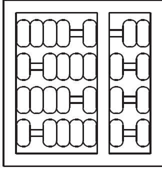

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

W-RBAC - A workflow security model incorporating controlled overriding of constraints

Jacques Wainer *Paulo Barthelmess*
Akhil Kumar

Technical Report - IC-01-013 - Relatório Técnico

October - 2001 - Outubro

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

W-RBAC - A workflow security model incorporating controlled overriding of constraints

Jacques Wainer[*] Paulo Barthelmess[†] Akhil Kumar[‡]

2001-10

Abstract

This paper presents a pair of role-based access control models for workflow systems, collectively known as the W-RBAC model. The models described here contains both static and dynamic (history based) constraints, which is integrated with a workflow system. The W0-RBAC model describes our concept of dynamic constrains, and the integration of the access control system with the workflow.

The W1-RBAC model extends the W0-RBAC model by allowing for a controlled overriding of constraints, which we argue are necessary in workflow applications in order to cope with exceptional situations. Finally we discuss a Prolog implementation of the access control models.

1 Introduction

Workflow management systems, or workflow for short, are systems that allow for the definition and enactment of business procedures. A workflow stores a definition of a procedure, in terms of its activities, which applications must be called to perform each activity, which data is available for each activity, the partial ordering on activities, and who should perform them. For the proposes of this paper the last characteristic is the central one: given that an activity can start (because the activities which it depends on are finished) who are the users that can perform that activity, and among them who is the one most preferred according to some criteria.

The paradigm behind the ideas discussed in this paper is that of a complete workflow management system that communicates with a permission system. The workflow's tasks are to define when an activity can start, and query the permission system as to which users can perform it. The permission system works as an organizational model, it knows about users, roles, privileges and rights, business constraints on assigning users to roles, which user belong to which department and who is in charge of that department, which user is working for which project, and so on. The permission system answers the workflow query and order the users that satisfy the query according to some criteria.

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

Our permission system is based on the RBAC model, with some extensions. In the RBAC model, roles are defined as a meaningful set of abilities and rights that can be assigned to users, so that they can perform their tasks. Roles in RBAC are organized in a hierarchy of inheritance. In workflow system besides this definition of roles, there is the need to refer to subordinate/superior relation among user. For example, it is very common in a business procedure that an authorization step must be performed by a superior or a responsible to the interested party. For example, a travel reimbursement process may include an approval activity performed by a superior of the user who is seeking the reimbursement, or by the responsible for the project on behalf of which the travel was taken.

It is possible to artificially explode the RBAC role structure and include such roles as "member of the legal department" and "responsible for the legal department", or "member of the quality project" and "head of the quality project", but we prefer a second alternative. We will define a structure of "organizational units" which contains objects like "legal department" and "quality project team". These organizational units are ordered by a partial order of inclusion, that is a organizational unit may be part of a larger organizational unit. Finally there will be two relations between organizational units and users: users are members of organizational units, and users are responsible for organizational units.

With this added structure, we will define a constraint language W0-RBAC that would allow the expression of business and process constraints. Because we are focusing on a workflow domain, we will have access to a very important concept, that of a case, an instance of a business procedure. By including the case in the W0-RBAC language we will be able to define both static and dynamic constraints, with a granularity that could not be achieved in other approaches that uses the RBAC concept of session (for example [1]).

We will define the relationship between the workflow and the permission system, where the workflow stores knowledge about work procedures, task, temporal relations, and the such, and the permission system is the embodiment of the organizational knowledge and rules. The synergy between both systems allow for new functionality, such as, ordering possible executor of tasks according to organizational knowledge of roles, subordinance, and so on.

Finally we will extend the W0-RBAC language to define the W-RBAC language that includes the concept of overriding constraints in a controlled way. That is particularly useful in the workflow domain because of what is refereed to as exception handling in the workflow literature: it is common that a particular instance of a business process may have to be altered so that work can proceed. If a important order is getting late it may be necessary to change the order of activities, or remove some activities, or attribute some activity to users that normally would not be allowed to perform it. In particular this last form of exception handling is the one that involves the constraints of the permission system: it may be necessary to violate some constraints in order to be able to assign an activity to a particular user. Of course this violation must be controlled, an important constraint cannot be violated, and even less important constraint can only be violated if someone with appropriate rights ask for it.

The W-RBAC language allows for different levels of importance on constraints, includes the appropriate rights to violate constraints, and is able to deal with the potentially contradictory set of facts and constraints that are created when constraints are violated.

This paper is organized as follows. Section 2 describes a constraint model for security which covers static constraints. Section 2.3 covers dynamic constraints. Section 3 gives an architecture for a permission service and shows how it will interface with a workflow model. Section 4 describes our approach for controlled overriding of constraints. In Section 5 we discuss implementation of our proposal. Section 6.1 discusses future work, and Section 6 discusses related works.

2 W0-RBAC Model: The Extended RBAC model

We base our model on the Role Based Access Control model (RBAC) with extensions. The basic RBAC model can be described in terms of: 1) **entities**: *users*, *roles*, and *privileges*, 2) **relationships** between these entities, and 3) **constraints** over these relationships. A meta-model is displayed in Figure 1. This meta-model is a representation of a graph-based model presented by Nyanchama and Osborn in [19].

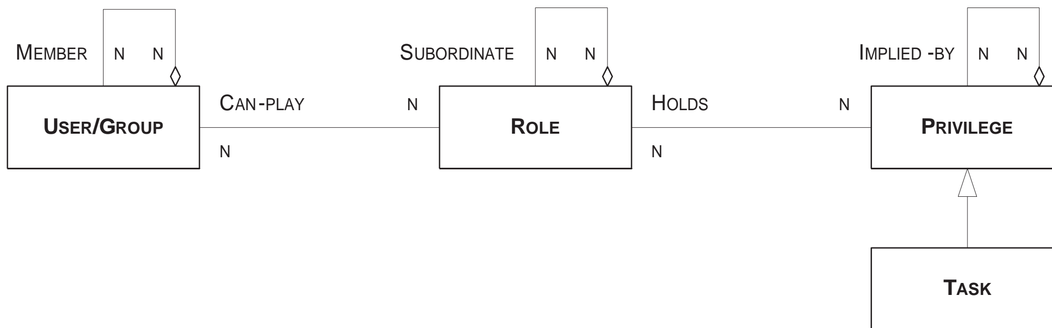

Figure 1: Meta model.

- user (U) represent individual users.
- privileges (P) represent classes of rights to perform operations, tasks, access data and so on, possibly with explicit attributes. For example, *travel-approval*(US$500) represents the class of all approval of travel expenses tasks up to US$500.
- role (R) describe meaningful groupings of privileges or abilities that can be assigned to users, e.g., the role of *manager*.
- organizational units (OU) describe meaningful organizational structures with members and responsible people.

In the case of privileges, in this paper we are interested in just two forms of privileges: the right to execute an activity, and the right to delegate activities. 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 will be outside the scope of this paper.

We will define the following relations:

- can-play(U,R), state that the user U can play the role R. can-play(*mary*, *manager*) says that Mary plays the role of manager.
- is-a-r(R1,R2), states that role R1 is a kind of (and thus inherits all privileges) role R2. is-a-r(*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 possible more. This is the standard inheritance partial order among roles as defined in RBAC.

  If is-a-r(R1,R2) is true, we will say that R1 is an **immediate superior** to R2. If R1 and R2 are related by the transitive closure of is-a-r we will say that R1 is a **superior** of R2.
- hold(R,P), state that role R holds certain privilege P.
- imply(P1,P2), state that privilege P1 is stronger, or supersedes, or includes P2. For example the right to approve travel expensed up to US$1000 implies (is stronger than) the right to approve travel expenses up to US$500.

  If P1 and P2 are related by the transitive closure of implies we will say that P1 is **stronger** than P2
- include(OU1,OU2), state that the organizational unit OU1 includes the organizational unit OU2. Thus include(*engineering dept*, *project x12 team*) states that the project 12 team is a part of the engineering department.
- member(U,OU), state that the user U is a member of the organizational unit OU.
- head(U,OU), states that user U is the head, or the responsible for the organizational unit OU.

There are some implicit inheritance structures in our model. The first one is defined by the is-a-r relation among roles, that follows the RBAC model: if is-a-r(R1,R2) then for all P such that hold(R2,P) it is also the case that hold(R1,P).

The second inheritance structure is defined by the imply relation. If *approve travel expenses up to $1000* implies *approve travel expenses up to $500* than anybody that can perform the former can also perform the latter.

2.1 Constraints

The model described above can confer broad privileges on users and occasionally this may not be desirable. Hence, the need for a mechanism to fine-tune the model. For example, some instantiations of the model may cause "conflicts of interest" [19]) in the form of incompatible privileges. The classic case occurs when a traveler who is claiming travel reimbursement ends up receiving the privilege to approve his/her own expense claim. Thus,

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

We will represent an integrity constraint as

$$\bot \leftarrow C$$

where C describes an invalid situation.

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

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

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

As an example, the constraint that R3 (a right) can only be held by users that head an organizational unit is represented as:

$$\bot \leftarrow \mathsf{hold}(r, \mathrm{R3}), \mathsf{can\text{-}play}(u, r),\ \textbf{not}\ \mathsf{head}(u, ou)$$

The formula above should be read as "the following situation is invalid: there exists a role (r) which holds the privilege R3 and there exists a user u which can play that role, and such user is not the head of a organizational unit."

Also, auxiliary predicates can be defined. A particularly useful one is boss(U1,U2) which is true is U1 is the head of on of the organizational units to which U2 is a member:

$$\begin{aligned}\mathsf{include^*}(x, y) &\leftarrow \mathsf{include}(x, y)\\ \mathsf{include^*}(x, y) &\leftarrow \mathsf{include}(x, z), \mathsf{include^*}(z, y)\end{aligned}$$

and

$$\begin{aligned}\mathsf{boss}(x, y) &\leftarrow \mathsf{head}(x, ou), \mathsf{member}(y, ou),\ \textbf{not}\ x = y\\ \mathsf{boss}(x, y) &\leftarrow \mathsf{head}(x, ou), \mathsf{member}(y, ou'), \mathsf{include^*}(ou, ou'),\ \textbf{not}\ x = y\end{aligned}$$

Constraints can be established over any of the relationships of the meta-model and can be broadly classified into *static* and *dynamic*. We start by describing the static ones.

2.2 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 to be introduced. The name *static* comes from the fact that these constraints do not depend from any actual execution of actions on objects. They control the structure of the security model independently of any dynamic behavior.

The reason behind these constraints can be reduced to just two: 1) to prevent grouping of incompatible users, and 2) to prevent some user to accumulate too much power, i.e., to indirectly hold incompatible privileges, such as *request* and *approve*. This correspond to the usual concept of *separation of duties* [22, 19, 6, 23]. We want to avoid that a single person would be able to do some damage by holding certain combinations of privileges.

Incompatible privileges can be obtained in many ways: by users that can play a role that holds incompatible privileges; by roles that inherit from multiple subordinate roles that in turn hold incompatible privileges; by coarser granularity privileges that imply multiple incompatible privileges; or finally, by users that can play multiple roles that together hold the incompatible privileges. The constraints are placed to avoid these invalid relationships from ever happening.

For example, the static constraint that no user can have both the privileges of request and approve is represented as:

$$\begin{aligned} \bot \quad \leftarrow \quad & \mathsf{can\text{-}play}(U, R1), \mathsf{can\text{-}play}(U, R2), \\ & \mathsf{hold}(R1, request), \mathsf{hold}(R2, approve), \end{aligned}$$

that is, it is inconsistent if there is a user U who can play both the roles $R1$ and $R2$ (not necessarily distinct), and where $r1$ holds the privilege *request* and $R2$ holds *approve*.

Static constraints are enforced when tuples are added or removed. After a transaction that inserts of new users, roles, and so on, and the tuples that represent their relations to each other and to the users, roles, privileges, etc, already in the system, and other objects and tuples are deleted, the static constraints must be all checked. Using the constraint rules are described in the paper, the result of checking the static constraints would be just the existence of not of violations. But of course a practical implementation of the rule above would not only verify that a violation exists but also tell which instantiation(s) of the variables $U, R1$, and $R2$ above cause the violation.

2.3 Dynamic Constraints

The limitations of static constraints is well known in the security literature. A static constraint may forbid a user to hold the roles of pilot and navigator of a plan, but that is not exactly what is needed. A pilot can be a navigator if needed, but 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 the concept of *session*, that is, binding between user and roles fixed in some time interval. In this case one my 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 has the role of *requester* and *approver* for *the same reimbursement request process*, 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 the her reimbursement and approving Carol's, but that is acceptable if these roles are being played in different *reimbursement instances*.

To be able to refer to an *instance of a process* we will add to our model a new class case, as described in Figure 2 (case is one of the standard ways instances of a procedure are refer to in workflow systems [26]). We also define a new trinary relation doer(U,P,C) that state that a user (U) exercised a particular privilege (P) on 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 the exercised the privilege P on instance C.

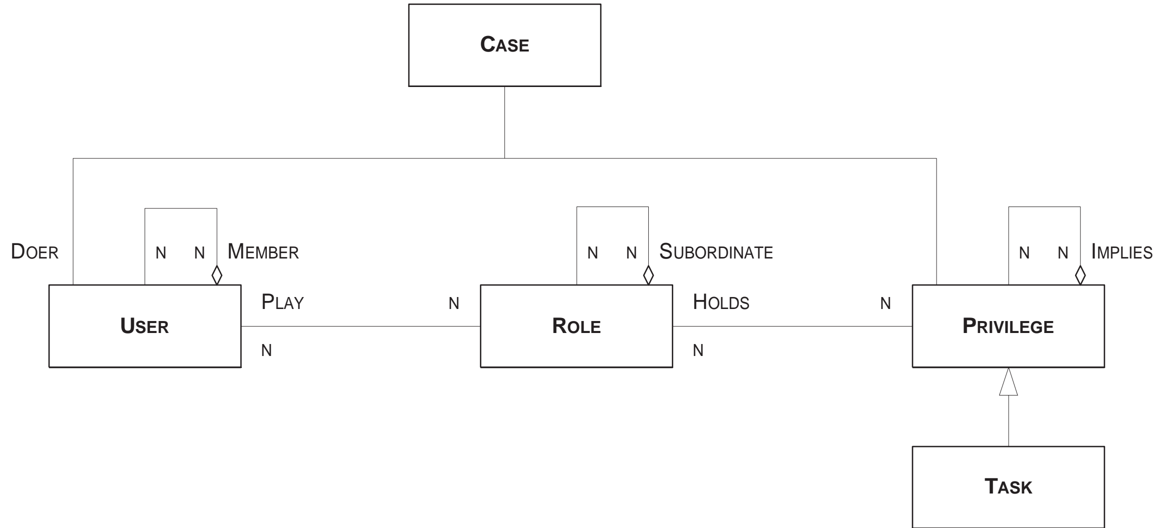

Figure 2: Meta model.

Dynamic constraints can be either *negative* or *positive* - users and roles are either blocked from performing some actions, and/or required to perform specific actions on a case, respectively, depending on 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*, whenever 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 that 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 define the constraint:

$$\perp \leftarrow \mathsf{doer}(u, \mathrm{T2}, c), \mathsf{doer}(u', \mathrm{T3}, c),\ \mathbf{not}\ u' = u \tag{1}$$

History based security matches quite naturally workflow systems, which usually employ role based mechanisms independently of security reasons, as a means to distribute work. History is also naturally kept by most workflow systems as well, for auditing and recovery purposes. As a consequence, the bulk of the workflow related security is concerned with history based mechanisms, e.g., [5, 7, 20, 18].

3 An access control system integrated with a WFMS

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

From now on, we will equate tasks and rights/privileges. In other words, we are only interested in the right/privilege of performing a task. We will refer to these right/privileges by a symbol stating with the letter "T".

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

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

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

Ideally the system as a whole works as follows: as soon as an activity for a case is finished, the workflow is informed. The workflow following its process model, computes the next activity that needs to be performed, and asks the permission system who are the users that can perform that task, and receives back an ordered list of users. The workflows decides who among the users in the list will perform the next task, and sends that information to the user's work list, and informs the permission system, by sending the doer tuple. When

the last activity of a process instance is finished, the workflow sends the done information to the permission system.

Figure 3 shows a diagram of the interchanges between the workflow engine, the permission system and users.

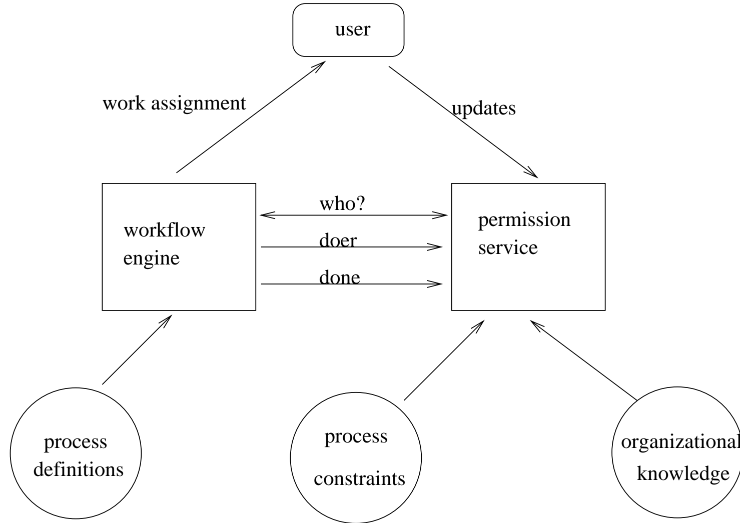

Figure 3: Interaction between system components and users.

3.1 Answering the queries from the workflow system

With the concepts presented so far, we can define what is the appropriate answer to the query who?(Q,O,T,C) posed by the workflow. The permission system should return all users u that satisfy a property Q, and for which doer(u,T,C) does not violate any integrity constraint. Furthermore, the permission system should order the users according to the order O, to be explained shortly.

The property Q used in the workflow query to the permission system is part of the specification of the process itself. The specification may state that potentially only users that satisfy some property (and do not violate any constraint) can perform an activity. This property may involve roles, organizational units, privileges, and even the past history of the case itself. Thus the workflow passes this specification of potential executers to the permission system, as the property Q. The permission system will verify which users satisfy the property and which further do not violate any constraint. The property is a two place predicate, where the first argument is a user, and the second the case identity, defined by means of one or more clauses in logic programming.

To illustrate, let us assume a simple process of three activities: *apply* for reimbursement, *verify* expenses and *approve* reimbursement. The activity *verify* can be executed by anyone that can play the role of *auditor*. Finally the *approve* can be performed by anyone that is the boss of the applicant or any VP. As constraints we require that neither the approver

not verifier can be the applicant himself. That is the constraints are:

$$\begin{aligned} \bot &\leftarrow \mathsf{doer}(u, \mathit{apply}, c)\mathsf{doer}(u, \mathit{verify}, c) \\ \bot &\leftarrow \mathsf{doer}(u, \mathit{apply}, c)\mathsf{doer}(u, \mathit{approve}, c) \end{aligned}$$

The predicate that define the set of potential executors of the activity *verify* is

$$P_1(x, c) \leftarrow \mathsf{can\text{-}play}(x, \mathit{auditor})$$

This predicate makes no use of the case identity information. The property that defines the potential executors of the activity *approve* is defined by the predicate

$$\begin{aligned} P_2(x, c) &\leftarrow \mathsf{doer}(a, \mathit{apply}, c), \mathsf{boss}(x, a) \\ P_2(x, c) &\leftarrow \mathsf{can\text{-}play}(x, \mathit{VP}) \end{aligned}$$

When Dana finishes the *apply* task, workflow is informed and determines that *verify* is the next activity to be performed for this case (which we assume is has the identity *c120*). The workflow queries the permission system with $\mathsf{who?}(P_1, O_1, \mathit{verify}, \mathit{c120})$. We will discuss the order O_1 below. The permission system returns an ordered list 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 $\mathsf{doer}(\mathit{Eric}, \mathit{verify}, \mathit{c120})$. When this activity is over the workflow will query the permission system with $\mathsf{who?}(P_2, O_2, \mathit{approve}, \mathit{c120})$ and so on.

This example also illustrate the need for organizational units in the model. One could make the argument that organizational units and other organizational information, such as user properties [21], a more complex role model [10] are important concepts in real life use of RBAC systems, but are not part of the core RBAC model. But because of the requirement of the separation of concerns between the workflow and the permission system, we can make the point that the inclusion of organizational units is not only desirable but necessary.

Let us suppose that we do not include the organizational units hierarchy into the model. A standard RBAC representation for an organization that has two different projects, with different responsibilities, would be to explode the role hierarchy with two different sublattices for each project, as shown in figure 4.

But for such a model, the workflow rule that the *approve* action must be performed by the boss of the applicant cannot be written without the knowledge of the role structure. The predicate that defines the potential executors of the *approve* action would not be as defined above, but:

$$\begin{aligned} P_2(x, c) &\leftarrow \mathsf{doer}(a, \mathit{apply}, c), \mathsf{can\text{-}play}(a, \mathit{project\ 1\ member}), \mathsf{can\text{-}play}(x, \mathit{project\ 1\ leader}) \\ P_2(x, c) &\leftarrow \mathsf{doer}(a, \mathit{apply}, c), \mathsf{can\text{-}play}(a, \mathit{project\ 2\ member}), \mathsf{can\text{-}play}(x, \mathit{project\ 2\ leader}) \\ P_2(x, c) &\leftarrow \mathsf{can\text{-}play}(x, \mathit{VP}) \end{aligned}$$

Thus if a new project were to be created, the appropriate definition of the P_2 predicate would have to added into the *workflow* definitions, which violates the requirement that the workflow should encapsulate the process knowledge and the permission system the organizational knowledge.

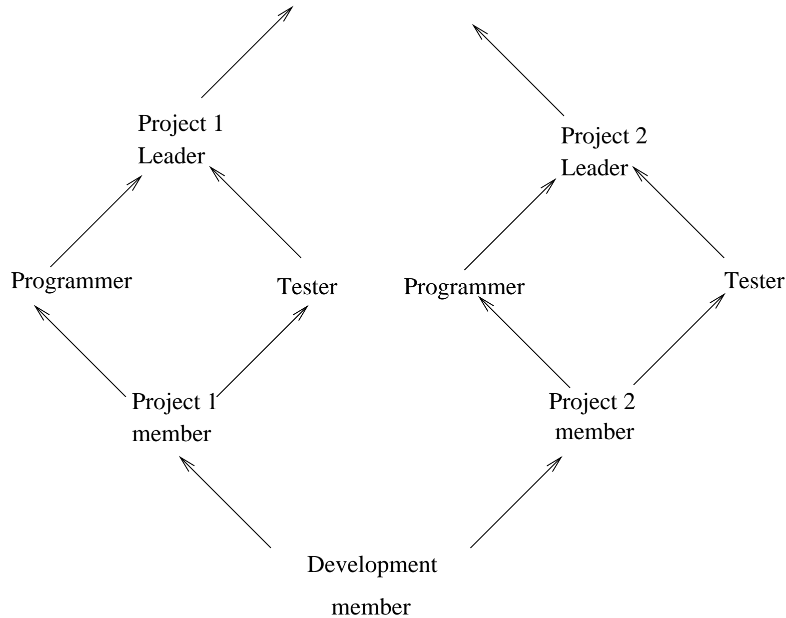

Figure 4: A role hierarchy for a two project organization.

3.2 Ordering the answers

An important feature of our permission system is that it will rank the users who can perform a task based on their suitability and the knowledge stored in the permission system. For example, it may happens 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 systems is to state that the least superior role that hold a privilege is the most appropriate to execute that privilege. Thus if *compile* is the privilege that is needed, one must choose the most specific role that hold that privilege, say *programmer*, instead of superiors of that role, say *C-programmer*.

But in this domain, the most specific role rule does not translate itself is 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 role of a C-programmer and programmer. Should Falco or Gail be the most preferred as the possible performer of the task *compile*? Or, as another example, let suppose the task to be executed is *approve-reimbursement($2000.00)*, that is to approve a reimbursement of $2000.00. Hector by playing the role of manager can approve reimbursement up to $5000.00, and Ingrid, playing the role of project leader can approve reimbursements up to $7000.00, and project leader and manager are roles with no is-a-r relation between them. Which of Hector and Ingrid are preferred users to perform the task? And does that answer changes if Hector is the boss of Ingrid?

The basic principle behind most of the generic answers to the problem is the idea of the

more "specific" user that can perform the task, or use the privilege, should be preferred to more "generic" users. But there are many possible definitions of what it means for a user to be more "specific" than others. In particular, our model suggests three implicit or explicit orderings: the superior/inferior ordering between roles, the stronger/weaker ordering between privileges, and the boss relation between users, and there may be no correspondence among those three orderings.

For example if the database contains the tuples:

```
USERS     ROLES     PRIVILEGES     ORGANIZATIONAL UNITS

 a,b      r,s       x,y            m

                    imply(x,y)

USERS x ROLES        ROLES x PRIVILEGES   USERS X UNITS

can-play(a,r)        hold(r,x)             member(a,m)
can-play(b,s)        hold(s,y)             member(b,m)
                                           head(b,m)
```

and the workflow query is to find out users that can perform `y`, then:

- under a role-centered preference above, 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-r relation between them.
- under the privilege-centered preference, `b` is preferred to `a` because the privilege that `b` will hold is more specific than the one held by `a`.
- under a boss-centered preference, `a` is preferred to `b` because in organizational unit `m`, `b` is the boss of `a`
- if we add `is-a-r(r,s)` which only adds to the database the fact that now `r` holds also `y` because of the definition of is-a-r, then under a role-centric preference, `b` is preferred because the role `a` plays to exercise `y` is superior to the one `b` plays.
- finally, if we further add the tuples `role(q)`, `can-play(b,q)`, `is-a-r(q,r)`, 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 superior to the higher role `a` could have played (which is `r`).

Since there is not a 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 total order among users, that is, O must be asymmetric and transitive, and for each pair of users u and v that satisfy P, either $O(u, v)$ or $O(v, u)$ must be true. $O(u, v)$ state that u is preferred to v (according to some criteria). The permission system must order the users in decreasing order O. We will see in the next section that a partial order can be completed to become a total order and thus can be used as the O relation.

3.3 Some preference relations

Now, we shall formally define the three orders illustrated above, the role, the absolute role, and the privilege order. For roles R and S we will say that R $<$ S if S is a superior of R, that is S and R are related buy the transitive closure of is-a-r. For privilege X and Y we will say that X $<$ Y if Y is stronger than X.

Definition 1 *A* **maximal role with respect to privilege X** *for a user U is the role R such that*

- *can-play(U,R)*
- *hold(R,X)*
- *there is no other R' such that can-play(U,R') and hold(R',X) and R $<$ R'*

Intuitively, a maximal role is the *highest* role (most superior) that user U can play that holds privilege X.

Definition 2 *A* **absolute maximal role** *for a user U is the role R such that*

- *can-play(U,R)*
- *there is no other R' such that can-play(U,R') and R $<$ R'*

Intuitively, an absolute maximal role is the *highest* role that user U can play.

Definition 3 *A* **maximal privilege belonging to user U that is stronger than privilege X** *is a privilege Y such that:*

- *implies(Y,X)*
- *there exists a role R such that can-play(U,R) and holds(R,Y)*
- *there is no privilege Y' and role R' such that can-play(U,R') and holds(R',Y') and Y $<$ Y'*

Intuitively, the maximal privilege Y is the *strongest* privilege that implies X that user U may hold by playing some role (R).

These three definitions attempt to define how one user can be more "generic" than another one for performing a task X. A user U1 can be more "generic" than a user U2 if U1 has a higher maximal role with respect to X, or a higher absolute maximal role, or a stronger maximal privilege that implies X. But regardless of the definition of "generic", a *less* generic user is always preferred for doing a task than a *more* generic user.

We can define a partial ordering based on the definition above:

Definition 4 *We will say that user U1 is preferred to user U2 (for privilege X) under the* **role preference** *(ORP_X) ORP_X(U1,U2) if the maximal role for U2 with respect to privilege X is superior to the maximal role for U1 (with respect to X), and the reverse does not happen.*

Definition 5 *We will say that a user U1 is preferred to a user U2 (for privilege X) under the* **privilege preference** *(OPP_X) OPP_X(U1,U2) if the maximal privilege stronger than X of U2 is stronger than the maximal privilege stronger than X of U1.*

Definition 6 *We will say that user U1 is preferred to user U2 (for privilege X) under the* **absolute role preference** *(OARP) OARP(U1,U2) if the absolute maximal role for U2 is superior to the absolute maximal role for U1, and there exists a role that holds the privilege X and U1 can play it, and the same is true for U2.*

These are only partial orders. For the permission system we need a total order among the users. One can complete these partial orders in many ways. For example, the order ORP_X+LEX is a total order that completes ORP_X by a lexicographical order among users, such that

$$\langle x,y\rangle \in \text{OR}_X + \text{LEX} \quad \text{if} \quad \begin{cases} \langle x,y\rangle \in \text{OR}_X \\ \langle x,y\rangle \notin \text{OR}_X \text{ and } \langle y,x\rangle \notin \text{OR}_X \text{ and } \langle x,y\rangle \in \text{LEX} \end{cases}$$

where LEX is some lexicographical ordering among all users. Or in a different notation, which will be used from now on:

$$\langle x,y\rangle \in \text{ORP}_X + \text{LEX} \quad \begin{cases} \text{if } \langle x,y\rangle \in \text{OR}_X \\ \text{else if } \langle x,y\rangle \in \text{LEX} \end{cases}$$

One can define other compositions also. For example, first verify if user x is preferred to y under the absolute maximal role. If this is not true, then try the maximal role with respect to the task X; if that is not true, then try the maximal preference; and, if that is not true then order them lexicographically. We call this ordering OARP+ORP+OPP+LEX, and it is defined as

$$\langle x,y\rangle \in \text{OARP+ORP+OPP+LEX} \quad \begin{cases} \text{if } \langle x,y\rangle \in \text{ORA} \\ \text{else if } \langle x,y\rangle \in \text{OR}_X \\ \text{else if } \langle x,y\rangle \in \text{OP}_X \\ \text{else if } \langle x,y\rangle \in \text{LEX} \end{cases}$$

3.4 Answering the workflow system

Finally we can define what is the answers to the query WHO?(Q,O,T,C).

Definition 7 *The permission system's* **answer** *to the workflow query who?(Q,O,T,C) is the ordered list of users $\langle u_1, u_2, u_3, \ldots u_n\rangle$ such that*

- *for each $u_i \in \langle u_1, u_2, u_3, \ldots u_n\rangle$, the pair $\langle u_i, C\rangle$ satisfy the property Q*
- *for each u_i, adding doer(u_i, T, C) does not violate any constraints*
- *and O(u_i, u_{i+1}) for all $i < n$.*

4 W-RBAC: Controlled overriding of rules

In a previous section 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 common that the specifications of a process as implemented in a workflow must be violated in order to get things done. This is usually referred as exception handling. For example, the client on behalf of whom the process is being executed is a very important client and is in a hurry; in such a case certain activities may be removed from the process, the order in which others are performed changed, and different people, who are available now, may be assigned to activities that would usually not be allowed to perform.

These different forms of exception handling actions, the one that is relevant to the permission system is the assignment of activities to users that are not allowed to perform them. For example, the constraints state that the activities of receiving a client query and answering it should be performed by the same person, so the client has a feeling of personal touch. For example, Jose received a technical query from Kensinton Corp., which is a important client. The process of constructing the answer proceeded normally but Jose was out in vacation when the answer was ready and had to be returned to the client. In this case it may be more important to answer the query promptly than to give the personal touch. An thus, it was decided that Ling Mai would contact the client, violating the binding of rule constraint regarding the two activities. It would have been less likely that need to a expedite answer would allow one to violate a separation of duties constraint, say that the hiring of an external specialist must be approved by someone else than the one that decided on the hiring.

It is clear that some constraints are more important than others, and some situations may require more or less constraints to be violated. The novel feature of our proposal in that we allow association of priorities with constraints. Thus, a user may "allow" the system to override the constraints in a controlled manner and in a subjective way based on what constraints she considers to be of low priority.

4.1 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

$$\bot \leftarrow C$$

a numeric label that expresses how important is the rule. 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:

$$\bot \leftarrow C \quad \text{priority} \;\; 7$$

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 the labeling we are able to define what is the level of compliance of a formula (say some additions to the data base) in relation to the constraints.

Definition 8 (temporary). *If KB is the set of tuples 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*

$$\begin{aligned} KB \wedge N \wedge \mathcal{C}_{j,j>i} &\not\models \bot \\ KB \wedge N \wedge \mathcal{C}_{j,j>i} \wedge \mathcal{C}_i &\models \bot \\ KB \wedge \mathcal{C}_{j,j>i} \wedge \mathcal{C}_i &\not\models \bot \end{aligned}$$

The formulas state that KB and N are not contradictory to the constraints labeled with $j > i$, but they are contradictory with the constraints labeled i and above, and the KB by itself was not contradictory with these constraints. If New does not violate any constraint, we will say that its level of compliance is 0.

We will discuss below that some 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 turn the whole knowledge base as useless, since anything 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, and possible lower level constraints.

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

Definition 8 (final) *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*

$$\begin{aligned} \textit{for each } j > i \quad \bigcup_{l<j} K_l \wedge N \wedge \mathcal{C}_j &\not\models \bot \\ \bigcup_{l<i} K_l \wedge N \wedge \mathcal{C}_{j,j>i} \wedge \mathcal{C}_i &\models \bot \end{aligned}$$

4.2 Right to override

Which are the rules 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 override(N), which allows the user to override the rules with priority equal or smaller than N.

The intuition, as we mentioned above, is that more important rules 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 rules that cannot be overridden or, in our model, rules whose priority levels are high enough so that no role has the right to override them.

Definition 9 *The* **max override level** *of a user U is higher N such that the user can hold the privilege* override(N)*. That is:*

$$\exists R \textit{ such that } \textsf{can-play}(U,R) \textit{ and } \textsf{hold}(R, \textsf{override}(N))$$
$$\textit{if } \textsf{can-play}(U,R) \textit{ and } \textsf{hold}(R, \textsf{override}(N')) \textit{ then } N' < N$$
$$\textit{for all } R' \neq R \textit{ if } \textsf{can-play}(U,R) \textit{ and } \textsf{hold}(R, \textsf{override}(n)) \textit{ then } n \leq N$$

4.3 Interaction

The basic interaction to the permission system extends the W0-RBAC interaction with a new query/command assign?(U1,U2,T,C) sent by the workflow which states that user U1 wants to assign user U2 as the executor of task T for case C. If user's U1 max override level is lower than at least one of the constraints that would be violated by asserting doer(U2,T,C), then the command returns false, and no update is performed. If on the other hand user U1 does have enough permission to override the constraints that are violated by doer(U2,T,C), then the command asserts into the knowledge base doer(U2,T,C) and its compliance level, and returns to the workflow its compliance level.

The security model for the assign? operation is still incomplete. For example, nothing was stated about the properties of user U1 in the operation assign?(U1,U2,T,C) except that he should have the appropriate max override level. We have identified at least three modes of use of the assign? operation, and we will briefly describe them here. We developed the full model for these modes of use elsewhere.

The first mode of use is *delegation*. A user knows he will be unavailable to perform a particular task that will still happen in the future, and thus he preemptively delegates that task (for a particular case C) to someone else. This preemptive delegation may both violate the permission model and the constraints. For example Maria knows that she may perform the task T for a case C sometime next week, but she will be unavailable by then. She can delegate that task to Ngome. But it may be the case that a) because of separation of duties constraints, Ngome cannot perform T on C and b) to perform T one needs to play the roles of programmer or tester, and Ngome cannot play either role. The case a) is an example of the delegation violating the constraints, and case b) is an example of the delegation violating the basic permission relation themselves. Thus to delegate the task T to Ngome, in the case a) Maria must have the permission to temporarily expand the permission model so that Ngome can perform T. (Whether Ngome gets all rights associated with programmers or testers by this delegation is some of the issues that a delegation model must resolve). More relevant to this paper, in the case b) Maria must have permission to violate at least the constraint that binds Ngome from performing T for C.

The second mode of use is *forwarding*. If the task was already assigned to Maria, and it is in her inbox, she may realize that Ngome was better prepared to perform it. By forwarding it to Ngome she may incur on the two violations described in the delegation mode. But forwarding may have some further complications. In delegation Ngome was one of the possible executors of the task T. It may the case that because Maria had to (and was allowed to) violate both the permission model and the constraints, Ngome would loose priority in being chosen to perform the task. But by forwarding the task implies that

necessarily Ngome will perform it. It is possible that because of this certainty, forwarding may require higher permissions that delegation would.

Finally the third mode of use is *transfer*. If the task 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 behalf of whom the process is being executed, may have to remove the task from Maria's inbox and assign it to some one else, say Ngome. Again all the violations of delegation and forwarding may be present, and further the right to transfer may only be assigned to a few roles in the organization. Different than delegation and forwarding, the user performing the transfer may not have the right to perform the task itself.

5 Implementation

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

As an example of how straight forward the implementation is, below is the code for the definition of the answer to the workflow query without delegation (definition 7).

The `who?`(R,O,T,C) query is implemented as a Prolog query with a fifth 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 `X` is the expression of X in Prolog syntax, and `C` is the case identification.

```
% definition 12

'who?'(Q,O,T,C,U) :-
        findall(X,Q(X,C),L),          % find all users that satisfy the
                                      % predicate Q
        filter(L,T,C,Lout),      % filter out those who cannot perform
                                 % T for case C
        predsort(O,Lout,U).      % order the result by O


% selects from a set of users those that can perform the task for the case
filter([],_,_,[]).
filter([A|RA],T,C,B) :-
        can_do(A,T),
        ( 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 is a user can perform a task
can_do(U,T) :-
        can_play(U,R),
        holds(R,T).
```

The definitions that involve overriding some constraints are based on the concept of the compliance level of a formula (definition 8), which is implemented below:

```
% given Formula, determines its complience Level
complience_level(Case, Formula, Level) :-
         asserta(Formula),
         aux_find_level(Case,Level),
         retract(Formula).

% finds the highest ranking violation.
aux_find_level(Case, Level) :-
     countdown(20,1,X),
     violation(Case,X),
     !, Level = X .
aux_find_level(_,0).

% counts from A down to B on backtracking
countdown(A,_,A).
countdown(A,B,X) :-
        A > B,
        AA is A-1,
        countdown(AA,B,X).
```

If the binding of duties constraint in 2.3 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 is straight forward from the formal definitions in the paper.

5.1 Complexity, running time, and optimizations

As a general introduction to the complexity of a logic program based implementation, one has to 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:

$$\begin{array}{l} N_{A_1}T_{A_1}+ \\ N_{A_1}N_{A_2}T_{A_2}+ \\ \ldots \\ N_{A_1}N_{A_2}\ldots N_{A_k}T_{A_k}+ \\ N_{A_1}N_{A_2}\ldots N_{A_k}T_{B_1}+ \\ N_{A_1}N_{A_2}\ldots N_{A_k}T_{B_2}+ \\ \ldots \\ N_{A_1}N_{A_2}\ldots N_{A_k}T_{B_l} \end{array}$$

Let us analyze the complexity of a `who?`(R,O,T,C) query in W0-RBAC (without the overriding of constraints). The query is implemented as:

1. find all users that satisfy the predicate R
2. remove from that set all users u for which `doer`(u,T,C) is contradictory, that is, proves `violation`.
3. order the remaining users using the relation O.

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

The second step is more costly. Let us assume that the average number of users returned by such query is U_R, where $U_R \leq U$. For each user u in the set, one needs to verify if adding `doer`(u,t,c) violates any constraint, that is, if does not prove `violation(c)`.

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

$$\begin{array}{lll} \bot & \leftarrow & doer_1, doer_2, \ldots doer_k, other_1, \ldots other_l \\ & & \textbf{not } doer_{k+1}, \ldots \textbf{ not } doer_{k+x}, \textbf{ not } other_{l+1}, \ldots \textbf{ not } other_{l+y} \end{array}$$

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 or to disprove it, takes just an 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:

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

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

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

that is 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 at most contributed with a additive constant to that complexity. Thus if the constraint rule requires very complex computation regarding the organizational structure (roles, users, organizational units), that computation is the one that 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 total worst case running time for this step is $U \times C \times \alpha$

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 $T_3 = U \times \log(U) \times T(O)$.

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

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

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 by decreasing order of priority, but in the worst case, where there is no violation, all the C violation causes will checked.

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

But there are possible optimizations that may be attempted if such computation is too expensive. Among them:

- off-line computation. Since these predicates refer to the static component of the rules, they do not change as new activities or cases are executed, only if there is a change on the role, privilege, organizational units, or users hierarchies. Since such changes are of low frequency it may be convenient to compute off line these predicates. In the worst case that will not change the number of different solutions for there predicates, but will change the time to compute each solution, which will now be an access to the Prolog fact base, and thus reducing 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(X,Y),can-play(Y,auditor)` is a segment 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(X,Y)` predicate, may result in much less than U^2 pairs of users. Such reduction in the number of solutions may be significant enough to warrant the of line computation.

- memoization. it may not be worth it to pre-compute all possible values for the static predicates that appear in the rule, but once the values for a set of input arguments have been computed, it may be worth it 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 computed.

- partial evaluation. Once enough of the activities 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 worth it 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 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 are will reduce the overall cost of computing the `who?` query, or even if any of those optimizations are necessary on the "average case."

6 Related Work

This work is closely related to work proposed by Bertino, Ferrari, and Atluri [5, 4] and the differences between our work and theirs must be stressed. [5] proposes a constraint based security model whose language allow for somewhat different expressivity than the one presented here. For example, their language and model allow one to refer to many instances of the activation of a task within a case. This activations of a task represent the execution in parallel of the same task by many (possible different) executors. This can be achieved in our model by defining each parallel activation of the task as a different task. Thus three parallel activations of the task T in their model, would be implemented in out model as three tasks T1, T2 and T3.

The language in [5] has no clear concept of case. It is assumed that the access control system runs for a single case; different cases would require running different instances of the system. By making the concept of case explicit, we allow for multiple cases to be run by the same instance of the system, and open the door for future extensions that include inter-case constraints.

[5] also discuss the ordering of answers, but they stop at the role level: they propose that more subordinate roles are given higher priority to execute a task, but they do not extend this to users, as we did. And by extending the basic idea of giving higher priority to the users that have the least privilege enough to perform the task can yield different definitions, as we described.

A large section of the [5] describes what we would call optimizations of the basic constraint verification mechanism. Some of such optimizations are performed by the system, but some are done by the user themselves. For example the constraint that T1 and T2 cannot be performed by the same persons, which in our language is expressed as:

$$\bot \leftarrow \mathsf{doer}(u, \mathrm{T1}, c),\ \mathsf{doer}(u, \mathrm{T2}, c) \tag{2}$$

would be expressed in their model, something like[1]

$$\mathsf{cannot\text{-}do}(u, \mathrm{T2}, c) \leftarrow \mathsf{doer}(u, \mathrm{T1}, c) \tag{3}$$

This last expression embodies the knowledge that in the workflow specification T1 happens before T2. Thus if it is asserted that John was the executor of T1, the system may determine that John cannot be the executor of T2. To verify if John can be the executor of a task, the system verifies first if it is known that he cannot be the executor of that task, by verifying the cannot-do predicate. In a formal way, the expression 3 is the partial evaluation of 2 given the fact that doer(u,T1,c) will be asserted.

As we discuss above, we do not believe that optimizations on the dynamic components of the constraints are necessary, and thus we did not include such optimizations. Furthermore by including such optimizations in the language itself, causes a non-separation of information that we believe should be kept separated. 3 only work if T1 happens before T2, which is a workflow knowledge. If the ordering of the activities changes, the permission system

[1] Of course the language of [5] is different; the expression below is a translation to our language of the concepts in their model.

must be modified. [5] also discuss some of what we called in the previous section, off-line optimizations, or the pre-computation of some of the static predicates, and even discuss in which conditions such pre-computations are not worth it.

We believe that our model, by providing a “cleaner” definition of the constraint, allows one to bring in many of the tools for logic program optimizations (as discussed in the previous section), and not leave it to the user’s responsibility.

Finally, [5] do not include in their work, any form of overriding constraints.

Nyanchama and Osborn [19] present a rich discussion of conflicts of interest in terms of users/groups, roles, privileges and their inter-relationships. Dynamic mechanisms requiring history are acknowledged but not discussed.

Ferraiolo et al. [8] presents the National Institute of Standards and Technology (NIST) RBAC reference model and a web-based implementation. The model is based on the properties presented in [9] and do not consider history based extensions as well.

Thomas and Sandhu [25] proposed in 1993 the shift from the traditional subject-object-right paradigm used in database security to a transaction oriented one, introducing (a limited form) of history based security mechanisms. In [22], four reference RBAC models are presented and different constraint types are categorized.

Simon and Zurko [23] survey and summarize separation of duty constraint types, both static and dynamic. The different types of constraints are then implemented in Adage, a rule-based authorization system for distributed applications.

Castano et al. [7] propose an active-rule based model, that is implemented on top of the Wide workflow system [24]. ECA rules are employed to specify instance, time, history and event constraints. The active rule mechanism is also employed to propagate privileges along the role graph, emulating the inheritance that is usually assumed along the *subordinate* relationship.

Atluri and Huang [2, 3, 13] introduce dynamic activation and revocation of privileges based on the flow of execution. The idea is that users should only be granted privileges to access objects in the context of authorized tasks. An author, for instance, should only have access to a *paper* object while preparing it for submission, Once it has been submitted, the author should not be allowed to have access to it anymore.

An approach similar to Atluri and Huang’s is followed by other researchers, e.g., Karpalem and Hung [14, 17]. These authors also discuss the trade-off between security and robustness of a system and present a metric to evaluate such trade-off.

Pernull and Herrman [12, 11] extend the different perspectives of workflow systems (informational, functional, dynamic and organizational) with security specific features.

At the University of Georgia, the Meteor system is being extended to include security features, in a joint effort with the Naval Research Laboratory [18, 27, 16, 15].

6.1 Future work

In this section we discuss further extensions to the W-RBAC model that we feel are particularly urgent.

- inter-case constraints. Some simple kinds of inter-case constraints can be handled by our present framework but more complex ones cannot. For example, we may wish

to state a constraint that a CEO can appoint any two of her three executive officers, but not all three. Each appointment is a case, and there are constraints that relate different cases.

- different activations of the same activity within the same case. It is frequent 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 new activation of the the test activity. It is possible that one would like to place constraints on the different activations of an activity (of a case). The language of W1-RBAC is not able yet to refer to different activations of the same activity (for a same case).

- verification of the satisfiability of constraints. It is common that the specification of constraints and of the potential users of certain activities to incur in inconsistencies. A typical one is to require that it is the immediate boss of the employer who wants a reimbursement that should approve the request. But some users may not have a 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 may not be able to play the role of approver, because one would not want to bother them with such tasks.

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

Most of the contributions described in the paper and the ones in the future work list were presented in terms of a permission system working in conjunction with a workflow system. But most of the ideas are independent of this particular workflow application.

An example is graceful degradation. Such concept was presented here in conjunction with delegation, but graceful degradation is an important concept in any constraint based system: if the constraints are too strong and do not allow or a solution, or if in some particular cases some constraints should not be applied, then it should be possible to in a controlled way violate some constraints. Again there is still work to be done in developing the concept described herein to generic constraint based systems.

References

[1] Gail-Joon Ahn and Ravi Sandhu. The RSL99 language for role-based separation of duty constraints. In *ACM RBAC Workshop 99*, pages 43–54, 1999.

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

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

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

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

[6] 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):26–65, 1998.

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

[8] D.F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2(1):34–64, 1999.

[9] D.F. Ferraiolo and D. R. Kuhn. A role-based access control. In *Proc. of the 15th Annual Conference on National Computer Security.* National Institute of Standards and Technology, 1992.

[10] C. Goh and A. Baldwin. Towards a more complete model of roles. In *3rd ACM Workshop on Role-Based Access*, pages 55–61, 1998.

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

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

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

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

[15] M. H. Kang, J. N. Froscher, B. J. Eppinger, and I. S. Moskowitz. A strategy for an mls workflow management system. In *Proc. of the 13th Annual IFIP WG11.3 Working Conference on Database Security*, 1999.

[16] M. H. Kang, J. N. Froscher, A. P. Sheth, K. Kochut, and J. A. Miller. A multilevel secure workflow management system. In *11th International Conference on Advanced Information Systems Engineering - CAiSE'99*, volume 1626 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 1999.

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

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

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

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

[21] S. Osborn and Y. Guo. Modeling users in role-based access control. In *ACM RBAC 2000*, pages 31–37, 2000.

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

[23] R. Simon and M. E. Zurko. Separation of duty in role-based environments. In *Proc. of the 10th Computer Security Foundations Workshop (CSFW '97)*, 1997.

[24] G. Snchez. The wide project: Final report. Technical report, Wide Consortium, 1999.

[25] R. Thomas and R. Sandhu. Towards a task-based paradigm for flexible and adaptable access control in distributed applications. In *Proc. on the 1992/1993 ACM SIGSAC on New security paradigms workshop*, pages 138–142, 1993.

[26] Workflow Management Coalition. Workflow management coalition specification - the workflow reference model. Technical report, February 1995. http://www.aiai.ed.ac.uk:80/WfMC/glossary.html.

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