Expressing User Access Authorization Exceptions in Conventional Role-based Access Control

Xiaofan Liu[1,2], Natasha Alechina[1], and Brian Logan[1]

[1] School of Computer Science, University of Nottingham, Nottingham, NG8 1BB, UK
[2] School of Computer and Communication, Hunan University, Hunan, 410081, P. R. China
{lxx,nza,bsl}@cs.nott.ac.uk

Abstract. In this paper we present a systematic categorization of the user access authorization exceptions which may occur in conventional role-based access control models. We propose a slightly revised NIST RBAC model which allows us to express all the authorization exceptions we consider. We give a formal definition of the model and show how it can be implemented in DATALOG with negation to give simple and efficient algorithm for computing authorization decisions. As an illustration, we present a simple case study from the domain of medical informatics and show how a range of different kinds of authorization exceptions that may arise in such a domain can be expressed in our approach.

1 Introduction

Role-based access control (RBAC) models have been advocated as a way of reducing the complexity in discretionary and mandatory access control. In role-based access control, users are assigned to roles, and roles are associated with sets of permissions. A user request for access to a particular object (resource) is *authorized* if the user is assigned to a role that has the appropriate permission for the object. To simplify the management of permissions associated with roles, such models frequently utilize a role hierarchy which allows senior roles to implicitly include all the permissions associated with junior roles in the hierarchy. Conventional RBAC models, such as RBAC96 [15] and NIST RBAC [7] (which is based on RBAC96), adopt a closed policy, that is, a user has a particular permission if one of the user's roles has the permission, otherwise the user does not have the permission. However, in many practical applications, the positive authorizations allowed by an RBAC policy admit *user access authorization exceptions*, or *authorization exceptions* for short. For example:

Example 1. A medical records system has a set of roles including "Doctor" and "Cardiologist", some of which are associated with permissions that allow a user to read a patient's records. However, patient Alice may stipulate that a particular doctor, Tom (who happens to be her brother-in-law), should not have access to her records whichever role Tom is assigned to in order to protect Alice's privacy.[3]

[3] Experience in the UK suggests that patients sometimes wish to restrict access to their health records by their relatives or particular health care workers [2].

Conventional RBAC models do not support such authorization exceptions to default access policies [2, 3, 13].

In RBAC models with role hierarchies (called *Hierarchical RBAC* in [7]), a user assigned to a role may be granted permissions both through their assigned role, and through junior roles from which their role inherits. In such models, in addition to the authorization exceptions discussed above, authorization exceptions may also result from role inheritance. Specifically, while users assigned to senior roles inherit all permissions associated with junior roles, some permissions associated with one or more junior roles should not be granted to users assigned to a senior role. For example:

Example 2. Assume that the role "IT supervisor" is senior to (and inherits permissions from) the role "IT professional". However, a permission that allows a user to alter source code associated with "IT professional" should not be inherited by "IT supervisor", because users assigned to "IT supervisor" have a background in management rather than in computer science.

Authorization exceptions in Core and Hierarchical RBAC have long been recognized, and proposals for handling some of those exceptions can be found in the literature [15, 8, 11, 2, 3, 6, 13]. However, to the best of our knowledge, there has been no systematic analysis of authorization exceptions in conventional RBAC. In this paper we provide an analysis of the range of possible authorization exceptions in conventional RBAC and propose extensions to the RBAC model that allow both core and inherited exceptions to be formalized. In addition, we show how our formalization of RBAC with exceptions can be expressed as a stratified program of recursive DATALOG with negation, giving a simple and efficient algorithm for computing authorization decisions.

The remainder of this paper is organized as follows. We briefly introduce the NIST RBAC model in Section 2. Authorization exceptions in core RBAC are explored in Section 3, and authorization exceptions existing in hierarchical RBAC are discussed in Section 4. In section 5, we introduce DATALOG with negation and show how to express our model in it. We conduct a case study in Section 6. Then, related work is shown in Section 7. Finally, we present our conclusions and indicate some potential areas of future work in Section 8.

2 NIST RBAC

The NIST RBAC model [7] (adopted as ANSI standard ANSI INCITS 359-2004) is arguably the most influential approach to RBAC. The NIST RBAC reference model is defined in terms of four model components: *Core RBAC*, *Hierarchical RBAC*, *Static Separation of Duty Relations* and *Dynamic Separation of Duty Relations*. We focus on the first two model components, Core RBAC and Hierarchical RBAC. In Core RBAC permissions are associated with roles and roles are assigned to users. Core RBAC is mandatory for all RBAC models. Hierarchical RBAC adds a role hierarchy, which defines an inheritance relation among roles, to Core RBAC. Informally, role r_1 inherits from role r_2 if users assigned to r_1 have all permissions associated with r_2. In what follows, we broadly follow the principles, definitions and reference models given in the NIST RBAC model. However, as proposed by Li et al [10] and Power et al [12], we do not consider sessions as defined in the Core RBAC model.

3 Authorization Exceptions in Core RBAC

In this section, we introduce the formal definition of the NIST Core RBAC model defined in [7]. We then define authorization exceptions in Core RBAC and show how to express these exceptions by adding new constructs into the NIST Core RBAC model definition.

3.1 NIST Core RBAC

The NIST RBAC model defines Core RBAC as follows [7]:

Definition 1. (NIST Core RBAC)

- $USERS$, $ROLES$, ACS, *and* OBS *denote sets of users, roles, actions, and objects, respectively.*
- $UA \subseteq USERS \times ROLES$ *is a many-to-many mapping user-to-role assignment relation.*
- $assigned\_users(r : ROLES) \rightarrow 2^{USERS}$ *maps a role r onto a set of users. Formally,* $assigned\_users(r) = \{u \in USERS \mid (u, r) \in UA)\}$.
- $PRMS = 2^{ACS \times OBS}$ *is the set of permissions.*
- $PA \subseteq PRMS \times ROLES$ *is a many-to-many mapping permission-to-role assignment relation.*
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}$ *maps a role r onto a set of permissions. Formally,* $assigned\_permissions(r) = \{p \in PRMS \mid (p, r) \in PA)\}$.
- $Ac(p : PRMS) \rightarrow \{ac \subseteq ACS\}$ *is the permission-to-action mapping, which gives the set of actions associated with permission p.*
- $Ob(p : PRMS) \rightarrow \{ob \subseteq OBS\}$ *is the permission-to-object mapping, which gives the set of objects associated with permission p.*

The definition of Core RBAC given in Definition 1 is somewhat redundant, as pointed out in [10]. Given $USERS$, $ROLES$, ACS, OBS, UA and PA, all the other relations and functions in the NIST definition of Core RBAC are definable. In the interests of brevity, we therefore omit the definable relations and functions. For technical reasons (the ease of expressing definitions in first-order logic and translation to DATALOG) we also "flatten" the set of permissions: instead of $PRMS = 2^{ACS \times OBS}$ we set it to be $PRMS \subseteq ACS \times OBS$.[4] This means that PA becomes a relation between a single permission tuple and a role, rather than between a set of tuples and a role.

More importantly, in the definition above, there is no relation explicitly connecting a user to a permission. Every user assigned to a role implicitly has all the permissions associated with that role. This key idea of RBAC allows a compact and transparent representation of an access control policy. However it does not allow authorization exceptions to be expressed. We therefore modify the NIST Core RBAC definition as follows: we add a predicate $AUTH$ which stands for "user is authorized". We add a condition (Core RBAC User Authorization) which states that if a user is assigned to a role, and a role is associated with a permission, then the user has (is authorized for) this permission:

$$\forall p \forall u \forall r(UA(u, r) \wedge PA(p, r) \rightarrow AUTH(p, u))$$

[4] In a given system, there may exist some action-object pairs that are not permissions.

Definition 2. (Core RBAC with User Authorization Relation)

- $USERS$, $ROLES$, ACS, OBS, UA *are as in Definition 1.*
- $PRMS \subseteq ACS \times OBS$ *is a set of permissions (a subset of the set of action-object pairs).*
- $PA \subseteq PRMS \times ROLES$ *is a many-to-many mapping permission-to-role assignment relation.*
- $AUTH \subseteq PRMS \times USERS$ *is a many-to-many permission-to-user authorization relation.*
- *The following condition (Core RBAC User Authorization) holds:*

$$(CRBAC\ C) \quad \forall p \forall u \forall r(UA(u, r) \wedge PA(p, r) \rightarrow AUTH(p, u))$$

3.2 Authorization Exceptions in Core RBAC

Previous work has focused on a single type of authorization exception in Core RBAC [2, 13], specifically, "a particular user should not be authorized for a particular permission, such as reading a patient's record, irrespective of the role the user is assigned to", as illustrated in Example 1. We refer to this kind of authorization exception as *core exceptions for all roles*. However, for some access control policies, core exceptions for all roles provide insufficient granularity. For example:

Example 3. Assume that, in addition to roles "Doctor" and "Cardiologist", a medical records system has an additional role "Accident & Emergency Doctor", and Alice is being treated in the Emergency department. Though Alice may stipulate that "Tom does not have permission to access Alice's record", it is reasonable that Tom can access Alice's record when Tom is working in the Emergency department, i.e., when Tom is assigned to the role "Accident & Emergency Doctor".

Clearly, in the example above, it is inappropriate to disallow Tom to read Alice's record for all roles. Instead, an access control policy should stipulate an authorization exception for a user when assigned to a particular role. We refer to this kind of authorization exception as *core exceptions for one role*, and we refer to core exceptions for a role and for all roles collectively as *core exceptions*.

3.3 Expressing Core Exceptions in Core RBAC

Core exceptions can be expressed using a relation $CE_{p,u,r}(p, u, r)$, denoting that u assigned to r is not authorized for p. It may seem that *core exception for all roles* should be expressed as $CE_{p,u}(p, u)$ meaning u is not authorized for p in any role. However, logically speaking, $CE_{p,u}(p, u)$ can be derived from $CE_{p,u,r}(p, u, r)$, i.e., $CE_{p,u}(p, u) =_{def} \forall r\, CE_{p,u,r}(p, u, r)$. We therefore only need to add $CE_{p,u,r}(p, u, r)$ to Definition 2 to express core exceptions as below:

1. we add $CE_{p,u,r} \subseteq PRMS \times USERS \times ROLES$, an authorization exception relation.
2. we replace Core RBAC User Authorization condition (CRBAC C) with Core RBAC User Authorization with Core Exceptions:

$$(CRBAC\ C) \quad \forall p \forall u \forall r(UA(u, r) \wedge PA(p, r) \wedge \neg CE_{p,u,r}(p, u, r) \rightarrow AUTH(p, u))$$

We can now define Core RBAC with core exceptions as follows:

Definition 3. (Core RBAC with Core Exceptions)

- *$USERS$, $ROLES$, ACS, OBS, UA, $PRMS$, PA and $AUTH$ are the same as in Definition 2.*
- *$CE_{p,u,r} \subseteq PRMS \times USERS \times ROLES$ is an authorization exception relation.*
- *The following condition (Core RBAC User Authorization with Core Exceptions) holds:*

$$(CRBAC\ C) \quad \forall p \forall u \forall r(UA(u,r) \wedge PA(p,r) \wedge \neg CE_{p,u,r}(p,u,r) \rightarrow AUTH(p,u))$$

Both examples 1 and 3 can be expressed in core RBAC with core exceptions, by stating core exceptions

$$CE_{p,u,r}(\text{'Read Alice's Record'}, Tom, \text{'Doctor'})$$
$$CE_{p,u,r}(\text{'Read Alice's Record'}, Tom, \text{'Cardiologist'})$$

The only difference between the two cases is that in example 3 we do not have the core exception

$$CE_{p,u,r}(\text{'Read Alice's Record'}, Tom, \text{'Accident \& Emergency Doctor'})$$

so if Tom is assigned to the accident and emergency doctor role, he will be authorized to read Alice's record.

4 Authorization Exceptions in Hierarchical RBAC

In the Hierarchical RBAC model, users assigned to a role have the permissions associated with the role, as well as permissions associated with all junior roles. In a hierarchical model, authorization exceptions may result not only from the role the user is assigned to, i.e., *core exceptions*, but also from role inheritance, i.e., *inheritance exceptions*.

In this section, we first briefly recall the Hierarchical RBAC model presented in [7] and discuss authorization exceptions in Hierarchical RBAC in detail. We then add some new constructs to Definition 3 to incorporate exceptions arising from role inheritance.

4.1 NIST Hierarchical RBAC

In NIST Hierarchical RBAC, roles are hierarchically organized into a role-subrole relationship called a role hierarchy. Based on the role hierarchy, role inheritance is interpreted using a graph where each node represents a role and a directed edge from role r_1 to r_2 indicates that role r_1 inherits the permissions associated with role r_2. Role inheritance is a partial order that is reflexive, transitive, and antisymmetric. Inheritance is reflexive because a role inherits its own permissions; transitive because permissions are inherited along the role hierarchy; and antisymmetry rules out cycles in the role hierarchy, that is, roles can not inherit from each other.

The NIST RBAC model distinguishes both general and limited role hierarchies. However, in the interests of generality, we consider only general role hierarchies below. NIST Hierarchical RBAC defined in [7] is introduced as follows:

Definition 4. (NIST RBAC with General Role Hierarchies)

- *$USERS$, $ROLES$, ACS, OBS, UA, $PRMS$, PA are the same as in Definition 1.*
- *$RH \subseteq ROLES \times ROLES$ is a partial order on ROLES called the inheritance relation. $RH(r_1, r_2)$ means r_1 inherits from r_2.*
- *The following condition holds: if $RH(r_1, r_2)$ then any user assigned to r_1 is a member of r_2 and every permission assigned to r_2 is assigned to r_1. More precisely:*

$$(HRBAC\,1) \quad \forall u \forall r_1 \forall r_2 (RH(r_1, r_2) \wedge UA(u, r_1) \rightarrow UA(u, r_2))$$

$$(HRBAC\,2) \quad \forall p \forall r_1 \forall r_2 (RH(r_1, r_2) \wedge PA(p, r_2) \rightarrow PA(p, r_1))$$

As in NIST Core RBAC, there is no relation explicitly connecting a user with a permission in the definition above. In order to express inheritance exceptions, we therefore modify the NIST RBAC with General Role Hierarchies definition by adding an additional condition (Hierarchical RBAC with User Authorization from Role Inheritance) as follows:

$$(HRBAC\,I) \quad \forall p \forall u \forall r_1 \forall r_2 (UA(u, r_1) \wedge RH(r_1, r_2) \wedge PA(p, r_2) \rightarrow AUTH(p, u))$$

Based on Definition 2 and Definition 4, our revised RBAC model with authorization both from a single role and role inheritance is as follows.

Definition 5. (Hierarchical RBAC with User Authorization Relation)

- *$USERS$, $ROLES$, ACS, OBS, UA, $PRMS$, PA and $AUTH$ are the same as in Definition 2.*
- *The following condition (Core RBAC with User Authorization) holds:*

$$(CRBAC\,C) \quad \forall p \forall u \forall r (UA(u, r) \wedge PA(p, r) \rightarrow AUTH(p, u))$$

- *$RH \subseteq ROLES \times ROLES$ is a partial order on $ROLES$ called the inheritance relation. $RH(r_1, r_2)$ means r_1 inherits r_2.*
- *The following conditions hold: if $RH(r_1, r_2)$ then any user assigned to r_1 is a member of r_2 and every permission assigned to r_2 is assigned to r_1. More precisely:*

$$(HRBAC\,1) \quad \forall u \forall r_1 \forall r_2 (RH(r_1, r_2) \wedge UA(u, r_1) \rightarrow UA(u, r_2))$$

$$(HRBAC\,2) \quad \forall p \forall r_1 \forall r_2 (RH(r_1, r_2) \wedge PA(p, r_2) \rightarrow PA(p, r_1))$$

- *The following condition (Hierarchical RBAC with User Authorization from Role Inheritance) holds:*

$$(HRBAC\,I) \quad \forall p \forall u \forall r_1 \forall r_2 (UA(u, r_1) \wedge RH(r_1, r_2) \wedge PA(p, r_2) \rightarrow AUTH(p, u))$$

4.2 Authorization Exceptions in Hierarchical RBAC

It is easy to see that core exceptions may exist for each role in a Hierarchical RBAC model. In addition, a Hierarchical RBAC model may also have inheritance exceptions. In this section, we explore inheritance exceptions in detail.

Assume that role r_1 is a direct descendent of role r_2 in a role hierarchy, i.e., $RH(r_1, r_2)$,[5] $P_2 = \{p_1, p_2, \ldots, p_m\}$ is a set of permissions associated with r_2 and $U_1 = \{u_1, u_2, \ldots, u_n\}$ is a set of users assigned to r_1. In hierarchical RBAC, a user from U_1 has all permissions in P_2. However, in many cases, it is necessary to specify exceptions, for example, that no user from U_1 has permission $p_k \in P_2$. We refer to such exceptions as *direct inheritance exceptions for all users* since the exception applies to two directly related roles in a role hierarchy and affects all users assigned to the senior role (see example 2).

Inheritance exceptions have been extensively studied in the access control literature [15, 8, 11, 6, 13]. As far as we know, with the exception of [13], previous work on inheritance exceptions has focused on *direct inheritance exceptions for all users*. However, we believe that in some situations it can be useful to restrict inheritance exceptions between two directly related roles to a particular user. Consider the following example:

Example 4. Assume the role "Doctor" is a direct descendant of the role "Clinician" and hence *RH("Doctor", "Clinician")*. The permission "reading Alice's record" is associated with the role "Clinician" and George is assigned to the role "Doctor". Alice stipulates that George (who is a friend of Alice) should not be permitted to read Alice's record, while other users assigned to the role "Doctor" can.

We refer to such inheritance exceptions as *direct inheritance exceptions for one user*, as the exception applies to two directly related roles in a role hierarchy and affects only a particular user assigned to the senior role.

In a role hierarchy, an inheritance exception may apply not only between a role and its direct descendant, but also between a role and one of its indirect descendants. For example:

Example 5. Assume "Nurse in emergency department" is a direct descendant of "Nurse" and "Nurse" is a direct descendant of "Clinician", hence "Nurse in emergency department" is an indirect descendant of "Clinician". It may be regulated that users assigned to the role "Clinician" are authorized for a permission p like "updating patient's record" and users assigned to the role "Nurse in emergency department" can not be authorized for p while users assigned to the role "Nurse" can.

We can formulate the meaning of such inheritance exceptions as follows. Assume R is a set of roles, $r_1, \ldots, r_i, \ldots, r_j, \ldots, r_k, \ldots, r_n$ is a sequence of directly related roles in a role hierarchy defined by a partial order RH over R, and P_k is a set of permissions associated with r_k. By default, users assigned to r_i are also authorized for P_k. However, assume that a particular permission $p \in P_k$ should not be inherited by r_i. We refer to such inheritance exceptions as *indirect inheritance exceptions*, as they apply to two indirectly related roles. As in the case of direct inheritance exceptions, we distinguish two

[5] r_1 is a direct descendent of r_2 if there exists no role r_3 ($r_3 \neq r_1$, $r_3 \neq r_2$) in the role hierarchy such that $RH(r_1, r_3)$ and $RH(r_3, r_2)$

types of indirect inheritance exceptions: *indirect inheritance exceptions for all users* and *indirect inheritance exceptions for one user*. As with direct inheritance exceptions for a user, inheritance exceptions between two indirectly related roles have not been studied in the literature.[6]

4.3 Expressing Authorization Exceptions in Hierarchical RBAC

We observe that inheritance exceptions, either for all users or a particular user, can be denoted by a relation $IE_{p,u,r}(p, u, r)$ regardless of whether they are indirect or direct. It may seem that since inheritance happens between two roles, inheritance exception should be expressed as $IE_{p,u,r,r'}(p, u, r, r')$, which means u assigned to r is not authorized for a permission p which is assigned to r'. However, this would stop u being authorized for p in the direct case, but not in the indirect case, since u's role r will inherit p from the roles between r and r' in the role hierarchy. We could specify exceptions $IE_{p,u,r,r'}(p, u, r, r'')$ for all such r'', but in situations where many roles are involved, it is undesirable to have to explicitly state this for each role which inherits from r'. Thus, we simply use $IE_{p,u,r}$ to prevent a user assigned to r from inheriting p from any roles. $IE_{p,r}(p, r)$ denoting that any user assigned to r is not authorized for p is just a special case of $IE_{p,u,r}(p, u, r)$. Logically speaking, $IE_{p,r}(p, r) =_{def} \forall u IE_{p,u,r}(p, u, r)$.

Further, we adopt a single authorization exception relation $EXP_{p,u,r}(p, u, r)$ to replace both $IE_{p,u,r}(p, u, r)$ and $CE_{p,u,r}(p, u, r)$. Otherwise in order to prevent a user u from being authorized for p we may have to state both $IE_{p,u,r}(p, u, r)$ and $CE_{p,u,r}(p, u, r)$. For example, assume that $UA(u, r)$ and $PA(p, r)$, and we wish to prevent u from being authorized for p. If we state $CE_{p,u,r}(p, u, r)$, u will not be authorized for p by the Core RBAC User Authorization with Core Exception condition. However, the model would need to have a condition for authorization which stems from role inheritance, along the lines of

$$\forall p \forall r_1 \forall r_2 (UA(u, r_1) \wedge RH(r_1, r_2) \wedge PA(p, r_2) \wedge \neg IE_{p,u,r}(p, u, r) \rightarrow AUTH(p, u))$$

In our example, if we do not state explicitly that $IE_{p,u,r}(p, u, r)$ also holds, we get

$$UA(u, r) \wedge RH(r, r) \wedge PA(p, r) \wedge \neg IE_{p,u,r}(p, u, r) \rightarrow AUTH(p, u)$$

and since the antecedent holds by the reflexivity of RH, $AUTH(p, u)$ will be derived.

In order to express inheritance exceptions, we believe it is necessary to abandon condition HRBAC 1 of Definition 5.[7] The reason why HRBAC 1 is problematic is as follows.

Assume R is a set of roles, r_1,. . .,r_i, . . ., r_n is a sequence of directly related roles such that $RH(r_i, r_{i+1})$, r_n has permission p, and hence all roles from r_1 to r_n also have

[6] An exception is [13], in which authors discuss "nesting access policy statements", which is just a different expression of inheritance exception between two indirectly related roles.

[7] Note that although the NIST RBAC model adopts the most widely used definition of role hierarchy, many researchers agree that alternative interpretations may be appropriate in particular circumstances [14]. For example, [10] argues convincingly that only one of HRBAC 1 or HRBAC 2 should be used to define role inheritance.

p by HRBAC 2. Assume that u assigned to r_i should not be authorized for p. Such an exception could be expressed as $EXP_{p,u,r}(p, u, r_i)$. However, by (HRBAC 1), u could still get the permission from the roles r_{i+1} to r_n because u is implicitly assigned to $r_{i+1} \ldots, r_n$ by HRBAC 1. An alternative approach would be to retain condition HRBAC 1, and instead add exception statements $EXP(p, u, r')$ for all roles r' such that $PA(p, r')$ and $RH(r, r')$ (in other words, all roles r' which have permission p and to which u is implicitly assigned by HRBAC 1); however this may involve adding a large number of additional exceptions.

We now give our full definition of Hierarchical RBAC with authorization exceptions including core exceptions and inheritance exceptions.[8]

Definition 6. (Hierarchical RBAC with Unified Authorization Exceptions)

- *$USERS$, $ROLES$, ACS, OBS, UA, $PRMS$, PA and $AUTH$ are the same as in Definition 2.*
- *$RH \subseteq ROLES \times ROLES$ is a partial order on $ROLES$ called the inheritance relation. $RH(r_1, r_2)$ means r_1 inherits r_2.*
- *The following condition holds: if $RH(r_1, r_2)$ then every permission assigned to r_2 is assigned to r_1. More precisely:*

$$(HRBAC) \quad \forall p \forall r_1 \forall r_2 (RH(r_1, r_2) \wedge PA(p, r_2) \rightarrow PA(p, r_1))$$

- *$EXP_{p,u,r} \subseteq PRMS \times USER \times ROLES$.*
- *The following condition (Hierarchical RBAC with Unified Authorization Exceptions) holds:*

$$(HRBAC\,EXP) \quad \forall p \forall u \forall r (UA(u, r) \wedge PA(p, r) \wedge \neg EXP_{p,u,r}(p, u, r) \rightarrow AUTH(p, u))$$

5 Expressing HRBAC with Exceptions in DATALOG

In this section we show how Hierarchical RBAC with Unified Authorization Exceptions can be expressed as a stratified program of recursive DATALOG with negation. In fact, various extensions of DATALOG have been widely adopted in access control field because of their easy-to-read syntax and precise semantics [9]. For example, in [4], Bertino et al. show how to express access control models in D-DATALOG program. Formulating our model in DATALOG immediately gives us a simple and efficient algorithm for computing authorization decisions because DATALOG already has efficient query evaluation algorithm [1].

We first briefly introduce recursive DATALOG with negation programs and stratified programs from [1]. Then we show how to express Hierarchical RBAC with Unified Authorization Exceptions as a stratified program.

[8] For simplicity, we leave a single reflexive and transitive inheritance relation *RH* in our definition, but in our DATALOG implementation, we use a direct inheritance relation *DRH* instead and define *RH* as its reflexive transitive closure, again as suggested in [10].

5.1 Backgroud

Definition 7. *A* recursive DATALOG with negation program *is a finite set of* rules *of the form*

$$R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n)$$

where

- *An* atom *is a n-ary-predicate with n terms. A* literal *is an atom or negated atom.*
- $n \geq 1$, $R_1, \ldots, R_n$ *are literal names and* $u_1, \ldots, u_n$ *are free tuples of appropriate arities.*
- *Each variable occurring in* u_1 *must occur in at least one of* $u_2, \ldots, u_n$.
- $R_1(u_1)$ *is called the* head *of the rule and* $R_2(u_2), \ldots, R_n(u_n)$ *forms the* body.
- *A rule without a body is called a* fact.

Next we introduce the concept of a *stratified* program. Unlike arbitrary recursive DATALOG with negation programs, stratified programs have a well-behaved semantics (a unique minimal model where all the consequences the program are true). Let P be a DATALOG program. A predicate appearing only in the body of a rule is referred to as an *extensional* predicate, while an *intentional* predicate is a predicate occurring in the head of a rule. The *extensional schema*, referred to as $edb(P)$, consists of the set of all extensional predicate names, whereas *intensional schema*, denoted as $idb(P)$, consists of all the intensional ones. The union of $edb(P)$ and $idb(P)$ is called the *schema* of P which is denoted as $sch(P)$. The *semantics* of a DATALOG program is a mapping from database instances over $edb(P)$ to database instances over $idb(P)$.

Now consider a program P in which *idb* predicates are defined by one or more rules of P and negation applies to predicates, such as R, appearing both in the body and the head of a rule, i.e., $R \in idb(P)$. Then program P could be considered as consisting of several parts. Specifically, for each *idb* predicate R', if the part of P defining R' comes before the negation of R' is used, we can simply compute R' before its negation must be evaluated. Such a way of treating P is called a *stratification* of P and is precisely defined as follows.

Definition 8. *A stratification of a recursive* DATALOG *with negation program* P *is a sequence of* DATALOG *with negation programs* $P^1, \ldots, P^n$ *such that for some mapping* σ *from* $idb(P)$ *to* $[1..n]$.

- $\{P^1, \ldots, P^n\}$ *is a partition of* P.
- *For each predicate* R, *all the rules in* P *defining* R *are in* $P^{\sigma(R)}$ *(i.e., in the same program of the partition)*
- *If* $R(u) \leftarrow \ldots R'(v) \ldots$ *is a rule in* P, *and* R' *is an idb predicate, then* $\sigma(R') \leq \sigma(R)$.
- *If* $R(u) \leftarrow \ldots \neg R'(v) \ldots$ *is a rule in* P, *and* R' *is an idb predicate, then* $\sigma(R') < \sigma(R)$.

Given a stratification $P^1, \ldots, P^n$ of P, each P^i is called a *stratum* of the stratification, and σ is called the *stratification mapping*. Not all programs are stratifiable [1]. However it is straightforward to determine whether a program is stratifiable. Specifically, let P be a DATALOG with negation program. The *precedence graph* G_P of P is the labeled graph whose nodes are the *idb* relations of P. Its edges are the following:

- if $R(u) \leftarrow \ldots R'(v) \ldots$ is a rule in P, then $< R', R >$ is an edge in G_P with label + (called a positive edge).
- if $R(u) \leftarrow \ldots \neg R'(v) \ldots$ is a rule in P, then $< R', R >$ is an edge in G_P with label - (called a negative edge).

Proposition 1. *A recursive* DATALOG *with negation program P is stratifiable iff its precedence graph G_P has no cycle containing a negative edge.*

A proof is given in [1, p.380].

5.2 Result

To express Definition 6 in DATALOG with negation we need the following *edb* predicates: UA for user-role assignment, DPA (for direct assignment of permissions to roles), DRH (for direct inheritance relation), EXP (for exceptions), which will be used to state the facts concerning user assignment etc.[9] We also need *idb* predicates RH, PA and $AUTH$ defined as follows:

$$
\begin{array}{ll}
r_1 & RH(r_1, r_2) \leftarrow DRH(r_1, r_2) \\
r_2 & RH(r_1, r_2) \leftarrow DRH(r_1, r_3), RH(r_3, r_2) \\
r_3 & PA(a, o, r) \leftarrow DPA(a, o, r) \\
r_4 & PA(a, o, r_1) \leftarrow DPA(a, o, r_2), RH(r_1, r_2) \\
r_5 & AUTH(a, o, u) \leftarrow PA(a, o, r), UA(u, r), \neg EXP(a, o, u, r)
\end{array}
$$

Now, we can easily prove that the program implementing Hierarchical RBAC with exceptions is stratifiable.

Proposition 2. *The program implementing Hierarchical RBAC with exceptions is stratifiable.*

Proof. It is easy to see that there is no cycle containing a negative edge in the precedence graph. The only negative edge is from EXP to $AUTH$, and there is no edge from $AUTH$ for any predicate.

6 Case Study

We illustrate our approach using a simple medical informatics case study. The permissions are taken from the appendix of [5], which defines a vocabulary for permissions in healthcare information systems. We used the DATALOG Educational System (DES), a free PROLOG-based implementation of a basic deductive database system.[10]

In our case study, role "nurse in emergency department" inherits from role "nurse" and role "nurse" inherits from role "clinician". In the DATALOG program, the direct inheritance relation is encoded as follows:

[9] We use pairs (action, object) to stand for permissions, and omit a permission relation $P(a, o)$ for brevity.

[10] For more details of DES, see http://www.fdi.ucm.es/profesor/fernan/DES/index.html

```
drh(nurse,clinician).
drh(nurse_in_emergency_department,nurse).
```

We assume three users, Jessica who is a nurse in the Emergency Department, and Kate and Ellen who are nurses. This corresponds to the following user-role assignment:

```
ua(jessica,nurse_in_emergency_department).
ua(kate,nurse).
ua(ellen,nurse).
```

The direct role-permission assignment is given below. Recall that the first two arguments constitute a permission, namely an action and an object (patient in this case), and the third argument is a role:

```
dpa(read_patient_test_report,alice,clinician).
dpa(read_patient_test_report,sherry,clinician).
dpa(read_patient_test_report,mina,clinician).
dpa(read_patient_test_report,katherine,clinician).

dpa(sign_history_and_physical,alice,clinician).
dpa(sign_history_and_physical,sherry,clinician).
dpa(sign_history_and_physical,mina,clinician).
dpa(sign_history_and_physical,katherine,clinician).

dpa(create_history_and_physical,alice,clinician).
dpa(create_history_and_physical,sherry,clinician).
dpa(create_history_and_physical,mina,clinician).
dpa(create_history_and_physical,katherine,clinician).

dpa(update_progress_note,alice,nurse).
dpa(update_progress_note,sherry,nurse).
dpa(update_progress_note,mina,nurse).
dpa(update_progress_note,katherine,nurse).

dpa(append_progress_note,alice,nurse_in_emergency_department).
dpa(append_progress_note,sherry,nurse_in_emergency_department).
dpa(append_progress_note,mina,nurse_in_emergency_department).
dpa(append_progress_note,katherine,nurse_in_emergency_department).
```

We also assume the existence of the following authorization exceptions: users assigned to the role of nurse should not be authorized to sign a patient's history and physical. In addition, Kate should not be authorized to read patient test report of Alice. In DATALOG:

```
exp(sign_history_and_physical,alice,kate,nurse).
exp(sign_history_and_physical,sherry,kate,nurse).
exp(sign_history_and_physical,mina,kate,nurse).
```

```
exp(sign_history_and_physical,katherine,kate,nurse).

exp(sign_history_and_physical,alice,ellen,nurse).
exp(sign_history_and_physical,sherry,ellen,nurse).
exp(sign_history_and_physical,mina,ellen,nurse).
exp(sign_history_and_physical,katherine,ellen,nurse).

exp(read_patient_test_report,alice,kate,nurse).
```

The remaining predicates are defined as in the previous section: [11]

```
rh(R1,R2) :- drh(R1,R2).
rh(R1,R2) :- drh(R1,R3), rh(R3,R2).
pa(A,O,R1) :- dpa(A,O,R1).
pa(A,O,R1) :- dpa(A,O,R2), rh(R1,R2).
auth(A,O,U) :- pa(A,O,R), ua(R,U), not(exp(A,O,U,R)).
```

DATALOG computes 43 tuples for the `auth` predicate, 48 tuples for the `pa` predicate and 3 for the `rh` predicate. Note that even in this small example, the advantage of using rules to compute the role-permission relation rather than storing it explicitly is obvious: the size of the program is much smaller and it is easier to maintain (integrity is ensured by the rules). Note that if we did not use exceptions, we would need to eliminate the inheritance between Clinician and Nurse and introduce a special role for Kate which has fewer permissions than Nurse. In the worst case, when every role, and every user assigned to a role, is involved in some exception, Hierarchical RBAC would collapse to a non-hierarchical model, losing the advantages of a compact and easy to maintain formalization.

7 Related Work

Exceptions, or preventing particular users from performing actions which their role would normally authorize them to perform, have been extensively studied in the liter-

[11] The implementation given above is not very efficient — we used it as an illustration as it closely corresponds to the model. For a more afficient and compact representation, we could introduce a predicate `dpa1` which means 'permitted to perform an action' (omitting the object) if a role is given a permission to perform a given action on all possible objects, for example `dpa1(read_patient_test_report, clinician)`. Similarly, we could introduce a predicate `exp1` to say that a user is not authorized to perform a given action on any object, for example `exp1(sign_history_and_physical, kate, nurse)`. We would also need two versions of `pa`, one of the form `pa(Action, Object, Role)` and another `pa1(Action, Role)`, with the obvious definitions. Finally, we need two definitions of `auth`:

```
auth(A,O,U) :-
    pa(A,O,R), ua(U,R), not(exp(A,O,U,R)), not(exp1(A,U,R)).
auth(A,O,U) :-
    pa1(A,R), ua(U,R), p(A,O),
    not(exp(A,O,U,R)), not(exp1(A,U,R)).
```

where `p` is a permission predicate.

ature, see for example [15, 2, 3, 13]. Bacon et.al. address what we call *core exception for all roles* in [3] and propose supplementing an RBAC policy with an exception list in the OASIS model (which is based on NIST RBAC model). Our approach is similar to theirs in that we adopt an exception table (EXP predicate). They do not deal with the hierarchical exceptions.

In [15], Sandhu et.al. propose extending the RBAC96 model with "private roles" which allow *direct inheritance exceptions for all users*. Specifically, assume role r_1 inherits permissions from role r_2 which is associated with a set of permissions P including p. To let r_1 inherit only the set $P \setminus \{p\}$, a private role r_2' which has permission p can be introduced. r_1 inherits permissions from r_2, which is now assigned P $\setminus\{p\}$. In this way, *direct inheritance exception for all users* is solved. However, *indirect inheritance exception* can not be handled by simply adding a "private role".

In [15], Sandhu et.al. also mention that, in some systems, certain permissions are blocked to be inherited from any roles. A typical example is a RBAC96-based model with "oriented" permissions proposed by Crampton in [6]. Namely, a permission p assigned to role r can be barred from being inherited by any role which is above r in the hierarchy. It is easy to see that *direct inheritance exception for all users* could be solved in such scheme, because no user of the role r' which is a direct descendant of r will be authorized for p. However, *indirect inheritance exception* and *direct inheritance exception for one user* can not be handled this way. Indirect inheritance exception for some indirect descendant r'' of r is not handled properly because although p is not inherited by r'', it is also not inherited by *any* roles r between r'' and r in the role hierarchy. Similarly, we cannot use this mechanism to bar just one user u of a more senior role from being authorized for p (unless we create a private role just for u).

Reid et.al. [13] propose a modified NIST RBAC model to express so-called "nesting access policy statements" which are equivalent to what we call *indirect inheritance exceptions*. They present an authorization algorithm for the modified model. Their scheme can handle several types of authorization exceptions. However, *core exceptions* can not be expressed in their approach. Our approach, on the other hand, handles all types of authorization exceptions, and arguably has a simpler way of computing authorization decisions.

8 Conclusion and Future Work

Though authorization exceptions, which are of practical importance in medical informatics, have been discussed in the literature, to the best of our knowledge there has been no proposal for a uniform treatment of different types of exceptions. To address this problem, we propose a systematic classification of user authorization exceptions. We incorporate these exceptions in the RBAC model and show how to express them in DATALOG with negation.

In future work, we plan to extend our work to dynamic access control (incorporating temporal constraints). We also believe that exceptions will be useful in role mining (extracting roles from Access Control Lists) since this process usually produces too many too specifically defined roles instead of more "natural" roles precisely due to the

existence of a small number of exceptions for each natural role. We plan to look at modifying role mining procedures to produce roles with specified sets of exceptions.

9 Acknowledgement

We would like to thank Jason Crampton for his suggestions and comments on earlier version of this article, Bernd Blobel for directing us to the documents on RBAC in medical informatics.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, November 1994.
2. Jean Bacon, Michael Lloyd, and Ken Moody. Translating role-based access control policy within context. In *POLICY*, pages 107–119, 2001.
3. Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4):492–540, 2002.
4. Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, 2003.
5. HL7 Security Technical Committee. *Role Based Access Control (RBAC) Healthcare Permission Catalog*. HL7 Security Technical Committee, January 2010.
6. Jason Crampton. On permissions, inheritance and role hierarchies. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 85–92, 2003.
7. David F. Ferraiolo, Ravi S. Sandhu, Serban I. Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4:224–274, 2001.
8. Cheh Goh and Adrian Baldwin. Towards a more complete model of role. In *ACM Workshop on Role-Based Access Control*, pages 55–62, 1998.
9. Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. *ACM Transactions on Information and System Security*, 11:21:1–21:41, July 2008.
10. Ninghui Li, Ji-Won Byun, and Elisa Bertino. A critique of the ANSI standard on role-based access control. *IEEE Security & Privacy*, 5(6):41–49, 2007.
11. Jonathan D. Moffett and Emil Lupu. The uses of role hierarchies in access control. In *ACM Workshop on Role-Based Access Control*, pages 153–160, 1999.
12. David J. Power, Mark Slaymaker, and Andrew C. Simpson. On formalizing and normalizing role-based access control systems. *Computer Journal*, 52(3):305–325, 2009.
13. Jason Reid, Ian Cheong, Matthew Henricksen, and Jason Smith. A novel use of RBAC to protect privacy in distributed health care information systems. In *ACISP*, pages 403–415, 2003.
14. R. Sandhu, M. Bellare, and R. Ganesan. Password-enabled pki: Virtual smart cards versus virtual soft tokens. In *PKI Research Workshop*, April 2002.
15. 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.