Noname manuscript No.
(will be inserted by the editor)

A Relational Database Integrity Framework for Access Control Policies

Romuald Thion · Stéphane Coulondre

the date of receipt and acceptance should be inserted later

Abstract Access control is one of the most common and versatile mechanisms used for information systems security enforcement. An access control model formally describes how to decide whether an access request should be granted or denied. Since the role-based access control initiative has been proposed in the 90s, several access control models have been studied in the literature.

An access control policy is an instance of a model. It defines the set of basic facts used in the decision process. Policies must satisfy a set of constraints defined in the model, which reflect some high level organization requirements. First-order logic has been advocated for some time as a suitable framework for access control models. Many frameworks have been proposed, focusing mainly on expressing complex access control models. However, though formally expressed, constraints are not defined in a unified language that could lead to some well-founded and generic enforcement procedures.

Therefore, we make a clear distinction by proposing a logical framework focusing primarily on constraints, while keeping as much as possible a unified way of expressing constraints, policies, models, and reference monitors. This framework is closely tied to relational database integrity models. We then show how to use well-founded procedures in order to enforce and check constraints. Without requiring any rewriting previous to the inference process, these procedures provide clean and intuitive debugging traces for administrators. This approach is a step toward bridging the gap between general but hard to maintain formalisms and effective but insufficiently general ones.

R. Thion
Université de Lyon,
Université Lyon 1, LIRIS, UMR5205, F-69622, France
Tel. (+33) 472 447 936
Fax. (+33) 472 431 536
E-mail: romuald.thion@univ-lyon1.fr

S. Coulondre (Corresponding Author)
Université de Lyon,
INSA-Lyon, LIRIS, UMR5205, F-69621, France
Tel. (+33) 472 437 055
Fax. (+33) 472 438 713
E-mail: stephane.coulondre@insa-lyon.fr

Keywords Integrity Model · Dependencies · Authorization · Access Control · Inference

1 Introduction

1.1 Access control models, policies and constraints

Security policies are sets of laws and rules governing the security of organizations. They can cover areas from internal organizations rules to national laws, from structural (e.g., fire protection) to organizational aspects (e.g., emergency phone lines). An Access Control (AC) (or authorization) policy is a specialized form of security policy, dedicated to permission management. AC aims at enforcing confidentiality and data integrity.

Within information systems, an AC policy is structured according to an AC *model*, which formally describes the structure of the policy. A model defines how to decide whether an access request (i.e., an action on an object issued by a subject pertaining to a user) should be granted or denied using a set of rules. For instance, in the Role-Based Access Control (RBAC) models family, roles are assigned to users, and permissions are assigned to roles (Sandhu et al., 1996; Ferraiolo et al., 2003). An RBAC policy is a set of assignments between users and roles and between roles and permissions. The *core rule* of the RBAC models family states that an access request is granted *if and only if* the issuer endorses a role with this privilege.

Since the RBAC initiative, several models have been studied in the literature. These models have extended RBAC (e.g., *Generalized-Temporal*-RBAC (Joshi et al., 2005) or *Geographical*-RBAC (Damiani et al., 2007)), and have organized policies by additional concepts to enhance their expressive power and flexibility (e.g., Workflow-RBAC (Wainer et al., 2003, 2007), *Team*-BAC (Thomas, 1997), *Task*-BAC (Thomas and Sandhu, 1997), *Organization*-BAC (Miège, 2005)). Throughout these propositions, First-Order Logic (FOL) has been advocated as a general framework suitable to formalize AC models and policies.

In addition to innovative concepts and relations (e.g., roles and hierarchies) for organizing policies, AC models have integrated the concept of *constraints*. Constraints reflect some high level organization requirements that must be enforced within policies. With the development of AC models, several kinds of constraints have been defined. The most prominent one is the *mutual exclusion*, which has been proposed in order to enforce separation of duties (Li et al., 2004). Other kinds of constraints have been defined: some mutual exclusion variants, prerequisite constraints or constraints over hierarchies (Crampton, 2003; Jaeger and Tidswell, 2001).

Actually, constraints may express different requirements on policies. Generally speaking, constraints are policy properties that can be classified in the following basic classes:

- (conditional) absence of values (e.g., mutual exclusion),
- (conditional) existence of (constrained) values (e.g., prerequisite),
- (conditional) uniqueness of values (e.g., uniqueness of ancestor in a hierarchy).

These different classes of constraints share a common objective, which is to restrict the set of policies expressible over a model to the set of consistent ones. For instance, in the RBAC setting, the definition of roles a and b as mutually exclusive means that *no user* should be assigned to a and b. Thus, the set of expressible policies is restricted to the set of policies in which no user is assigned to both a and b.

1.2 Problem statement and contribution

Constraints attempt to ensure that policies are consistent. Verifying the consistency of policies is a paramount task. Indeed an inconsistent policy can lead to an unexpected behaviour of the AC system that may completely ruin the benefits of the AC system:

- illegitimate access: exclusion constraints aim at ensuring that some activities are carried out by different users, for instance to prevent that the same user from initiating a payment and authorizing a payment. If a policy does not satisfy the exclusion constraints, one unique user may be able to circumvent the separation of duties and gain unlegitimate access,
- deny of legitimate access: conversely, a constraint can express that every user has the right to log into the system. Failing to verify such a constraint may lead to users to be unable to do anything,
- difference between expected behaviour and real behaviour: some constraints are implicitly expected from a model. For instance, when dealing with a role hierarchy, it is relevant and quite natural to prevent graph cycles. Failing to verify such constraint may lead to circular inheritance that implies that all the roles have the same set of permissions. Moreover, in a RBAC setting, such a set would be the union of all permissions assigned to at least one role in the cycle,
- unpredictable behaviour: in the worst case, if core constraints like basic integrity requirements of policy (e.g., uniqueness of user related to a subject) are not satisfied, the whole system may be compromised.

In order to express these constraints in an homogeneous way, a formal language able to handle broad classes of constraints is necessary. It should allow the definition of *new classes* of constraints and should be able to capture general integrity requirements of AC models. This language must have clear semantics, and provide well-founded automated proof procedures for consistency checking.

To address these issues we adopt a top-down approach, starting with a framework focusing primarily on constraints. This framework relies on database integrity theory. We have chosen to present it in a logical setting, but other formalisms, such as *tableaux*, could have been used. AC models and policies are then formally defined using one of the Datalog languages, which is also a fragment of the database integrity language. This interesting property allows for expressing the whole AC system, including models, policies and constraints, in a single and homogeneous framework. AC models foundations and semantics are described in section 2. Without being universal, the obtained framework can still express several classical models and extensions found in the literature.

We then define AC constraints in section 3. We make use of *relational data dependencies*, in order to model AC constraints. Dependencies are able to capture complex integrity requirements in an homogeneous way. For instance, one of the properties considered as fundamental in the RBAC standard (Ferraiolo et al., 2003, property 3.2, p.60) (this property is quoted in section 3.5) can be modelled by means of data dependencies. To the best of our knowledge, the property (3.2) has not been modelled and taken into account in any other logical framework for AC.

In section 4, we define a set of well-founded operations that can be used to help AC models designers and policy administrators in making constraints design and administration easier. These operations are generic and can be used over any model built upon the structure defined in sections 2 and 3.

We have implemented the framework and validated our approach with automated formal proofs based on previous results in the literature that had been manually proved. These results are presented in section 4.6. Moreover, the proofs obtained are quite readable as no prior rewriting is performed (such as clausal form), for it may obfuscate human analysis.

Our approach tries to encompasses many concepts found in the AC literature, but it does not take into account some peculiarities of very dedicated models. For instance, our framework does not encompass authentication (Jim, 2001) and delegation (Li et al., 2003; Wainer et al., 2007). Section 6 evaluates the major design decisions of the framework. The very last section concludes this paper and gives the main directions for future work. For sake of clarity and applicability, most of the examples of in this paper are based upon the RBAC model and further extensions.

2 Access Control Framework

This section defines an access control framework able to express several classical models and extensions found in the literature. As we will see in the next section, this framework is a subset of the general framework proposed for access control constraints, that relies on database integrity theory.

Without lack of generality, we operate a clear distinction between *models* and *policies*: AC models defines structures and AC policies are instances of these structures. In such a perspective, AC *design* is the task of defining AC models, whereas *administration* of AC is the task of defining policies, which is up to administrators.

FOL has been advocated a suitable formal framework to formalize AC models and policies (Li et al., 2003; Jim, 2001; DeTreville, 2002; Bertino et al., 2003; Li and Mitchell, 2003; Barker and Stuckey, 2003; Halpern and Weissman, 2003; Miège, 2005). From the logical point of view, an AC model both define :

- a *vocabulary*, i.e., the set of sorts and relations between sorts used to organize access privileges (e.g, subjects, roles, permissions, assignments of roles to subjects)
- a set of so-called *rules* expressed in a first-order language built over the vocabulary, i.e., the policy that states how privileges are derived from base concepts and relations (e.g., a subject is granted some access only if it endorses some role with the corresponding access right).

Section 2.1 defines the vocabulary and states of an AC model. Section 2.2 defines rules and policies. The generic definition of an AC model is summarized in section 2.3.

2.1 Access control vocabulary and state

An AC model relies on *vocabulary*: a set of *sorts* and *relations* between sorts. Sorts are the main concepts used to organize rights (e.g., users and roles...). Relations define how sorts are related in the model (e.g., assignments between users and roles). The goal of an AC system is to determine whether an access (an action on an object) issued by a subject that represents a user is granted or not. Thus, the main sorts of subjects, users, actions and objects have to be defined in any AC model.

Sorts partition the set of constants in a policy. This property is tied to the *many-sorted* FOL framework. It is shown that many-sorted FOL can be reduced to one-sorted FOL (i.e., classical logic) by assigning a specific unary predicate symbol D_S

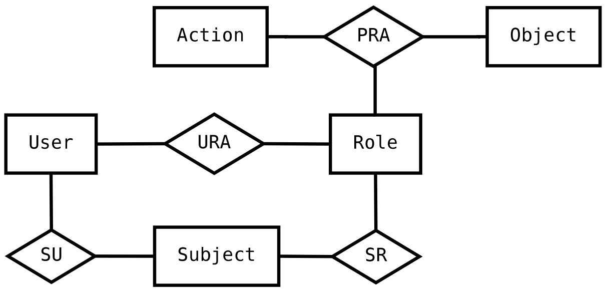

Fig. 1 The core vocabulary *edb* of Rbac.

called *domain predicate symbols* to each sort S (Gallier, 1986, chapter 10, p. 460). In our framework, we implicitly operate this transformation by assigning a unique unary predicate symbol to each sort. The vocabulary is represented in the *relational formalism* according to the standard terminology used in databases.

Definition 1 *Access control vocabulary. The* vocabulary *Voc of an* ac *model is the union of a set Sorts of* unary predicate symbols *called* sorts *and a set Rels of n-*ary predicate symbols *called* relations.
The sorts of users *(User),* subjects *(Subject),* actions *(Action) and* objects *(Object) must be present in any* ac *vocabulary.*

Figure 1 illustrates the core vocabulary of Rbac models. This vocabulary is composed of five sorts (drawn by rectangles in figure 1): *User*, *Subject*, *Role*, *Action* and *Object*. Moreover, four relations are defined (drawn by diamonds in figure 1): *URA* betwen *User* and *Role*, *PRA* between *Role*, *Action* and *Object*, *SU* between *Subject* and *User* and *SR* between *Subject* and *Role*.

We define the *state* of an ac model, which is a set of facts defined over the core vocabulary *edb*. The state of an ac model is the extensive part of a policy, which can practically be stored in a Relational Database Management System (rdbms). Following traditional axioms of logical interpretation of relational databases, we assume that constants are distinct and that states are finite.

Definition 2 *Access control state. To each sort $S \in$ Sorts is associated a set of constants* $\mathbf{S}$ *called its* domain. *Domains are pairwise disjoint:* $\mathbf{S} \cap \mathbf{S}' = \emptyset$ *for all* $S \neq S'$ *in Sorts. To each relation* $R \in$ *Rels of arity n between sorts* $S_1 \dots S_n$*, is associated the set* $\mathbf{R} = \mathbf{S}_1 \times \dots \times \mathbf{S}_n$.
An access control state $\mathbf{I}$ *on an* ac *vocabulary Voc is a mapping from each sort* $S \in$ *Sorts to a finite subset of* $\mathbf{S}$*, called its* active domain*, and from each relation* $R \in$ *Rels to a finite subset of* $\mathbf{R}$.

In the context of Rbac, the term ac *state* (a.k.a. Rbac *database*) has first been coined by (Gavrila and Barkley, 1998). A toy sample of an Rbac state $\mathbf{I}$ (over vocabulary shown in figure 1) is given in table 1. In this state, the sort *User* takes its values from the set {Alice, Bob, Charly}, *Role* from $\{\mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_3, \mathtt{r}_4\}$, *Action* from {r, w, x} and *Object* from {file1, file2, file3}

In this state, Bob is assigned both roles $\mathtt{r}_1$ and $\mathtt{r}_3$ and Bob endorses these roles in two different sessions, namely S2 and S3. The rules of Rbac models allow inferring that Bob can read file1, file2 and file3 as the role $\mathtt{r}_1$ is granted r access on file1, file2 and file3.

URA

User	Role
Alice	r_1
Alice	r_2
Bob	r_1
Bob	r_3
Charly	r_1
Charly	r_4

SR

Subject	Role
S1	r_1
S1	r_2
S2	r_1
S3	r_3
S4	r_4

PRA

Role	Action	Object
r_1	r	file1
r_1	r	file2
r_1	r	file3
r_2	w	file1
r_3	w	file2
r_3	r	file4
r_3	w	file4
r_3	x	file4
r_4	w	file3
r_4	r	file4
r_4	w	file4
r_4	x	file4

SU

Subject	User
S1	Alice
S2	Bob
S3	Bob
S4	Charly

Table 1 A sample RBAC state.

2.2 Access control rules and policies

In this section we define the AC *rules*. Rules express the *deductive principles* of an AC model. From a set of rules P and a state $\mathbf{I}$, it is possible to compute derived relations on which the access decision process is based. The state is the minimal knowledge from which a complete policy may be derived using the set of rules.

For instance, in order to prevent administrators from inserting redundancies in a policy, several algebraic properties of relations are commonly assumed: transitivity, reflexivity or symmetry for instance. These properties are expressed *in intenso* by means of rules. Thus, administators only have to insert the minimal knowledge in the state. The complete policy is obtained by applying the set of rules P as a closure operator producing new facts.

In the case of complex AC models, defining derived relations within the model is a mandatory prerequisite in order to obtain a maintainable state. This separation between extensive relations in the state and intensive relations in the policy is a design choice made according to the recommendations addressed to the RBAC standard (Li et al., 2007), in particular the *third suggestion*:

> **Suggestion 3**: *"The standard should make a clear distinction between base relations and derived relations."*

Several fragments of FOL have been used as formal languages for modelling AC rules. DATALOG-based models are considered expressive enough to capture complex AC policies (Bertino et al., 2003), (Miège, 2005), (Jim, 2001), (Barker and Stuckey, 2003). We focused our work on FOL rules of DATALOG^C, which has been recognized as fruitful to formalize AC models (Li and Mitchell, 2003).

We operate the main following specializations on FOL by selecting DATALOG^C as a formal language for rules: functions, negation and disjunction are not allowed. Formally, DATALOG^C sentences are formal expressions of the form:

$$\forall \tilde{X}\ R_1(X_1) \wedge \ldots \wedge R_n(X_n) \wedge \psi(\tilde{X}) \Rightarrow R_0(X_0)$$

where $\forall i \in 0..n$, R_i are symbols from Voc and X_i are sequences of logic variables of length equal to the arity of R_i. The left-hand side of a rule $(R_1(X_1) \wedge \ldots \wedge R_n(X_n) \wedge \psi(\tilde{X}))$ is called the *body* of the rule its right-hand side $(R_0(X_0))$ is called its *head*. $\tilde{X}$ is the set of variables that appear in the body. $\psi(\tilde{X})$ is a conjunction of linear arithmetic constraints (e.g., $\leq$, $\geq$, $=$, $\neq$) over variables and constants of same sorts.

Definition 3 *Access control rules. The* rules *of and* AC *model is a set P of* DATALOG^C *sentences.*
P defines a partition on the vocabulary Voc: $idb \cap edb = \emptyset$ and $idb \cup edb = Voc$. idb, for intensive database*, is the set of relations that appear in the heads of the rules and edb, for* extensive database*, is the set of relations that appear only in the bodies of the rules but not in their heads.*

The definition of the component P that captures rules is generic. It encompasses the main notions introduced in the literature to organize AC policies. Next subsections focus on three main derived relations that are commonly used in AC models : *authorization triple*, *hierarchies* and *mutual exclusion*. We can now formally define AC policies.

Definition 4 *Access control policy. The semantics of rules is given by the standard* FOL *model-theoretic interpretation of* DATALOG^C *rules.*
Given an AC *state* $\mathbf{I}$ *of a model $AC = (Voc, P)$, an* access control policy $\mathbf{I}'$ *over a given state* $\mathbf{I}$ *is a logical model of P stated* $\mathbf{I}' \models P$ *with* $\mathbf{I} \subseteq \mathbf{I}'$.

The existence of a decidable procedure which computes the interpretation $\mathbf{I}'$ from $\mathbf{I}$ ensures that it is always possible to compute the policy and consequently to answer whether an access request is granted or denied. DATALOG's (and its major extension such as DATALOG^C, C-DATALOG or DATALOG$^\neg$) restrictions ensure that there is a *unique minimal model* of P and that this model can be computed in a finite time (Abiteboul et al., 1995, theorem 12.5.2, p. 301). Uniqueness of the minimal model ensures that for given state and set of rules there is a unique derived policy. The finite time property ensures that computation of the policy if decidable and thus, it ensures that access control requests will be always answered.

2.2.1 Derivation of authorizations

We have defined the structure of an AC model, however deriving authorizations from state has not been explained yet. As coined by Lampson in his seminal paper (Lampson, 1974), the aim of access control is to take a boolean AC decision upon an AC request. An AC request is a triple subject, action, and object. The reference monitor which enforces AC acts as a Non-bypassable, Evaluatable, Always Invoked and Tamperproof (NEAT[1]) proxy between subjects and objects. It takes authorization decisions upon the policy.

[1] `http://www.ois.com/Products/MILS-Technical-Primer.html`

$SR(S,R) \wedge PRA(R,A,O)$	$\Rightarrow$	$Access(S,A,O)$
$URA(U,R) \wedge PRA(R,A,O)$	$\Rightarrow$	$Static(U,A,O)$
$SU(S,U) \wedge Access(S,A,O)$	$\Rightarrow$	$Dynamic(U,A,O)$

Table 2 Rules for fundamental triples in RBAC.

Definition 5 *Fundamental triples. The three following* fundamental triples *symbols must appear in the intensive database idb, and the set of rules P must define how to derive the corresponding relations Access* $\subseteq$ *Subject* $\times$ *Action* $\times$ *Object, Static* $\subseteq$ *User* $\times$ *Action* $\times$ *Object and Dynamic* $\subseteq$ *User* $\times$ *Action* $\times$ *Object.*
The relation Access is the set of AC *permissions granted to subjects. Static is the set of permissions granted to users independently of the subjects they use. Dynamic is the set of permissions granted to users through the subjects. The inclusion Dynamic* $\subseteq$ *Static* must hold *in any model.*

Authorizations decisions are based upon triples *Access* derived from the state. Thus, the reference monitor can be modelled as a function:

$$\mathcal{F} : Subject \times Action \times Object \rightarrow \{false, true\}$$
$$\mathcal{F}(s,a,o) = \begin{cases} true & \text{if } (s,a,o) \in Access \\ false & \text{otherwise} \end{cases}$$

The three rules showed in table 2 define the triples in the RBAC model. According to the sample toy state of table 1, user `Bob` is assigned to two roles $\mathtt{r}_1$ and $\mathtt{r}_3$. One may be interested by the authorizations statically granted to `Bob` which is $\{(\mathtt{r},\mathtt{file}_1),(\mathtt{r},\mathtt{file}_2),(\mathtt{r},\mathtt{file}_3),(\mathtt{w},\mathtt{file}_2),(\mathtt{r},\mathtt{file}_4),(\mathtt{w},\mathtt{file}_4),(\mathtt{x},\mathtt{file}_4)\}$ by means of the query:

$$\{(a,o) \mid Static(\mathtt{Bob},a,o)\}$$

2.2.2 AC *models hierarchies*

The RBAC standard contains two major features to make AC administration easier: role hierarchies and constraints. Hierarchies are a way to reduce redundant user and permission-role assignments. Roles are given a preorder (reflexive, transitive) $\preccurlyeq$ modelling an *is a* relationship. Relation $r_1 \preccurlyeq r_2$ means that every permissions granted to role r_1 are granted to r_2 and that each user who is a member of role r_2 is also a member of r_1. Our framework generalizes this approach and takes recommendations of Li et al. (2007) into account:

> **Suggestion 4**: *"The Reference Model should maintain a relation that contains the role dominance relationships that have been explicitly added, and update this relation when the role hierarchy changes."*

For any sort $S \in edb$ of an AC model, we define two relations: a dominance relation $SeniorD_S \in edb$ stored in the state, and an inheritance relation $Senior_S \in idb$ defined as the reflexive transitive closure of $SeniorD_S$. Please note that we use suffix D to distinguish between extensive and intensive predicate.

Definition 6 *Inheritance. A sort* $S \in edb$ *of an* AC *model is given an* inheritance *relationship if a* dominance *relation* $SeniorD_S$ *with* $SeniorD_S \subseteq S \times S$ *is defined in edb and if the following three rules are defined in P:*

$$\begin{array}{rcl} SeniorD_S(S,S') & \Rightarrow & Senior_S(S,S') \\ SeniorD_S(S,S') \wedge Senior_S(S',S'') & \Rightarrow & Senior_S(S,S'') \\ S(ID) & \Rightarrow & SeniorD_S(ID,ID) \end{array}$$

If defined, a $Senior_S$ relation should be used in the definition of AC triples. For instance, in the RBAC models with role hierarchies, the rule deriving the *Access* relation is redefined as follows, in order to take the role hierarchy into account:

$$SR(S,R) \wedge Senior_R(R,R') \wedge PRA(R',A,O) \Rightarrow Access(S,A,O)$$

State from table 1 is an instance of the *flat* RBAC model, without role hierarchy. Assume that we add $\mathtt{r}_5$ into roles and that $\mathtt{r}_5$ inherits $\mathtt{r}_4$ $Senior_{Role}(\mathtt{r}_5,\mathtt{r}_1)$. With the extended *Access* rule, any user who is assigned to $\mathtt{r}_5$ has at least `r` access on `file1`, `file2` and `file3` because these permissions are granted to $\mathtt{r}_1$.

2.2.3 Mutual exclusion relation

Another important feature introduced in the AC literature is *mutual exclusion*, which is the main constraint defined in the RBAC standard (Ferraiolo et al., 2003). For instance, one may define that no user can be assigned to both roles $\mathtt{r}_3$ and $\mathtt{r}_4$, because they stands for mutually incompatible roles.

As it is the case for hierarchies, mutual exclusion needs two relations: an extensive one *SoDD* and an intensive one *SoD* which is its symmetric closure. Moreover, when both an exclusion and an inheritance relation have been defined on the same sort, an additional rule must be defined in P, to ensure that exclusion is propagated via inheritance (Gavrila and Barkley, 1998).

Definition 7 *Mutual exclusion. A sort $S \in edb$ is given a* mutual exclusion *relationship if a core separation relation $SoDD_C \subseteq S \times S$ is defined in edb and if a relation $SoD_S \subseteq S \times S$ is defined with the following two principles in the set of rules P:*

$$\begin{array}{rcl} SoDD_S(S,S') & \Rightarrow & SoD_S(S,S') \\ SoD_S(S,S') & \Rightarrow & SoD_S(S',S) \end{array}$$

If an inheritance relation $Senior_S$ is also set on a sort S, then the following rule must be defined:

$$SoD_S(S,S') \wedge Senior_S(S'',S) \Rightarrow SoD_S(S,S'')$$

Last definition is still incomplete: we have not defined *which* constraint the mutual exclusion relation really enforces. We have neither expressed that an inheritance relation should be *antisymmetric* nor that a subject should be assigned to a *unique* user. As a matter of fact, we have only expressed *how* to define the policy $\mathbf{I}'$. We will show in the next section how to define the conditions upon which the policies are consistent according to a set of constraints.

For instance, in the toy RBAC state of table 1, one states that $\mathtt{r}_3$ and $\mathtt{r}_4$ are mutually exclusive by adding $SoD_{Role}(\mathtt{r}_3,\mathtt{r}_4)$ into the state. It may be a way to ensure that request (allowed to $\mathtt{r}_3$) and approval (allowed to $\mathtt{r}_4$) of major expenditure are done by two separate people. In the toy RBAC state, it should be inconsitent to state $SoD_{Role}(\mathtt{r}_1,\mathtt{r}_3)$ because user `Bob` is assigned to both.

2.3 Framework summary

Policies are *logical models* (in the model-theoretic sense) of a theory defined by an AC *model*. The word *model* is indeed prone to confusion. We use the term (AC) *model* to refer to the structure that describes how rights are organized and granted (i.e., the meaning of *model* in the AC literature). We explicitly use the term *logical model* to refer to a model-theoretic interpretation which satisfies a set of closed FOL formulae.

This preliminary modelling step is paramount for addressing further issues. The proposed framework is closely related to the *deductive database* paradigm. In this paradgim, a database is defined with a *schema* (i.e., the AC vocabulary Voc) and a set of *deduction rules* (i.e., expressed in DATALOG).

An AC state is considered as a set of *relational data* structured accordingly to a given vocabulary. A policy is a set of facts derived from a state and a set of rules. Formally, we have defined an AC model as a pair $AC = (Voc, P)$ composed as follows:

- Voc, the access control *vocabulary*: a set of unary relations called *sorts* and n-ary *relations* between these sorts. This vocabulary sets the *core* of the AC model used to organize the policies. An *intepretation* $\mathbf{I}$ of *edb* is a *state*.
- P: a set of DATALOG^C rules defining the AC model *rules*. These rules allow deducing consequent facts from $\mathbf{I}$, thus defining an *intensive vocabulary idb* and a *policy* $\mathbf{I}'$ over *idb*.

One of our main objectives is to treat constraints as first class citizens. In the next section, we will extend the definition of (simple) AC model to be a triple $AC = (Voc, P, \Sigma)$. This refinement includes a set Σ of FOL formulae modelling the AC model constraints. They are expressed by *data dependencies*, which subsume the expressive power of traditional deductive database such as DATALOG^C. As shown in this paper, this enhanced expressivity is needed to model complex integrity constraints found in AC models. Whereas formulae of P allow deducing the policy from a given state, those from Σ *restrict* $\mathbf{I}$ and $\mathbf{I}'$. The definition and usage of Σ are defined in the next section.

3 Access Control Constraints

Among the formal tools available in the database area, *data dependencies* (a.k.a *integrity constraints*) have been defined to capture integrity requirements on relational data. In a unification attempt, they have been defined as FOL sentences (Abiteboul et al., 1995, Chapter 10). In the proposed framework, dependencies are used to capture AC constraints: they capture formal integrity requirements of policies.

3.1 Data dependencies

Dependencies share common characteristics with DATALOG deduction rules, but they form a larger subclass of FOL sentences. Data dependencies are categorized into classes of increasing expressivity. The best known classes are *functional* (FD), *inclusion* (IND) and *multivalued* (MVD) dependencies (Abiteboul et al., 1995). Expressive classes have been developed to express complex statements on relational data. They can model semantic relationships in spatial, temporal or multimedia databases.

One of the most general form of dependencies is *constrained tuple-generating dependencies* (CTGD) (Maher and Srivastava, 1996) which are FOL sentences having the following syntax (we reuse convention from section 2.2):

$$\begin{array}{l}\forall\tilde{X}\ R_1(X_1) \wedge \ldots \wedge R_n(X_n) \wedge \psi(\tilde{X}) \Rightarrow \\ \exists\tilde{Z}\ Q_1(Y_1) \wedge \ldots \wedge Q_m(Y_m) \wedge \phi(\tilde{Y})\end{array}$$

where $\tilde{Z}$ does not designate the whole set of variables in the head but only those which are not already bound by a universal quantifier ($\tilde{Z} = \tilde{Y} - \tilde{X}$).

Special forms of dependencies considered in this paper are restriction of CTGD:

1. Constraint-Generating Dependencies (CGD), head is restricted to constraints:
$$\forall\tilde{X}\ R_1(X_1) \wedge \ldots \wedge R_n(X_n) \wedge \psi(\tilde{X}) \Rightarrow \phi(\tilde{X})$$
2. Nullity-Generating Dependencies (NGD), head is empty[2]:
$$\forall\tilde{X}\ R_1(X_1) \wedge \ldots \wedge R_n(X_n) \wedge \phi(\tilde{X}) \Rightarrow \perp$$
3. Full Tuple-Generating Dependencies or Total Tuple-Generating Dependencies (Full-TGD) do not include existentially quantified variables:
$$\forall\tilde{X}\ R_1(X_1) \wedge \ldots \wedge R_n(X_n) \Rightarrow Q_1(X'_1) \wedge \ldots \wedge Q_m(X'_m)$$
4. Tuple-Generating Dependencies (TGD) or Generalized Dependencies, generalize both FTGD and IND but do not allow constraints:
$$\forall\tilde{X}\ R_1(X_1) \wedge \ldots \wedge R_n(X_n) \Rightarrow \exists\tilde{Z}\ Q_1(Y_1) \wedge \ldots \wedge Q_m(Y_m)$$

Dependencies are used either to restrict authorized values in a policy when their heads do not include atoms (e.g., CGD, NGD), or to impose presence of tuples if some other ones are already present in the policy (e.g., IND, MVD, TGD). Note that NGD allow the expression of negative requirements as $\phi \Rightarrow \perp$ is logically equivalent to $\neg\phi$.

In our framework, the set of constraints over an AC model is denoted as Σ. It captures conditions that *must hold* in the policy. The expressivity of dependencies is needed to capture desirable properties which cannot be expressed in P, which relies on the DATALOG^C fragment, mainly due to the absence of existential quantifier. From the logical perspective the $P \cup \Sigma$ of rules and constraints is a logical theory, i.e., a set of closed FOL formulae.

Definition 8 *Access control model. An access control model is a triple $AC = (Voc, P, \Sigma)$, where Σ is a set of constraints expressed as* data dependencies.
The semantics of constraints is given by the standard FOL *model-theoretic interpretation of dependencies. An access control policy $\mathbf{I}'$ built from a state $\mathbf{I}$ is* consistent *iff $\mathbf{I}' \models \Sigma$.*

The next subsections describe how dependencies are used to express various security requirements of models in an homogeneous way: integrity of states, algebraic properties, semantics of mutual exclusion, constraints on authorizations and administrative prerequisites.

[2] $\perp$ stands for a logical antilogy (e.g., $0 = 1$)

$SR(S,R)$	$\Rightarrow$	$\exists U\ SU(S,U)$
$SU(S,U) \wedge SU(S,U')$	$\Rightarrow$	$U = U'$
$SU(S,U) \wedge SR(S,R)$	$\Rightarrow$	$URA(U,R)$

Table 3 Integrity requirements for *SU* relation in Rbac.

3.2 Integrity constraints on AC states

We define a category of constraints ensuring that relations are well-founded. For example, whenever a relation over sorts exists, the sorts must exist too. Thus, the sentence $URA(U,R) \Rightarrow User(U) \wedge Role(R)$ should be enforced in *any* Rbac policies: a role can be assigned to a user only if both the user and the role exist. This can be considered as an equivalent to *foreign key* constraints in RDBMS. These kinds of constraints ensure that what is actually stored is consistent. This leads to the definition of *well-founded state*.

Definition 9 *Well-founded state. Let Λ be a set of constraints $\Lambda \subseteq \Sigma$ involving only symbols of the extensive database edb, such that for each n-ary relation $R \in edb$ over sorts $S_1 \ldots S_n \in edb$, Λ contains an inclusion dependency of the form:*

$$R(ID_1,\ldots,ID_n) \Rightarrow S_1(ID_1) \wedge \ldots \wedge S_n(ID_n)$$

Let $\mathbf{I}$ *be a state. Then* $\mathbf{I}$ *is* well-founded *if* $\mathbf{I} \models \Lambda$.

For instance, in the Rbac standard it is defined that each subject has to be assigned to a *unique* user and that a role can be used by a subject *only if* the role is assigned to the user who owns the subject. These constraints can be defined by dependencies as shown in table 3. These requirements are satisfied by the Rbac state given in table 1, thus this state is a well-founded one.

The dependencies paradigm allows the formalization of the next suggestion found in the critique of the Rbac standard (Li et al., 2007):

> **Suggestion 2**: *"The standard should accommodate* Rbac *systems that allow only one role to be activated in a session"*

This constraints can be captured in a straightforward way by means of a *functional dependency* that enforces a key constraint $SR(U,R) \wedge SR(U,R') \Rightarrow R = R'$. Actually, this requirement is not satisfied by table 1, because `Alice` endorses two different roles in session `S1`.

3.3 Algebraic constraints of relations

AC models hierarchies are commonly defined as partial orders for avoiding cycles (e.g., (Kuhn, 1997) for Rbac or (Miège, 2005) for Orbac). Moreover, a mutual exclusion is defined as irreflexive to prevent a sort from being mutually exclusive with itself (e.g., (Li et al., 2004)). As example, antisymmetry of inheritance relation and irreflexivity of mutual exclusion in Rbac can be expressed by dependencies, as given in table 4.

Moreover, some additional properties may be desirable. For example some models (e.g., Lattice-Bac (Sandhu, 1993)) organize sorts using forests of trees, forests of inverted trees or lattices (posets in which any two elements have a join and a meet).

These additional requirements over $Senior_S$ and $SeniorD_S$ relations can be expressed by CGD. Note that the restriction of a hierarchy to a lattice could not be expressed in DATALOG-based framework, because these frameworks lacks existentially quantified variables, and the property states that *there exists* some meet and join for any given pair of elements.

Definition 10 *Restricted hierarchies. Let $Senior_S \in idb$ be an intensive inheritance relation computed from an extensive one $SeniorD_S \in edb$. $Senior_S$ is called a* limited hierarchy *of type:*

- forest of trees*, if Σ contains the following* FD*:*

$$SeniorD_S(S, S'), SeniorD_S(S, S'') \Rightarrow S' = S''$$

- forest of inverted trees*, if Σ contains the following* FD*:*

$$SeniorD_S(S', S), SeniorD_S(S'', S) \Rightarrow S' = S''$$

- lattice*, if Σ contains the following* FTGD*:*

$$\begin{array}{l} S(S'), S(S'') \Rightarrow \exists S_\perp \ Senior_S(S', S_\perp), Senior_S(S'', S_\perp) \\ S(S'), S(S'') \Rightarrow \exists S_\top \ Senior_S(S_\top, S'), Senior_S(S_\top, S'') \end{array}$$

3.4 Semantics of mutual exclusion

Although the intensive $SoD_S \in idb$ relation ensures that there is a mutual exclusion on sort S, the principles of mutual exclusion can be applied from different perspectives. In the RBAC model, if two roles r and r' are stated as mutually exclusive, several interpretations can exist (Crampton, 2003). All these different semantics can be stated respectively by means of CGD:

- no *user* could be assigned to both roles r and r':

$$SoD_{Role}(R, R') \wedge URA(U, R) \wedge URA(U, R') \Rightarrow \perp$$

- no *subject* could be assigned to both r and r':

$$SoD_{Role}(R, R') \wedge SR(S, R) \wedge SR(S, R') \Rightarrow \perp$$

- no common *permission* could be granted to both r and r':

$$SoD_{Role}(R, R) \wedge PRA(R, A, O) \wedge PRA(R', A, O) \Rightarrow \perp$$

- no action over a common *object* could be granted to both r and r':

$$SoD_{Role}(R, R') \wedge PRA(R, A, O) \wedge PRA(R', A', O) \Rightarrow \perp$$

For instance, according to the first semantics, it is inconsistent to add $SoD_{Role}(\mathtt{r_1}, \mathtt{r_2})$ in the state given in table 4, because `Alice` is granted both roles $\mathtt{r_1}$ and $\mathtt{r_2}$. According to the second semantics, there is an inconsistency because of session `S1`. However, according to the third semantics, there is no inconsistency because no permission is granted to two different roles.

$Senior_{Role}(R, R') \wedge Senior_{Role}(R', R)$	$\Rightarrow$	$R = R'$
$SoD_{Role}(R, R)$	$\Rightarrow$	$\perp$

Table 4 Antisymmetry of role hierarchy, and irreflexivity of mutual exclusion.

$Access(S, A, O)$	$\Rightarrow$	$\exists R\ SR(S, R) \wedge PRA(R, A, O)$
$Static(U, A, O)$	$\Rightarrow$	$\exists R\ URA(U, R) \wedge PRA(R, A, O)$
$Dynamic(U, A, O)$	$\Rightarrow$	$\exists S\ SU(S, U) \wedge Access(S, A, O)$

Table 5 Constraints on authorizations in Rbac.

3.5 Constraints on authorization relations

The Rbac model imposes that any authorization must be granted via a role (Ferraiolo et al., 2003):

> **Property 3.2**: *"A subject s can perform an operation op on object o* only if *there exists a role r that is included in the subject's active role set and there exists an permission that is assigned to r such that the permission authorizes the performance of op on o".*

This property can be considered as fundamental in structured ac models. Defining intermediate sorts between users and permissions is a way to simplify administration tasks. Bypassing these sorts is error-prone and may lead to ambiguities within policies. Thus, we argue that such a property defined for Rbac should be generalized. To the best of our knowledge, this property is neither captured in logic-based modelling attempts of Rbac, nor in extended models.

Definition 11 *Property of authorizations. If there is a rule in P with head Access (resp. Static, Dynamic) and hypothesis ψ, then there is a corresponding constraint in Σ of the form Access $\Rightarrow \psi$ (resp. Static, Dynamic) that makes the rule an* if and only if *condition.*

According to the rules for triples given in table 2, the constraints of table 5 can be derived. These tgd use existentially quantified variables shared among multiple predicates. Such sentences can not be expressed in Datalog-based frameworks because they lack the existential quantifier. This quantifier is needed to capture so called *invented values* in the database terminology, i.e., unknown values that have to be present in a policy to ensure its consistency.

3.6 Administrative prerequisite

An administrative prerequisite enforces the presence of tuples *before allowing administrative operations* (Ferraiolo et al., 2003). Administrative operations consist in updates, insertions and deletions of tuples within the state **I**. If any administrative prerequisite, expressed by some dependencies, is violated, the transaction initiated by the administrator will not be committed.

Definition 12 *Administrative prerequisite. An administrative prerequisite constraint imposes the presence of tuples in $\mathbf{I}'$. A transitive prerequisite relation $Required_R \in idb$ over a relation R is defined as the transitive closure of a relation $RequiredD_R \in edb$. Prerequisite constraints can be modelled as (constrained)* tgd *in Σ by an auxiliary relation $RequiredD_R$ of the following form, where the predicate R is present in both the sentence body and head:*

$$\forall \tilde{X}\ R(X_R) \wedge \ldots \wedge Required_R(X) \wedge \phi(\tilde{X}) \Rightarrow \exists \tilde{Z}\ R(X'_R) \wedge \ldots \wedge \psi(\tilde{Y})$$

For instance, in the RBAC models, administrative prerequisites are used for preventing administrators from assigning roles to users if some other role has not already been defined. This is an example of a prerequisite over the relation URA. The following sentences express these statements. Defining $RequiredD(r_1, r_2)$ in $\mathbf{I}$ imposes that whenever a user is assigned to r_2, then there must be at least one user assigned to r_2.

$$\begin{array}{rcl} RequiredD(R, R'), Required(R', R'') & \Rightarrow & Required(R, R'') \\ URA(U, R), Required(R, R') & \Rightarrow & \exists U' \; URA(U', R') \end{array}$$

4 Constraint verification

From now on, we have defined the three basic components of an AC model. The task of defining the vocabulary Voc (the names of subjects, roles, assignments, etc.), the set of administrative rules P and the set of integrity constraints Σ is dedicated to the AC model designer. Administrators define only the state $\mathbf{I}$, as $\mathbf{I}' = P(\mathbf{I})$ is computed from $\mathbf{I}$ and P. The policy is consistent if $\mathbf{I}' \models \Sigma$. Thus, from a logical perspective, there are only a few differences between P and Σ which can be considered as a whole as a logical theory $T = P \cup \Sigma$.

4.1 Formal characterization

The main difference between P and Σ lays in their usage: rules are used for *computing* the policy from the state, whereas constraints are used to impose *restrictions* on authorized instances of $\mathbf{I}'$. Notice that the uniqueness and computability of $\mathbf{I}'$ is ensured by the properties of the fragment of FOL used for P.

In this section we rely on two theoretical problems over logical theories:

– the *satisfaction problem.* Answering whether a policy $\mathbf{I}'$ satisfies a given FOL sentence σ: $\mathbf{I}' \models \sigma$. This problem is central for computing $\mathbf{I}'$ from $\mathbf{I}$, for answering queries and for checking whether a policy satisfies the set of constraints for a given model. The satisfaction problem is decidable for the class of formulae used for $AC = (Voc, P, \Sigma)$.
– the *logical implication problem.* Answering whether a set of closed formulae T logically implies a closed formula σ: $T \models \sigma$ or, in other words, deciding if *any* policy model of a theory T is also a model of the single sentence σ. This problem is central for simplifying logical theory or for checking model consistency from an abstract perspective, *without considering any state.* This problem is decidable in the class of FTGD extended with constraints. However, it is semi-decidable (it may not halt for negative answers but will always halt for positive ones) for larger classes of dependencies such as CTGD.

As dependencies such as CTGD are strictly more expressive than DATALOG^C, and as they share the same FOL semantics, we will actually treat the sentences of P as dependencies to build and homogeneous logical theory T made of rules and constraints.Thus, it is possible to use the same proof procedures for both rules and constraints without distinction.

We have implemented the proof procedures for dependencies presented in (Beeri and Vardi, 1984), (Maher and Srivastava, 1996) and (Coulondre, 2003) to validate our approach. As shown in section 4.6, our prototype allows automatizing the consistency checking of RBAC policies and furnishing proof of previous results independently proposed (Gavrila and Barkley, 1998).

4.2 Administrative review

One of the features expected from an access control model implementation is the *administrative review*. This set of operations has been defined in the RBAC standard: "*When URA and PRA relation have been created, it should be possible to view the contents of those relations from both the user and role perspectives*[...]". In the present framework, administrative reviews are simple conjunctive queries over $\mathbf{I}'$. Therefore, the requirements of the standard definition are met.

Definition 13 *Administrative reviews. Any conjunctive query (from the standard database sense) built upon the vocabulary Voc is an* administrative review.

Examples of queries built upon the RBAC model vocabulary Voc are respectively (1) the set of permissions granted directly (without hierarchy) to users through their roles , (2) the set of users who can execute an object, whatever it is, (3) the set of users who are assigned the role $\texttt{r}_1$:

1. $\{(u, a, o) \mid \exists r\ URA(u, r) \wedge PRA(r, a, o)\}$,
2. $\{u \mid \exists s, o, r\ SU(s, r) \wedge SR(s, r) \wedge SeniorD(r, r') \wedge PRA(r', \texttt{x}, o)\}$
3. $\{u \mid URA(u, \texttt{r}_1)\}$

The answer to the last query, on the state given in table 1, is $\{\texttt{Alice}, \texttt{Bob}, \texttt{Charly}\}$. When $\mathbf{I}$ only is queried, we can rely upon any RDBMS to provide the querying mechanism, as the state is stored in extension in relational tables. The technical difficulty is to compute and to query the policy $\mathbf{I}'$. As far as DATALOG or DATALOG^C is chosen, $\mathbf{I}'$ can be computed quite efficiently. This computation is still valid as long as $\mathbf{I}$ is not modified, thus allowing some caching optimizations techniques. We suggest four approaches to compute the policy $\mathbf{I}'$ from a state $\mathbf{I}$ and rules P:

– use recursive queries and advanced features provided by common RDBMS. For instance, Microsoft SQL-Server offers the Common Table Expression system which allows (restricted) recursive queries. From the logical point of view, this kind of feature can be seen as restrictions of DATALOG (e.g., set-based operations are not provided),
– use triggers and stored procedures to compute $\mathbf{I}'$ on the fly on each modification of $\mathbf{I}$. This approach can be implemented for simple access control models (when cardinality of T is small enough), but will become hard to maintain if many roles are defined. Indeed it is needed to code specific procedures for each FOL sentence in P.
– use a deductive database engine. Several efficient engines have been developed, for instance DLV[3] or XSB[4]. They can compute $\mathbf{I}'$ and include recursive queries and simple forms of dependencies. However, expressive dependencies such as TGD are not handled.

[3] http://www.dbai.tuwien.ac.at/proj/dlv/
[4] http://xsb.sourceforge.net/

– develop an external inference engine out of an existing RDBMS while relying on the mechanisms from the RDBMS for access and modification of state. As basic operations are common to both computation of $\mathbf{I}'$ and logical inference over T (see next section), we have implemented this approach in a library dedicated to expression and inference of general classes of dependencies.

4.3 Policy completeness and consistency

The validity of an AC policy is checked by verifying, given a model $AC = (Voc, P, \Sigma)$, whether $\mathbf{I}' \models \Sigma$. Unsatisfaction of a set of constraints by a policy can fall into two categories:

– the policy is *inconsistent*: some CGD or NGD are not satisfied. For instance, a policy is said to be inconsistent if antisymmetry, irreflexivity or exclusion relations are not satisfied,
– the instance is *incomplete*: some TGD are not satisfied. For instance, a policy is said to be incomplete if some properties of authorization relations or administrative prerequisites are not satisfied.

If a policy is *inconsistent* or *incomplete*, administrators have to correct the state. Whenever a policy is inconsistent, deletion of existing facts or value updates should be favored. Whenever a policy is incomplete, addition of facts should be privileged. Examples of possibles corrections of inconsistent or incomplete policies are given in table 6.

Dependency		**Type**	**Correction in the state**
property	antisymmetry	EGD	*Deletion* of cycles
	irreflexivity	NGD	*Deletion* of edges
restrictions	tree hierarchy	EGD	*Deletion* of ancestors
	inverted-tree hierarchy	EGD	*Deletion* of descendant
	lattice hierarchy	TTGD	*Addition* of edges
constraints	prerequisite	CTGD	*Addition* of requirement
	exclusion	NGD	*Deletion* of assignments
	hierarchy/exclusion	NGD	*Deletion* of assignments

Table 6 Examples of inconsistent or incomplete policies corrections

4.4 Static policy comparison

Several AC models have introduced *dynamic* sorts. For instance, in the RBAC family, the unique *dynamic* sort is *subject*, the other ones (*role*, *user*, *action*, *object*, *permission*) are *static*, as well as relations between them (e.g., user-role assignment URA, permission-role assignment PRA). This specialization can be expressed *according to the rights granted to administrators* over the state. Sorts and relations in *edb* can be categorized as follows:

- *static*: *only* administrators can modify this part of the state. Static aspects are *stable according to the execution of the system* and do not depend on the end-user activity.
- *dynamic*: administrators *are not the only ones* having the right to modify this part of the state. For instance, the sort of *Subject* in Rbac acts on behalf of an user. A *SU* is created each time a user logged into the system. In ac models involving time or space, these sorts and relations are out of the sovereignty of the administrators.

It is straightforward to define static *comparison* of policies from the relational paradigm. In section 2.2.1, we have defined three fundamental authorization triples, i.e., *Access*, *Dynamic* and *Static*. Considering the last one, we can compare static restriction of policies by comparing the sets of tuples in *Static*.

Definition 14 *Static comparison of policies. Let be* $\mathbf{I_1}$ *and* $\mathbf{I_2}$ *two consistent and complete policies, expressed on two different* ac *models* AC_1 *et* AC_2.
Let $Static_1$ *be the set of static authorization triples of* $\mathbf{I_1}$, *and* $Static_2$ *be the set of static authorization triples of* $\mathbf{I_2}$. $Static_1 \in \mathbf{I'_1} \supseteq \mathbf{I_1}$ *et* $Static_2 \in \mathbf{I'_2} \supseteq \mathbf{I_2}$.
The policy $\mathbf{I_1}$ *is as or more* more restrictive *than* $\mathbf{I_2}$ iff*:*

$$Static_1 \subseteq Static_2$$

The policy $\mathbf{I_1}$ *is as or more* more permissive *than* $\mathbf{I_2}$ iff*:*

$$Static_1 \supseteq Static_2$$

The policies $\mathbf{I_1}$ *and* $\mathbf{I_2}$ *are* equivalent iff*:*

$$Static_1 \subseteq Static_2 \text{ and } Static_1 \supseteq Static_2$$

Actually, static sorts constitute the backbone of ac models. For instance, roles in Rbac, labels in Mac, organizations in Orbac or tasks in Trbac are static sorts. Thus, it is worth verifying static enforcement of ac policies, as this can ensure that most robust properties are valid in *any state* of the policies (Li et al., 2004).

4.5 Model properties

Whereas previous subsections have been devoted to *policy* checking, this section considers ac models from an abstract perspective, without reference to *any* particular policy or state. The main problem we address is *logical implication*, in particular for model simplification purposes. For example, in the case of a new tailored ac model, where many collaborative designers from different sites might be involved, the associated logical theory may become quite large (e.g., hundreds of rules and statements). Thus, for practical purposes, it is necessary to reduce the size of the theory.

Definition 15 *Redundancy in an* ac *model. Let be* $T = P \cup \Sigma$ *the logical theory of an* ac *model made of rules and constraints. Let be* $\sigma \in T$, *if* $T \backslash \{\sigma\} \models \sigma$ *then the dependency* σ *is said* redundant, *moreover* $T \backslash \{\sigma\}$ *and* T *have the same models.*

The authors of Maher and Srivastava (1996) give two bottom-up (also known as forward) chase procedures for solving the implication problem of CTGD: given a set of CTGD Σ, and a single CTGD σ (of the form $\phi \Rightarrow \psi$), determine whether in every policy where Σ is satisfied, σ is also satisfied. If the chase procedures successfully stop, then Σ logically implies σ, stated briefly as $\Sigma \models \sigma$.

The operational nature of proof procedures for CTGD is based on the concept of tuple (a grounded atom, with no variables). The procedures saturates a symbolic database made of tuples by repeated applications of dependencies, to produce a model of the set of dependencies. In a precise sense, this model is the most general possible one, it is a *canonical* universal one Calì et al. (2008).

Maher and Srivastava's procedures keep the same strategy as the original chase of Beeri and Vardi (1984) extended to deal with constraints. For sake of clarity, original algorithm is given on page 29. Its design is the core of other procedures for dependencies and it is conceptually simpler to understand. It's theoretical complexity has been shown to be exponential (Beeri and Vardi, 1984). The very basic outline of the chase is as follows, input is a set of dependencies Σ and a single dependency σ:

1. *initialization*: the chase picks a valuation μ to hypothesize the existence of some tuples so that the body ϕ of σ is satisfied,
2. *main loop*: treats Σ as a closure operator generating tuples $\Sigma(\phi)$, that is repeatedly applies dependencies from Σ to produce new facts,
3. *exit condition*: at the end of each loop, the following conditions are checked and three termination cases are possible:
   (a) if $\Sigma(\phi)$ contains an inconsistency, return $\Sigma \models \sigma$ vacuously,
   (b) if $\Sigma(\phi)$ contains an instance of ψ, that is there is a valuation ν that extends μ such that $\nu(\psi) \in \Sigma(\phi)$, return $\Sigma \models \sigma$,
   (c) if $\Sigma(\phi)$ neither produces new facts nor contains a instance of ψ, return $\Sigma \not\models \sigma$.

The CTGD implication problem is semi-decidable: the procedure always halt when the answer is positive, but may run forever if the answer is negative. However, there are some interesting decidability results holding in various subclasses of CTGD such as (Weakly) Guarded-TGD that include existentially quantified variables (Calì et al., 2008). For example, the chase is decidable for TGD having no existentially quantified variable (Beeri and Vardi, 1984).

4.6 Sample result

This section illustrates the proposed approach by simplifying integrity properties of mutual exclusion and inheritance defined over a common sort. For this illustation, we use the set of integrity properties for RBAC models defined by Gavrila and Barkley (1998). In the present approach, these properties are modelled by constraints in Σ. The authors have manually proved that the set of properties in table 7 can be reduced to a smaller set. Using some proof procedures for dependencies, we can simplify their logical theory by eliminating redundancies.

We provide a sample execution trace obtained using a prototype. This prototype, written in C++, can handle the CTGD as well as its subclasses. It relies on an external constraint solver over reals to handle constraints. Three different *chases* have been implemented : (Beeri and Vardi, 1984), (Maher and Srivastava, 1996) and (Coulondre, 2003). Given a logical formalization of an AC written using dependencies, the prototype

	Description
σ_1	*any two roles assigned for a same user are not in separation of duties* $URA(User, Role_1), URA(User, Role_2), SoD(Role_1, Role_2) \Rightarrow \perp$
σ_2	*no role is mutually exclusive with itself* $SoD(Role, Role) \Rightarrow \perp$
σ_3	*mutual exclusion is symmetric* $SoD(Role_1, Role_2) \Rightarrow SoD(Role_2, Role_1)$
σ_4	*any two roles in ssd do not inherit one another* $Senior(Role_1, Role_2), SoD(Role_1, Role_2) \Rightarrow \perp$
σ_5	*there is no role inheriting two roles in ssd* $SoD(Role_1, Role_2), Senior(Senior, Role_1), Senior(Senior, Role_2) \Rightarrow \perp$
σ_6	*If a role inherits another role and that role is in SSD with a third one, then the inheriting role is in SSD with the third one.* $Senior(Senior, Role_1), SoD(Role_1, Role_2) \Rightarrow SoD(Senior, Role_2)$.

Table 7 Logical characterization of RBAC constraints (Gavrila and Barkley, 1998)

is used to simplify the model, to compute the policy for a given state, to check the consistency of the policy and to answer administrative queries. Static comparison of policies have not been implemented yet.

The following trace is the result of the execution of the CTGD *chase* of Maher and Srivastava (1996) implemented in the prototype. Initially, the dependency base is loaded with σ_2, σ_3 and σ_6 of table 7. The goal is to prove that $\{\sigma_2, \sigma_3, \sigma_6\} \models \sigma_5$. The following trace is a formal proof of this entailment.

```
---------------------------Dependencies in base : 4----------------------------
[0] (for all)[R1,R2] SoDD(R1,R2)->SoD(R1,R2).
[1] (for all)[R1,R2] SoD(R1,R2)->SoD(R2,R1).
[2] (for all)[R] SoD(R,R)->error(reflex) {(1=\=1)}.
[3] (for all)[R,R1,R2] SoD(R1,R2),senior(R,R1)->SoD(R,R2).
-------------------------------------------------------------------------------
Tuples :
[0] SoD(_r1_0,_r2_0); -1;
[1] Senior(_r_0,_r1_0); -1;
[2] Senior(_r_0,_r2_0); -1;

seed : SoD(_r1_0,_r2_0),Senior(_r_0,_r1_0),Senior(_r_0,_r2_0)
goal :  {(1=\=1) }
tgdGoal : (for all)[R,R1,R2] SoD(R1,R2),Senior(R,R1),Senior(R,R2)-> {(1=\=1)}.
```

Hypothesize three tuples $SoD(\mathtt{r}_1, \mathtt{r}_2)$, $Senior(\mathtt{r}_0, \mathtt{r}_1)$ and $Senior(\mathtt{r}_0, \mathtt{r}_2)$ from the body of σ_5.

```
stepNumber : 1
++++++++++++++++++++++++++++++++++++++++
Treating [1]... '(for all)[R1,R2] SoD(R1,R2)->SoD(R2,R1).'
        Added to tuples: SoD(_r2_0,_r1_0); 1;
        Added to activations: {R1:_r1_0,R2:_r2_0}; 0;
...[1] treated

Treating [3]... '(for all)[R,R1,R2] SoD(R1,R2),Senior(R,R1)->SoD(R,R2).'
        Added to tuples: SoD(_r_0,_r2_0); 3;
        Added to activations: {R:_r_0,R1:_r1_0,R2:_r2_0}; 0,1;

        Added to tuples: SoD(_r_0,_r1_0); 3;
        Added to activations: {R:_r_0,R1:_r2_0,R2:_r1_0}; 3,2;
...[3] treated
```

With σ_3 (sentence `[1]`, *mutual exclusion is symetric*), $SoD(\mathtt{r}_2, \mathtt{r}_1)$ is derived. Next, applying σ_6 (sentence `[3]`, *exclusion is propagated through inheritance*) twice, $SoD(\mathtt{r}_0, \mathtt{r}_2)$ et $SoD(\mathtt{r}_0, \mathtt{r}_1)$ is derived.

```
stepNumber : 2
++++++++++++++++++++++++++++++++++++++++
Treating [1]... '(for all)[R1,R2] SoD(R1,R2)->SoD(R2,R1).'
        Added to tuples: SoD(_r2_0,_r_0); 1;
        Added to activations: {R1:_r_0,R2:_r2_0}; 4;

        Added to tuples: SoD(_r1_0,_r_0); 1;
        Added to activations: {R1:_r_0,R2:_r1_0}; 5;
...[1] treated

Treating [3]... '(for all)[R,R1,R2] SoD(R1,R2),Senior(R,R1)->SoD(R,R2).'
        Added to tuples: exclusion(_r_0,_r_0); 3;
        Added to activations: {R:_r_0,R1:_r2_0,R2:_r_0}; 6,2;
...[3] treated
```

Next, using σ_3 (sentence `[1]`), $SoD(\mathtt{r}_2, \mathtt{r}_0)$ and $SoD(\mathtt{r}_1, \mathtt{r}_0)$ are deduced. Thus, by σ_6 (sentence `[3]`), $SoD(\mathtt{r}_0, \mathtt{r}_0)$ is derived.

```
stepNumber : 3
++++++++++++++++++++++++++++++++++++++++
Treating [2]... '(for all)[R] SoD(R,R)->Error(reflex) {(1=\=1)}.'
        Added to tuples: Error(reflex); 2;
        Added to store: (1=\=1)
        Added to activations: {R:_r_0}; 8;
...[2] treated

--------------------------------------------------------------------------------------
there is an inconsistency in the constraint store F|=g vacuously (VACUOUSLY)
number of rules applied for closure F(l):7
this chase was :0.088197 seconds long
number of tuples generated:10
```

Finally, applying σ_2 (sentence `[2]`, *exclusion is irreflexive*), an antilogy is derived. Thus, the chase procedure proved that $\{\sigma_2, \sigma_3, \sigma_6\} \models \sigma_5$. The prototype can be used to derive other sample theorem from the dependencies of table 7. Let be Σ the six dependencies shown in the table. The chase procedures can prove that $\Sigma\backslash\{\sigma_4\} \models \sigma_4$, $\Sigma\backslash\{\sigma_5\} \models \sigma_5$ and that $\Sigma\backslash\{\sigma_4, \sigma_5\} \models \sigma_4, \sigma_5$.

This example illustrates the utility of the proposed framework. Given an AC model $AC = (Voc, P, \Sigma)$, an automated proof of non-trivial properties can be provided. For instance, besides the above obtained theorems, we have been able to derive the following results:

- *read and write* access over an objet in Mandatory Access Control (MAC) are granted to a subject iff the subject's clearance level is equal to the object's confidentiality level (Sandhu, 1993),
- a *root role* which inherits all other ones cannot exist in an RBAC policy where two roles are mutually exclusive (Benantar, 2006),
- dynamic authorizations are a subset of static authorizations in RBAC policies (Ferraiolo et al., 2003).

5 Related work

5.1 General comparison

Our work is closely related to (Barker and Stuckey, 2003) which expresses Rbac models in constraint logic programming, to (Halpern and Weissman, 2003) which identifies a subset of Fol for ac models, to (Bertino et al., 2003) which describes a C-Datalog framework for representing ac models, to (Jajodia et al., 2001) which defines a flexible access control framework, to (Li and Mitchell, 2003) which uses Datalog^C and (Miège, 2005) which uses the Datalog$^\neg$ for defining the Orbac model. Our main goals were to provide a framework for ac which:

- makes a clear *separation* between policies and models and has a clear semantics,
- is expressive enough to *capture* and *generalize* roperties of ac models,
- treats policy *constraints* as first-class citizens,
- provides readable algorithm execution traces for designers and administrators.

The background we have settled in section 2 is based on Datalog^C. This logical framework has been often treated as a *middle-ground formalism* to which many other logical based frameworks can be reduced, for instance in the work of (DeTreville, 2002; Jim, 2001; Bertino et al., 2003; Li and Mitchell, 2003). Moreover, Datalog^C has shown to be able to capture several ac models, and various extensions of Rbac model (Joshi et al., 2005; Damiani et al., 2007). As the present framework subsumes Datalog^C, it can basically capture these models.

The main difference with the above previous work is the fruitful use of data dependencies as a unifying logical framework which encompasses both traditional ac rules *and* integrity constraints. Thus, constraints are expressed in the very same model, and not expressed in an independent and different framework. As constraints are integrity requirements of policies, we argue that integrating them in the model as soon and as tightly as possible is a step towards ensuring ac robustness.

We rely on some known results for data dependencies, in order to provide well-founded tools for reasoning on policies. An interesting feature of these proof procedures is that they do not require any prior translation of Fol formulae of rules P and integrity constraints Σ (e.g., by means of Skolemization or rewriting rules). This property leads to native clear traces of automated proofs, as given in section 4.6. This greatly enhances the readability of inference results for design and maintenance purposes. Finally, by means of the prototype, we have been able to re-prove in an automated way some interesting results found in the literature.

5.2 Frameworks built upon Datalog

(Bertino et al., 2003) and (Li and Mitchell, 2003) have used the Datalog framework, respectively C-Datalog which allows object-oriented definitions, and Datalog^C which includes constraints. These frameworks are able to capture extended Rbac models such as Temporal-Rbac (Bertino et al., 2001). The basic components of the present proposition (sections 2 and 2.2) reuse and extend some of their formal models. (Li et al., 2003), (Jim, 2001), and (DeTreville, 2002) have extended Datalog with specific constructions which are reduced into standard Datalog formulae. However, we have not used rewriting procedures which may cloud the debugging steps and puzzle administrators.

5.3 Other frameworks built upon logic

(Jajodia et al., 2001), (Barker and Stuckey, 2003) and (Halpern and Weissman, 2003) describe logical programs able to express general AC models. These propositions address the problems arising with the use of *closed* policies (in which access is by default denied and authorizations are only positive), *open* policies (access is granted by default) or *hybrid* policies (authorizations and denial can be both explicitly defined). Miège (2005) use DATALOG$^{\neg}$ to deal with negative authorization (prohibition). These logical frameworks allow a permissive use of negation in formal sentences, whereas we chose to favor existential quantifiers. However, our framework is able to support a restrictive use of negation by means of NGD for instance. Our framework captures complex integrity requirements considered as fundamental in RBAC, which are not expressible in other ones. Moreover, we provide an effective procedure to automatize administrative operations.

Our framework is tightly linked to the database area and makes a clear distinction between policies and models. This separation, advocated by (Li et al., 2007), is not explicit in the work of Barker and Stuckey (2003) and Halpern and Weissman (2003), as well as in the logic programming paradigm on a more general point of view. Finally, we have reused and extended some definitions of (Ferraiolo et al., 2003) (properties of RBAC) and results of (Gavrila and Barkley, 1998) (consistency of RBAC databases) which were not integrated into other frameworks.

5.4 Access control constraints

Constraints have received much attention in AC models. RBAC is argueably one of their most prominent representatives for which many kinds of constraints have been proposed (Gligor et al., 1998; Ahn and Sandhu, 1999; Crampton, 2003). These suggestions either define and categorize new kinds of constraints (e.g., the variations on mutual exclusion outlined in section 3.4) or proposed specification languages for constraints. Most of these constraints are defined as a way to enforce *separation of duties*, and are variants of mutual exclusion.

The algebra of Li and Wang (2008) for specifying constraints encompasses these approaches but consider constraints as high-level organization requirements. Our approach is quite different, in the sense that constraints are defined closer to the model at a lower level. Thus, we can capture intrinsic properties (e.g., constraints on authorization triples, prerequisites) as well as general integrity requirements (e.g., states well-foundedness, algebraic properties of relations) which are usually taken into account separately.

6 Discussion

This section discusses the main choices we made. We divide this section into the AC structure design choices and the FOL fragment choice.

6.1 Design and administration

By drawing a distinction between an AC model and an AC policy instanciated from a model, we can define two AC-related activities:

- *model design* is the task of defining the AC *model*. The model designer sets up the sorts and relations of the model (Voc), its rules (P) and the properties that must be enforced (Σ). For instance, a model designer may define that tasks and roles are used to structure policies. As an analogy in the database field, the model designer counterpart is the database *administrator*, who defines tables (Voc), views (P) and keys between tables (Σ).
- *policy administration* is the task of defining the *state* $\mathbf{I}$. The policy administrator sets up the facts of an AC model instance, but can not define $\mathbf{I}'$ directly because it is automatically derived from $\mathbf{I}$ using P. According to the above example, a policy administrator sets that `physician` and `surgeon` are roles, and is responsible for assigning these roles to individuals. As an analogy in the database field, the policy administrator counterpart is an *end-user*, whose job is to query and modify the policy, but who is not able to modify the schema.

6.2 Model structure

In section 2, we introduced the structure of $AC = (Voc, P, \Sigma)$ made of three components and we defined the major design choices we have made:

- AC models are defined as logical theories while being narrowed to decidable fragments without negation, disjunction nor function symbols in P,
- the vocabulary is partitioned into *sorts* and *relations* in a many-sorted FOL paradigm,
- *rules* (P) and *constraints* (Σ) are distinguished,
- the difference between a state $\mathbf{I}$ and a policy $\mathbf{I}'$ is logically made explicit.

One may argue that these choices are somewhat restrictive. The administrator's tasks are limited to extensive policies management, i.e., to tuple management in the AC state. It might be possible to allow them to expressed their own formulae in P or Σ. However, we think this is not desirable, as this would blur the responsabilities between designers and maintainers and would introduce higher policy checking complexity. In such an approach, the stability of the AC models would not be guaranteed through the policies lifetime, in particular consistency and interoperability of updated models.

6.3 Fragments of logic for access control

The logical fragment of FOL we used to define P and Σ is quite restrictive: negation, disjunction and function symbols are not allowed. The main argument is that we obtain a *unique* and *computable* model of P. Computability is necessary because the reference monitor has to answer each access request. Uniqueness for integrity checking purposes is required, in order to avoid checking the satisfaction of Σ over multiple models of P. Moreover, uniqueness may ease the understanding of intensive policies by administrators.

Furthermore, we chose to have a more expressive framework for constraints than for rules, by favoring *existential quantification* over *negation*. The main goal is to be able to model some of the most important constraints commonly identified for AC models, which cannot be expressed in DATALOG models. Even with a quite simple AC model (for instance, with a few sorts, relations and principles), most of the properties given in section 3 require existentially quantified variables (e.g., sessions integrity properties, restricted hierarchies, authorizations properties and administrative prerequisites).

7 Conclusions and perspectives

In this paper, we presented a logical framework $AC = (Voc, P, \Sigma)$ for AC models, which relies on three abstract components. The key idea is to express and handle AC models, policies and constraints in an homogeneous way within the very same logic background *by means of* database integrity theory (not to deploy AC policies *in* databases). We then showed how to use well-founded procedures in order to enforce and check constraints, and to provide a formal trace of inferences. The proposed framework is not universal because of the choice we made to focus on access control constraints and to treat them as first-class citizens. It however allows for expressing several classical models and extensions found in the literature. It also takes into account some major recommendations that had been previously addressed to RBAC models. Moreover, the expressivity of general classes of dependencies allows capturing most of the AC model properties considered as fundamental.

We envision several extensions and perspectives:

- extension of the FOL subclasses, to capture new properties of models. However, algorithms decidablity and tractability should be taken into account. As an example, it may be interesting to use some decidable subclass of TGD by imposing some restrictions on existentially quantified variables, as it is the case for negation in stratified-DATALOG for instance,
- explore automated maintenance of databases (Chomicki and Marcinkowski, 2005), and data integration (Fagin, 2006) to fix non-consistent policies. Data integration may be a fruitful perspective for policies creation expressed in different models (Li et al., 2009),
- broadening the scope of policies. Indeed, it would be valuable to model administrative policies, which define administrator rights over AC policies. This could be useful for instance when a huge policy requires several administrators, each of them being allowed to handle only a part of the policy.
- another emerging topic is *usage control* and *privacy protection*. The basic components and definitions we presented can be used to define next generation AC models and policies, as it has been done with RBAC models (Ni et al., 2009).

References

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, Boston, 1995. ISBN 0-201-53771-0.

Gail-Joon Ahn and Ravi S. Sandhu. The RSL99 language for role-based separation of duty constraints. In *RBAC'99: 4th ACM workshop on Role-based access control*, pages 43–54, New York, 1999. ACM Press. ISBN 1-58113-180-1. doi: http://doi.acm.org/10.1145/319171.319176.

Steve Barker and Peter J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Transactions on Information & System Security*, 6(4):501–546, 2003.

Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

Messaoud Benantar, editor. *Access Control Systems - Security, Identity Management and Trust Models*. Springer-Verlag, 2006.

Elisa Bertino, Piero A. Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information & System Security*, 4(3): 191–233, 2001.

Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information & System Security*, 6(1):71–127, 2003.

Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In Gerhard Brewka and Jérôme Lang, editors, *KR*, pages 70–80. AAAI Press, 2008. ISBN 978-1-57735-384-3.

Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information & Computation*, 197(1-2):90–121, 2005.

Stêphane Coulondre. A top-down proof procedure for generalized data dependencies. *Acta Informatica*, 39(1):1–29, 2003.

Jason Crampton. Specifying and enforcing constraints in role-based access control. In *SACMAT'03: 8th ACM Symposium on Access Control Models and Technologies*, pages 43–50. ACM Press, 2003. ISBN 1-58113-681-1. doi: http://doi.acm.org/10.1145/775412.775419.

Maria Luisa Damiani, Elisa Bertino, Barbara Catania, and Paolo Perlasca. GEO-RBAC: A spatially aware rbac. *ACM Transactions on Information & System Security*, 10(1), 2007.

John DeTreville. Binder, a logic-based security language. In *SP'02: IEEE Symposium on Security and Privacy*, page 105, Washington, 2002. IEEE Computer Society. ISBN 0-7695-1543-6.

Ronald Fagin. Inverting schema mappings. In Stijn Vansummeren, editor, *PODS'06: 25th ACM SIGACT-SIGMOD-SIGART Symposiumon Principles of Database Systems, Chicago, Illinois*, pages 50–59. ACM Press, 2006. ISBN 1-59593-318-2.

David F. Ferraiolo, Richard D. Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House Publishers, 2003. ISBN 1-58053-370-1.

Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving. Revised on-line version 2003.* Harper & Row, 1986. ISBN 0-06-042225-4. http://www.cis.upenn.edu/ jean/gbooks/logic.html.

Serban I. Gavrila and John F. Barkley. Formal specification for role based access control user/role and role/role relationship management. In *RBAC'98: 3rd ACM workshop on Role-based access control*, pages 81–90, 1998.

Virgil D. Gligor, Serban I. Gavrila, and David F. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *1998 Symposium on Security and Privacy*, pages 172–183, Oakland, California, 1998. IEEE Computer Society Press.

Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. In *CSFW'03: 16th IEEE Computer Security Foundations Workshop, Pacific Grove, California*, pages 187–201. IEEE Computer Society, 2003. ISBN 0-7695-1927-X.

Trent Jaeger and Jonathon E. Tidswell. Practical safety in flexible access control models. *ACM Trans. Inf. Syst. Secur.*, 4(2):158–190, 2001. ISSN 1094-9224. doi: http://doi.acm.org/10.1145/501963.501966.

Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26:214–260, June 2001. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/383891.383894. URL `http://doi.acm.org/10.1145/383891.383894`.

Trevor Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

James Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge & Data Engineering*, 17(1):4–23, 2005.

Richard D. Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *RBAC'97: 2nd ACM workshop on Role-based access control*, pages 23–30, New York, 1997. ACM Press. ISBN 0-89791-985-8. doi: http://doi.acm.org/10.1145/266741.266749.

Butler W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/775265.775268.

Ninghui Li and John C. Mitchell. DATALOG with constraints: A foundation for trust management languages. In Verónica Dahl and Philip Wadler, editors, *PADL'03: 5th International Symposium on Practical Aspects of Declarative Languages, New Orleans*, volume 2562 of *Lecture Notes in Computer Science*, pages 58–73. Springer-Verlag, 2003. ISBN 3-540-00389-4.

Ninghui Li and Qihua Wang. Beyond separation of duty: An algebra for specifying high-level security policies. *J. ACM*, 55(3), 2008.

Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information & System Security*, 6(1):128–171, 2003.

Ninghui Li, Ziad Bizri, and Mahesh V. Tripunitara. On mutually-exclusive roles and separation of duty. In *CCS'04: 11th ACM conference on Computer and communications security*, pages 42–51, New York, 2004. ACM Press. ISBN 1-58113-961-6. doi: http://doi.acm.org/10.1145/1030083.1030091.

Ninghui Li, Ji-Won Byun, and Elisa Bertino. A critique of the ANSI standard on role-based access control. *IEEE Security and Privacy*, 5(6):41–49, 2007. ISSN 1540-7993. doi: http://doi.ieeecomputersociety.org/10.1109/MSP.2007.158.

Ninghui Li, Qihua Wang, Wahbeh Qardaji, Elisa Bertino, Prathima Rao, Jorge Lobo, and Dan Lin. Access control policy combining: theory meets practice. In *SACMAT '09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 135–144. ACM, 2009. ISBN 978-1-60558-537-6. doi: http://doi.acm.org/10.1145/1542207.1542229.

Michael J. Maher and Divesh Srivastava. Chasing constrained tuple-generating dependencies. In Richard Hull, editor, *PODS'96: 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Montreal, Canada*, pages 128–138. ACM Press, 1996. ISBN 0-89791-781-2.

Alexandre Miège. *Dénition d'un environnement formel d'expression de politiques de sécurité : modèle Or-BAC et extensions.* PhD thesis, Ecole Nationale Supérieure des Télécommunications,Paris, 2005.

Qun Ni, Elisa Bertino, and Jorge Lobo. Privacy-aware RBAC - leveraging RBAC for privacy. *IEEE Security and Privacy*, to appear, 2009.

Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/2.241422.

Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and CharlesE. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

Roshan K. Thomas. Team-based access control (tmac): a primitive for applying role-based access controls in collaborative environments. In *RBAC'97: 2nd ACM workshop on Role-based access control*, pages 13–19, New York, 1997. ACM Press. ISBN 0-89791-985-8. doi: http://doi.acm.org/10.1145/266741.266748.

Roshan K. Thomas and Ravi S. Sandhu. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented autorization management. In Tsau Young Lin and Shelly Qian, editors, *IFIP'98: 11th International Conference on Database Security, Lake Tahoe, California*, volume 113 of *IFIP Conference Proceedings*, pages 166–181. Chapman & Hall, 1997. ISBN 0-412-82090-0.

Jacques Wainer, Paulo Barthelmess, and Akhil Kumar. W-RBAC - a workflow security model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems*, 12(4):455–485, 2003.

Jacques Wainer, Akhil Kumar, and Paulo Barthelmess. DW-RBAC: A formal security model of delegation and revocation in workflow systems. *Information Systems*, 32 (3):365–384, 2007.

A Chase algorithm

```
Input  : $\Sigma$ : a set of TGD and EGD
Input  : $\sigma \equiv \phi \to \psi$ : a TGD or and EGD
Output: $true$ iff $\Sigma \models \sigma$ or $false$ only if $\Sigma \not\models \sigma$
begin
    {initialization}
    let $\mathbf{I}$ an empty (symbolic) instance of $\mathbf{R}$
    let $\nu$ an arbitrary valuation of $\phi$'s variables
    let $\Gamma$ the set of valuations used for each dependency
    for atom $R_i(\mathtt{c_1}, \ldots, \mathtt{c_n})$ in $\nu(\phi)$ do
        $\mathbf{I} \leftarrow \mathbf{I} \cup R_i(\mathtt{c_1}, \ldots, \mathtt{c_n})$
    {main loop}
    while $\mathbf{I}' \neq \mathbf{I}$ do
        $\mathbf{I}' \leftarrow \mathbf{I}$
        for $\alpha \in \Sigma$, $\alpha := a \to b$ do
            for valuation $\mu$ such that $\mu(a) \in \mathbf{I}$ do
                if $(\alpha, \mu) \notin \Gamma$ then
                    if $\alpha$ is a TGD then
                        for $R_i(\mathtt{c_1}, \ldots, \mathtt{c_n})$ in $\mu(b)$ do
                            if $R_i(\mathtt{c_1}, \ldots, \mathtt{c_n}) \notin \mathbf{I}$ then
                                $\mathbf{I} \leftarrow \mathbf{I} \cup R_i(\mathtt{c_1}, \ldots, \mathtt{c_n})$
                    else
                        for $\mathtt{c_i} = \mathtt{c_j}$ in $\mu(b)$ do
                            replace all occurences of $\mathtt{c_i}$ in $\mathbf{I}$ by $\mathtt{c_j}$
                    $\Gamma \leftarrow \Gamma \cup (\alpha, \mu)$
            {exit condition}
            if $\sigma$ is a TGD then
                for $R_i(\mathtt{c_1}, \ldots, \mathtt{c_n})$ in $\nu(\psi)$ do
                    if $R_i(\mathtt{c_1}, \ldots, \mathtt{c_n})$ in $\mathbf{I}$ then
                        return $true$
            else
                for $\mathtt{c_i} = \mathtt{c_j}$ in $\nu(\psi)$ do
                    if $\mathtt{c_i} = \mathtt{c_j}$ or if ($\mathtt{c_i}$ or $\mathtt{c_j}$) is not in $\mathbf{I}$ then
                        return $true$
end
return $false$ (fixpoint is reached)
```

Algorithm 1: Chase algorithm from Beeri and Vardi (1984)