B}, \mathrm{F}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 4), \mathrm{C}),$ $\mathsf{delegate}(\mathrm{F}, \mathrm{J}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 2), \mathrm{C}),$ $\mathsf{delegate}(\mathrm{J}, \mathrm{G}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 1), \mathrm{C}) \rangle$.

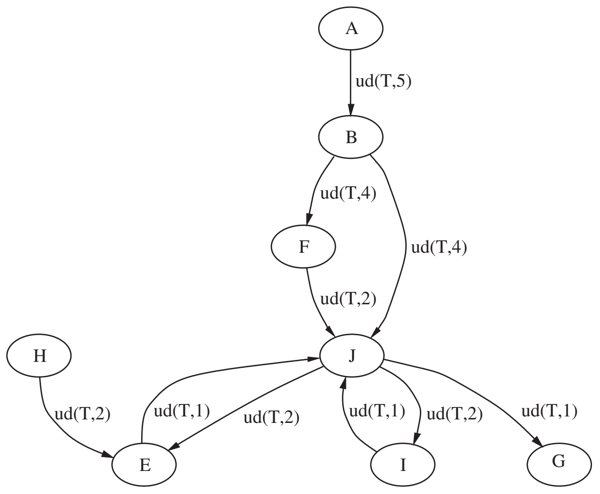

Fig. 3. Example of dependence of delegations.

The support chain for $\mathsf{delegate}(\mathrm{J}, \mathrm{E}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 2), \mathrm{C})$ is only

- $\langle \mathsf{delegate}(\mathrm{A}, \ \mathrm{B}, \ \mathrm{T} + \mathrm{ud}(\mathrm{T}, 5), \mathrm{C}),$ $\mathsf{delegate}(\mathrm{B}, \mathrm{J}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 4), \mathrm{C}),$ $\mathsf{delegate}(\mathrm{J}, \mathrm{E}, \mathrm{T} + \mathrm{ud}(\mathrm{T}, 2), \mathrm{C}) \rangle$.

Definition 9. A delegation d_j is *dependent* on a delegation d_i if all support chains for d_j contain d_i.

Definition 10. The effect of $r(d)$, the revocation of the delegation d, given the set of accepted delegations $\mathscr{A}$ is a new set of accepted delegations $\mathscr{A}'$ defined as

$$\mathscr{A}' = \mathscr{A} - \{d\} - \{x \mid x \text{ is dependent on } d\}.$$

For example, the effect of revoking the delegation of $ud(T, 4)$ from B to J in Fig. 3 is illustrated in Fig. 4. The delegations from J to I, from I to J, and from J to E, all depend on the delegation from B to J, and were, therefore, removed from $\mathscr{A}$. I is left as an unsupported delegation now.

5.1. Algorithm for revocation

A naive algorithm to compute the effect of revoking a delegation $d = \mathsf{delegate}(\mathrm{A}, \mathrm{B}, \mathrm{T} + \mathrm{X}, \mathrm{C})$ can be implemented from the definition:

- Given the set of accepted delegations for task T and case C construct all chains of delegations.
- For each delegation x verify if all support chains for x contain d.

- Remove from the set of accepted delegations the ones dependent on *d*.

The problem with this naive algorithm is that each chain through B must be examined and there may be an exponential number of chains (in relation to the number of delegations). Moreover, the graph contains cycles. We will present below a quadratic algorithm for computing the set of dependent delegations of a delegation *d*. The algorithm is based on an acyclic graph, called the delegation graph, that implicitly represents all chains in which each delegation is a denoted by a vertex and a chain is a path in the graph.

5.1.1. The delegation graph

A *delegation graph* is a graph in which each node is an accepted delegation assertion, and each directed arc is a support relation between two delegations. A delegation graph for a task T and a case C ($DG(T, C)$) is a graph whose nodes are all delegations of the form delegate($U_1, U_2, T' + D$, C), where U_1 and U_2 are users, T' are task rights such that each T' is either $T' \geqslant T$ or $T \geqslant T'$ and D is a delegation right. Each such node is represented as $dg(U_1, U_2, D)$.

There is an edge from $dg(U_1, U_2, D_1)$ to $dg(U_3, U_4, D_2)$ if

- delegate($U_1, U_2, T_1 + D_1$, C) supports delegate ($U_3, U_4, T_2 + D_2$, C),
- it is *not* the case that user U_3 has direct right to D_2.

All nodes that have no incoming arcs are called *sources*. Fig. 5 is the delegation graph corresponding to the set of delegations depicted in Fig. 3. The labels in the nodes correspond to the labels in the arcs and the position of the nodes is close to the position of the corresponding arc in Fig. 3. The two sources are denoted by the ellipses with wider borders.

The delegation graph has the property that all paths in the delegation graph starting from a source correspond to a chain of delegations.

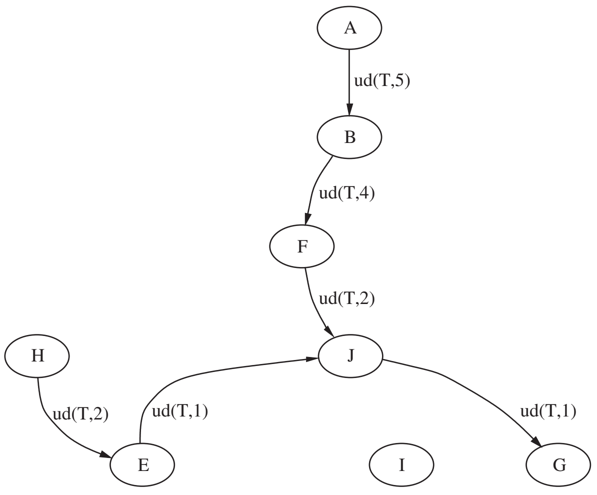

Fig. 4. Delegations after the revocation of $ud(T, 4)$ from B to J.

5.1.2. The algorithm

The algorithm to compute the set $\mathscr{A}'$ given the revocation of a delegation *d* for task T and case C is as follows:

```
Algorithm revoke($d$)
  $DG(T, C)$ ← the delegation graph for task T and
  case C.
  $X$ ← the reachable nodes in $DG(T, C)$ from all
  sources.
```

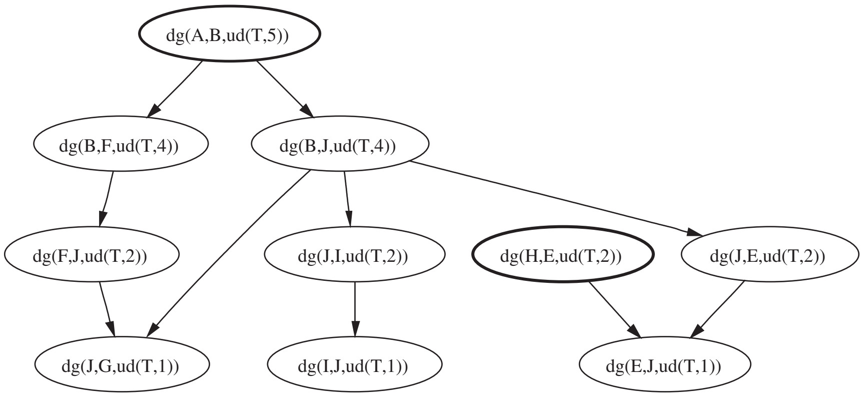

Fig. 5. Delegations graph of the delegations in Fig. 3.

$DG^{-}(T,C) \leftarrow$ remove d and its incoming and outgoing edges from $DG(T,C)$.
$X^{-} \leftarrow$ the reachable nodes in $DG^{-}(T,C)$ from all sources.
$\mathscr{A}' \leftarrow \mathscr{A} - \{d\} - (X - X^{-})$.

The total cost for the computation of the revocation of delegation d for task T on case C is in the worst case $O(n^2)$ where n is the set of delegations for tasks that are stronger or weaker than T for case C. The creation of the delegation graph takes $O(n^2)$ using a naive algorithm—for each delegation scan all other delegations to verify if there is an edge between them. Computing the set of reachable nodes from the sources takes $O(n)$.

5.1.3. Correctness of the algorithm

Next, we argue the correctness of the above algorithm as follows. *All removed nodes correspond to a dependent delegation*: given that every path from a source in the graph corresponds to a chain, if Y got disconnected by the removal of X, then there is no path from a source to Y that does not go through X; thus, Y depends on X, and Y should be deleted.

All dependent nodes are removed: suppose there is a dependent delegation Y that was not deleted. Then there is path from a source node that does not go through X—this is a chain of support to Y that does not go through X—thus Y is not dependent on X, a contradiction.

5.2. Property of revocation

The important property of revocation is that it will leave no one with unsupported rights. For example, if the right to hire a new programmer has been delegated throughout the organizations such that eventually Alice received such right, and all users that started a chain eventually revoked the delegation, then Alice will lose the right she received.

In order to prove that, we need the following lemma:

Lemma 1. *If a sequence of delegations D is followed by a sequence of revocations R, then all remaining accepted delegations will either be the source of a chain (called a starter) or will have at least one support chain.*

Proof. By induction on the number of revocations in R.

Base case (if there are no revocations): For a delegation d to be accepted, either the grantor has directs rights to the appropriate delegation, and thus, d is a starter of a chain, or there is another delegation d' that supports it. In turn, d', is also either a starter that defines the chain of support for d, or it has a supporting delegation d'', until a *starter* is reached.

Inductive case: Now we show that if the claim of the lemma is true for a sequence of delegations D and a sequence of revocations R, then it remains true for the new sequence of revocations $R + r(d)$ obtained after a new revocation $r(d)$. By assumption, before the revocation, d had at least one chain of support. After the revocation, node d is removed from the delegation graph, and all nodes that are unreachable as a result are also removed. Now, clearly, the only nodes that are unreachable are the ones for which all the support chains passed through d. If there was a support chain for a node, say d_x, that did not pass through d, then d_x would still be reachable. Hence: (1) all delegations that are dependent on d are removed; while (2) all delegations that have an alternate support chain are not removed. □

Note that this lemma is general for any combination of delegations and revocation assertions, and in any sequence. We can now state and prove the main theorem based on the lemma.

Theorem 2. *If there is an ordered set of delegations $D = \langle d_1, d_2 \ldots d_x \rangle$ of the right to execute task T followed by an ordered set of revocations $R = \langle r(d_{i1}), r(d_{i2}) \ldots r(d_{ik}) \rangle$ such that, for all d_k, starter of a chain in D, there is a corresponding $r(d_k)$ in R, then no user U that received the right T through some delegation $d_x \in D$ will have the right after all the revocations.*

Proof. From Lemma 1, if the set of remaining accepted delegations is still non-empty, then the remaining delegations must have a support chain. But since all starter nodes have been deleted, and all support chains start at the starter nodes, there cannot be any support chain left. Hence, there are no unsupported delegations still remaining after R. □

5.3. Handling generic revocations

So far we were dealing with case-specific delegations. The revocation of a generic delegation d_g has a double effect: it removes from the set of accepted generic delegations the ones that depend on d_g, and it recursively revokes all spawns from these dependent generic delegations.

Formally, one defines dependence of generic delegations (*g-dependent*) in a very similar way to case specific dependence, but making reference to the set $\mathscr{A}_g$ of accepted generic delegations. Let us define the set of generic delegations of d_g, given $\mathscr{A}_g$ as g-dep$(d_g) = \{x \in \mathscr{A}_g \mid x$ is g-dependent on $d\}$.

Definition 11. The effect of revoking the generic delegation d_g given the current set of accepted generic delegations $\mathscr{A}_g$, and the current set of accepted specific delegations $\mathscr{A}$ are the new sets $\mathscr{A}'_g$ and $\mathscr{A}'$ defined as

- $\mathscr{A}'_g = \mathscr{A}_g - \{d_g\} -$ g-dep(d_g),
- $\mathscr{A}'$ is the result of revoking all spawns of each $x \in$ g-dep(d_g).

It is important to recognize that the cost of revoking a generic delegation can be substantial. The revocation of a generic delegation g must revoke all spawns of the generic delegation and all g-dependent generic delegations of g.

The algorithm discussed above to determine the dependent delegations of a specific delegation can be adapted to determine all g-dependent delegations of a generic delegations. One can construct a generic delegation graph $GDG(T)$ in a similar way to constructing the $DG(T, C)$ (but considering only generic delegations for task T' such that either $T' \geqslant T$ or $T \geqslant T'$. The algorithm will determine in $GDG(T)$ all g-dependent delegations of a generic delegation of T. As above, this algorithm is quadratic in the number of generic delegations of rights T or stronger.

However, for each g-dependent delegation x removed from $\mathscr{A}_g$, all spawns of x must be revoked. Spawns are specific delegations that are automatically generated when a new case is created: for each generic delegation delegate(A,B,R), a specific delegation delegate(A,B,R,C) is added to $\mathscr{A}$ when case C is created. Each spawn is a specific delegation which may generate other specific delegations. On the other hand, when the case finishes, all spawns can be removed. Thus, revoking all spawns of delegation x may involve revoking k spawns (and delegations the spawns generated), where k is the number of active cases in the workflow.

6. Extensions

In this section, we discuss our ideas of transfer, strong delegation and revocation, revocation with downgrades and time-restricted delegations. These extensions have been incorporated into our framework—but we decided to present them separately not make the text too complex.

6.1. Transfer

A *transfer* is an operation by which a user, who has been selected to execute a task T for case C, assigns another user to perform it, and forfeits her own right to do the task once the transfer is accepted by the system. Transfer is similar to delegation in the sense that not only is the right to execute the task being delegated, but the duty to do it is being transferred as well. In order to transfer a task, the user must have the right to transfer, and the receiver must not violate any of the dynamic constraints. Note that we do not treat transfer as a "right", rather we assume that a delegation right automatically gives transfer rights also provided certain conditions are met. In particular, if the user has any delegation right stronger than ud(T,0) for case C (that is, any right of the form cd(T,Q,n) or ud(T,n)), then she can transfer T to another user.

The transfer operation is denoted by transfer(S,D,T,C), where S is the *source* of the transfer (the user transferring the task), D is the *destination* of the transfer (the user receiving the task), T is the task and C is the case. By its nature, transfer is only a case specific operation, and there is no notion of a generic transfer.

Definition 12. A transfer transfer(S,D,T,C) can be accepted if

- S has been selected to perform task T for case C,
- has(S,R,C) for some right $\mathrm{R} > \mathrm{ud}(\mathrm{T}, 0)$,
- for all rights T^- such that $\mathrm{T} \geqslant \mathrm{T}^-$ adding has-right$(\mathrm{D}, \mathrm{T}^-)$ does not violate any generic constraint,
- adding doer(D,T,C) does not violate any dynamic constraints,
- if R is of the form cd(T,Q,n) or $\mathrm{cd}^*(\mathrm{T}, \mathrm{Q})$, then D must also satisfy the predicate Q.

The effect of accepting a transfer is that D now becomes the selected executor of task T for case C. The workflow system and user D must be notified of this change. We will discuss how to do this in Section 7.

6.2. Strong delegation and strong revocation

In our present framework, in order to delegate a right R, the user must both have the right to R *and* also the right to delegate R. But there are many situations in which it is desirable that a user may delegate a right even if he does not possess the right to it. This can often happen in management situations—the user responsible for a process must delegate a particular task that is late, or cannot be performed by the assigned person to some other user, even though the person in charge does not have the right to perform it himself. The chief of medicine of a hospital should be able to delegate the right to perform an emergency eye operation to someone else, say an ophthalmologist, even though the chief, being an oncologist, is not qualified for eye surgery, and thus, should not have the right to operate on an eye himself.

We introduce the new rights sud(R,n), sud*(R), scd(R,Q,n), scd*(R,Q) as the "strong" versions of the delegation rights already discussed. The meaning of a scd(R,Q,n) is that the grantor can delegate the task R and the right $\mathrm{cd}(R, Q, n-1)$ to any delegate that satisfies the property Q, even if the grantor himself does not have the right to R. Notice that the *strongness* of the right is not itself delegated—if A strong-delegates R to B, it is assumed that B already has the right to R and does not need to strong delegate further.

Revocation in our framework is not a standard right in the same sense as delegation, and is not represented explicitly in the RBAC data structures. However, revocation is an "automatic" right for anyone that made a delegation, i.e. an individual that made a delegation can also revoke it. However, for administrative purposes, it is important that delegations made by one individual be revocable by another. We call this a *strong revocation* right. Further details of strong revocation are left for future work.

6.3. Revocation with downgrade

There is another possibly more complex definition of revocation. Let us see the example in top part of Fig. 6. When the delegation from A to B is revoked, so are the delegations form B to D, D to E, D to G, and E to H, because they are all dependent on $\mathrm{delegate}(A, B, T + \mathrm{ud}(T, 5), C)$. Clearly, B could not delegate T + ud(T,4) to D because with A's revocation, she has no such right. But with the "remaining" delegation from F, at least B could delegate T + ud(T,1) to D, and D could have delegated *T*, to E and G. The result is illustrated on the bottom part of Fig. 6. Revocation with downgrade will thus weaken some delegations rather than removing them when an alternative support chain (albeit a weaker one) is present. Of course, all delegations that cannot be downgraded in this way will necessarily be removed from the set of accepted delegations. New algorithms are required for handling situations of revocation with downgrade.

6.4. Time-restricted delegations

A further possible extension to the model proposed here is that of time-bound delegations such that: *A delegates to B the right R for the next H hours*. Thus, instead of expressing a delegation as:

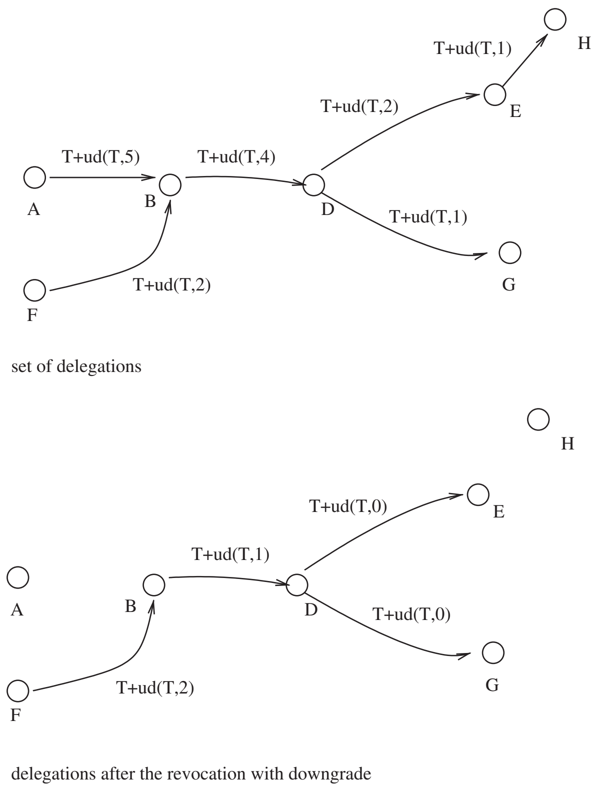

Fig. 6. Delegations before and after a revocation with downgrade.

delegate(U1,U2,R,C), we would modify it to delegate(U1,U2,R,C, start_time, end_time) or delegate(U1, U2, R, C, now(), now() + H), where *now()* is a standard function that returns the current time. One simple implementation is to transform the time-bound delegation into a standard delegation and have a trigger to revoke it at the expiry time with a standard revocation. Such an implementation guarantees that the delegation will only last for the desired H hours, but does not allow one to reason about it. For example, in this simple implementation a user cannot ask the system for how long he will hold a delegated right, since such temporal information is not represented in the system.

An authorization model for temporal and derived data has been proposed by Atluri and Gal [11]. This model enables temporal assertions to be made for read, update and insert accesses to data items in a database. Algorithms are given to ensure that only accesses that do not violate these assertions can occur. Our intent is to extend some of these notions to the level of operations as opposed to data items. Thus, we would like to be able to express delegations such as: "delegate Beth's rights to Carla while Beth is on vacation, but for no more than 1 week." Such temporal assertions would naturally need the notion of constraints, which is not present in the proposal in [11].

7. Implementation

A proof-of-concept prototype of the DW-RBAC system was implemented. We now describe the implementation architecture and the various interfaces between the user, the DW-RBAC module and an existing workflow system.

Fig. 7 is a simplified version of the DW-RBAC implementation architecture. We assume a standard workflow system module with additional interfaces to make calls on and receive calls from the DW-RBAC component. The three circles represent the DW-RBAC databases. The workflow component would have its own database for storing the process definitions, case data, etc., but they are not shown here. The workflow component used in the implementation was a simple prototype developed by the authors. An important requirement of the architecture is that all tasks represented in the workflow processes must have a corresponding right associated with them in the RBAC struc-

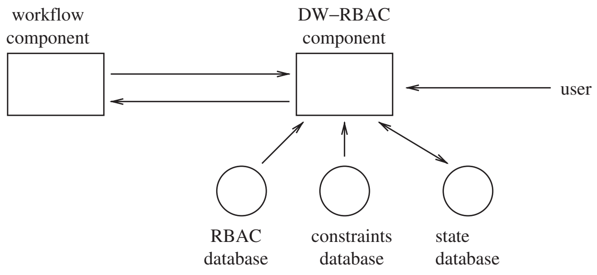

Fig. 7. The implementation architecture.

ture. The DW-RBAC component accesses three databases:

- The RBAC database which stores users, roles, and rights, and the relations among them.
- The constraint database which stores the dynamic and generic constraint as a logic program clause, together with all relevant auxiliary predicate definitions.
- The state database which stores the accepted delegations, and the identities of the executors of each task for each case.

Next we define the various interfaces required to implement this architecture. On the user side, DW-RBAC receives the following calls (where G is the grantor, D the delegate, T a task, C a case, and the return value is boolean):

- delegate(G,D,T,C): delegation assertion for a case
- delegate(G,D,T): a generic delegation assertion
- revoke(G,D,C): a specific revocation—revoke the delegations from G to D of case C
- revoke(G,D): a generic revocation
- transfer(G,D,T,C): transfer task T of case C from user G to user D

On the workflow system interface, the DW-RBAC component receives the following calls from the workflow system:

- p-executor(T,C,O): this function returns the *potential* executors for task T of case C, ordering the executors according to ordering O. The DW-RBAC will then verify all users that have the right to T for C, and do not violate any dynamic constraints, and order them according to the order O given. The ordered list is returned to the workflow. For more on orderings, see [6].

- start-case(C): notify DW-RBAC that case C has started. DW-RBAC will spawn a specific delegation for each appropriate accepted generic delegation.
- end-case(C): notify the DW-RBAC component that case C has ended. All accepted delegations for this case are deleted.
- executor(U,T,C): notify DW-RBAC that U is the selected executor for task T of case C. This data is stored in the state database and possibly used later to verify dynamic constraints.

The DW-RBAC makes the following call on the workflow system:

- new-executor(U,T,C): notify the workflow system that there was a transfer from the selected executor to user U. This is the result of an accepted transfer operation.

A prototype of the system was implemented in Prolog. The RBAC data is stored in a database and accessed through an ODBC interface. The constraint store is a set of Prolog programs that are loaded with the DW-RBAC component, and the state data is represented in memory as Prolog facts and also kept in a permanent data storage (using Berkeley DB). Prolog is well suited for the implementation of the algorithms and definitions presented. As an illustration, Appendix A gives sample code for verifying if a delegation can be accepted.

8. Discussion and related work

The management of delegations, at least at a first approximation, does not seem to be difficult. We envision that each user will enter his/her own delegation 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 she does not hold the appropriate permissions).

In this paper, we have extended our previous work related to fine-grained user-to-user delegation in RBAC [7]. In particular, we have introduced the notion of workflow case and also dynamic constraints that allow dynamic separation and binding of duties on a case by case basis.

Related research on delegation has been somewhat limited. Barka and Sandhu [2] present a framework with the objective of identifying interesting cases that could be used for building role-based delegation models. In terms of their framework, our model is: temporary (in the sense that a delegation can be revoked, but not in the sense that it has s predefined validity), monotonic, partial, self-acted, both single- and multi-step (with a fine control over the multi-step delegation using different delegation rights), multiple-delegation (which we call delegation to a group with the inclusion of the workflow component to control the execution of the delegated activities), unilateral agreement, a version of cascading revocation (based on our definition of dependency), and grant-dependent. Further extensions of this approach are presented in [3]. Our model has a complex structure regarding the multi-step delegation issue which is not discussed in previous work, and includes components related to dynamic constraints which are especially relevant to a workflow context.

In [3], the RBDM0 model (role-based delegation model zero) is proposed as an extension of RBAC (more specifically RBAC0 model of the RBAC96 family) to include delegation. Being derived from RBAC0, it is restricted to flat roles and does not allow hierarchies. Moreover, 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. Some extensions are discussed: grant-dependent revocation, delegatable and non-delegatable permissions, and two-step delegation, as well as delegation in hierarchical roles. This work is still further extended in [12] to a permission based delegation model (PBDM).

The role-based delegation model of Zhang, et al. [13] is very interesting and useful. Their proposal allows delegations of the form DLGT(User1,Role1,User2,Role2), where User1 who holds Role1 delegates her role to User2 who holds Role2. Moreover, they also allow generic constraints in a manner similar to ours. They also allow grant-dependent and grant-independent revocations. Our proposal in this paper is an effort to complement the previous work. In particular, our objective is to allow a more "flexible" sort of delegation. The delegation of a role results in an "all or nothing" situation, i.e., either the grantor must delegate *all* the privileges of his or her role or none. In practice, a grantor may wish to delegate only certain privileges. A president may wish to delegate her privilege to sign checks, approve expenses and authorize capital expenditures, but not the privilege to hire new employees. This would be difficult if the entire role is being delegated. Since delegation is normally a somewhat unusual and special situation, a somewhat fine-grained kind of control over the privileges of the grantor, we feel, would be useful. Furthermore, our scheme offers more flexibility in that a president may delegate some of the tasks such as signing of checks and approval of expenses to a vice-president with a right to further delegate *to one level lower*, but the right to approve capital expenditures may not be further delegated.

In another noteworthy work on delegation [14], the authors present a graph based approach where a delegation from A to B is shown as a link between nodes A and B on a graph. The line joining A and B can be of three types depending upon whether the delegation of the right is positive, negative, or delegatable (i.e., the right can be further delegated). The authors give various rules for correctness and consistency of delegation, along with algorithms. One difference in our approach is that it can *fine tune* the number of levels to which a right can be delegated (as opposed to making it binary). Moreover, we allow constraints and conditions as ways to exert still finer control over the delegation. In [15], delegation is achieved by the notion of an *appointment*. A user acting in the appointer role grants another user called the appointee a credential which may be used to activate one or more roles. Role activation takes places based on rules. Thus, it is possible for an appointer to give access to privileges that she does not possess. Still other related work on role hierarchy supporting restricted permission inheritance may be found in [16]. This approach is based on dividing a single role hierarchy into inter-related hierarchies for controlling permission inheritance behavior. Approaches for delegation in the context of trust management systems are discussed in [17,18]. Another method of delegation that can constrain the "shape of delegation chains" through regular expressions is discussed in [19].

Our proposal for revocation differs from the standard SQL revocation mechanism. The issue centers around multiple chains of delegation. The SQL revocation mechanism uses a time stamp to identify which delegations causally derived from others. Thus if delegation d_1 is derived from d_0, which among other things means that d_1 must have happened after d_0, then the revocation of d_0 also revokes d_1. In opposition to this causality-based view, we follow a logic (or atemporal) view: the revocation of d_0 will only revoke d_1 *now*, if d_0 is the only support for d_1. In our view, if there is another delegation $d_{0'}$ which supports d_1 (even if $d_{0'}$ was accepted after d_1, and thus, was not "really" used to determine if d_1 was accepted) then the revocation of d_0 will not revoke d_1.

Bertino et al. [8,20] propose a constraint based security model for workflow systems, whose language allows for more expressive power than the one presented here. Their language and model refer to the many instances of the activation of a task within a case, which was not considered here, and their work discusses in detail optimizations that can be performed off-line so that queries performed on line would have a good chance of being answered quickly (but the worst case is still exponential). However, delegation is not addressed in any form in [8,20].

Security in workflow systems are also considered e.g. by Castano et al. [21], Atluri and Huang [22–24], Karlapalem and Hung [25,26], Pernull and Herrman [27,28] and the Meteor project [29–31]. Each of the models mentioned above explore different aspects of workflow related security, but none addresses delegation issues, the focus of the present paper. Finally, Kumar [32] introduces issues of user to user delegation support in workflow systems and related security implications in a non-RBAC framework.

The work of Atluri et al. [33] has some similarities to this one. The authors propose a delegation framework in workflow systems that correspond to our generic delegations, added to "delegation rules" which roughly correspond to our generic constraints. Their framework does not contemplate specific delegation, delegation rights, chaining, and revocation.

9. Conclusions and future work

Researchers have recently noted [2–4] that the current RBAC model does not handle delegation well and has various shortcomings in this respect. In this paper, we extended our previous on WRBAC [6] to incorporate delegation in a workflow context in an RBAC environment. The DW-RBAC model allows a fine- grained delegation, in which rights to execute a task for a workflow case, and delegation rights are delegated among users in an ad hoc way, controlled by general, organization-level constraints. A novel aspect of our approach lies in providing two types of constraints: *generic constraints* for managing overall delegation policies, and *dynamic constraints* for implementing case-specific policies that pertain to a specific case. Case specific policies are important when binding or separation of duties has to be enforced. In addition, we discussed algorithms for revocation of delegations, and also described a proof of concept implementation done in Prolog to test our proposal.

The paper also discussed our ideas for how the framework can easily be extended to incorporate other variants of delegations such as: transfers, strong delegations, and temporal delegations. Another extension that would be useful and deserves consideration is delegation to a group, such that the group (as opposed to an individual) is required to perform the delegated task. For example, a VP may delegate a task (say, approval of a loan request) to two of her department heads, with the provison that they must review it independently and unless both give their approval, the decision will be negative. The problems with delegation to a group is that it involves creating a subworkflow for the members of the group to execute the delegated task. Further work is required with regards to these extensions.

Appendix A. Details of Prolog implementation

Prolog is well suited for the implementation of the algorithms and definitions presented. For example the code for verifying if a delegation can be accepted is simply written as

```
%% acceptance of a specific delegation

accept-delegation(Grantor,Delegate,
                                 T+D,Case)
:-
    task-right(T),
    accept-delegation(Grantor,
               Delegate, T, Case),
    accept-delegation(Grantor,
                Delegate,D,Case).


accept-delegation(Grantor,Delegate,
                                  R,Case)
:-
      can-accept(Grantor,Delegate,
                                R,Case),
         asserta(accepted-del(
          Grantor,Delegate,R,Case)),
      add-to-storage(accepted-del(
       Grantor, Delegate, R, Case)).
```

```
% verify if can accept task right

can-accept(Grantor,Delegate,T,Case)
:-
    task-right(T),
    has-max(Grantor,TT,Case),
    greq(TT,T),
    has-max(Grantor,D1,Case),
    ( D1 = ud(TPlus,_)
    ; D1 = cd(TPlus,Q,_)),
    satisfy(Delegate,Q)
    ),
    greq(TPlus,T),
    ok-generic-constr(Delegate,T).


% verify if can accept a delegation
right

can-accept(Grantor,Delegate,D,Case)
:-
    delegation-right(Delegate,T),
    has-max(Grantor,TT,Case),
    greq(TT,T),
    has-max(Grantor,DD,Case),
    decrement(DD,Dminus),
    greq(Dminus,D),
    ( Dminus = ud(TPlus,_)
    ; Dminus = cd(TPlus,Q,_),
    satisfy(Delegate,Q)
    ),
    greq(Tplus,T),
    ok-generic-constr(Delegate,D).
```

where

- accepted delegations are asserted as the fact `accepted-del(Grantor,Delegate,R,Case)` both in the Prolog fact base and in the permanent storage (which is performed by the `add-to-storage` predicate).
- `has-max`(U,R,C) if R is a maximal right that user U has for case C. For direct rights it verifies the list of pre-computed maximal rights for a user (independent of the case), and for delegated rights it returns the maximally stronger rights received by the user.
- `greq`(R1,R2) verifies that $R1 \geqslant R2$. The code for `greq` follows from the definition.
- `satisfy`(U,Q) is true if user U satisfy all predicates listed in Q
- `ok-generic-const`(U,R) verifies if user U does not violate any generic constraint, if she receives the right R.

References

[1] R. Sandhu, E. Coyne, H. Feinstein, C. Youman, Role-based access control models, IEEE Comput. 29 (2) (1996) 38–47.

[2] E.S. Barka, R. Sandhu, Framework for role-based delegation models, in: 16th Annual Computer Security Applications Conference, December 2000. http://www.acsac.org/2000/abstracts/34.html.

[3] E.S. Barka, R. 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.

[4] W. Yao, K. Moody, J. Bacon, A model of oasis role-based access control and its support for active security, in: Proceedings of Sixth ACM Symposium on Access Control Models and Technologies, SACMAT 2001, 2001, pp. 171–181.

[5] C. Ramaswamy, R. Sandhu, Role-based access control features in commercial database management systems, in: National Information Systems Security Conference, 1998.

[6] J. Wainer, A. Kumar, P. Barthelmess, WRBAC—a workflow security model incorporating controlled overriding of constraints, Int. J. Coop. Inf. Syst. 12 (4) (2003) 455–486.

[7] J. Wainer, A. Kumar, A fine-grained, controllable, user-to-user delegation method in RBAC, in: Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT 05), 2005.

[8] E. Bertino, E. Ferrari, V. Atluri, The specification and enforcement of authorization constraints in workflow management systems, ACM Trans. Inf. Syst. Secur. 2 (1) (1999) 65–104.

[9] A. Kumar, W.M.P. van der Aalst, H.M.W. Verbeek, Dynamic work distribution in workflow management systems: how to balance quality and performance?, J. MIS 18 (3) (2002) 157–193.

[10] G. Baggio, J. Wainer, C. Ellis, Applying scheduling techniques to minimize the number of late jobs in workflow systems, in: SAC '04: Proceedings of the 2004 ACM symposium on Applied computing, ACM Press, New York, NY, 2004, pp. 1396–1403.

[11] V. Atluri, A. Gal, An authorization model for temporal and derived data: securing information portals, ACM Trans. Inf. Syst. Secur. 5 (1) (2002) 62–94.

[12] X. Zhang, S. Oh, R. Sandhu, in: PBDM: a flexible delegation model in RBAC, SACMAT '03: Proceedings of the Eighth ACM symposium on Access Control Models and Technologies, ACM Press, New York, 2003, pp.149-157.

[13] L. Zhang, G.-J. Ahn, B.-T. Chu, A rule-based framework for role-based delegation and revocation, ACM Trans. Inf. Syst. Secur. 6 (3) (2003) 404–441.

[14] C. Ruan, V. Varadharajan, Resolving conflicts in authorization delegations, in: Seventh Australian Conference on Information Security and Privacy, Lecture Notes in Computer Science, vol. 2384, Springer, Berlin, 2002, pp.271–285.

[15] J. Bacon, K. Moody, W. Yao, A model of oasis role-based access control and its support for active security, ACM Trans. Inf. Syst. Secur. 5 (4) (2002) 492–540.

[16] J.S. Park, Y.L. Lee, H.H. Lee, B.N. Noh, A role-based delegation model using role hierarchy supporting restricted permission inheritance, in: Proceedings of the International Conference on Security and Management, SAM '03, 2003, CSREA Press, pp.294-302.

[17] R. Tamassia, D. Yao, W.H. Winsborough, Role-based cascaded delegation, in: Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies, ACM, 2004, pp. 146–155.

[18] W. Yao, Fidelis: a policy-driven trust management framework, in: Trust management, First International Conference, iTrust, Lecture Notes in Computer Science, vol. 2692, Springer, Berlin, 2003, pp.301–317.

[19] O.L. Bandmann, B.S. Firozabadi, M. Dam, Constrained delegation, in: IEEE Symposium on Security and Privacy, 2002, pp. 131–142.

[20] E. Bertino, E. Ferrari, V. Atluri, A flexible model supporting the specification and enforcement of role-based authorization in workflow management systems, in: Proceedings of the Second ACM workshop on Role-Based Access Control, 1997, pp. 1–12.

[21] S. Castano, F. Casati, M. Fugini, Managing workflow authorization constraints through active database technology, Inf. Syst. Front. 3 (3) (2001).

[22] V. Atluri, W-K. Huang, An authorization model for workflows, in: Proceedings of the Fifth European Symposium on Research in Computer Security, Lecture Notes in Computer Science, vol. 1146, Springer, Berlin, 1996, pp.44–64.

[23] V. Atluri, W-K. Huang, A petri net based safety analysis of workflow authorization models, J. Comput. Secur. 8 (2/3) (1999).

[24] W-K. Huang, V. Atluri, Secureflow: a secure web-enabled workflow management system, in: Proceedings of the Fourth ACM Workshop on Role-Based Access Control on Role-Based Access Control, 1999, pp. 83–94.

[25] P.C.K. Hung, K. Karlapalem, J.W. Gray III, A study of least privilege in CapBasED-AMS, in: Proceedings of the Third IFCIS International Conference on Cooperative Information Systems, 1998, pp. 208–217.

[26] K. Karpalem, P. Hung, Security enforcement in activity management systems, in: NATO Advanced Study Institute (Ed.), Advances in Workflow Management Systems and Interoperability, 1997, pp. 166–194.

[27] G. Herrmann, G. Pernul, Viewing business-process security from different perspectives, Int. J. Electron. Commer. 3 (3) (1999) 89–103.

[28] G. Herrmann, Security and integrity requirements of business processes—analysis and approach to support their realisation, in: Proceedings of CAiSE*99 Sixth Doctoral Consortium on Advanced Information Systems Engineeering, 1999, pp. 36–47.

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

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

[31] G.-J. Ahn, R. Sandhu, M.H. Kang, J.S. Park, Injecting RBAC to secure a web-based workflow system, in: Fifth ACM Workshop on Role-Based Access Control, Berlin, Germany, July 2000. http://citeseer.nj.nec.com/ahn00injecting.html.

[32] A. Kumar, A framework for handling delegation in workflow management systems, in: Proceedings of Workshop on Information Technologies, 1999.

[33] V. Atluri, E. Bertino, E. Ferrari, P. Mazzoleni, Supporting delegation in secure workflow management systems, in: IFIP WG 11.3 Conference on Data and Application Security, 2003.