International Journal of Cooperative Information Systems
Vol. 12, No. 4 (2003) 455–485
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

W-RBAC — A WORKFLOW SECURITY MODEL INCORPORATING CONTROLLED OVERRIDING OF CONSTRAINTS

JACQUES WAINER

Institute of Computing, State University of Campinas
Campinas, 13083-970, São Paulo, Brazil
wainer@ic.unicamp.br

PAULO BARTHELMESS

Department of Computer Science, University of Colorado
Boulder, Colorado, 80309-0430, USA
barthelm@colorado.edu

AKHIL KUMAR

The Smeal College of Business Administration, The Pennsylvania State University
University Park, PA 16802-3009, USA
akhilkumar@psu.edu

This paper presents a pair of role-based access control models for workflow systems, collectively known as the W-RBAC models. The first of these models, W0-RBAC is based on a framework that couples a powerful RBAC-based permission service and a workflow component with clear separation of concerns for ease of administration of authorizations. The permission service is the focus of the work, providing an expressive logic-based language for the selection of users authorized to perform workflow tasks, with preference ranking. W1-RBAC extends the basic model by incorporating exception handling capabilities through controlled and systematic overriding of constraints.

Keywords: Workflow management; access control; RBAC; constraints.

1. Introduction

1.1. *Workflow management systems*

Workflow management systems, or workflow systems for short, allow for the definition and enactment of business procedures. A workflow system stores definitions of procedures in terms of their tasks, definitions of users that should perform the tasks (usually as roles), and a partial ordering of tasks that establishes constraints on task execution. More formally, we will define a workflow procedure W to be a tuple $(\mathcal{T}, \mathcal{P}_{\prec})$, where $\mathcal{T}$ is a set of tasks and $\mathcal{P}_{\prec}$ is a partial order relation on $\mathcal{T}$.

Example 1. Take for instance a typical process for *reimbursement of operational costs*, $W = (\{request, auditing, approval1, approval2, refund, rejection\}, \mathcal{P}_{\prec})$, with

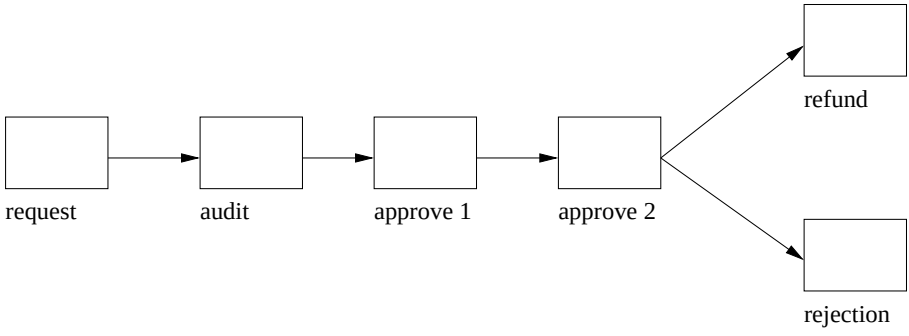

Fig. 1. Ordering of tasks in a reimbursement process.

$\mathcal{P}_{\prec}$ defined as shown in Fig. 1. A reimbursement process is started by a *request* task in which an employee requesting reimbursement inputs information. The request is then analyzed in an *auditing* task, followed by two *approvals*. Depending on the results of the approval tasks, either a *rejection* task is initiated to format the reasons for denial and inform the requester, or a *refund* is issued. The partial ordering of tasks establishes constraints on task execution, so that, e.g. *request*, *auditing* and the two rounds of *approval* may be required to be executed strictly sequentially, and *rejection* or *refund* (which are mutually exclusive), can be started as soon as the second round of approvals has been completed.

Typically, workflows have a mix of automatic tasks and tasks that require performance by humans. In the reimbursement process, for instance, most tasks require human intervention, and perhaps *rejection* and *refund* might be automatic, either generating an email to be sent to the requester, or sending out an electronic order for the payment, respectively.

Once a workflow procedure W such as the one just described is defined, a workflow system automates (or enacts) it by creating instances (also known as *cases* in the workflow literature). At any time, zero or more instances of a workflow procedure might be enacted. Each instance is a forward-recoverable long transaction that has a private state that is isolated from potentially other instances of the same process, e.g. "Anna's request for travel expenses reimbursement for her trip to Cairo." We will define $\mathcal{C}$ as the set of all cases of W. $c_i \in \mathcal{C}$ represent an instance (or single case) of W.

Each case c_i might potentially be executing a different set of tasks at each moment. Instances of tasks are created as their preconditions are met (e.g. as the previous tasks they depend upon are completed), followed by a user selection phase for those tasks that require human intervention. Once a specific user is selected, a workflow system makes available for her the data and applications that are required for the performance of the task. Once a user declares a task completed, the workflow system updates its state and looks for other tasks that might have become enabled for execution, and the instantiation and user selection process is

repeated. Execution state is stored in persistent logs, allowing forward recovery in the presence of execution disruptions.

The focus of the present paper is on the selection of users to 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). While any employee can, in principle, fill out a reimbursement request, only certain users should perform other various tasks. *Auditing*, for example, should only be performed by *auditors*. The same applies to the remaining *approval* tasks. Clearly, it is necessary to select users who have enough authority to grant the approval. At the same time, it is necessary to avoid situations in which, for instance, an employee gets to audit or approve his own requests.

Conventional workflow systems rely on simple mechanisms that are usually based on assigning users to roles and roles to tasks, so that only those users that can play a role that is associated with a task will be able to perform it. In other words, in these systems, a task $T \in \mathcal{T}$ is defined as a tuple (D, R), where D is some sort of description and $R \in \mathcal{R}$ is one element from a set of roles $\mathcal{R}$ (of course a task may include information on other aspects as well, but here we restrict the discussion to the elements that are related to user selection). Users are in turn associated to one or more roles they can play. In the usual way, such systems select users to execute a task is to "offer" new tasks to all employees that can play the associated role R, by inserting a new item in their electronic "in baskets". Once one of the potentially many users starts executing the item, it is retracted from other users' in baskets and this user becomes the assigned executor of the task instance. This mechanism, while flexible, does not take into consideration authorization constraints on the roles, such as *binding* and *separation of duties*. Separation of duties is a traditional business technique that tries to minimize fraud by spreading the responsibility and authority for an action or task over multiple users, thereby decreasing the risk involved in committing a fraudulent act by requiring the involvement of more than one individual.[1] Binding of duties, on the other hand, refers to having the same user that performed a task be responsible for one or more other related tasks, e.g. because of the exogenous knowledge that the user acquires while performing the first task, or to simplify the interaction with a client. The conventional user selection mechanism employed by workflow systems does not allow for such a fine grained specification of a user selection criterion.

Example 2. In real world scenarios, one would wish to specify execution requirements for Example 1 above such as:

(1) The *request* task can be performed by any employee.
(2) The *auditing* task can be executed by anyone that can play the role of *auditor* (2.1), provided this user is not the requester herself (2.2). Furthermore, it is

preferred that both the requester and the auditor be members of the same organizational unit (2.3).

(3) The first *approval* task should be executed by a head of the organizational unit from which the employee is requesting reimbursement (3.1), and she cannot be the requester herself (3.2). Lets call this user *Approver1*.

(4) The second *approval* needs to be performed by somebody at the same or at a superior hierarchical level of *Approver1* (4.1). It cannot be the requester(4.2) or *Approver1* (4.3).

where the number in parenthesis is a reference to the (sub) requirement. It is important to notice that not only categorical requirements are needed, for example, Secs. 2.1 and 2.2, but also preference requirements such as Sec. 2.3.

It is clear that the simple role assignment used by conventional workflow systems is not expressive enough to handle even simple requirements as the ones just enunciated. There is therefore a clear need for enhancements to the user selection mechanisms, both in terms of security and expressiveness.

1.2. *Role-based access control model*

The Role-Based Access Control model (RBAC) (for example, Sandhu *et al.*[17]) is receiving attention as a systematic way of implementing the security policy of an organization. It groups individual users into roles and assigns permissions to various roles according to their stature in the organization. Roles are organized in a hierarchy or lattice; other additional structures, such as groups and privilege hierarchy offer additional expressive power (e.g. Nyanchama and Osborn[14]). RBAC offers therefore an interesting and well understood paradigm that extends the simple use of roles employed in conventional workflow systems, yet shares the notion of employing roles as the semantic construct that is at the center of authorization.

Formally an RBAC model is described by (1) **entities**: *users* $\mathcal{U}$, *roles* $\mathcal{R}$, *privileges* $\mathcal{P}$, (2) **relationships** between these entities, and (3) **constraints** over these relationships. A meta-model is displayed in Fig. 2.

- a user $u \in \mathcal{U}$ represents individual users;
- a privilege $p \in \mathcal{P}$ represents classes of rights to perform operations, tasks, access data, and so on, possibly with explicit attributes. For example,

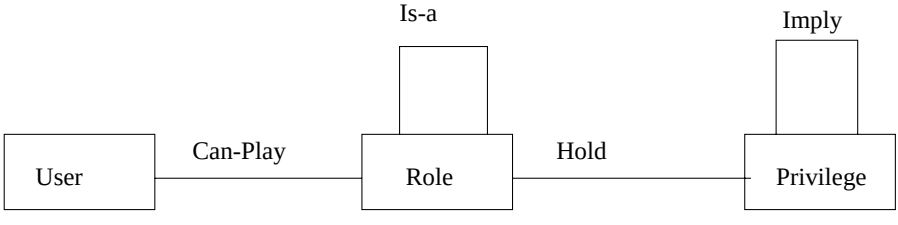

Fig. 2. Meta model.

travel-approval(US$500) represents the right to execute the task of approving travel expenses up to US$500;

- a role $r \in \mathcal{R}$ describes meaningful groupings of privileges or abilities that can be assigned to users, e.g. the roles of *manager*, *auditor*, and so on.

RBAC defines the following relations:

- can-play(u, r), $u \in \mathcal{U}, r \in \mathcal{R}$, states that user u can play the role r. can-play(*mary*, *manager*) says that Mary can play the role of manager.
- is-a(r_1, r_2), $r_1, r_2 \in \mathcal{R}$, states that role r_1 is a kind of (and thus inherits all privileges) role r_2. is-a(*c-programmer*, *programmer*) states that *c*-programmers are a kind of programmer, and thus a *c*-programmer has all the rights that a programmer has, and possibly more.

  If is-a(r_1, r_2) is true, we will say that r_1 is **larger** than r_2, and conversely that r_2 is **smaller** than r_1. These are not standard RBAC names, but we feel that they are neutral enough not to convey any ordering of importance between the roles. r_1 is larger than r_2 because it hold all the privileges that r_2 does, and more.
- hold(r, p), $r \in \mathcal{R}, p \in \mathcal{P}$, states that role r holds the privilege p.
- imply(p_1, p_2), $p_1, p_2 \in \mathcal{P}$, state that privilege p_1 is stronger, or supersedes, or includes p_2. For example, the right to approve travel expenses up to US$1000 implies (is stronger than) the right to approve travel expenses up to US$500.

  If imply(p_1, p_2) is true we will say that p_1 is **stronger** than p_2

There are some implicit inheritance structures in RBAC. The first one is defined by the is-a relation among roles, that follows the RBAC model: if is-a(r_1, r_2) then for all p such that hold(r_2, p), it is also the case that hold(r_1, p).

The second inheritance structure is defined by the imply relation: if imply(p_1, p_2) then in every situation in which p_2 is the appropriate right to accomplish something, p_1 can also be used to accomplish that.

The RBAC meta model also includes the concept of constraints that can restrict the co-occurrence of different relations. For example, one may restrict the number of different roles that any user can play to at most five. This constraint would not allow the co-occurrence of more than five instances of the can-play relation for any user.

The RBAC meta-model contains another entity called *session*, which represent the (temporarily bounded) exercise of a role by a user. In the RBAC model, constraints that refer to session entities are called *dynamic* constraints, and constraints that do not refer to sessions, are called *static*.

1.3. *Goals and contributions of this paper*

Workflow systems have a clear notion of history (through the persistent logs employed for recovery, among other uses), which matches quite well the notions of *binding of duties* and *dynamic separation of duties*, usually handled in the RBAC

literature through the association of *sessions* (e.g. in Goh and Baldwin,[1] Ferraiolo *et al.*[7]). During a session, users are associated with a (sub)set of roles the user can play. While users are not prevented from holding roles that conflict (because they hold incompatible privileges such as *request* and *audit*, for instance), they are dynamically prevented from playing these conflicting roles in the context of the same session. Sessions are not so clearly useful in the context of business processes: one does not want to prevent a user from assuming the roles of requester and auditor *at the same time*, but one wants to prevent one for assuming both roles *for the same reimbursement process*. The concept of a "reimbursement process" is typically a workflow concept; there is therefore a clear advantage in combining the complementary strengths of workflow systems and RBAC. This is essentially what is described in the present work. The present paper is therefore related to similar efforts, e.g. Bertino, Ferrari, and Atluri,[3,4] Castano *et al.*[6] and others (see the Related Work section for details and a comparison of approaches).

We start by defining a basic model — W0-RBAC, 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 in the first paragraphs of this introduction), the permission service handles the selection of authorized, most appropriate users to execute each task, based on an organizational and authorization model. W0-RBAC defines a protocol that regulates the interaction between these two components, 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, thus making the administration of authorization data easier.

The permission service is based on the RBAC model with extensions and is built upon an expressive meta-model that includes users, hierarchies of organizational units, roles, and privileges. The latter is taken in the present work to represent the privilege to execute a task. Users are prevented from exercising incompatible privileges through the specification of *static* and *dynamic constraints* expressed in a logic-based language. The use of logic as a specification mechanism has a long history, including its use for specifying workflows, e.g. Wainer,[20] representing authorization, e.g. Li *et al*,[12] and permission constraints in workflow systems, e.g. Bertino *et al.*[4]

A powerful user selection language allows a fine-grained specification of executors for each task. An "on-the-fly" ordering can be imposed to rank users according to an arbitrary preference criteria.

W1-RBAC extends W0-RBAC to include a mechanism for controlled overriding of constraints. It is well-known that workflow enactment is subject to deviations or unanticipated situations, known in the workflow literature as *exceptions*. W1-RBAC incorporates systematic and controlled exception handling capabilities by supporting the assignment of levels of priority for each constraint, and by defining a privilege to override constraints up to a particular level. In the presence of exceptions users can override constraints up to their authorized levels, allowing the

work to flow forward despite exceptions. Finally, W1-RBAC is able to deal with the potentially contradictory set of facts and constraints that are created when constraints are overridden.

In summary, contributions of this work are:

- A framework that couples a powerful permission service and a workflow component with clear separation of concerns for ease of administration of authorizations.
- The framework allows for the definition of preferences in the selection of users to perform tasks.
- Controlled and systematic overriding of constraints for exception handling.
- A prototype implementation of the system.
- The discussion of the complexity of the solution and possible optimizations.

1.4. *Organization of the paper*

This paper is organized as follows. Section 2 discusses related works. Section 3 describes the basic extensions we propose to the RBAC model, which include organizational units and cases. Section 4 gives an architecture for a permission service and shows how it will interface with a workflow model. Section 5 describes our approach for controlled overriding of constraints. In Sec. 6, we discuss implementation of our proposal. Section 7 discusses future work.

2. Related Work

The pioneering work of Sandhu[18] introduced separation of duties in the context of multi-step transactions through the notion of *transaction control expressions*. In this work, Sandhu associates transactions steps to roles. Each user executing a step in a transaction had to be different. To enforce this, the history of the execution of each transaction sequence had to be maintained. This early model can be related to simple role-based workflows with the strong restriction that users could execute only a single task for each case.

Ferraiolo *et al.*[7] has a similar concept of *operational separation of duties*, requires that, for all the operations associated with a particular business function (equivalent to a procedure in a workflow), no single user can be allowed to perform all of these operations. This permission mechanism is not expressive enough to allow implementation of the detailed assignments that arise in real-world business processes like the one presented in Example 2 above.

Bertino, Ferrari, and Atluri[3,4] propose an interesting and powerful constraint-based security model also based on logic predicates, that allows for somewhat different expressivity than the one presented here. Predicates in constraint expressions include predicates over a role graph and predicates over history (user and role that executed some task), task activation and task outcome (success, abort).

Because the language we propose includes the concept of *case* (workflow instance), we are able to express inter-case constraints, for example, *reciprocal*

separation of duties: if in a case, Amanda approved Beth's travel reimbursement, then Beth cannot be the approver of Amanda's request (Sec. 3.5). This is clearly a desirable constraint in a real-world application, which cannot be handled by Bertino *et al*'s model, because it can only express constraints within workflow instances, and not across them as required here.

We also include in our ontology the concept of *organizational units*, which is an extension of *groups* proposed in Nyanchama and Osborn's,[14] thus allowing one to refer to the power hierarchy in the organization. Operations on the role hierarchy such as the ones provided by Bertino *et al.*[4] are not enough to represent real world constraints (please see discussion in Sec. 3.2).

A large section of Bertino *et al.*[4] describes optimizations of the basic constraint verification mechanism. We believe that at this point in time there is not enough experience in this domain to further evaluate which method or combination of methods would reduce the overall cost of authorization related predicates, or even if any of those optimizations are necessary for the "average case". The present work discusses, nonetheless, possible alternatives for such optimization, should they prove to be necessary (Sec. 6.1).

Bertino *et al.*[4] orders users according to a single criteria, based on the default assumption that if a role r_i is larger than role r_j in the role partial order, then r_j should be given higher priority over r_i when assigning a role to a task. We discuss the need for alternative orderings and propose a flexible solution to implement them (Sec. 4.2).

Also the cited work uses an uncommon early definition of the executors of an activity: at the start of the case the mapping between activity and its executors is defined. We followed a more standard approach of delaying this binding to the moment the activity becomes enabled. Finally, that work does not include any form of overriding constraints, which is essential for exception handling, and, despite the name, exceptions are actually very common in workflow enactment.

Castano *et al.*[6] propose an active-rule based workflow security model which is implemented on top of the Wide workflow system.[16] ECA (event-condition-action) rules are employed to specify instance, history and event constraints. Selection of agents to which tasks are assigned is discussed, e.g. first trying to assign tasks to authorized users placed in lower positions of the role/level hierarchies. Actual support for policies is not presented in their paper. Castano *et al.* also include the notion of temporal constraints. Each of the relations of the meta-model can be annotated with a time specifier so that the validity of a relation, say can-play, between user and role, only holds during the specified time period. Such constraints could be easily incorporated in our model, but we consider that the usefulness of such constraints would be to express revocation of rights, which is a much broader issue on its own, and dealt with elsewhere.[21]

Hung and Karlapalem[9,10] present the security features of the CapBasED-AMS workflow system and discuss the trade-off between security and risk of a system and present a metric to evaluate such trade-off. The risk factor is equated to the

number of tasks any user executes in a given instance (case). The rationale is that users that perform more tasks are more knowledgeable about the task being performed and thus pose a higher security risk. Failure resilience, the ability to complete a task, on the other hand, would depend on more than one user being able to execute each task. A greedy algorithm is proposed, that determines task assignments that would achieve high failure resilience and low security risk factor according to these definitions. Under this perspective, our ordering mechanism, coupled with controlled overriding of constraints, would provide dynamic failure resilience, by broadening the choices of users who are able to execute each task. Overriding of constraints is not considered by Hung and Karlapalem, and neither is the flexible selection and ordering of users that we propose here.

The need for overriding a workflow security system is also discussed in Miller *et al.*,[13] in the context of a health care workflow application. The mechanism, implemented as part of the METEOR workflow system, is called "Break-Glass Procedure", and allows certain authenticated users to temporarily assume greater privileges or higher roles. System response in this case is to employ maximal auditing/tracking and inform a workflow administrator. Our approach allows overriding to be controlled in a progressive way, allowing therefore for a more controlled, graceful degradation than the all-or-nothing approach proposed there.

A few commercial workflow systems offer some functionality related to role authorization constraints. We borrow our description from Castano *et al.*[6]:

- IBM WebSphere MQ (formerly MQSeries workflow) — allows the definition of binding of duties constraints [www-3.ibm.com/software/ts/mqseries/].
- Staffware2000 — allows static definition of binding of duties, with some restrictions — the definition holds for all instances and cannot be used in tasks that join flows from multiple tasks. Authorizations that are valid for a restricted period of time are also supported [www.staffware.com].
- InConcert — allows the definition of external applications that are invoked at user selection time to determine the role to which the task should be assigned.
- Cosa — allows the definition of role hierarchies, in which authorizations can be inherited along the hierarchy. Binding, separation of duties and time-restricted authorizations can be defined by using a language [http://www.ley.de/cosa].

The user selection languages employed by these commercial systems are not so expressive as the ones discussed in the present work. Mechanisms for overriding constraints such as the ones discussed here are also not offered by any of these commercial workflow systems.

3. W0-RBAC Model: The Extended RBAC model

The W0-RBAC model adds to the RBAC meta-model (Sec. 1.2), the new entities: *case* $\mathcal{C}$ and *organizational unit* $\mathcal{OU}$. W0-RBAC also adds the relations: include, member, head, and doer.

As for privileges, in this paper, we are interested in just a single form of privilege — the right to execute a task. An important set of rights that will not be discussed in this paper are the administrative rights, that is, the right to add new users, roles, constraints, and so on. In fact, all aspects of administration of the system are outside the scope of this paper.

3.1. *Case and doer relation*

The limitations of static constraints are well known in the security literature. A static constraint may forbid a user from holding the roles of pilot and navigator of a plane, but that is not exactly what is needed. A pilot can be a navigator if needed; what one would like to forbid is for the same person to be both the pilot and the navigator *in the same flight*. This particular constraint can be captured in RBAC by making use of sessions, that is, the binding between user and roles fixed in some time interval. In this case, one might forbid a session in which a user is bound to both pilot and navigator.

In workflow applications, the concept of a session is less clear because the temporal bound is less well defined. For example, one would like to forbid the situation in which the same user executes the tasks of *request* and *approve* for *the same reimbursement request process*, or in terms of roles, one would like to avoid that the same user should play both the *requester* and *approver* roles, for the same reimbursement request. But of course Beth may be the approver for Carol's request, and may herself be the requester of a different reimbursement process, which must be approved by her boss Amanda. The concept of session is unclear here because Beth may be *at the same time* requesting her reimbursement and approving Carol's, and that is acceptable provided these roles are being played in *different* reimbursement instances.

To be able to refer to an *instance of a process* we add to our model a new class case, as described in Fig. 3 (recall that case is one of the standard ways instances of a procedure are referred to in workflow systems). We also define a new ternary relation $\mathsf{doer}(u, p, c)$, $u \in \mathcal{U}, p \in \mathcal{P}, c \in \mathcal{C}$, which means that a user (u) exercised a particular privilege (p) on a particular case (c). Alternatively, given the association between privileges and rights to perform a tasks, $\mathsf{doer}(u, t, c)$ states that user u executed the task t for a particular case 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 he exercised the privilege p on instance c.

3.2. *Organizational units*

Within organizations, and thus in workflow applications, the concept of a hierarchy of people/roles is prevalent. People are placed in one or more units such as departments, divisions, or groups, and they have different bosses, at different hierarchical levels. Thus, it is very common in a business process that a reimbursement request made by an employee must be approved by the head of the unit that employee is

statically assigned to, or by the head of the project the employee is dynamically assigned to and on behalf of which the expense was made, and so on.

While workflow systems as a rule include some form of organizational modeling capabilities, RBAC by itself does not have such an hierarchy modeled. The closest concept is that of a *group* presented in extended models such as Nyanchama and Osborn's[14] and Osborn and Guo's.[15] To that we add an explicit notion of who the head of the group is to allow us to model a power relation within the permission component. This power hierarchy allows us to express security constraints and ordering preferences that are based on it (see examples below).

It is important to discuss the differences between the proposed organizational unit hierarchy and the one that is based on RBAC's role hierarchy, that is more commonly used (e.g. by Bertino *et al.*[3,4]). The is-a relation among roles is an inheritance relation, useful for administrative purposes, which must not be confused with a power relation. is-a (*c-programmer*, *programmer*) states that a *c*-programmer has all privileges of a programmer, not that a *c*-programmer is the boss of a programmer. In some particular examples the is-a relation may be overloaded into a power relation, but that is only possible in very small examples where higher levels in the organization accumulate the privileges of those under their control. In any non-trivial organization, the president of the company, although the boss of the database administrator, does not have the privileges of that role. Similarly, the head of the hospital, although the boss of every physician, does not have the right to operate on a patient.

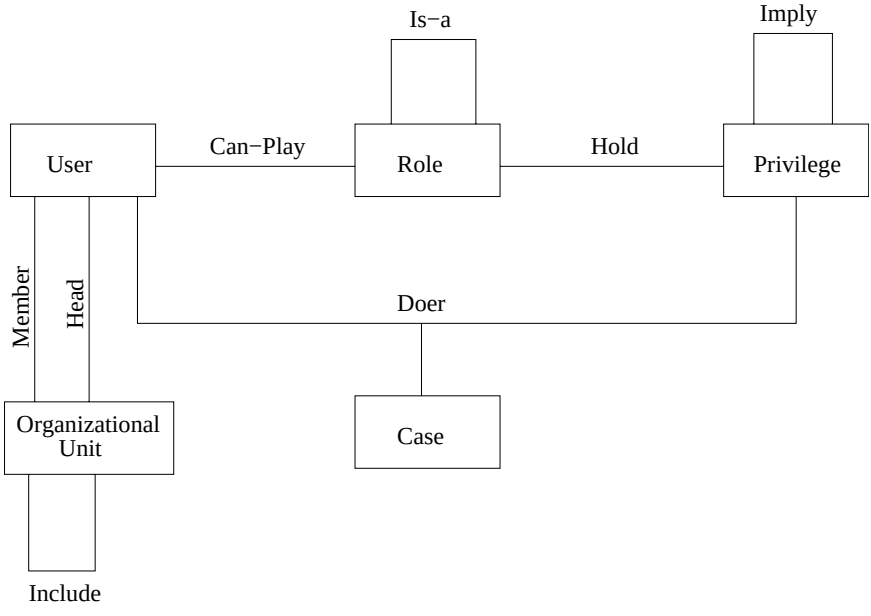

Fig. 3. W0-RBAC meta model.

There have been other research that proposed extensions to the RBAC role and to the user concepts as presented in the core RBAC model, in order to facilitate the use of RBAC in real life situations, e.g. Kappel *et al.*[8] and Osborn and Guo.[15] The proposal of organizational units should be viewed with this perspective. In order to model requirements such as the ones in Example 2, a boss hierarchy must be modeled, and that can be accomplished using the organizational units. Furthermore, organizational units seem to deal with both the static assignment of users to groups, departments, divisions, etc. as well as matrix like organizations, where users are dynamically assigned to one or more projects.

We define the following relations:

- $\mathsf{include}(d_1, d_2)$, $d_1, d_2 \in \mathcal{OU}$, states that the organizational unit d_1 includes the organizational unit d_2. For example, $\mathsf{include}$(*engineering dept*, *project 12 team*) states that the project 12 team is a part of the engineering department.
- $\mathsf{member}(u, d)$, $u \in \mathcal{U}, d \in \mathcal{OU}$, states that user u is a member of the organizational unit d.
- $\mathsf{head}(u, d)$, $u \in \mathcal{U}$, states that user u is the head, or the responsible party for the organizational unit d.

The W0-RBAC meta model is shown in Fig. 3.

Example 3. Some of the requirements in Example 2 are defined using the basic tools of W0-RBAC (or plain RBAC). For example, requirement 1 is modeled by defining a least role *employee* so that all other roles inherit from it, and ensure that hold (*employee*,*request*) is true.

Requirement 2.1 is modeled by defining an *auditor* role and asserting hold(*auditor*,*auditing*).

3.3. *Constraints*

In W0-RBAC, constraints are expressed as a standard logic program clause Cn, that is false, represented by the predicate $\perp$. Thus,

$$\perp \leftarrow Cn$$

states that the situation described in Cn is invalid, that is, it violates the constraint. The clause Cn is expressed in standard logic program form, that is,

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

where either k or 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 in the W0-RBAC meta model, 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, organizational units, or privileges). Variables

are represented in italics, such as x and c, and constants are represented in normal font, like "approve".

Example 4. The constraint that the requester and the executor of the first approval task cannot be the same person (requirement 3.2 in Example 2) is represented as:

$$\bot \leftarrow \mathsf{doer}(x, \text{request}, c), \mathsf{doer}(y, \text{approve1}, c), x = y\,.$$

The formula above should be read as "the following situation is invalid: the executor of task *request* for case c is x and the executor of task *approve1* for case c is y and x and y are the same".

Also, auxiliary predicates can be defined. A particularly useful one is $\mathsf{boss}(u_1, u_2)$ which is true if u_1 is the head of one of the organizational units of which u_2 is a member:

$$\mathsf{include}*(x, y) \leftarrow \mathsf{include}(x, y)$$
$$\mathsf{include}*(x, y) \leftarrow \mathsf{include}(x, z), \mathsf{include}*(z, y)$$

and

$$\mathsf{boss}(x, y) \leftarrow \mathsf{head}(x, ou), \mathsf{member}(y, ou),\ \textbf{not}\ x = y$$
$$\mathsf{boss}(x, y) \leftarrow \mathsf{head}(x, ou), \mathsf{member}(y, ou'), \mathsf{include}*(ou, ou'),\ \textbf{not}\ x = y\,.$$

Another useful auxiliary predicate verifies whether two users have the same hierarchical level, which is modeled as having a common boss, and the same number of intermediary bosses between the common boss and the users.

$$\mathsf{include2}*(x, y, 0) \leftarrow \mathsf{include}(x, y)$$
$$\mathsf{include2}*(x, y, \mathit{level}) \leftarrow \mathsf{include}(x, z), \mathsf{include2}*(z, y, \mathit{sublevel}), \mathit{level} = \mathrm{s}(\mathit{sublevel})$$

$$\mathsf{boss2}(x, y, 0) \leftarrow \mathsf{head}(x, ou), \mathsf{member}(y, ou),\ \textbf{not}\ x = y$$
$$\mathsf{boss2}(x, y, \mathit{level}) \leftarrow \mathsf{head}(x, ou), \mathsf{member}(y, ou'), \mathsf{include2}*(ou, ou', \mathit{level}), \textbf{not}\ x = y$$

$$\mathsf{same\text{-}level}(x, y) \leftarrow \mathsf{boss2}(z, x, l1), \mathsf{boss2}(z, y, l2), l1 = l2,\ \textbf{not}\ x = y\,.$$

In this predicate, numbers are represented using the constant 0 and the successor function "s", so 1 is represented as "s(0)" and so on.

Constraints can be established over any of the relationships of the meta-model and can be broadly classified into **static** and **dynamic**. A constraint is dynamic if any of the predicates in the clause is doer, because doer is the only predicate that makes reference to a running case.

3.4. *Static constraints*

Static constraints forbid the introduction of ill-formed relationships between users, roles, organizational units, and privileges, by specifying conditions under which such

relationships should not be allowed. The name *static* comes from the fact that these constraints do not depend on the execution of tasks. They control the structure of the security model independently of any dynamic behavior.

Example 5. For example, the static constraint that no user can have both the privileges of request and approve, which is *not* what is required in Example 2, is represented as:

$$\bot \leftarrow \mathsf{can-play}(u, r_1), \mathsf{can\text{-}play}(u, r_2), \\ \mathsf{hold}(r_1, \text{request}), \mathsf{hold}(r_2, \text{approve}),$$

that is, it is inconsistent to have a user u who can play both the roles r_1 and r_2 (not necessarily distinct), and where r_1 holds the privilege *request* and r_2 holds *approve*.

Static constraints are enforced when tuples are added or removed. After a transaction that inserts new users, roles, and the tuples that represent new instances of the relations, the static constraints must be checked. Using the constraint described in the paper, the result of checking the static constraints would be just the existence or absence of violations. Of course, a practical implementation of the constraint above should not only verify that a violation exists, but also determine which instantiation(s) of the variables u, r_1, and r_2 above cause the violation.

3.5. *Dynamic constraints*

In Sec. 3.1, we discussed that the concept of an instance of a process (a case) is necessary to fully describe dynamic constraints in the context of a business process.

Dynamic constraints may either block users and roles from performing some actions, or require them to perform specific actions on a case, depending upon their previous actions over that case:

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

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

- *Binding of duties* is just the opposite — a user who performed some action is bound to execute other related actions in the future, *for the same case*. 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, one can express this constraint as:

$$\bot \leftarrow \mathsf{doer}(u, \mathrm{T2}, c), \mathsf{doer}(u', \mathrm{T3}, c),\ \textbf{not}\ u' = u\,. \tag{3.1}$$

- *Inter-case constraints* allows constraints to refer to different case, for example, to refer to the number of times an activity was performed by someone. If the CEO

can appoint any two of her three executive officers, but not all three, then the business rule must be modeled as a constraint that relates different cases:

$$\begin{aligned}\bot \leftarrow\ & \mathsf{can\text{-}play}(u, \mathrm{CEO}), \mathsf{doer}(u, \text{appoint-exec-off}, c_1),\\ & \mathsf{doer}(u, \text{appoint-exec-off}, c_2), \mathsf{doer}(u, \text{appoint-exec-off}, c_3),\\ & \mathbf{not}\ c_1 = c_2,\ \mathbf{not}\ c_1 = c_3,\ \mathbf{not}\ c_2 = c_3\,.\end{aligned}$$

- *Reciprocal separation of duties* (a special case of inter-case constraints) can prevent coalitions across cases, for example, if in a case Amanda approved Beth's travel reimbursement, then Beth cannot be the approver of Amanda's request. The constraint is represented as:

$$\begin{aligned}\bot \leftarrow\ & \mathit{doer}(u, \text{request}, c_1), \mathit{doer}(v, \text{approve}, c_1),\\ & \mathit{doer}(v, \text{request}, c_2), \mathit{doer}(u, \text{approve}, c_2), \mathbf{not}\ c_1 = c_2\,.\end{aligned}$$

Example 6. Some of the requirements of Example 2 are represented as dynamic constraints. The number in parentheses before the expression refers to the requirements in Example 2.

$$\begin{aligned}&(2.2)\quad \bot \leftarrow \mathsf{doer}(x, \text{request}, c), \mathsf{doer}(y, \text{audit}, c), x = y\\ &(3.1)\quad \bot \leftarrow \mathsf{doer}(x, \text{request}, c), \mathsf{doer}(y, \text{approve1}, c),\ \mathbf{not}\ \mathsf{boss}(y, x)\\ &(3.2)\quad \bot \leftarrow \mathsf{doer}(x, \text{request}, c), \mathsf{doer}(y, \text{approve1}, c), x = y\\ &(4.1)\quad \bot \leftarrow \mathsf{doer}(x, \text{approve1}, c), \mathsf{doer}(y, \text{approve2}, c), \mathsf{lower\text{-}level}(y, x)\\ &(4.2)\quad \bot \leftarrow \mathsf{doer}(x, \text{request}, c), \mathsf{doer}(y, \text{approve2}, c), x = y\\ &(4.3)\quad \bot \leftarrow \mathsf{doer}(x, \text{approve1}, c), \mathsf{doer}(y, \text{approve2}, c), x = y\end{aligned}$$

boss is an auxiliary predicate defined in Sec. 3.3. lower-level is an auxiliary predicate that is true when the user in the place of the first argument is at a lower hierarchical level than the user in the place of the second argument. lower-level can be implemented in a similar fashion to the same-level predicate in Sec. 3.3.

4. An Access Control System Integrated with a WFMS

Our basic framework consists of an *access control service* or *permission system* attached to a workflow engine. The workflow system contains the knowledge about the processes, the ordering of tasks, deadlines, and so on. The permission service knows about the organizational structures, roles, permissions, etc. Figure 4 shows a diagram of the interchanges between the workflow engine, the permission service and users.

Workflow systems that address the issues of security[4,6,9,13] combine access control with workflow functionality in different ways. Our approach emphasizes a clear separation of duties between these two complementary functionalities.

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:

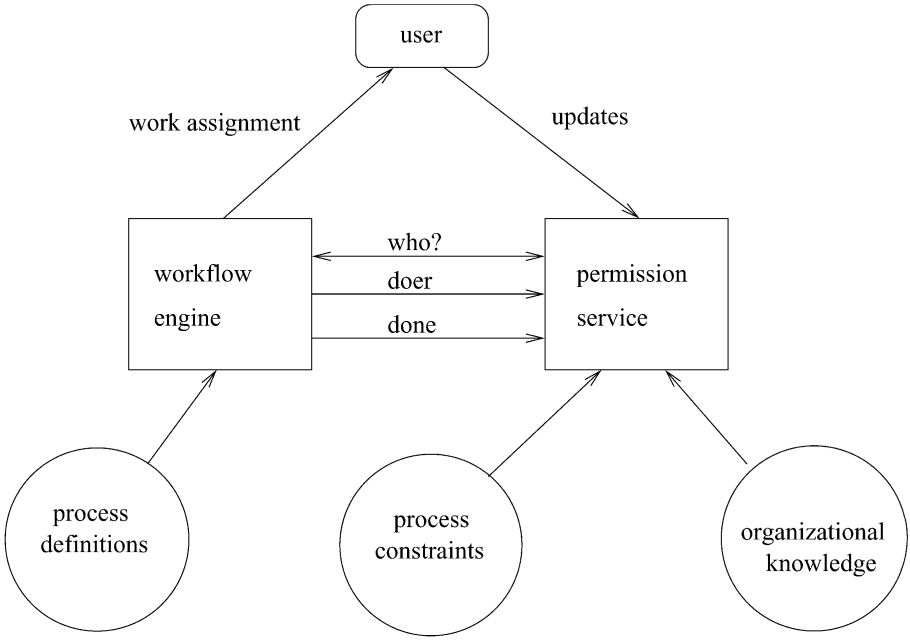

Fig. 4. Interaction between the system components and users.

- doer(u, t, c), $u \in \mathcal{U}, t \in \mathcal{T}, c \in \mathcal{C}$, which states that user u has done the task t for the process instance c.
- done(c), $c \in \mathcal{C}$ which states that the instance c has 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?(O, t, c), that is, who are the users that can perform the task t, for instance, c. The workflow receives back an ordered list of groups of users that satisfy all constraints, ordered according to the ordering scheme O, to be explained shortly.

The permission system stores the W0-RBAC tuples and the doer tuples sent by the workflow system in a data base (or knowledge base), which we will abbreviate as KB.

Recall that instances of new tasks are created as their preconditions are met (e.g. as previous tasks they depend upon are completed), followed by a user selection phase, for those tasks that require human intervention. The user selection in W0-RBAC is initiated by a query sent from the workflow component to the permission service, asking who are the users that can perform that task. The permission service computes a response based on the current KB, and returns an ordered list of groups of users, such that, users in each group are equally preferred (according to the specified ordering criteria). The workflow decides who among the users in the list will perform the next task, sends that information to the user's work list.

When the user informs the workflow that she accepts the task, the workflow informs the permission service by sending the doer tuple. When the last task of a process instance is finished, the workflow sends the done information to the permission service.

4.1. *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?(O, t, c) posed by the workflow. The permission system should return all users u, such that there exists a role r and can-play(u, r) and hold(r, t), and, further, doer(u, t, c) does not violate any integrity constraint. Furthermore, the permission system should order the users according to the two-place predicate O. Formally:

Definition 1. The permission system's **answer** to the workflow query who? (O, t, c) is the ordered list of groups of users $\langle g_1, g_2, g_3, \ldots g_n \rangle$, where $g_1 = \{u_{1,1}, u_{1,2}, \ldots u_{1,n_1}\}$ and $g_2 = \{u_{2,1}, u_{2,2}, \ldots u_{2,n_2}\}$ and so on, such that

- for each $u \in \{g_1 \cup g_2 \cup \cdots \cup u_n\}$, can-do$(u, t)$ is true.
- for each u, adding doer(u, t, c) does not violate any constraints.
- for all $u_a, u_b \in g_i$ both $O(u_a, u_b)$ and $O(u_b, u_a)$ are true (u_a and u_b are equally preferred by O).
- and for all $u_a \in g_i$ and $u_b \in g_j$ $O(u_a, u_b)$ is true for $i < j$.

Example 7. When Dana finishes the *request* task, the workflow system is informed and determines that *audit* is the next task to be performed for this case (which we assume has the identity *c120*). The workflow queries the permission system with who?(O_1, *audit*, *c120*), where O_1 is a two place predicate defined by the user who defined the workflow, which models the preference ordering for the executors of the task *audit*. The permission system returns an ordered list of groups of users (ordered according to O_1) that can perform *verify* for *c120*. The workflow will select the executor, say Eric, and inform the permission system using doer(*eric*, *audit*, *c120*). When this task is over, the workflow will query the permission system with who?(O_2, *approval1*, *c120*), and so on.

4.2. *Ordering the answers*

An important feature of our permission system is that it ranks the users who can perform a task based on their suitability and the knowledge stored in the permission system. For example, it may happen that a large set of users can execute a task t for a case c. How to order this set of users, so the workflow is informed of which user is the most appropriate to perform the task?

A standard practice in RBAC-based workflow systems[3,4,6] is to state that the "least specific" role that holds a privilege is the most appropriate one to execute that privilege. Thus if *compile* is the privilege that is needed, one must choose the

smallest role that holds that privilege, say a *programmer* role, instead of larger roles, such as a *C-programmer* role.

But in the business domain the most specific role rule does not translate itself into a unique answer to the problem of ordering *users*. Let us suppose that Falco, the head of the quality project team can play the role of programmer, and Gail, a member of the team, can play the roles of a *C*-programmer and programmer. Should Falco or Gail be the most preferred as the possible performer of the task *compile*?

There are many possible definitions of what it means for a user to be preferred than other users. In particular, our model suggests three implicit or explicit orderings: the smaller/larger ordering between roles (the user that plays the smaller role is preferred), the stronger/weaker ordering between privileges (the user that can exercise the weaker privilege is preferred), and the boss relation between users (the subordinate user is preferred), and there may be no correspondence among those three orderings.

Example 8. Figures 5 and 6 (based on Nyanchama and Osborn[14]) show graphically an example of the use of a power structure as described above. Figure 5 describes an example of one organizational unit m, two users a and b, two roles r and s, two privileges x and y, and the relations among them. The relation between organizational unit and user is member, the relation between user and role, can-play, and the one between role and privilege is hold. The directed line in the privilege set represents the imply relation, that is, a line from x to y states that x implies y, or in other words, that x is stronger than y. In Fig. 6, a line from r to q states that r is smaller than q.

For the relations shown in Fig. 5, let us suppose that the workflow wants to find out the users that posses privilege y.

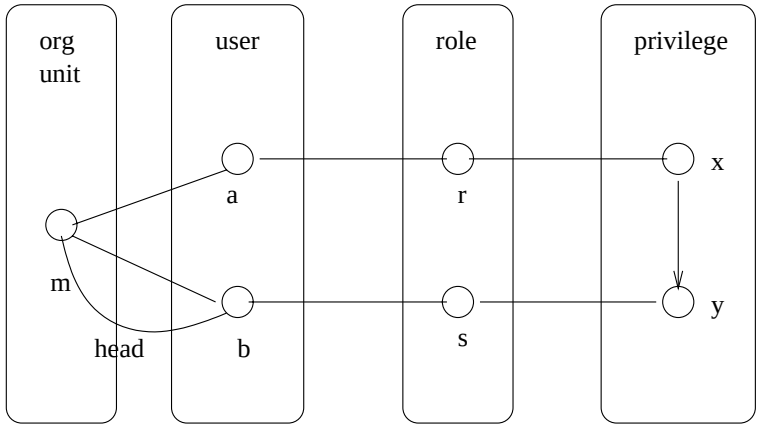

Fig. 5. Hierarchies for Example 8 (an arrow from x to y in the privilege hierarchy states that x is stronger than y)

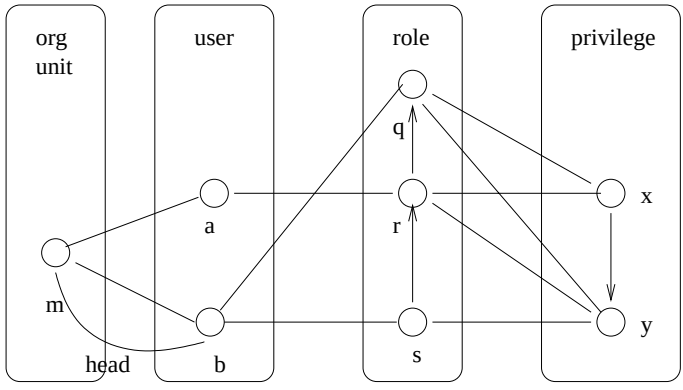

Fig. 6. Further hierarchies for Example 7 (an arrow from s to r in the role hierarchy states that s is smaller than r).

- under a *role-centered preference scheme*, there is no difference between a and b because to exercise y, a must play the role r and b must play s, which have no is-a relation between them.
- under a *privilege-centered preference scheme*, b is preferred to a because the privilege that b will hold is weaker than the one held by a.
- under a *boss-centered preference scheme*, a is preferred to b because in organizational unit m, b is the boss of a.
- if we add is-a(r,s) (see Fig. 6 which only adds to the database the fact that now r also holds y because of the definition of is-a), then under a role-centered preference, b is preferred because the role a plays to exercise y is larger to the one b plays.
- finally, if we further add the tuples role(q), can-play(b,q), is-a(q,r) (see Fig. 6) which would imply that q can hold both x and y, then there is a role-centered argument that would prefer a to b: to exercise y, b could have played the role q which is larger than the largest role a could have played (which is r).

Since there is no single reasonable ordering, it is left to the workflow to define which is the best ordering for that particular query. The workflow defines O, a two place predicate, that is, a preference relation among users, that is, O must be reflective, transitive, and complete (that is, for each pair of users u and v, either $\mathrm{O}(u, v)$ or $\mathrm{O}(v, u)$ must be true). $\mathrm{O}(u, v)$ state that u is at least as preferred as v (according to some criteria). If both $\mathrm{O}(u, v)$ and $\mathrm{O}(v, u)$ then u and v are indifferent or equivalent regarding the preference relation O. The permission system must order the users in groups of equivalent users, and each group decreasing order of O.

4.3. *Some preference relations*

Now, we shall formally define the orders illustrated above, which represent some standard orderings. As we will see later, the workflow (or the user that defined the

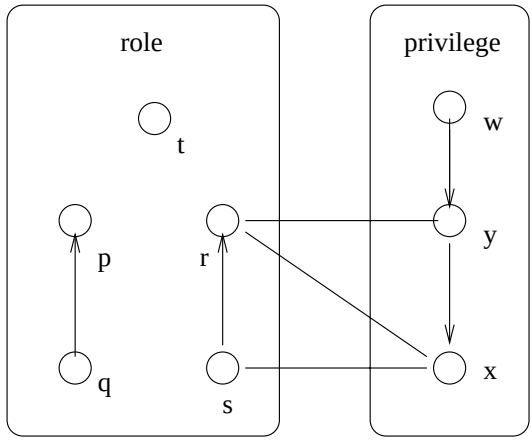

Fig. 7. Example of role/privilege hierarchies.

workflow) may define any ordering, which may or may not use the orders defined in this section.

For roles r and s, we will say that $r \prec s$ if s is larger than r, that is, s includes all privileges that r does, and more. For privileges x and y, we will say that $x \prec y$ if y is stronger than x. To illustrate the definition, we will use the diagram in Fig. 7. We assume that user a can-play all the roles in the diagram.

The set of **maximal roles** a user can play is defined as: $r \in \mathsf{MAXROLE}(u)$ iff $\mathsf{can\text{-}play}(u, r)$ and there is no other r' such that $\mathsf{can\text{-}play}(u, r')$ and $r \prec r'$. Intuitively, the maximal roles are the *largest* roles that user u can play. In Fig. 7, if we assume that a user u can play all the roles p, q, r, s, t, then $\mathsf{MAXROLE}(u) = \{p, t, r\}$.

The set of **maximal roles with respect to privilege** x is defined as: $r \in \mathsf{MAXROLEP}(u, x)$ iff $\mathsf{can\text{-}play}(u, r)$, and $\mathsf{hold}(r, x)$, and there is no other r' such that $\mathsf{can\text{-}play}(u, r')$ and $\mathsf{hold}(r', x)$ and $r \prec r'$. Intuitively, a maximal role w.r.t. x is the *largest* role that user u can play that holds privilege x. In Fig. 7, $\mathsf{MAXROLEP}(u, x) = \{r\}$.

The **maximal privilege with respect to privilege** x a user can hold is defined as: $y = \mathsf{MAXPRI}(u, x)$ iff $\mathsf{implies}(y, x)$, and there exists a role r such that $\mathsf{can\text{-}play}(u, r)$ and $\mathsf{hold}(r, y)$, and there is no privilege y' and role r', such that $\mathsf{can\text{-}play}(u, r')$ and $\mathsf{hold}(r', y')$ and $y \prec y'$. Intuitively, the maximal privilege w.r.t. x is the *strongest* privilege that implies x, which the user may hold using any role she can play. In Fig. 7, $\mathsf{MAXPRI}(u, x) = w$.

The **hierarchical level** of a user u is defined as: $\mathsf{HLEV}(u) = 0$ if there is no user $x \neq u$ such that $\mathsf{member}(u, k)$ and $\mathsf{head}(x, k)$; or $\mathsf{HLEV}(u) = n' + 1$ where n' is the smallest hierarchical level of all x such that $\mathsf{member}(u, k)$ and $\mathsf{head}(x, k)$. Intuitively, the hierarchical level is the shortest sequence of immediate bosses a user has until the sequence reaches someone that has no boss herself.

Given the definitions above one can create different preference orderings. For

example, to order the users in decreasing order of their MAXPRI w.r.t the privilege x one would define $O_{MAXPRI(x)}(u, v)$ to be true if imply(MAXPRI(v, x), MAXPRI(u, x)) or MAXPRI(v, x) = MAXPRI(u, x).

One can define composite preference orders, for example, which would prefer users with higher hierarchical levels, and among two users with the same hierarchical level, prefer the one with smaller MAXROLEP with respect to a privilege x. Let us call this order O_{new}. Thus $O_{\text{new}}(u, v)$ is true if HLEV$(u) >$ HLEV(v) or if HLEV$(u) =$ HLEV(v) and there exists a pair $r_u \in$ MAXROLEP(u, x) and $r_v \in$ MAXROLEP(v, x), such that r_u is smaller than r_v, and there is no other pair $s_u \in$ MAXROLEP(u, x) and $s_v \in$ MAXROLEP(v, x) such that s_u is larger than s_v. Finally, it is possible for the workflow to define completely *ad hoc* orderings to solve particularities of different tasks in the process.

For example, to represent the preference of both requester and auditor being from the same organizational unit in requirement 2.3 of Example 2, one would define an *ad hoc* ordering $O_{31,c}$ for the *auditing* task as: $O_{31,c}(u, v)$ is true if there exists a organizational unit o such that doer$(w, \textit{request}, c)$ and member(w, o) and member(u, o) and not member(v, o). Unfortunately, this preference ordering is very coarse: it orders the potential users into two groups, the ones that are members of the requester's (w) organizational unit and the ones that are not, with no ordering among them. One can create a new ordering $O_{32,c}$ which will further order by hierarchical level, for example. Thus $O_{32,c}(u, v)$ is true when there exists a organizational unit o such that doer$(w, \textit{request}, c)$ member$w, o)$ and (a) member(u, o) and not member(v, o) or (b) member(u, o) and member(v, o) HLEV$(u) \geq$ HLEV(v) or (c) not member(u, o) and not member(v, o) HLEV$(u) \geq$ HLEV(v). Of course, more complex orderings can be defined.

5. W1-RBAC: Controlled Overriding of Constraints

In previous sections, we discussed constraints at length. Clearly, some constraints are more important than others. In certain situations, it may be acceptable to override the less important constraints.

The workflow literature acknowledges that in real situations it is often the case that the specifications of a process as implemented in a workflow must be violated in order to get things done. This is usually referred to as exception handling. If, for example, the client on behalf of whom the process is being executed is a very important client, and is in a hurry, certain tasks may be skipped from the process, the order in which others are performed may be changed, or different people, who are available, may be assigned to tasks that they would usually not be allowed to perform. Among different forms of exception handling actions, the one that is relevant to the permission system is the assignment of tasks to users who are not usually allowed to perform them.

W-RBAC, as most workflow systems, only decides who will execute a task when the task becomes enabled. Such late binding of executers and tasks is the standard

in workflow applications because relevant information such as availability and work load of users can be brought to bear in this decision. But such late binding may lead to "dead ends". An "unfortunate" choice of executor in an earlier task may cause a later task not to have any potential executors because of separation or binding of duties. Here again, there will be the need to override some of the constraints that created the "dead end" situation so the case can move forward.

Example 9. For example, an organization policy might require that the tasks of receiving a client query and answering it should be performed by the same person, thus giving the client a feeling of personal touch. For example, Jose received a technical query from Kensington Corp., which is an important client. The process of constructing the answer proceeded normally but Jose was out on vacation when the answer was ready to be returned to the client. In this case, it may be more important to answer the query promptly than to wait for Jose to return and give the answer personally. And thus, it may be decided that Ling Mai should contact the client, violating the binding of duties constraint between these two tasks (receiving and responding a client's query). It would have been less likely that the need to expedite an answer would allow one to violate a separation of duties constraint, say the hiring of an external specialist, which must be approved by someone else in addition to the person that decided on the hiring.

It is clear that some constraints are more important than others, and different situations may allow different degrees of violation. The constraint described above reflects more of a preference, rather than a requirement. But when the binding constraint is based on legal or security reasons, it would be more important to enforce it. In other words, the model allows priorities to be assigned to constraints, according to how important they are.

5.1. *Levels of priority on constraints*

We will extend the formalism discussed above to include the idea of levels of priority or importance of rules and constraints. The idea is to associate with each integrity constraint rule

$$\bot \leftarrow Cn$$

a numeric label that expresses how important the rule is. The higher the label the more important the rule is. Thus, if the integrity constraint above has importance 7, we will label it with that integer:

$$\bot \leftarrow Cn \quad \text{priority } 7.$$

We can collect all constraints with label i in a set $\mathcal{C}_i$. We will assume that the labels are non-zero, positive integers.

With this labeling, we are able to define what is the level of compliance of a formula in relation to the constraints.

Definition 2. (Initial version). Let us define KB as the set of tuples in the database. If N is a formula to be added to the database, then i is the **level of compliance** of N, if i is the largest integer such that

$$\text{for each } j > i \quad KB \wedge N \wedge \mathcal{C}_j \not\models \bot$$
$$KB \wedge N \wedge \mathcal{C}_i \models \bot .$$

The formulas state that KB and N do not contradict any of the constraints labeled with $j > i$, but they do contradict at least one constraint labeled i. If N does not violate any constraint, we will say that its level of compliance is 0, i.e. full compliance.

We will discuss below that certain users may have the right to override some constraint, and thus, to add to the knowledge base a statement that contradicts some constraint. But in standard logic, the inclusion of an inconsistent statement would render the whole knowledge base worthless, since nothing could be inferred from it. To control the effects of adding inconsistent statements to the knowledge base, we partition it into different sets of statements, indexed by their level of compliance. Thus K_i is the set of statements with compliance level i, that is statements that override some constraint of level i. Or in other words, we stratify the KB based on the compliance level. Intuitively K_0 contains the "normal" facts (or tuples), the ones that do not violate any constraint. K_i, for a high i, contains very "exceptional" facts, the ones that violate an important constraint (with priority i).

Thus the level of compliance i of a new statement N, is defined as:

Definition 2 (Final version). If K_j is the set of tuples with compliance level j in the data base, and N is a formula to be added to the data base, then i is the **level of compliance** of N, if i is the largest integer such that

$$\text{for each } j > i \quad \left(\bigcup_{m<j} K_m\right) \wedge N \wedge \mathcal{C}_j \not\models \bot$$
$$\left(\bigcup_{m<i} K_m\right) \wedge N \wedge \mathcal{C}_i \models \bot .$$

If there is no such number, then the compliance level is 0.

The formula states that some constraint in C_i will be violated if N is included with the facts that are known not to violate any constraint of priority i or higher.

For example, let us assume that the set of constraint is:

$$\bot \leftarrow b, d \qquad \text{priority } 1$$
$$\bot \leftarrow a, b \qquad \text{priority } 2$$
$$\bot \leftarrow b, c \qquad \text{priority } 3 .$$

Let us also assume that the data base is empty, that is, $K_i = \emptyset$, for all i. In this case, the fact a has compliance level 0, since it does not contradict any constraint.

Now suppose that a is added, thus $K_0 = \{a\}$. Now, the fact b has compliance level 2, since it violates a constraint of priority 2. Adding b one would obtain $K_2 = \{b\}$. Now, c has compliance level 3, and once added, d would have compliance level 0 (it does not have level 1, because $\bot \leftarrow b, d$ is not contradictory with the facts in K_0, nor has it level 2 or above because the relevant constraint is assigned level 1).

5.2. *Right to override*

Which are the constraints that can be overridden, and who can override them? We assume that this is itself a privilege that can be attributed to roles, and indirectly to users. Roles are attributed privileges of the form $\mathsf{override}(n)$, which allows the user to override the constraints with priority equal or smaller than n.

The intuition, as we mentioned above, is that more important constraints are tagged with higher priorities levels. On the other hand, more responsible or powerful roles can hold higher override privileges. Of course, there should be constraints that cannot be overridden, or, in our model, whose priority levels are high enough so that no role has the right to override them.

Definition 3. The **max override level** of a user u is the highest n such that the user can hold the privilege $\mathsf{override}(\mathrm{N})$. That is, there exists r such that $\mathsf{can\text{-}play}(u, r)$ and $\mathsf{hold}(r, \mathsf{override}(n))$ and for all $r' \neq r$ if $\mathsf{can\text{-}play}(u, r')$ and $\mathsf{hold}(r', \mathsf{override}(n'))$ then $n' \leq n$.

5.3. *Interaction with the permission system*

The need to override constraints may appear when no one can perform an activity, that is, when the query $\mathsf{who?}(\mathrm{O,T,C})$ returns an empty list (Sec. 4.1). In this case, someone with responsibility over the process and the appropriate authorization would forcibly assign an executor to the activity. The assigned executor must necessarily be a user with the privilege to perform the activity but was eliminated from consideration by the permission system because of constraints such as binding or separation of duties. It is important to notice that the forced assignment does not violate the basic RBAC authorization mechanism. For this one would need to extend RBAC to include delegation or transfer of rights (e.g. Goh and Baldwin,[8] Kumar,[11] Barka and Sandhu,[2] Wainer *et al.*,[21] among others).

To allow users to perform this forced assignment, the basic interaction model of the workflow with the permission system (in Sec. 4) was extended with a new query/command $\mathsf{assign?}(u_1, u_2, t, c)$, sent by the workflow which states that user u_1 wants to forcibly assign user u_2 as the executor of task t for case c.

If user u_1's max override level is lower than the compliance level (Definition 2) of $\mathsf{doer}(u_2, t, c)$, then the command returns false, and no update is performed. If, on the other hand, user u_1 does have enough permission to override the constraints that are violated by $\mathsf{doer}(u_2, t, c)$, then the command asserts into the knowledge base $\mathsf{doer}(u_2, t, c)$ and returns its compliance level to the workflow system.

The assign? operation can also be used to transfer an activity, which was already assigned to a user, to another one. If a task t has already been assigned to Maria, and she is taken ill, a manager of the process, or a manager of the client's account on whose behalf the process is being executed, may have to remove the task from Maria's inbox and assign it to someone else, say Ngome. Again, assigning t to Ngome may violate some constraint which is overridden by the manager's override privileges. In this example, there is also the need to remove doer$(maria, t, c)$ from the data base.

6. Prototype Implementation

The definitions stated in this paper are straightforwardly implemented in Prolog. In fact, a proof-of-concept prototype of the system was implemented.

As an example of how straightforward the implementation is, below is the code for the definition of the answer to the workflow query (definition 1). The who?(O,T,C) query is implemented as a Prolog query with a fourth argument, which will contain the permission system's answer to the query. A constraint $\bot \leftarrow X$ is represented by a clause `violation(C) :- X`, where `C` is the case identification.

```
%
% Answer to the workflow system (definition 1)

'who?'(O,T,C,U) :-
        findall(X,can-do(X,T),L),  % find all users that can perform
                                   % task T
        filter(L,T,C,Lout),     % filter out those who cannot perform
                                % T for case C because of dynamic constraints
        preference_sort(O,Lout,U).     % order the result by ordering scheme O


% selects from a set of users those that can perform the task for the case
filter([],_,_,[]).
filter([A|RA],T,C,B) :-
        ( consistent(C, doer(A,T,C))
          ->   B = [A|RB],filter(RA,T,C,RB)
          ;    filter(Ra,T,C,B)).

% verify if a formula does not violate any constraint
consistent(Case, Formula) :-
        assert(Formula),
        ( violation(Case)
        -> retract(Formula),!,fail
        ;  retract(Formula)).

% verify if a user can perform a task
can_do(U,T) :-
        can_play(U,R),
        hold(R,T).
```

The translation of the formulas of W1-RBAC would have to include an extra argument which represents the compliance level of the fact. The algorithms would have to take that information into consideration. For example, if the binding of duties constraint in expression 3.1 (Sec. 3.5) is defined as having priority 3, such constraint would be represented as:

```
violation(C,3) :-
        doer(U1,t2,C,L1), L1 < 3,
        doer(U2,t3,C,L2), L2 < 3,
        U1 \= U2.
```

where `doer(U,T,C,L)` is used to represent that doer(U,T,C) was asserted with compliance level L.

Space limitations do no allow us to list all definitions as Prolog predicates, but one can see that the implementation follows in a straightforward way from the formal definitions given in the paper.

6.1. *Complexity, running time, and optimizations*

As a general introduction to the complexity of a logic program based implementation one should realize that for a clause such as:

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

the worst case running time is the one in which the "last solution" generated by the predicates A_1 to A_k is the only one that also satisfies **not** B_1 to **not** B_l. That is, also the worst case running time to disprove the clause — all possible solutions are generated and not even the "last one" is satisfied. Lets assume that each predicate A_i can generate N_{A_i} different solutions for its free variables, and takes, in the worst case, T_{A_i} units of time to compute each of these solutions. Lets also assume that each of the B_i predicates takes, in the worst case, T_{B_i} units of time to compute **not** B_i. Thus the worst case running time to compute C in this clause is:

$$\begin{aligned}
&N_{A_1} T_{A_1} + \\
&N_{A_1} N_{A_2} T_{A_2} + \\
&\ldots \\
&N_{A_1} N_{A_2} \ldots N_{A_k} T_{A_k} + \\
&N_{A_1} N_{A_2} \ldots N_{A_k} T_{B_1} + \\
&N_{A_1} N_{A_2} \ldots N_{A_k} T_{B_2} + \\
&\ldots \\
&N_{A_1} N_{A_2} \ldots N_{A_k} T_{B_l}\ .
\end{aligned}$$

Let us analyze the complexity of a who?(O, t, c) query in W0-RBAC. The query is implemented as:

(1) find all users that satisfy `can-do` the task t.
(2) remove from that set all users u for which $\mathsf{doer}(u, t, c)$ leads to a contradiction, that is, proves `violation`.
(3) order the remaining users according to the relation O.

The first step above can be implemented as a database query. The Prolog implementation of such a query (using `findall`) is not as efficient as running a database query, but if we assume that there are U users, the query can compute its answer in $T_1 = U \times T(cd)$, where $T(cd)$ is the average time to compute the truth of the query `can-do` for any user.

The second step is more costly. For each user u returned by the first step, one needs to verify if adding $\mathsf{doer}(u, t, c)$ violates any constraint, that is, if it does not prove `violation(c)`.

If we consider a `violation(c)` clause, it has the form:

$$\bot \leftarrow doer_1, doer_2, \ldots doer_k, other_1, \ldots other_l$$
$$\textbf{not}\ doer_{k+1}, \ldots\ \textbf{not}\ doer_{k+x},\ \textbf{not}\ other_{l+1}, \ldots\ \textbf{not}\ other_{l+y}\,.$$

It is important to notice that the `doer` predicates will in almost all situations have at most one free variable, the one representing the user. Both the case and the task are bounded when the `violation` predicate is queried. Since there is at most one user that is the executor of a given task for a given case, the `doer` predicate is deterministic, that is, it generates only a solution and to prove it (or disprove it), it takes only one access to the Prolog fact base. If we take the time to access the Prolog fact base as T, the worst case time to compute the violation clause is:

$$\begin{aligned}
&kT+\\
&N_{other_1}T_{other_1}+\\
&N_{other_1}N_{other_2}T_{other_2}+\\
&\ldots\\
&xN_{other_1}N_{other_2}\ldots N_{other_l}T\\
&N_{other_1}N_{other_2}\ldots N_{other_l}(T_{other_{l+1}}+\ldots T_{other_{l+y}})\,.
\end{aligned}$$

That is, complexity of computing the violation clause (α) is dominated by:

$$N_{other_1}N_{other_2}\ldots N_{other_l}(T_{other_{l+1}}+\ldots T_{other_{l+y}}+\text{constant})\,.$$

Therefore, only the predicates that do not refer to the dynamic component of the model are the ones responsible for the complexity of the query. The dynamic component of the model contributes with at most an additive constant to that complexity. Thus, if the constraint rule requires a very complex computation regarding the organizational structure (roles, users, organizational units), that computation will dominate the complexity of the violation clause.

If there are C constraints, then there are C violation clauses and in order to verify that a user does not violate any constraint, all violation clauses must be tested. This test must be performed for all users that satisfy the previous query; thus, the worst case total running time for this step is bounded by $U \times C \times \alpha$.

The third step is a sort, where the basic comparison predicate O may not take a constant time to compute. The upper bound for that step is $T_3 = U \times \log(U) \times T(O)$.

Now, the total time to compute the query who?(O,T,C) is bounded by

$$U \times T(cd) + U \times C \times \alpha + U \times \log(U) \times T(O)$$

where the dominating term is likely to be $U \times C \times \alpha$.

For the W1-RBAC model the total time to compute the query is the same as the above. The W1-RBAC will verify the violation clauses in decreasing order of priority, but in the worst case, where there is no violation, all the C violation clauses will be checked.

Unfortunately, we do not have enough examples of constraints to even approximate the value of α in the formulas above, but as mentioned, this complexity will be derived from the number of different solutions of the static predicates of the constraint rule.

There are possible optimizations that may be attempted if such computation is too expensive, all of which are standard techniques for optimizing a logic programs. Among them:

- off-line computation. Since these predicates refer to the static component of the rules, they do not change as new tasks or cases are executed, unless there is a change in the role, privilege, organizational units, or user hierarchies. Since such changes might occur infrequently, it may be convenient to compute these predicates off-line. In the worst case, that will not change the number of different solutions for these predicates, but will change the time to compute each solution, which will require an access to the Prolog fact base, and thus reduce the computation time by a constant factor. But more likely, not all solutions generated by one of these predicates will be accepted by another. For example, if boss(u, v), can-play$(v,$ *auditor*$)$ is a clause of a violation rule, and we consider each predicate in isolation, the first will generate at most U^2 pairs of users, and the second, at most U users, but taken together, by defining a boss-of-auditor(u, v) predicate, it may result in much fewer than U^2 pairs of users. Such reduction in the number of solutions may be significant enough to warrant the off-line computation.
- memoization. While it may not be worthwhile to pre-compute all possible values for the static predicates that appear in the rule, it may be worthwhile to store the results of this computation, so that, if the same computation is needed (for a different case), the results can be retrieved instead of being recomputed.
- partial evaluation. Once enough of the tasks of a case have already terminated, the violation clauses may be automatically transformed into new clauses, by partial evaluation, since some of its internal variables are already bounded. For example, for the clause:

  ```
  violation(C) :- doer(U1,task1,C),same_unit(U1,U2),doer(U1,task3,C)
  ```

  represent the constraint that `task1` and `task3` cannot be performed by users that belong to the same unit. If we know that `task1` has already been finished, for

case `c456`, it may be worthwhile to partially evaluate `violation(c456)`, which depending on how smart the partial evaluator is, could result in the (automatic) creation of the rule

```
violation(c456) :- doer(U1,task3,c456),
                   member(U1,[alice,bob,carol,david]).
```

That is, the partial evaluation of a violation clause, at the correct moment, would generate a particular rule for that case in which the computation of the `same-unit` predicate is performed in advance, so that when needed, the execution of the clause becomes just a membership check.

Unfortunately we believe that there is not enough experience in this domain to further evaluate which method, or combination of methods, would reduce the overall cost of computing the who? query, or even if any of those optimizations are necessary for the "average case".

7. Extensions, Discussion and Conclusions

A simple extension of W-RBAC is to include the concept of different activations of the same activity within the same case. It happens frequently that workflows have cycles of do-test activities, in which something is done, and in a different activity (usually with separation of duties constraints between them) the results are tested. If they fail the tests, a new *activation* of the *do* activity is performed, followed by a new activation of the *test* activity, etc. It is possible that one would like to place constraints on the different activations of an activity (of a case).

We added a fourth argument to the doer relation $\mathsf{doer}(u, t, c, n)$ to indicate that user u performed the nth activation of the task t for case c. With this extension, one can represent the constraint that the programming and the subsequent testing of the program cannot be performed by the same person, but there is no constraint on who can do the programming, if it fails the tests.

$$\bot \leftarrow \mathsf{doer}(u, \mathit{programming}, c, n), \mathsf{doer}(u, \mathit{testing}, c, n).$$

Similarly one could represent the constraint that the same user cannot be the programmer of two subsequent activation of programming activity.

A possible future work within the W-RBAC framework is the verification of the satisfiability of constraints. It is possible for the specification of the constraints and of the potential users of certain tasks to lead to inconsistencies. A typical one arises from the requirement that the immediate boss of the employee who wants a reimbursement should approve the request. However, some users may not have an immediate boss, either because they are not assigned to any organizational unit, or because the user is the president of the organization. A more subtle problem may arise because users higher up in the boss hierarchy should not play the role of approver because one would not want to bother them with such task.

It is very important that the system should verify at constraint definition time that such definition of potential executors of the approve task will not create a situation in which the reimbursement requests of some user cannot be approved. Similarly, but now regarding constraints, if the organization has only one auditor, and there is a constraint that whoever performs the auditing cannot be the requester, then this auditor will not be able to make any reimbursement request in the organization.

We acknowledge that placing constraints in a linear order (using the priority concept), and defining a privilege of overriding constraints on this linear order (the max override level) is limiting. For instance, the chief medical officer of a hospital may be allowed to override high priority *medical* constraints, and the chief accounting officer may be allowed to override high priority *accounting* constraints, but one cannot override the constraints of the "other domain". Still it is unclear at the moment how changes in the definition of right to override may impact the determination of when a fact violates a constraint or not (see definition 2).

Finally, to summarize, the main contributions of this paper are:

- We defined a framework that mixes concepts of RBAC and Workflow with a different expressive power than previous models, which allow simpler specification of real life business processes.
- The framework also allows for a cleaner separation of concerns between the permission aspects and the workflow aspects of the system.
- The framework allows for the definition of preferences in the selection of users to perform tasks.
- We extended the basic framework to deal with controlled overriding of constraints, which give us a partial solution to some problems of exception handling in workflows.
- We developed a prototype implementation of the system, discussed the complexity of the solution, and pointed out possible optimizations that may be used.
- We extended the framework to deal with multiple activations of tasks.

References

1. G.-J. Ahn and R. Sandhu, The RSL99 language for role-based separation of duty constraints, *ACM RBAC Workshop 99*, 43–54, 1999.
2. E. Barka and R. Sandhu, Framework for role-based delegation models, *16th Annual Computer Security Applications Conference*, 2000.
3. E. Bertino, E. Ferrari and V. Atluri, A flexible model supporting the specification and enforcement of role-based authorization in workflow management systems, *Proc. of the Second ACM Workshop on Role-Based Access Control*, 1997, 1–12.
4. 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 (1999) 65–104.
5. E. Bertino, S. De Capitani Di Vimercati, E. Ferrari and P. Samarati, Exception-based information flow control in object-oriented systems, *ACM Transactions on Information and System Security* **1**, 1 (1998) 26–65.

6. S. Castano, F. Casati and M. Fugini, Managing workflow authorization constraints through active database technology, *Information Systems Frontiers* **3**, 3, 2001.
7. D. Ferraiolo, J. Cugini and R. Kuhn, Role-based access control (RBAC): Features and motivations, *Proceedings of 11th Annual Computer Security Application Conference*, New Orleans, LA, December 1995, 241–248, http://hissa.ncsl.nist.gov/rbac/newpaper/rbac.ps.
8. C. Goh and A. Baldwin, Towards a more complete model of roles, *3rd ACM Workshop on Role-Based Access*, 1998, 55–61.
9. P. C. K. Hung, K. Karlapalem and J. W. Gray III, A study of least privilege in CapBasED-ams, *Proc. of the 3rd IFCIS International Conference on Cooperative Information Systems*, 1998, 208–217.
10. K. Karlapalem and P. Hung, Security enforcement in activity management systems, *Advances in Workflow Management Systems and Interoperability*, 1997, 166–194.
11. A. Kumar, A Framework for handling delegation in workflow management systems. *Proc. of Workshop on Information Technology and Systems (WITS)*, 1999.
12. N. Li, J. Feigenbaum and B. Grosof, A logic-based knowledge representation for authorization with delegation (extended abstract), *Proc. 12th Int. IEEE Computer Security Foundations Workshop*, 1999.
13. 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-LDIS Technical Report, University of Georgia, 1999.
14. M. Nyanchama and S. Osborn, The role graph model and conflict of interest, *ACM Transactions on Information and System Security* **2**, 1 (1999) 3–33.
15. S. Osborn and Y. Guo, Modeling users in role-based access control, *ACM RBAC 2000*, 2000, 31–37.
16. G. Sanchez, The wide project: Final report, Technical Report, Wide Consortium, 1999.
17. R. Sandhu, E. Coyne, H. Feinstein and C. Youman, Role-based access control models, *IEEE Computer* **29**, 2 (1996) 38–47.
18. R. Sandhu, Transaction control expressions for separation of duties, *Proc. of the Fourth Computer Security Applications Conference*, 1988, 282–286, http://citeseer.nj.nec.com/84322.html.
19. R. Simon and M. E. Zurko, Separation of duty in role-based environments, *Proc. of the 10th Computer Security Foundations Workshop (CSFW '97)*, 1997.
20. J. Wainer, Logic representation of processes in work activity coordination, *Proceedings of the ACM Symposium on Applied Computing, Coordination Track*, 2000, 203–209.
21. J. Wainer, A. Kumar and P. Barthelmess, Security management in the presence of delegation and revocation in workflow systems, Technical Report IC-01-14, Instituto de Computação, UNICAMP, 2001, http://www.ic.unicamp.br/ic-tr-ftp/2001/Titles.html.