RBAC Administration in Distributed Systems

M.A.C. Dekker[1,3], J. Crampton[2], and S. Etalle[3]

[1] Security Group, TNO ICT, The Netherlands,
[2] Information Security Group, Royal Holloway, University of London, UK,
[3] Distributed and Embedded Systems Group, University of Twente, The Netherlands

Abstract. Despite a large body of literature on the administration of RBAC policies in centralized systems, the problem of the *administration* of a distributed system has hardly been addressed. We present a formal system for modelling a distributed RBAC system and its administration. We define two basic requirements for distributed RBAC systems, based on safety and availability. We present a transition-system modelling the actual implementation of administrative commands and we show that it preserves those requirements. We also indicate how the system can be used as a basis for a practical implementation.

1 Introduction

Role-based access control (RBAC) [1, 5, 11] is the de-facto access control standard. RBAC simplifies the assignment of access rights to users, by grouping users in a relative small number of roles, which are ordered in a *role-hierarchy*. Nevertheless, in practice RBAC policies can be very large, and the administration of the RBAC policy can be a complex task. For example, in companies or hospitals an RBAC policy may involve thousands of roles and assignments [5], involving multiple systems across the organization.

The RBAC standard defines a basic set of administrative functions and controls [1]. Several lines of research extend these administrative functions. In the ARBAC administrative model [10], administrative privileges are defined using ranges of roles that can be changed. Crampton and Loizou [3] take a different approach by defining the *administrative scope* of a role. A role can only have administrative privileges over the roles in its administrative scope. In a similar approach, Wang and Osborn [12] divide the role-graph (a type of RBAC policy) into administrative domains. Each administrative domain has an administrator, who has privileges on the roles in that domain.

Although there is a large body of literature about many aspects of the delegation of administrative authority and administrative policies [3–5, 10, 12], there is no literature that deals with the implementation of

administrative policies in a distributed system (see Related work). To see why this is not trivial consider that a distributed system consists of different subsystems which in turn protect different sets of objects. (Think for instance of an organization with several departments, using a number of different databases and file systems.) The administration mechanism for the distributed system must fulfil a number of desiderata. First, each subsystem should store only the parts of the RBAC policy concerning the objects that it protects. Indeed, it is good engineering practice to see that the subsystems do not store excessive policy information; this helps to reduce the chances of inconsistencies. Second, when a policy change is made, updates to affected subsystem should be timely and efficient. In this phase, re-sending the whole RBAC policy to every subsystem is undesirable, because of efficiency reasons. On the other hand, simply making the policy change at each individual subsystem is not possible, since the peripheral subsystems do not store the whole RBAC policy, and they do not have enough information to carry out the policy change correctly.

In this paper, we propose a model for the administration of a distributed RBAC system. Our approach is based on a formal transition system. We formalize the desiderata (soundness, completeness and leanness), and we define the update process using an abstract form of message-passing progression between an administrative reference monitor and the periphery subsystems in the distributed system. Using this model, we define a transition function modelling the administration of a distributed RBAC system, showing that it preserves the desiderata, whilst keeping to a minimum the amount of policy information that is sent to the different subsystems in the distributed system. Finally, we show how our transition function can be translated into procedures for the administrative and the non-administrative reference monitors in a practical implementation of a distributed RBAC system.

The rest of this paper is structured as follows: In the following preliminary section we give the basic definitions for the RBAC model we use throughout this paper. In Section 3 we introduce administrative policies, and a basic model for an administrative reference monitor. In Section 4 we extend this basic model to the setting of a distributed system. We define basic requirements for the distributed system, based on safety and availability, and we implement the execution of administrative commands in such a way that the basic requirements are preserved. In Section 5 we demonstrate a practical implementation of an administration point. Sections 6 and 7 contain related work and our conclusions.

2 Preliminaries

We first introduce the *General Hierarchical RBAC* model, as defined in the ANSI RBAC standard, because it is the most commonly used RBAC model [1, 5].

The goal of an RBAC policy is to specify which users are permitted to perform which actions on which objects. We denote the sets of users, roles, actions, and objects, by U, R, A, and O. Permissions for performing actions on objects are called *user privileges*, forming a set $P \subseteq A \times O$,

An RBAC policy assigns users to roles, roles to user privileges, and it defines an order on the roles; the *role-hierarchy*[4].

Definition 1 (Non-administrative Policies). *Let U, R, and P, be sets of users, roles, and user privileges,* a non-administrative RBAC policy *ϕ is a tuple*

$$\phi = (UA, RH, PA),$$

where $UA \subseteq U \times R$ is a set of user assignments, $RH \subseteq R \times R$ a role-hierarchy, and $PA \subseteq R \times P$ a set of privilege assignments.

The set of RBAC policies is denoted $\Phi_{U,R,P}$. To simplify our exposition we treat a policy ϕ as a *directed graph*, defined by the *set* of directed edges $UA \cup RH \cup PA$. If there is a path from one vertex v to another v' we write $v \rightarrow^*_\phi v'$.

The RBAC *reference monitor* uses the policy ϕ as follows. Any user u can start a *session*. The reference monitor allows the user to activate a role r in a session iff $u \rightarrow^*_\phi r$. In that case the privileges of the session are all privileges p such that $r \rightarrow^*_\phi p$. In other words, a user u, in a role r, can perform an action a on an object o, iff $u \rightarrow^*_\phi r$ and $r \rightarrow^*_\phi (a, o)$. We abbreviate this by $u \rightarrow^*_\phi r \rightarrow^*_\phi (a, o)$. Sessions are an important safety mechanism, allowing users to apply the *principle of least privilege* [1]. Here we do not go into details about the session mechanism.

Example 2 (Basic RBAC). Consider a system s protecting files. The RBAC policy of the system s is defined by

$$\begin{aligned} UA &= \{(alice, prof)\}, \\ RH &= \{(prof, stud)\}, \\ PA &= \{(prof, (write, foo)), (stud, (read, foo))\}. \end{aligned}$$

[4] In the RBAC standard the relation RH is defined to be acyclic, reflexive and transitive, i.e. it is defined as a partial order. On the other hand, Li et al. showed that this definition causes problems when changes are made to the role-hierarchy [7]. Here, for the sake of generality we do not assume that RH is a partial order.

Alice starts a session at system s. Now, if Alice activates the role *prof*, then the system s allows her to perform the actions *write* and *read* on the object *foo*. If, on the other hand, Alice activates only the role *stud*, then the system allows only the latter action.

In the sequel we will use the following notation: Given a policy ϕ, the *upper closure* of a role r, is the set

$$(\uparrow_\phi r) = \{(v, v') : (v, v') \in \phi,\ v' \rightarrow_\phi^* r\}.$$

Notice that the upper closure of a policy is again a policy.

3 Administration of Centralized RBAC

In this section we present a simple model for the *administration* of a centralized RBAC system. This will serve as a basis for the distributed systems we will introduce in the next section.

The RBAC standard specifies a number of *administrative functions and controls*, which can be used by administrative users to make policy changes [1]. In this paper we use administrative privileges to model which users (or roles) can make which policy changes and we assign administrative privileges to roles just like the user privileges are assigned to roles in standard RBAC. This approach is also advocated in the literature [3, 12] and the intuition behind it is that the RBAC policy can also be used to specify who can change the RBAC policy. We model privileges that allow users to make changes to the sets UA, RH or PA. We do not model the changes to the sets U, R or P as we can safely assume that these sets are large and fixed. For example the set of user names can be chosen as the set of all lower-case strings starting with *uid*.

Definition 3 (Administrative Privileges). *Given the sets U, R, and P, administrative privileges form a set*

$$P^\circ \subseteq A^\circ \times O^\circ,$$

where $A^\circ = \{assign, revoke\}$, and $O^\circ = (U \times R) \cup (R \times R) \cup (R \times P)$.

For example the administrative privilege $(assign, (r, r'))$ allows the addition of an edge from role r to role r'.

The definition of RBAC policies is extended accordingly.

Definition 4 (Administrative Policies). *Let U, R, P by sets of users, roles and user privileges,* an administrative RBAC policy ϕ *is a tuple*

$$\phi = (UA,\ RH,\ PA \cup PA^\circ),$$

where $UA \subseteq U \times R$ is a set of user assignments, $RH \subseteq R \times R$ a role-hierarchy, $PA \subseteq R \times P$ are the assignments to user privileges and $PA^\circ \subseteq R \times P^\circ$ are the assignments to administrative privileges.

The set of administrative policies is denoted by $\Phi^\circ_{U,R,P}$, which is a superset of the policy set $\Phi_{U,R,P}$ defined in the RBAC standard.

We can now model the administrative reference monitor. Administrative policies allows users in roles with administrative privileges to make changes to the policy of the reference monitor. We model the administrative reference monitor by a queue of administrative commands, and an (administrative) policy. The reason for using a command queue will become clear in the next section.

Definition 5 (Administrative Commands). *Let U, R, A, O be sets of users, roles and user privileges, an* administrative command *is a term*

$$\texttt{cmd}(u, r, a, e),$$

where $u \in U$, $r \in R$, $a \in A^\circ$ and $e \in O^\circ$. A command queue *is a list of administrative commands, denoted $cq = \texttt{cmd}(u, r, a, e) : \texttt{cmd}(u', r', a', e')...$, where $:$ denotes the list constructor.*

The set of command queues is denoted CQ. The empty command queue is denoted ε. When a user makes an administrative command, it is placed at the end (the right hand side) of the queue.

Definition 6 (Local Administration). *Let $cq \in CQ$ be a command queue, and $\phi \in \Phi^\circ$ an administrative policy,* local administration *is a transition function $\Rightarrow$: $CQ \times \Phi^\circ \to CQ \times \Phi^\circ$ where,*

$$\begin{aligned}
\langle \texttt{cmd}(u, r, assign, e) : cq,\ \phi \rangle &\ \Rightarrow\ \langle cq,\ \phi \cup e \rangle, \text{ if } u \to^*_\phi r \to^*_\phi (assign, e).\\
\langle \texttt{cmd}(u, r, revoke, e) : cq,\ \phi \rangle &\ \Rightarrow\ \langle cq,\ \phi \setminus e \rangle, \text{ if } u \to^*_\phi r \to^*_\phi (revoke, e).\\
\langle \texttt{cmd}(\ldots) : cq,\ \phi \rangle &\ \Rightarrow\ \langle cq,\ \phi \rangle, \text{ otherwise.}
\end{aligned}$$

A user u can place any administrative command at the end of the queue, but the administrative system executes the command only if it is allowed by the policy ϕ. If an administrative command is not allowed, then the command is removed from the queue, without changing the policy ϕ. A set of consecutive transitions, corresponding to repetitive executions of commands on the queue, is called a *run*. Below a run is denoted by $\Rightarrow^*$.

4 Administration of Decentralized RBAC

Having specified a basic model for the adminstration of centralized RBAC, we can now address the issue of administration of decentralized RBAC. Consider the setting of a large distributed system composed of databases, file systems etc. In such a setting it is impossible to use a central reference monitor to decide about all the actions. Each action would involve contacting the central reference monitor, resulting in a bottleneck. On the other hand, if we allow each subsystem to have its own reference monitor, and its own RBAC policy, we need to manage these policies in an efficient and consistent way. This is the problem we tackle in this section.

We use a set of names S for the different subsystems in the distributed system. The subsystems S are non-administrative reference monitors, which decide about whether or not to allow users to perform actions on the objects they protect. Note that the subsystems in S do not execute administrative commands. To allow changes to be made to the RBAC policies of these reference monitor, we assume the presence of a single *administrative reference monitor*. Here and in the sequel we use the fixed sets U, R, P to denote users, roles, and user privileges across the distributed system. The model for the distributed system is as follows.

Definition 7 (Distributed System). *A distributed system is a tuple*

$$(S,\ sm,\ \phi,\ \overline{\psi}\ ),$$

where $sm : S \to \mathcal{P}(P)$ *is a function that maps each subsystem to a set of user privileges,* $\phi \in \Phi^{\circ}_{U,R,P}$ *is an administrative policy, and* $\overline{\psi} : S \to \Phi_{U,R,P}$ *is a function that maps each subsystem to a non-administrative policy.*

The set of distributed systems $(S,\ sm,\ \phi,\ \overline{\psi}\ )$ is denoted DS. The service mapping sm defines which subsystems *protect* which objects. If a user privilege $(a, o) \in sm(s)$, then we say that the subsystem s protects object o. For the sake of generality, we do not assume that $sm(s)$ and $sm(s')$ are disjoint for different subsystems s and s', so in principal one object could be protected by multiple subsystems. The set of privileges $sm(s)$ are referred to as the *relevant user privileges* for the subsystem s. The administrative policy ϕ is the policy of the administrative reference monitor. It is referred to as the administrative policy of the distributed system. The function $\overline{\psi}$ is the *distribution* of policies across the subsystems S, so $\overline{\psi}(s)$ is the the policy of the RBAC reference monitor of subsystem s. In the

rest of this section we implement the execution of administrative commands that change $\overline{\psi}$ and ϕ. Let us first give a simple practical example of the mapping sm.

Example 8 (Relevant User Privileges). A university department has a network consisting of a database named *Sqil*, a fast computer *Qalc* and a printer *Inq*. The mapping $sm : S \to P$ is as follows.

$$\begin{aligned} sm(Sqil) &= \{(gradetable, view), (gradetable, insert)\} \\ sm(Qalc) &= \{(job, halt), (job, start)\} \\ sm(Inq) &= \{(black, print), (color, print)\} \end{aligned}$$

The administrative policy ϕ is depicted in the upper circle of Figure 1. Each subsystem has a different RBAC policy specifying which users can access which objects or resources. They are depicted in the lower three circles in Figure 1. These policies constitute the policy distribution $\overline{\psi}$. For the sake of simplicity, the user assignments to roles are not shown in the figure.

We proceed as follows. We define two basic requirements for the distribution $\overline{\psi}$ with respect to the administrative policy ϕ. The two requirements are motivated by the principles of safety and availability.

Definition 9 (Soundness and Completeness). *Given a distributed system $(S, sm, \phi, \overline{\psi})$, the distribution $\overline{\psi}$ is* sound *with respect to the central policy ϕ, iff*

$$\bigcup_{s \in S} \overline{\psi}(s) \subseteq \phi.$$

The distribution $\overline{\psi}$ is complete *with respect to the central policy ϕ iff for every subsystem $s \in S$,*

$$\text{if } p \in sm(s) \text{ then } (\uparrow_\phi p) \subseteq \overline{\psi}(s)$$

where $(\uparrow_\phi p)$ denotes the upper closure of p in ϕ (cf. Preliminaries).

Soundness is important from the viewpoint of safety. Soundness ensures that subsystems allow access only when it is allowed by the administrative policy ϕ, allowing safety analysis (of policy changes): It implies that if $u \to^*_{\overline{\psi}(s)} r \to^*_{\overline{\psi}(s)} p$ then $u \to^*_\phi r \to^*_\phi p$.

Completeness, on the other hand, is important from the viewpoint of availability. Completeness ensures that the subsystem protecting object o allows access to the object o, iff it is allowed by the administrative policy: It implies that, for any $p \in sm(s)$, if $u \to^*_\phi r \to^*_\phi p$ then $u \to^*_{\overline{\psi}(s)} r \to^*_{\overline{\psi}(s)} p$.

Remark 10 (Trivial Distribution). A trivial distribution $\overline{\psi}$ that meets the soundness and completeness requirements is the distribution where $\overline{\psi}(s) = \phi$ for all $s \in S$. Here, all the subsystems have the same policy. However, this distribution would not follow the basic engineering principle that a component should only store what is strictly necessary. Moreover, each policy change would require updating all the different subsystems, which is inefficient.

Example 11 (Practical Policy Distribution). Let us continue our previous

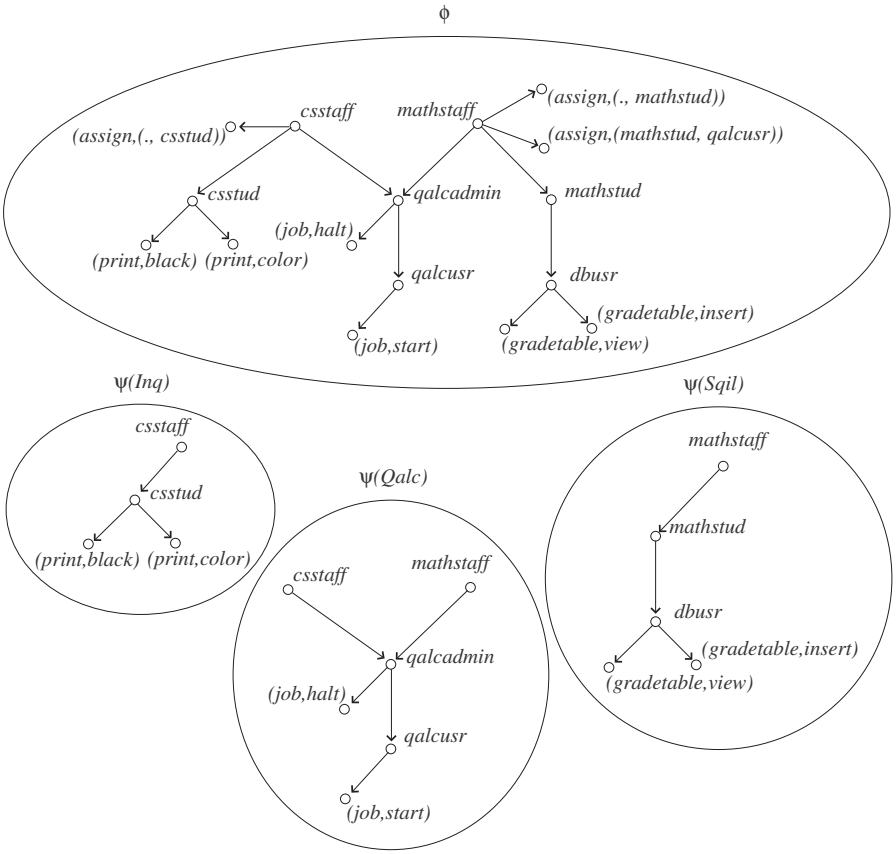

Fig. 1. Sound and complete RBAC policy distribution.

example (cf. Example 8). According to the mapping *sm* (cf. Example 8), the distribution $\overline{\psi}$ is sound and complete. Take for example the complete-

ness requirement regarding the RBAC policy of the subsystem *Inq*. The printer *Inq* does not protect database tables of *Sqil*, nor the processing resources of *Qalc*. Completeness requires only that the policy of *Inq* contains the upper closure of the printing privileges. The other parts of ϕ are *in practice* irrelevant for *Inq*. Notice that also that the administrative privileges are irrelevant for all subsystems (cf. Example 8), because the subsystems can not be used for administrative operations.

Now we come to the execution of administrative commands in our distributed system model. In a distributed system, the administrative reference monitor may need to update the subsystems about policy changes: We model this by changing the command queue defined earlier to include message commands. Message commands propagate parts of the central policy ϕ to update the subsystems about policy changes.

Definition 12 (Distributed Administrative Commands). *Given a distributed system* $(S, sm, \phi, \overline{\psi})$*, the set of distributed administrative commands is*

$$\{\texttt{cmd}(u, r, a, o), \texttt{msg}(S', add, \delta), \texttt{msg}(S', remove, \delta)\},$$

where $u \in U$*,* $r \in R$*,* $a \in A^\circ$*,* $o \in O^\circ$*,* $S' \subseteq S$*, and* $\delta \subseteq \phi$*. The command queue in the distributed setting is a list of distributed administrative commands.*

The administrative command $\texttt{cmd}(u, r, a, o)$ changes the policy ϕ as defined in the previous section. The command $\texttt{msg}(S', add, \delta)$ adds δ to $\overline{\psi}(s)$ for $s \in S'$, while the command $\texttt{msg}(S', remove, \delta)$ removes δ from $\overline{\psi}(s)$. The new set of command queues is denoted CQ_D, which is a superset of CQ. We can now define a transition function for our distributed system model.

Definition 13 (Distributed Administration). *Given a distributed system* $(S, sm, \phi, \overline{\psi})$*, let* $cq \in CQ_D$ *be a command queue. The distributed administration transition is a function* $\Rightarrow_D$*:* $CQ_D \times DS \rightarrow CQ_D \times DS$*,*

$$\langle cq, S, sm, \phi, \overline{\psi} \rangle \;\Rightarrow_D\; \langle cq', S, sm, \phi', \overline{\psi}' \rangle$$

where,

if $cq = \texttt{cmd}(u, r, assign, (v, v')) : cq''$, *and* $u \rightarrow_\phi^* r \rightarrow_\phi^* (assign, (v, v'))$, *then*
$cq' = \texttt{msg}(S', add, (v, v') \cup (\uparrow_\phi v)) : cq''$,
where $s \in S'$ *iff there is a privilege* $p \in sm(s)$ *such that* $v' \rightarrow_\phi^* p$,
$\phi' = \phi \cup (v, v')$,
$\overline{\psi}' = \overline{\psi}$.

if $cq = \texttt{cmd}(u, r, revoke, (v, v')) : cq'',$, *and* $u \rightarrow_\phi^* r \rightarrow_\phi^* (assign, (v, v'))$, *then*
$cq' = \texttt{msg}(S, remove, (v, v')) : cq''$,
$\phi' = \phi \setminus (v, v')$,
$\overline{\psi}' = \overline{\psi}$.

if $cq = \texttt{msg}(S', add, \delta) : cq''$,
$cq' = cq''$,
$\phi' = \phi$,
$\overline{\psi}'(s) = \overline{\psi}(s) \cup \delta$ *for* $s \in S'$ *else* $\overline{\psi}'(s) = \overline{\psi}(s)$.

if $cq = \texttt{msg}(S', remove, \delta) : cq''$,
$cq' = cq''$,
$\phi' = \phi$.
$\overline{\psi}'(s) = \overline{\psi}(s) \setminus \delta$ *for* $s \in S'$ *else* $\overline{\psi}'(s) = \overline{\psi}(s)$.

if otherwise, and $cq' = cq''$, $\phi' = \phi$ *and* $\overline{\psi}' = \overline{\psi}$.

Basically, in the execution of the administrative command, not only the administrative policy ϕ of the administrative reference monitor changes, but a message command is *prepended* to the command queue (contrary to administrative commands which are appended). The message commands model the updates in a practical implementation which are sent by the administrative reference monitor to the subsystems. In the case of assignment, the update is sent to the subsystems in S', which are all the subsystems with relevant privileges in the lower closure of the policy change. The update contains the upward closure of the policy change. Notice that the content of the message is smallest when the command is a user assignment, since the upper closure of a user is always empty. In practice this is also the most frequently used administrative command [5].

While updates following an assignment are sent only to those subsystems that are 'affected' by it, revocations are broadcast to all the

subsystems to ensure soundness of $\overline{\psi}$. In some cases a basic expiration mechanism can be used to reduce the number of revocations [9]. Although we do not go into details about time or expiration here, we would like to mention that an expiration mechanism can be implemented straightforwardly in our model: The administrative command for assignment should also specify an expiration date for the edge that is being added. The definition of policies can be extended straightforwardly to allow edges to carry expiration dates. The administrative system, and the subsystems in S should have a way to check if edges have expired. It is important to note that the transition system defined above can remain exactly the same. Soundness and completeness are preserved when edges expire. Practical considerations determine which expiration dates should be used for which assignments. For example, inside an organization where employees go away frequently and unexpectedly, it would be appropriate to make only user assignments that expire quickly.

Let us give a practical example of the distributed administration transition.

Example 14 (Updates in practice). We continue using the setting described in the previous examples. Consider the administrative privileges depicted in the administrative policy ϕ in Figure 1. Bob, who is a member of *mathstaff*, wants to allow all members of *mathstud* the possibility to use the fast computer *Qalc*, say for mathematical modelling. We describe the transitions step by step: The first command in the queue is

$$\texttt{cmd}(\mathit{Bob}, \mathit{mathstaff}, \mathit{assign}, (\mathit{mathstud}, \mathit{qalcusr})).$$

After executing this command, the new policy ϕ' contains the new edge $(\mathit{mathstud}, \mathit{qalcusr})$ and the command on the queue is replaced by the command

$$\texttt{msg}(\mathit{Qalc}, \mathit{add}, (\mathit{mathstud}, \mathit{qalcusr}) \cup (\uparrow_{\phi} \mathit{mathstud})).$$

The message command is executed, updating the policy of *Qalc*. The new policy for *Qalc* includes the upper closure of *mathstud*, i.e. the new edge $(\mathit{mathstud}, \mathit{qalcusr})$, as well as the members of *mathstud*.

Note that Bob's administrative command changes the policies ϕ and $\overline{\psi}(\mathit{Qalc})$, but it does not affect the policies of *Inq* or *Sqil*. These subsystems do not receive updates. The policy changes corresponding to Bob's action are depicted in Figure 2 by dashed edges.

The administration mechanism defined above preserves the safety and availability requirements for the policy distribution $\overline{\psi}$ (without sending to

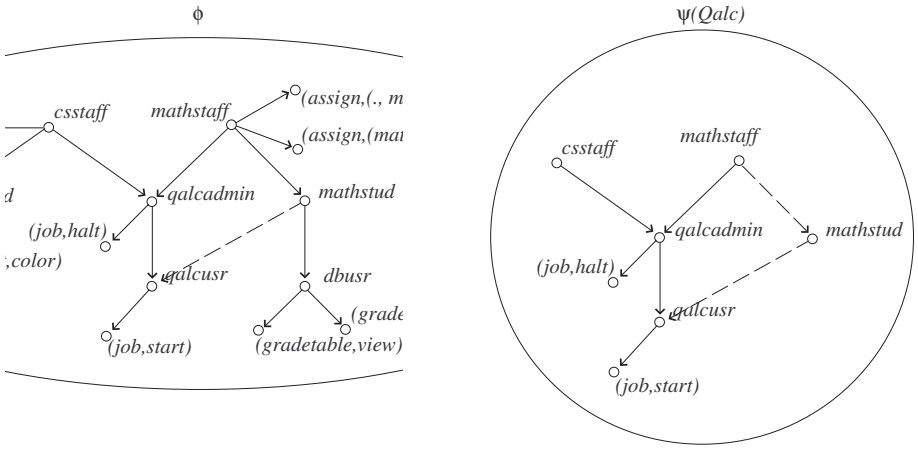

Fig. 2. An update, and its effect on the policy of subsystem *Qalc*.

subsystems parts of ϕ that are irrelevant for them). As before, we define a *run* as the consecutive execution of commands on the queue. A run in the distributed system is denoted by $\Rightarrow_D^*$. It can be shown that a run from a queue with administrative commands, $cq \in CQ$, preserves soundness and completeness.

Theorem 15. *Given a distributed RBAC system* $(S,\ sm,\ \phi,\ \overline{\psi}\ )$*, let* $cq \in CQ$ *be a command queue. If*

$$\langle cq,\ S,\ sm,\ \phi,\ \overline{\psi}\ \rangle \Rightarrow_D^* \langle \varepsilon,\ S,\ sm,\ \phi',\ \overline{\psi}'\ \rangle,$$

holds, then the following statements hold:

1. $\langle cq,\ \phi\ \rangle \Rightarrow^* \langle \varepsilon,\ \phi'\ \rangle$.
2. *If* $\overline{\psi}$ *is sound wrt.* ϕ*, then also* $\overline{\psi}'$ *is sound wrt* ϕ'.
3. *If* $\overline{\psi}$ *is complete wrt.* ϕ*, then also* $\overline{\psi}'$ *is complete wrt* ϕ'.

Proof. (Sketch) We sketch the proof for a command queue with a single command. Multiple commands follow by induction.

The first statements states that the changes to the administrative policy are the same as in the centralized setting. This holds since both commands $\texttt{cmd}(u, r, assign, (v, v'))$ and $\texttt{cmd}(u, r, revoke, (v, v'))$ change the administrative policy of the distributed system in the same way as in the

centralized setting (cf. Section 3), and the message commands that are added to the queue do not change the administrative policy.

The second statement says that soundness is preserved. We can assume that $\overline{\psi}$ is sound wrt ϕ. Let us distinguish according to the first command in the queue. In the case the command is $\texttt{cmd}(u, r\mathit{assign}, (v, v'))$, it is replaced with the message command $\texttt{msg}(S', (v, v') \cup (\uparrow_\phi v))$ on the queue, and the administrative policy is changed to $\phi' = \phi \cup e$ (cf. the first item in Definition 13). After this message command has been processed, we'll have that the policy of a subsystem $s \in S'$ is changed to $\overline{\psi}(s) \cup (v, v') \cup (\uparrow_\phi v)$. Since $(v, v') \cup (\uparrow_\phi v) \in \phi'$ and $\overline{\psi}$ is sound, also $\bigcup_{s \in S} \overline{\psi}'(s) \subseteq \phi'$, i.e. $\overline{\psi}'$ is sound wrt ϕ'. In the case the command is $\texttt{cmd}(u, r, \mathit{revoke}, (v, v'))$, it is replaced by the message command $(S, \mathit{remove}, (v, v'))$ on the queue, and the administrative policy is changed to $\phi' = \phi \setminus (v, v')$. After this message command is executed the the edge (v, v') is removed form all the policies of the subsystems, which ensures that $\cup_{s \in S} \overline{\psi}'(s) \subseteq \phi'$, i.e. that $\overline{\psi}'$ is sound wrt ϕ'.

The third statement says that completeness is preserved. We can assume that $\overline{\psi}$ is complete wrt ϕ. We distinguish cases for the first command in the queue. In case the first command is $\texttt{cmd}(u, r, \mathit{assign}, (v, v'))$, it is replaced by the message command $\texttt{msg}(S', \mathit{add}, (v, v') \cup (\uparrow_\phi v)$ on the queue, and the administrative policy changes to $\phi' = \phi \cup (v, v')$. If the message command is processed then the policy of a system in $s \in S'$ changes to $\overline{\psi}(s) \cup (v, v') \cup (\uparrow_\phi v)$, and the policies of the other subsystems remain unchanged. Completeness requires that the policy of each subsystem contains the upper closure of each relevant user privilege. Take a subsystem s not in S'. None of the relevant privileges are below v', therefore the upper closures of its relevant privileges do not change. Now take a subsystem $s \in S'$. There are relevant privileges below v, so the upper closure of the relevant privileges now contains also $(v, v') \cup (\uparrow_\phi v)$. Due to the message command, the new policy of $s \in S'$ also contains $(v, v') \cup (\uparrow_\phi v)$. Hence the distribution $\overline{\psi}$ is complete. Finally in case the first command is $\texttt{cmd}(u, r, \mathit{revoke}, (v, v'))$, it is replaced by the message command $(S, \mathit{remove}, (v, v'))$ on the queue, and the administrative policy is changed to $\phi' = \phi \setminus (v, v')$. Processing the message command removes the edge from the policies of the subsystems. The upper closure of any vertex may change, but in that case it decreases in the subsystem's policy in the same way as in the administrative policy. □

Leanness The mapping sm determining which privileges are relevant for which subsystems, can also be used to define the smallest policy distribution that satisfies soundness and completeness.

Definition 16 (Leanness). *Given a distributed system* $(S,\ sm,\ \phi,\ \overline{\psi}\ )$*, the distribution* $\overline{\psi}$ *is called lean iff for every subsystem* $s \in S$*,*

$$\overline{\psi}(s) = \bigcup_{p \in sm(s)} (\uparrow_{\phi} p)$$

A *lean* policy distribution has the advantage that components of the distributed system only have parts of the RBAC policy which are needed to decide correctly about allowing or denying user actions. This means that the decision procedure implemented at the reference monitor is not slowed down by evaluating irrelevant policy definitions. It can be shown that a lean distribution is the smallest policy distribution that satisfies both soundness and completeness. It is straightforward to see that a run starting with a command queue containing no revocations preserves leanness. If revocations are performed, leanness may not be preserved. Consider for example Figure 2. Suppose that Bob, after realizing a mistake, revokes the edge $(mathstud, qalcusr)$ by giving the command $\texttt{cmd}(Bob, mathstaf, (mathstud, qalcusr))$. The transition system will remove this edge from $\overline{\psi}(Qalc)$, however, the members of *mathstud* are not removed from $\overline{\psi}(Qalc)$ (which are now superfluous for the subsystem $Qalc$). This means that $\overline{\psi}(Qalc)$ is no longer lean. It can be shown that in order to preserve leanness it is sufficient to add to our transition system a new transition (independently from the message commands), which takes care of removing irrelevant parts of the subsystem's policy. This step can be carried out independently of the administrative reference monitor. This corresponds to the intuition that each subsystem can decide for itself to remove the irrelevant bits of its policy, for example after edges have been revoked, or when edges expire. The procedure for a subsystem s is as follows. For each edge (v, v') in the policy the system checks whether or not $v' \rightarrow^* p$ for a relevant privilege $p \in sm(s)$. If not, then the edge can be discarded.

5 Implementation

Here we show how the distributed administration transition system can be used in a straightforward way to define actual procedures for the administrative system to execute the administrative commands. We do this

by specifying the procedures in pseudocode for the administrative reference monitor and for the non-administrative reference monitors of the subsystems.

Definition 17 (Administrative Reference Monitor).

```
procedure(policy, cmd) if cmd = assign(user, role, v1, v2){
   l1 = lower(policy,user);
   l2 = lower(policy,role));
   if role in second(l1) and "(assign,(v1,v2))" in second(l2) then {
       l3 = upper(policy,v1);
       l4 = lower(policy,v2);
       for s in systems do {
           if sm(s, second(l4)) then
                    msg(s,"add",l3+"(v1,v2)");
           else nothing;}
       return(policy + "(v1,v2)");}
   else
       return("command not allowed");}
if cmd = revoke(user, role, v1, v2){
   l1 = lower(policy,user);
   l2 = lower(policy,role));
   if role in second(l1) and "(assign,(v1,v2))" in second(l2) then {
          for s in systems do msg(s,"remove","((v1,v2)"));
          return(policy - "(v1,v2)");}
   else
       return("command not allowed");}
```

Let us explain the procedure in detail. Vertices `v1, v2,...` (users, roles, and privileges) are assumed to be strings, edges are bracketed pairs of such strings `(v1, v2)`, and policies are lists of edges. In the first step the lower closure of the vertex `user` and that of the vertex `role` are calculated. The function `lower(a, b)` returns a list of elements from the policy `a` which are in the lower closure of `b`: i.e. ($\downarrow_a b$). In the second step it is checked that the operation is allowed. The expression `a in b` is true if an element `a` is in the list `b`, while the function `second` takes a list of pairs, and returns a list of only the second element of each pair.

Now if the user in that role is allowed to perform the operation, then the lower closure of `v1` and the upper closure of `v2` are calculated. The lower closure of `v2` is used to select systems that must receive an update (denoted by `msg`). The function `sm` takes a list and a system name as input and it returns true if the list contains relevant privileges for the system.

The upper closure of `v1` is included in the update (cf. Definition 13). The operators `+` and `-` denote appending an element to a list, and removing, respectively. Finally, the procedure terminates by returning either the new administrative policy, or if the operation was not allowed, an error message.

The procedure for the non-administrative system is more simple. There are two types of commands: A message commands from an administrative system, or a user command, to perform an action on an object, denoted by the command `execute`.

Definition 18 (Non-Administrative Reference Monitor).

```
procedure(policy, cmd) if cmd = msg(add, delta){
       return(policy + delta);}
if cmd = msg(remove, delta){
        return(policy - delta);}
if cmd = (user, role, action, object){
   l1 = lower(policy,user);
   l2 = lower(policy,role));
   if role in second(l1) and "(action,object)" in second(l2) then {
          execute(action, object);
   else
       return("command not allowed");}
```

6 Related Work

In this paper we have focussed on the issue of administration of RBAC in a distributed system. This issue has hardly been addressed in existing RBAC literature.

Most closely related to our work is a paper by Bhamidipati and Sandhu which discusses how RBAC can be used in a number of different architectures with multiple servers in a network [2]. They focus however on the capabilities of the servers (whether or not RBAC is supported), while treating the role-hierarchy, without privilege assignments, as a central service. We do not make this distinction, instead we use the definition of privileges in the RBAC standard (and safety and availability considerations) to determine how to distribute the RBAC policy across the distributed system.

dRBAC is a decentralized trust management and access control mechanism for systems that span multiple administrative domains [6]. It is targeted at settings where independent organizations form dynamic coalitions (This setting is also addressed by the TM models discussed below.)

In dRBAC, local RBAC policies in one administrative domain can be used in another domain, by using trust statements about the remote domain. In this paper, on the other hand, we focus on the setting of a distributed system *within* a single organization, and on how to implement administration within a single administrative domain.

There are several lines of research about administrative RBAC policies (aiming to extend the RBAC standard with delegation of administrative authority to multiple users). It is not clear how these administrative models can be used in a distributed system. We mention one of the proposals. Wang and Osborn introduce administrative domains for *role graphs*, a class of RBAC policies with a single lowest and single highest role, , called *minrole* and *maxrole* respectively. Each administrative domain is defined by one role, and it contains all the roles below it, except *minrole*. Administrative domains can not overlap, unless one domain includes the other completely. Wang and Osborn justify this restriction by arguing that it should not be allowed for different domain administrators to make changes to the same roles. On the other hand they also stress that this is a disadvantage of their model, arguing that in practice one would like to have overlapping domains, for example when one resource is shared by different departments (see Figure 1). In the administrative domains model, *calcadmin* can only be managed by the highest domain administrator, which manages also *mathstaff* and *csstaff*. Although we agree that there may be practical settings where such administrative models can be used, we do not adopt a particular model in this paper.

Role-based Trust Management (TM) [8] and distributed certificate systems [9], such as SDSI [9], are somewhat related lines of research. In these systems, a number of agents exchange security statements, and they may create hierarchies similar to those used in RBAC. In TM it is assumed that users are free to utter security statements, while the focus is on whether to trust such statements (which involves some trust calculation by the receiver of such statements). In RBAC this assumption is inappropriate, because statements changing the RBAC policy are explicitly guarded by administrative privileges. The central issue of this paper, that is, to ensure that users can perform the actions they are allowed to, without necessarily broadcasting the entire security policy, has also been researched in trust management models: Administrative actions in RBAC correspond to issuing TM credentials. In some TM models the user is expected to collect the credentials needed for access, in others *credential chain discovery* algorithms are used, which are procedures to retrieve missing credentials from remote locations.

7 Conclusion

Although there is a large body of literature on the different aspects of specification of administrative RBAC policies [3–5, 10, 12] there is no literature about how to use administrative RBAC policies in a distributed system. We believe that RBAC, and efficient administrative mechanisms, can be very useful in a distributed system, as in practice distributed systems may have a large number of users and objects.

In this paper we have presented a formal model for a distributed RBAC system. We have formalized a number of desiderata, motivated by safety and availability, about the policy distribution across the subsystem of the distributed system. We have defined how administrative commands can be executed in a distributed system, using an abstract form of message-passing progression between the administrative system and peripheral subsystems. We have shown that our mechanism preserves the desiderata mentioned. Our administration mechanism is efficient in the sense that it updates subsystems only about the relevant parts of the RBAC policy. Finally we have shown an easy translation of our administration mechanism into procedures for the administrative reference monitor and the reference monitors of the subsystems in the distributed system.

For the sake of simplicity we have restricted our model to the setting of a single administrative reference monitor. The soundness and completeness requirements can also be applied in the more general setting of multiple distinct administrative reference monitors. We believe that this is an interesting possibility for future research, for example to address settings where one can not designate a single administrative system.

Acknowledgements

We would like to thank J.G. Cederquist for discussing about early drafts of this paper. This research was in part funded by TNO, and Senter Novem's IOP Gencom program (*the PAW project*).

References

1. RBAC Standard, ANSI INCITS 359-2004, 2004.
2. V. Bhamidipati and R. Sandhu. Push architectures for user role assignment.
3. J. Crampton and G. Loizou. Administrative scope: A foundation for role-based administrative models. *Transactions on Information System Security (TISSEC)*, 6(2):201–231, 2003.

4. M.A.C. Dekker, J. Cederquist, J. Crampton, and S. Etalle. Extended privilege inheritance in RBAC. In *Proc. of the Symp. on Information, Computer and Communications Security (ASIACCS)*, page to be published. ACM Press, 2007.
5. D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-based Access Control*. Computer Security Series. Artech House, 2003.
6. E. Freudenthal, T. Pesin, L. Port, E. Keenan, and V. Karamcheti. dRBAC: Distributed role-based access control for dynamic coalition environments. In *Int. Conf. on Distributed Computing Systems (ICDCS)*. IEEE Computer Society Press, 2002.
7. N. Li, J. Byun, and E. Bertino. A critique of the ANSI standard on role based access control. *IEEE Security and Privacy*, page in press.
8. N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In P. Samarati, editor, *Proc. of the Conf. on Computer and Communications Security (CCS)*, pages 156–165. ACM Press, 2001.
9. R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rump session, 1996.
10. R. S. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.
11. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
12. H. Wang and S. L. Osborn. An administrative model for role graphs. In *Proc. of the IFIP TC-11 WG 11.3 Annual Working Conference on Data and Application Security (DBSec)*, pages 302–315. Kluwer, 2003.