Concrete and Abstract Based Access Control

Yacine Bouzida[1], Luigi Logrippo[1], and Serge Mankovski[2]

[1]Université du Québec en Outaouais, Gatineau, Québec, Canada
[2]CA Labs, Thornhill ON, Canada

June 14, 2011

Abstract

Access control models allow expressing access control rules (also called policies) stating that certain subjects (or users) have or do not have the right (or privilege) to access certain objects in order to execute certain actions under certain conditions. Several existing models allow expressing rules only for specific subjects, objects and actions. Role Based Access Control (RBAC) introduced the notion of role, which is an abstraction over subjects. Organization Based Access Control (OrBAC) generalized further, by allowing specifying rules involving abstract subjects, abstract actions and abstract objects. We propose here a model that allows expressing rules involving any combinations of abstract or concrete subjects, actions, and objects, as well as conditions over them. For this reason, our model is called Concrete and Abstract Based Access Control model (CABAC). The semantics of our model is expressed in terms of first-order predicate logic. Temporal, spatial, knowledge, and historical contexts can be specified and combined. We show how in this model it is possible to express hierarchies of subjects, objects and actions as well as propagation of policies over hierarchies. Further, while in most models subjects, objects and actions, whether concrete or abstract, must be specified statically, it is possible in our model to specify subjects, actions and objects dynamically, i.e. according to conditions that can vary over time. Access control rules can also be explicitly revoked and subjected to different types of constraints, among which are cardinality constraints and separation of duties.

1 Introduction and related work

Access control models allow expressing access control rules (also called policies) stating that certain subjects (users) have or do not have the right (or privilege) to access certain objects in order to execute certain actions under certain conditions. However, managing security policies that associate privileges to subjects is complex as these policies are likely to change over time. Several existing models allow expressing rules only for specific subjects, objects and actions. Role Based Access Control (RBAC) [14, 21] introduced the notion of role, which is an abstraction over subjects. In RBAC, privileges are assigned to roles. The different roles are stable while the different subjects assigned to them might change. In addition, the concept of role is associated with the concept of functional role. In a hospital such roles could be doctors, nurses, etc. Defining relations between users and roles on one hand and roles and permissions on the other hand allows flexibility for managing security policies in organizations. For example, a security officer can express an exclusion relation between two roles in order to enforce separation of duty policies. He may also define inheritance between two roles to allow delegation of authority [10], by which a user may assign some rights he owns to another user.

However, the simplest RBAC models do not allow defining permissions that may be granted to roles when certain context conditions are satisfied. For instance, *"nurses are allowed to read medical records only in emergency cases"* is one example of such a rule. Extensions of RBAC that take into account temporal contexts over roles are investigated in the Temporal Based Access control (TRBAC) model [4] and its extension Generalized Temporal Based Access Control (GTRBAC) model [17]. The corresponding roles of the RBAC extension models could be available at certain times and unavailable at other times and it is possible to express a variety of temporal constraints over roles including periodic and duration constraints. For example, GTRBAC can handle a policy that says: *"a part time nurse may be authorized to work from* $09h00$ *to* $13h00$ *during working days"*. Such a requirement may be supported to specify when the corresponding part time nurse can be enabled and then may be activated by a legitimate user who is authorized to play the role of a part time nurse. GEORBAC —a spatially aware RBAC— is introduced in [5] as an extension of the RBAC model to deal with spatial aspects of applications such as mobile applications.

Park and Sandhu [20] have presented a complex access control model, called UCON-ABC, which includes the concepts of authorizations, obligations, and conditions. UCON supports role and group hierarchies, but does not explicitly support the concept of different levels of abstraction for subjects, actions and objects, nor dynamic definition of these, which are the main concerns of CABAC. Privilege inheritance is not explicitly discussed in [20], implying that UCON adopts the inheritance model that is present in RBAC. In this area, our paper generalizes over RBAC by introducing the concept of "propagation direction" (Section 6). Since the areas of application for the two models are essentially the same, and the predicate logic semantics are similar, future research will have to attempt a combination of what will prove to be the most useful aspects of them.

In spite of all these extensions, there is not a general RBAC model that allows expressing different privileges such as permissions, prohibitions and obligations that apply only when certain conditions are satisfied. This is one of the reasons why the OrBAC model [1] was introduced. The different privileges (i.e. permissions, prohibitions, etc.) that are expressed within OrBAC are specified using contexts. In addition, the OrBAC model provides means to specify within a unique framework the fact that organizations may be structured into sub-organizations having their own specified security policies.

In this paper, we introduce the concept of Concrete and Abstract Based Access Control model (CABAC) which inherits from the concept of role in RBAC and OrBAC, and from the concept of concrete entities such as subjects and objects in Discretionary Access Control (DAC) [19].

Most access control models specify static security policies. An exception is the OrBAC model that uses contexts over privileges (permissions, prohibitions and obligations) for dynamic activation of these privileges. Dynamic environments need such expressiveness for deploying real time access control policies. To our best knowledge, OrBAC is the only access control model known in the literature, that offers this expressiveness. As an example, the OrBAC high level policy specification has been used as the final stage in a voice over IP (VoIP) framework in order to definitively react against intrusions [11]. However, OrBAC expresses only contexts over privileges. Consider as example a rule that says that *"only emergency nurses and emergency doctors that are currently in the emergency ward are allowed to use the emergency line"*. To express this in the OrBAC formalism, or in all extensions of RBAC we know, we need two rules, one for each role: nurses and doctors. More generally, if there is a general rule that applies to certain roles, then there will have to be as many rules as there are roles that should be considered. We propose to improve such multiplication of rules by introducing in CABAC the notion of dynamic role assignment. This is performed by assigning dynamically defined roles to subjects and then assigning privileges to these roles. In such situations, the simplification made possible by the use of the CABAC model with respect to RBAC and OrBAC is comparable to the simplification made possible by RBAC models with respect to DAC. Notice that dynamic assignment also applies to actions and objects that can be grouped into abstract actions and abstract objects respectively.

In Section 8, we introduce the notion of dynamic revocation that allows revoking dynamically a role from a subject when some conditions are satisfied. This is useful when considering threats inherent in computer networks. For example, the *http* server role may be revoked from a machine that plays this role if it is attacked. The contextual condition in this case is the event of attack. Once the server role is revoked from the attacked machine, the latter loses all privileges of that role and appropriate policies are executed for the machine whose goal is to counter the effects of such attacks. One of the simplest actions that may be the result of such policies is shutting down the attacked machine or stopping the corresponding *http* dæmon.

Finally, the CABAC model allows specifying policies at different levels of abstraction. This flexibility allows specifying exceptions and constraints for each level of abstraction.

The rest of this paper is organized as follows. Section 2 presents the general CABAC model.

Section 3 presents the manner by which we specify high level access control policies, Sections 4 and 5 explain how access control policies can be specified at different levels. Section 5.3 presents the specification of contexts and combination of different contexts that should be satified in order to grant the privileges. Section 6 presents hierarchy inheritance over subjects, actions and objects. While contexts are only expressed with privilege specification in OrBAC [1], in CABAC they are expressed with dynamic entities, as presented in Section 7. The dynamic assignment and revocation of entities are respectively discussed in Sections 7 and 8. Section 9 shows how to specify different constraints and Section 10 contains a comparison between OrBAC (and thus implicitly RBAC) and CABAC. Finally, Section 11 concludes the paper and discusses some future work.

2 Concrete and abstract based access control - CABAC

The main goal of access control policies is to specify the privileges (permissions, prohibitions, obligations, recommendations and faculties) that regulate the different actions that may be performed by subjects on objects. These privileges may be expressed using first-order predicates or deontic logic modalities. $permission(s, a, o)$ (resp. $prohibition(s, a, o)$, $obligation(s, a, o)$, $faculty(s, a, o)$ or $recommendation(s, a, o)$) means that a subject s is permitted (is prohibited, is obliged, has the faculty or is recommended) to perform action a on object o. For instance, *permission(doctor,read,medical_record)* means that any doctor may read any medical record. In general, for many purposes, we would like to be able to specify contextual conditions on subjects, objects and actions. An expression such as $doctor(Alice)$ can be used to state that Alice is a doctor, or *medical_record(document99.doc)* can be used to state that *document99.doc* is a medical record. Privileges are granted when all such conditions are true. Rule-based languages that allow to specify such conditions have been proposed in [13, 15, 16] where the general format to express privileges is:

$$\forall s \in S, \forall a \in A, \forall o \in O, (condition) \rightarrow privilege(s, a, o)$$

where *privilege* may be a permission (resp. prohibition, obligation, recommendation or faculty), S is a set of subjects, A a set of actions and O a set of objects. A positive or a negative authorization, an obligation, a recommendation or a faculty are *privileges* in our model. Hence, the above rule means that for any subject s, action a and object o, if the provided condition is satisfied, then for subject s it is permitted (prohibited, obligatory, facultative or recommended) to perform action a on object o.

Notice that prohibition is the negation of permission; this corresponds to the following equivalence $\neg permission(s, a, o) \overset{\text{def}}{=} prohibition(s, a, o)$ meaning that the fact that a subject s is not permitted to perform action a on object o is equivalent to the fact that subject s is prohibited to perform action a on object o.

Since we may grant privileges (permissions, prohibitions, obligations, faculties and recommendations) according to certain conditions involving subjects, actions, objects and various combinations of them, CABAC as a general and robust access control model should specify different types of constraints. These constraints should be satisfied for applying the corresponding privileges. Each constraint is expressed as a logical expression. For instance, the following rule:

$$\begin{aligned} R: & \, doctor(s) \wedge medical\_record(o) \wedge identity\_patient(o, p) \wedge \\ & \, different\_ward(s, p) \rightarrow prohibition(s, read, o) \end{aligned}$$

states that a doctor is not allowed to read a medical record of a patient if she/he is not in the same ward. In this example, we have a subject constraint corresponding to the predicate $doctor(s)$, meaning that subject s is a doctor, an object constraint corresponding to the predicate $medical\_record(o)$ meaning that object o is a medical record and a subject-action-object constraint defined as follows: $identity\_patient(o, p) \wedge different\_ward(s, p)$, where $identity\_patient(o, p)$ is an application dependent predicate saying that object o is a medical record of a patient p and $different\_ward(s, p)$ is an application dependent predicate stating that subject s and patient p are located in different wards.

Each of the different models mentioned in Section 1 including RBAC [21], OrBAC [1] and DAC [19], specifies policies either at the abstract level as in the case of RBAC and OrBAC or at the concrete level as in the DAC model. Our proposal, the CABAC model (Concrete and Abstract Based Access Control), allows specifying access control policies at different levels of abstraction, namely at both concrete and abstract levels. To each concrete entity (subject, object and action) corresponds an abstract entity (abstract subject, abstract object and abstract action). At the abstract level, the security policies are expressed using abstract entities whereas concrete entities are used to express policies at the concrete level. Both concrete and abstract levels for different entities can be used in a single rule. For additional flexibility, we introduce in our model the notion of dynamic abstract entities (dynamic abstract subjects, abstract objects and abstract actions). In this way, we get a dimension of policy specification that is not present in previously known access control models. This point is detailed in Section 5.

In the following, we discuss the different aspects of the proposed CABAC model.

3 Expressing high level access control rules

In the CABAC model, the different constraints over subjects, actions and objects correspond respectively to conditions mentioning that in an organization, an abstract subject is assigned to a subject, an abstract action is assigned to an action and an abstract object is assigned to an object.

The different constraints over subjects, actions and objects are specified by means of the following relations expressed as predicates:

$1 -$ *assign_subject* is a relation predicate over domains $Org \times S \times AS$, where Org is a set of organizations, S is a set of subjects and AS a set of abstract subjects.

If org denotes an organization, s denotes a subject and as denotes an abstract subject, then relation $assign\_subject(org, s, as)$ means that in organization org, abstract subject as is assigned to subject s.

$2 -$ *assign_action* is a relation predicate over domains $Org \times A \times AA$, where Org is a set of organizations, A is a set of actions and AA a set of abstract actions.

If org denotes an organization, a an action and aa an abstract action, then $assign\_action(org, a, aa)$ means that in organization org, abstract action aa is assigned to action a. The ternary relation *assign_action* can be used to specify different semantics to the same abstract domain in different organizations. For instance, the abstract action *"consulting"* may correspond in hospital *H_Gatineau* to action *"read"* that can be executed over files while, in hospital *H_Aylmer*, it may correspond to action *"select"* that can be executed over a relational database.

$3 -$ *assign_object* is a relational predicate over domains $Org \times O \times AO$, where Org is a set of organizations, O is a set of objects and AO is a set of abstract objects.
If org denotes an organization, o denotes an object and ao denotes an abstract object, then $assign\_object(org, o, ao)$ means that in organization org, abstract object ao is assigned to object o. This ternary relation is useful since it makes it possible to assign different semantics to policies by changing the definitions of abstract entities. For instance, in hospital *Aylmer_hospital*, the abstract object *medical_record* may be specified as an *Excel* document whereas in hospital *Gatineau_hospital* it is used as a record in a relational database. This is modeled as follows:

- Gatineau hospital: $assign\_object(Gatineau\_hospital, doc99.xls, medical\_record)$
- Aylmer hospital: $assign\_object(Aylmer\_hospital, doc99.rec, medical\_record)$

Constraints that combine subjects, actions, and objects are modeled using the notion of context. We note that our definition of context is quite similar to that of the OrBAC model [1]. From now on, the context will be specified using the predicate *occurs* that is defined as follows:

$4 -$ *occurs* is a relational predicate that is defined over $Org \times S \times A \times O \times C$, where C is a set of contexts. If org is an organization, s a subject, a an action, o an object and c a context then $occurs(org, s, a, o, c)$ specifies that context c is satisfied for subject s, action a and object o.

The conditions that should be satisfied in order to relate a context within an organization to a subject, action and object are expressed using logical rules. Section 5.3 presents different examples for such rules. For instance, a *"default"* context is defined when no condition should be satisfied to grant the corresponding authorization. In organization org (i.e. if $org \in Org$), a default context is used such that[1]:

$$\forall s \in S, \forall a \in A, \forall o \in O, occurs(org, s, a, o, default) \leftarrow .$$

Meaning that the default context is always satisfied. This context may be used for general rules that should be applied unconditionally. For instance, a default context may be used to specify a rule that says that *"nurses are allowed to read medical records"*.
As another example, *patient_doctor* is a context that may be defined as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O, occurs(hospital, s, a, o, patient\_doctor) \leftarrow patient(s, o)$$

The above specification means that, in organization *hospital*, context *patient_doctor* is satisfied between subject s, action a and object o if o is a patient of doctor s.

Policy rules definition As presented in Section 2, each rule is expressed as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O, ((condition) \rightarrow privilege(s, a, o))$$

Constraint *condition* corresponds to the following conjunctive expression:

$$assign\_subject(org, s, as) \wedge assign(org, a, aa) \wedge assign\_object(org, o, ao) \wedge occurs(org, s, a, o, context)$$

Therefore, we can specify that *"a doctor can prescribe medicine to his patients"* as follows:
$assign\_subject(hospital, s, doctor) \wedge assign\_action(hospital, a, prescribe)$
$\wedge assign\_object(hospital, o, patient) \wedge occurs(hospital, s, a, o, patient\_doctor)$
$\rightarrow permitted(s, a, o)$
However, we do not express privileges directly on concrete subjects, objects and actions for specifying high level access control rules. We first specify a privilege (permission, prohibition, obligation, faculty and recommendation) between abstract entities (abstract subject, abstract action and abstract object) and contexts. This high level privilege is a relation that is defined over domains $Org \times AS \times AA \times AO \times C$. For instance, $permission(org, as, aa, ao, c)$ means that within organization org, abstract subject as is granted the permission to perform abstract action aa over abstract object ao within context c. For convenience and for differentiating between high level and concrete level privileges, we use the relation *permission* for expressing high level permission and *permitted* for expressing permission at the concrete level. Similar conventions are defined for other privileges (using the relations *prohibition*, *obligation*, *faculty* and *recommendation* for the abstract level and *permitted*, *prohibited*, *obliged*, *facultative* and *recommended* for the concrete level respectively).

Using these high level authorizations such as *permission* (resp. *prohibition*, *obligation*, *faculty* and *recommendation*), the concrete level privilege *permitted* (resp. *prohibited*, *obliged*, *facultative* and *recommended*) is derived from the *permission* (resp. *prohibition*, *obligation*, *faculty* and *recommendation*) assigned to abstract subjects, abstract actions and abstract objects by the relation *permission* (resp. prohibition, obligation, faculty, recommendation). Now, we can specify permission policies in the following general form:

$$\begin{aligned}&\forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O, \forall as \in AS, \forall aa \in AA, \forall ao \in AO, \forall c \in C,\\ &permission(org, as, aa, ao, c) \wedge assign\_subject(org, s, as) \wedge assign\_action(org, a, aa) \wedge\\ &assign\_object(org, o, ao) \wedge occurs(org, s, a, o, c)\\ &\rightarrow permitted(s, a, o)\end{aligned}$$

With the same syntax we can derive: concrete prohibitions (denoted by relation *prohibited*), concrete obligations (denoted by relation *obliged*), concrete faculties (denoted by relation *facultative*) and concrete recommendations(denoted by relation *recommended*).

[1]Following [16] and the subsequent literature, we write $B \leftarrow A$ to mean that from A one can infer B and $B \leftarrow .$ to mean that B is always true.

We notice that *permission* is defined over $Org \times AS \times AA \times AO \times C$. We have used predicate overloading as is common practice in object oriented programming in order to simplify the notation for taking into account different levels of abstraction. Therefore, *permission* is globally defined over $Org \times AS \cup S \times AA \cup A \times AO \cup O \times C$.

4 Expressing low level access control rules

Low level policies should be defined when authorizations need to be granted to concrete subjects, concrete actions and concrete objects within a context. In these cases, it is possible to define an abstract subject (singleton abstract subject) for a unique concrete subject. However, this solution is not interesting since it complicates the model with useless concrete-abstract subjects (resp. concrete-abstract actions and concrete-abstract objects) assignments. Our model proposes defining directly low level policies in addition to the high level ones specified above. Low level access control policy rules are expressed as follows:

$$\begin{aligned}&\forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O, \forall c \in C,\\ &permission(org, s, a, o, c) \wedge occurs(org, s, a, o, c)\\ &\rightarrow permitted(s, a, o)\end{aligned}$$

Low level control policies are useful for many purposes. The first one consists in reducing the number of active abstract subjects (abstract actions and abstract objects) that might be needed for expressing some specific policy types. These policies may be expressed only using the corresponding concrete entity (subject, action or object). Also, the CABAC model handles the notion of exception. For instance, let us consider the following hypothetical access rules from the medical sector. In Aylmer hospital, there are four different rules specified by the security officer.

(1) Doctors can consult medical records. (2) Doctors can use the laser machine. (3) Bob is a doctor. And finally, (4) Bob is a doctor in Aylmer hospital who cannot use the laser machine. Since Bob is the only individual to which this last rule applies, we should avoid creating an abstract subject for this rule only. By the same token, the laser machine is also a concrete object, hence we should avoid creating a new abstract object for it. Thus, exceptions are handled by specifying low level policies that are defined directly over concrete entities. The rules above can be more precisely specified as follows:

- $permission(Aylmer\_hospital, doctor, consult, medical\_records, default)$; meaning that doctors, in Aylmer hospital, are allowed to read medical records in all circumstances.
- $permission(Aylmer\_hospital, doctor, use, laser\_machine, default)$; doctors are also allowed to use the laser machine in all circumstances in Aylmer hospital.
- $assign\_subject(Aylmer\_hospital, Bob, doctor)$; meaning that abstract subject doctor is assigned to Bob for allowing him to get all privileges of doctors.
- $prohibition(Bob, use, laser\_machine, default)$; i.e. Bob is not allowed to use the laser machine in any circumstances. This policy is a low level policy that is defined only over concrete entities (subject, action and object).

So far, we have shown how to represent access control policies that are exceptions. However, there are many other policies that cannot be precisely specified using only high or low level entities, namely all the policies that deal with different levels of abstraction. As an example, in a policy we may specify that in Aylmer hospital doctors are not allowed to consult *Topsecret_H1N1_99.doc*. Notice here that the subject (*doctor*) and action (*consult*) are abstract entities whereas the object, the H1N1 file, is at the concrete level. This is expressed as follows:

$$prohibition(Aylmer\_hospital, doctor, consult, Topsecret\_H1N1\_99.doc, default)$$

As another example: *"nurses are not allowed to access room18"*. In this rule there is one abstract entity (i.e. *nurses*) and two concrete entities (i.e. action *access* and specific object *room18*). Combinations of the different levels are considered so that rules may be expressed with the different levels of entity abstraction.

By combining different levels of entities it becomes possible to express policies at different levels of abstraction as described in the following section.

5 Specifying levels of policies and contexts

5.1 Expressing policies at different levels

In Sections 3 and 4, we have introduced two distinct levels for specifying policies. The first contains concrete policies such as those defined within the DAC model and the other specifies policies only at the abstract level as in the OrBAC model and partially in the RBAC model.

Since we want to be able to express policies at different levels, we need to specify the generic function that generates concrete policies from abstract ones. This generic function , which generalizes the corresponding function defined in OrBAC [1], is as follows:

$$\begin{array}{l} \forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O, \forall as \in S \cup AS, \forall aa \in A \cup AA, \forall ao \in O \cup AO, \forall c \in C \\ permission(org, as, aa, ao, c) \wedge assign\_subject(org, s, as) \wedge assign\_action(org, a, aa) \wedge \\ assign\_object(org, o, ao) \wedge occurs(org, s, a, o, c) \\ \rightarrow permitted(s, a, o) \end{array}$$

Notice that this function should be carefully used since *assign_subject(org, s, as)* (resp. *assign_action(org, a, aa)*, *assign_object(org, o, ao)*) is included in the derivation only if $as \notin S$ (resp. $aa \notin A$, $ao \notin O$). This yields eight possibilities, the ones shown in Table 1, of which four are:

- Case 1: $\forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O, \forall c \in C,$
  $permission(org, s, a, o, c) \wedge occurs(org, s, a, o, c)$
  $\rightarrow permitted(s, a, o)$
- Case 2: $\forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O, \forall as \in AS, \forall c \in C,$
  $permission(org, as, a, o, c) \wedge assign\_subject(org, s, as) \wedge occurs(org, s, a, o, c)$
  $\rightarrow permitted(s, a, o)$
- Case 3: $\forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O, \forall aa \in AA, \forall c \in C,$
  $permission(org, s, aa, o, c) \wedge assign\_subject(org, s, as) \wedge$
  $assign\_action(org, a, aa) \wedge occurs(org, s, a, o, c)$
  $\rightarrow permitted(s, a, o)$
- Case 8: $\forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O, \forall as \in AS, \forall aa \in AA, \forall ao \in AO, \forall c \in C,$
  $permission(org, as, aa, ao, c) \wedge assign\_subject(org, s, as) \wedge$
  $assign\_action(org, a, aa) \wedge assign\_object(org, o, ao) \wedge occurs(org, s, a, o, c)$
  $\rightarrow permitted(s, a, o)$

The last case corresponds to specifying policies in the OrBAC model [1] where policies are only specified at the abstract level over abstract entities (subjects, objects and actions). The second

case number	Subject (s)/ abstract subject (as)	action (a)/ abstract action (aa)	object (o)/ abstract object (ao)	Corresponding model
case 1	s	a	o	DAC, MAC, CW, **CABAC**
case 2	as	a	o	RBAC, **CABAC**
case 3	s	aa	o	**CABAC**
case 4	as	aa	o	**CABAC**
case 5	s	a	ao	**CABAC**
case 6	as	a	ao	**CABAC**
case 7	s	aa	ao	**CABAC**
case 8	as	aa	ao	OrBAC, **CABAC**

Table 1: Different levels of security policy rules specification by CABAC and comparison with existing models

case corresponds to specifying policies in the RBAC [21] model except that the context is always set to *default* since contexts are not expressed in RBAC.

5.2 First Comparison with existing models

The CABAC model is more flexible than previously known models since it can specify policies either at the abstract level or at the concrete level. It is also possible to combine policies at the two levels over different entities when considering different levels of abstractions. Table 1 presents the different possibilities for the combined use of the two levels.

Table 1 shows that CABAC is the only access control model that can specify access control policy rules at all levels of abstraction. The concrete rules are deployed in PEPs (Policy Enforcement Points). These PEPs may be any security component. Examples are firewalls, intrusion detection systems, SBCs (Session Border Controllers), etc. that are deployed in computer networks, reference monitors that deploy access control lists in operating systems or any other hardware or software security components for accessing buildings.

Some access control models specify policies at the abstract level when dealing with organization entities whereas others specify the policies at the concrete level. In OrBAC [1] access rules are specified at the abstract level, and privileges are expressed over abstract entities, namely abstract subjects, abstract actions and abstract objects. Conflicts are detected and resolved at the abstract level, and the concrete level is derived from the abstract level. In CABAC, instead, both levels can be explicitly and directly specified, also conflicts can be detected and resolved at both levels. As well, levels can be freely mixed as we show in several examples.

RBAC [21], on the other hand, groups functionally identical subjects in order to reduce management overhead. However, the different policies within the RBAC model are expressed over abstract subjects, concrete actions and concrete objects. The only derivation from the abstract level to the concrete level is done over roles while this derivation is generalized to actions and objects within the OrBAC model.

Section 10 will contain a final comparison of CABAC with OrBAC and RBAC, including examples.

5.3 Different types of contexts

The different privileges apply when the corresponding constraints are satisfied. As presented in Section 3, the first three constraints correspond to separate conditions over subject, action and object. However, the fourth constraint in the condition part of the rule is expressed as a constraint over subject, action and object. This constraint corresponds to a set of elementary contexts that must be satisfied for applying a privilege, each of which is defined over a subject, action and object. In the following, we present four different context types:

- *Temporal context* specifies the time constraint that must be satisfied for the subject to be granted with the requested access. We suppose that we have a trusted *"Clock"* that provides us with the accurate time. The following attributes may be obtained from *"Clock"*: Time, Weekday, Monthday, Month, Monthweek, Yearweek. Two other basic functions, over the time set T, are used to express the temporal context: *from_time(t)* and *until_time(t)* where:

  - $\forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O$
    $\forall t, t' \in T, occurs(org, s, a, o, from\_time(t)) \leftarrow Time(Clock, t') \wedge t' \geq t$
  - $\forall org \in Org, \forall s \in S, \forall a \in A, \forall o \in O$
    $\forall t, t' \in T, occurs(org, s, a, o, until\_time(t)) \leftarrow Time(Clock, t') \wedge t' \leq t$

  Using the basic temporal contexts, we can define composed contexts by combining elementary contexts as specified in Section 5.4 below. For instance, let us consider the *visitinghours* context defined in the following security policy rule: *"Receptionists can locate patients during visiting hours where visiting hours temporal context corresponds to the morning hours from 11h00 to 12h00, only on the first two Mondays of each month"*. This temporal context is expressed as follows:

  $visitinghours = from\_time(11h00) \wedge until\_time(12h00) \wedge on\_weekday(Monday) \wedge$
  $(on\_monthweek(1) \vee on\_monthweek(2))$

- *Spatial context* expresses the spatial location constraints of the subject and object. This context defines the constraints, depending on the subject or object location, that should be satisfied in order to grant the requested access privilege. We assume that we have a trusted localization system (GPS or an access control system to a building and the different places within the building) that indicates the effective position of the subject or the object. Many spatial contexts may be defined. For instance, we may define a country, continent, town, street address, emergency ward of a hospital, etc. as a spatial context. We use different attributes for this context such as country, town, ward, street, etc. For example, to specify that a subject s (or object o) must be located in the emergency ward, we use the predicate *is_located* to get this information from the localization system.

  Let us express the following context condition: *"the doctor and the patient must be in the same ward"*. This context may be used in a hospital to allow doctors to prescribe medication to a patient if doctor and patient are in the same ward. In this example, *are_in_same_ward* might be defined as follows:

  $occurs(hospital, s, a, o, are\_in\_same\_ward) \leftarrow in\_ward(s, w) \wedge in\_ward(o, w)$
  where $in\_ward(s, w) \leftarrow is\_located(GPS, s, w)$ and $in\_ward(o, w) \leftarrow is\_located(GPS, o, w)$
  Notice that we assume that our GPS contains a sophisticated function that returns the exact ward of a staff member based on her geolocalization; also, we take the patient to be the object of the prescription action.

- *Knowledge context* depends on information that may be provided by the information system such as the information stored in the information system database. For instance, *"a doctor can operate a patient only if he has at least 19 years of experience"*. The corresponding context *has_*19*_years_experience* may be expressed as follows in hospital:

  $\forall s \in S, \forall a \in A, \forall o \in O$
  $occurs(hospital, s, a, o, hasmorethan\_19\_years\_experience)$
  $\leftarrow experience(s, years) \wedge years \geq 19$
  where $experience(s, years)$ is a basic function that retrieves from the information system database the number of practice years of subject s.

- *Historical context* depends on the actions that have already been performed. For example, some access requests cannot be granted unless some actions are performed before the request is presented. A database logging the different actions (with the corresponding subjects, objects and timestamps) is used for this goal. For instance, a doctor cannot operate a patient unless he has already diagnosed him. The corresponding context *has_diagnosed* of this rule may be expressed as follows:

$\forall s \in S, \forall a \in A, \forall o \in O,$
$occurs(hospital, s, a, o, has\_diagnosed)$
$\leftarrow log(s, diagnose, o)$

where $log(s, diagnose, o)$ is a dependent predicate that says that action *diagnose* has already been performed by s over o. The different actions that are performed are stored in an event log database.

Of course, context types are not limited to the the ones just mentioned. Our examples have given an idea of what can be done by using different context notions. Other context notions can be defined including weather conditions, urgency considerations when dealing with accidents in hospitals or threat contexts when dealing with intrusions in information systems.

5.4 Combining Contexts

Elementary context functions can be combined by using standard propositional logic operations ($\wedge$, $\vee$ and $\neg$).
For example, the policy that says: *In Aylmer hospital, nurses may consult medical records during working_hours and only in emergency situations* may be expressed as follows:

$$permission(Aylmer\_hospital, nurse, consult, medical\_record, working\_hours\_emergency)$$

where:

$$\begin{aligned}&occurs(Aylmer\_hospital, Alice, read, H1N1\_Bob\_99.doc, working\_hours\_emergency)\\&\leftarrow from\_time(8h00) \wedge until\_time(18h00) \wedge \neg on\_weekday(Saturday) \wedge\\&\quad \neg on\_weekday(Sunday) \wedge emergency(H1N1\_Bob\_99.doc)\end{aligned}$$

In this example, Alice is a nurse in *Aylmer hospital* (i.e. *assign_subject(Aylmer_hospital, Alice, nurse)*), *read* is an action of type *consult* (i.e. *assign_action(Aylmer_hospital, read, consult)*), *H1N1_Bob_99.doc* is a file that belongs to medical_record (i.e. *assign_object(Aylmer_hospital, H1N1_Bob_99.doc, medical_record)*) and *emergency(file)* is a predicate that is true if the corresponding file is that of a patient that is in an emergency case.

6 Hierarchies

Hierarchies on abstract entities (abstract subjects, abstract actions and abstract objects) are represented in our model. Every hierarchy corresponds to a partial order over the different abstract entities (subjects, actions and objects).

6.1 Abstract subject hierarchies

The notion of hierarchy is introduced within the RBAC model [21] as a basic concept of privilege inheritance. Hierarchies are a natural manner of structuring roles to reflect the organization main flow of responsibilities and authorities. Hierarchy of roles in OrBAC is the same as that of RBAC but is expressed using relation predicates to inherit privileges from a role by its sub role. In RBAC and OrBAC models, all privileges are propagated from the general roles to senior roles.

If we consider the RBAC or OrBAC model and the abstract subject hierarchy presented in Figure 1, then all privileges of a role are inherited by its senior role (or sub role). For example, all permissions assigned to nurses are inherited by the sub roles head nurse and then clinical staff manager. In other words, all privileges that are assigned to a role are automatically inherited by all its senior roles (or sub roles) in the hierarchy. The inheritance is always going down in such models. However, some organizations may need to specify that the inheritance of permissions will go down in hierarchies and the inheritance of prohibitions takes the other direction. In the above hierarchy example, we may wish to specify that all prohibitions that apply to clinical staff manager also apply to all its corresponding junior roles in the hierarchy *Clinical_Staff*. Therefore, the direction of the propagation of privileges should be specified.

The concept of policy propagation direction is modeled by using a propagation policy relation that we call *prop* that is defined over domains $Org \times Priv \times H \times D$, where Org denotes the

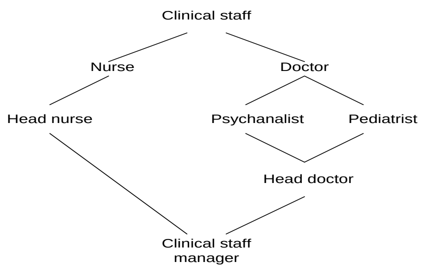

Figure 1: Abstract subject hierarchy *H_ClinicalStaff*.

set of organizations, *Priv* denotes the set of privileges (i.e. permission, prohibition, obligation, recommendation and faculty), *H* denotes the set of hierarchies –*H_ClinicalStaff* is such a hierarchy example– and *D* denotes the set of the propagation directions: {*UP, DOWN*}.

For our example, we may specify policy propagation as follows:

$$prop(Aylmer\_Clinic, permission, H\_ClinicalStaff, DOWN)$$

The different relations that are used to specify hierarchies within the CABAC model are expressed as follows. We denote the hierarchy between abstract subjects by using the predicate $sub\_abs\_subject(org, hs, as_1, as_2)$ meaning that in organization org, abstract subject as_1 is a sub-abstract subject of as_2 within hierarchy hs. Therefore, we get the following authorization inheritance cases according to the hierarchy:

- DOWN case: $\forall org \in Org, \forall as_1 \in AS, \forall as_2 \in AS, \forall aa \in AA, \forall ao \in AO, \forall c \in C, \forall hs \in HS$, where HS is a set of hierarchies over abstract subjects:
  $sub\_abs\_subject(org, hs, as_1, as_2) \wedge prop(org, authorization, hs, DOWN) \wedge$
  $authorization(org, as_2, aa, ao, c))$
  $\rightarrow authorization(org, as_1, aa, ao, c))$

- UP case: $\forall org \in Org, \forall as_1 \in AS, \forall as_2 \in AS, \forall aa \in AA, \forall ao \in AO, \forall c \in C, \forall hs \in HS$,
  $sub\_abs\_subject(org, hs, as_1, as_2) \wedge prop(org, authorization, hs, UP) \wedge$
  $authorization(org, as_1, aa, ao, c))$
  $\rightarrow authorization(org, as_2, aa, ao, c))$

6.2 Abstract action hierarchies

Accordingly, we can also define abstract action and abstract object hierarchies. The abstract action hierarchy is defined by using the predicate $sub\_abs\_action(org, ha, aa_1, aa_2)$ meaning that in organization org, abstract action aa_1 is a *sub_abstract_action* of aa_2 within abstract action hierarchy ha.

In the following example, we specify hierarchies within organization CA_GS (CA Global Switch).

Let us consider the update, or configuration command, of which there can be several subtypes as presented in Figure 2. This action consists of updating the configuration of computer and network security devices within an operator. This hierarchy describes a configuration command of which there can be two types: (i) GUI Configure Command or (ii) Configure Command Line. Again, there can be two types of each one of the last two abstract actions (i.e. Secured Configure Command and Unsecured Configure Command), etc.

We consider two directions in this hierarchy. The first is UP which says that any permission of an abstract action is inherited by its junior abstract action (if it is present). The second corresponds to the DOWN direction which says that any prohibition of an abstract action is inherited by its senior abstract action (if it exists).

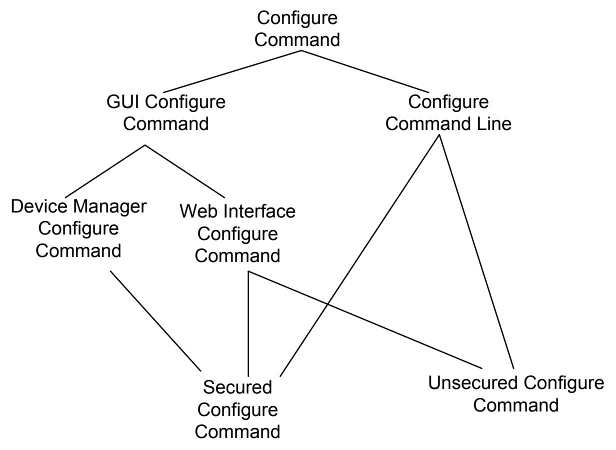
Configure Command
GUI Configure Command
Configure Command Line
Device Manager Configure Command
Web Interface Configure Command
Secured Configure Command
Unsecured Configure Command

Figure 2: Abstract action hierarchy *H_ConfigureCommand.*

The two hierarchy propagations for this example can be specified as follows:
$prop(CA\_GS, permission, H\_ConfigureCommand, UP)$
$prop(CA\_GS, prohibition, H\_ConfigureCommand, DOWN)$

Meaning that in organization CA_GS, if a subject or an abstract subject is permitted to perform an abstract action on an object or an abstract object, then she has the permission to perform its junior abstract action on the corresponding object or abstract object. As an example, if it is permitted to X (subject or abstract subject) to perform an Unsecured Configure Command on Firewall FW then X is allowed to perform both Web Interface Configure Command and Configure Command Line and so on.

However, if X (a subject or an abstract subject) is prohibited from performing an abstract action on an object (or an abstract object) then X is prohibited to perform its senior abstract action on the corresponding object or abstract object. As an example, if it is prohibited for X (subject or abstract subject) to perform any configuration on Firewall FW (or any security device), then X is prohibited from performing any configuration command, in the whole hierarchy presented in Figure 2, on the corresponding FW (or any security device).

6.3 Abstract object hierarchies

An abstract object hierarchy can be defined by using the predicate $sub\_abs\_object(org, ho, ao_1, ao_2)$ meaning that in organization org, abstract object ao_1 is a *sub_abstract_object* of ao_2 within abstract object hierarchy ho.

Let us consider the following example related to a Datacenter within the global switch of the company CA:
We define two hierarchy propagations for this example:
$prop(CA\_GS, permission, H\_DataCenter, UP)$
$prop(CA\_GS, prohibition, H\_DataCenter, DOWN)$

In case of UP propagation, a permission of an abstract object is inherited by its junior abstract object if it is present. And in case of DOWN propagation, then a prohibition of an abstract object is inherited by its senior abstract object if it is present.

So a subject or an abstract subject is permitted to access an abstract object then he/she has the permission to access its junior abstract object. As an example, if it is permitted to X (subject or abstract subject) to access the LAN Datacenter then X is allowed to access Telco Datacenter and Global Switch Datacenter.

However, if X (a subject or an abstract subject) is prohibited from accessing an abstract object

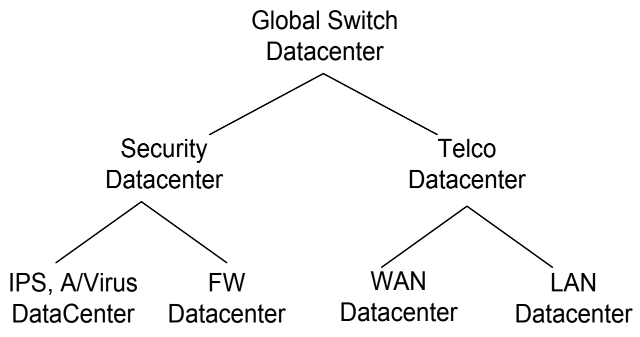

Figure 3: Abstract object hierarchy *H_DataCenter*.

then X is prohibited to access its senior abstract object. As an example, if it is prohibited for X (subject or abstract subject) to access the Global Switch Datacenter, then X is prohibited from accessing Telco Datacenter and both WAN and LAN datacenters.

UP and DOWN inheritance, especially if used together, can lead to inconsistency. Such inconsistencies can be prevented by using priority rules, a subject that we leave for further work.

7 Dynamic abstract subjects, dynamic abstract actions and dynamic abstract objects

When specifying high level policies, subjects are statically assigned to predefined subjects. While this is useful to define static roles as in RBAC [21] or OrBAC [1], some other abstract entities may be defined dynamically according to specific contexts. For instance, we may want to specify a rule policy that says that *"all subjects in the emergency ward can read all medical records and cannot prescribe medicine to patients"*. In conventional systems, in order to do this it is necessary to define as many high level policies as there are predefined abstract subjects. We propose the notion of dynamic abstract subject, which is not defined statically but is dynamically activated using contexts defined over subjects. Once a dynamic abstract subject is activated, it will be automatically assigned to subjects satisfying the corresponding specified context. This is modeled using the predefined predicates defined below.

– *occurs_dynamic_abs_subject* is a predicate that is defined over domains $Org \times S \times SC$ (where SC is a set of contexts that are defined over domain S). If s is a subject and sc is a context over subject s then $occurs\_dynamic\_abs\_subject(org, s, sc)$ specifies that context sc is satisfied over subject s in organization org.

Then the corresponding dynamic abstract subject das_{sc} is implicitly assigned to subject s as follows:

$$assign\_subject(org, s, das_{sc}) \leftarrow occurs\_dynamic\_abs\_subject(org, s, sc)$$

We also define two other predicates *occurs_dynamic_abs_action* (resp. *occurs_ dynamic_abs_object*) for dynamically activating abstract actions (resp. abstract objects):

– *occurs_dynamic_abs_action* is a predicate that is defined over domains $Org \times A \times AC$ (where AC is a set of contexts that are defined over domain A). If a is an action and ac a context over action a then *occurs_dynamic_abs_action* specifies that context ac is satisfied over action a.

The corresponding dynamic abstract action daa_{ac} is implicitly assigned to action a as follows:

$$assign\_action(org, a, daa_{ac}) \leftarrow occurs\_dynamic\_abs\_action(org, a, ac)$$

– *occurs_dynamic_abs_object* is a predicate that is defined over domains $Org \times O \times OC$ (where OC is a set of contexts that are defined over domain O). If o is an object and oc a context over object o then $occurs\_dynamic\_abs\_object(org, o, oc)$ specifies that context oc is satisfied over object o.
The corresponding dynamic abstract object dao_{oc} is implicitly assigned to object o as follows:

$$assign\_object(o, dao_{oc}) \leftarrow occurs\_dynamic\_abs\_object(org, o, oc)$$

Notice that $DAS \subseteq AS$, $DAA \subseteq AA$ and $DAO \subseteq AO$ where DAS is the set of activated dynamic abstract subjects, DAA is the set of activated dynamic abstract actions and DAO is the set of activated dynamic abstract objects. AS denotes the set of all abstract subjects (i.e. activated dynamic abstract subjects and statically defined abstract subjects), AA the set of all abstract actions and AO the set of all abstract objects.

Dynamic abstract subjects (but not objects nor actions) have been implemented in at least one commercially available system: CA's Embedded Entitlement Manager (CA-EEM) [9].

8 Policy rules revocation

The need for policy revocation Most access control models such as RBAC or OrBAC only specify role assignment to subjects and rarely specify revocation, which in practice is performed by administrators at local sites. We explicitly introduce the notion of revocation in CABAC in order to make it possible to revoke an abstract subject from a subject (respectively an abstract action from an action and an abstract object from an object) statically or dynamically. The notion of session is used within the RBAC model by which a user, after the authentication phase, activates the different roles that are necessary to perform specific tasks. However, neither RBAC nor OrBAC model revocation of already assigned roles from subjects. This notion is useful and should be taken into account by modern access control models. For instance, in a dynamic system that faces intrusions, some subjects, which play roles of *http* servers, may be attacked. In such cases, the deployed security policy should be modified as long as the threat remains present. As a solution, some prohibitions or obligations such as shutting down *http* servers (meaning that all *http* servers should be stopped) could be performed. This may be done by adding the following abstract rule using the above specification

$$obligation(LRSI\_Lab, http\_server, shut\_down, http\_daemon, http\_threat)$$

Using the above rule, all *http* servers within the LRSI_Lab would be shut down when one of them is attacked. A more conservative solution would be to stop only the attacked server. This can be done by revoking the attacked server (let us call the attacked server *marasai*) from *http* abstract_subject.
$revoke\_subject(LRSI\_Lab, marasai, http\_server)$

However, as the abstract subject assignment is performed either manually by the administrator or automatically (using dynamic subjects), the revocation may also be done manually by the administrator (as in the above specification) or automatically based on contexts. After revoking the web server *marasai* from the abstract subject *http_server*, one can add a concrete rule allowing us to stop the *marasai http_server* without generalizing the action of stopping all *http_servers*. The concrete obligation rule to stop the attacked server may be specified as follows:

$$obligation(LRSI\_Lab, marasai, killall, http\_process\_Unix\_id, http\_threat)$$

Notice that one should verify that the corresponding dæmon process is actually stopped. In addition, further steps of verification may be recommended if other functionalities are threatened by the attack. In this case, a more in depth verification process should be performed for further actions that allow us to reduce (or stop definitively) the impact of the threat. This will be discussed in forthcoming papers.

Specifying revocation in CABAC Specifying revocation is quite similar to specifying the assignment of abstract entities to concrete entities. We define two different methods for specifying revocation. The first method is static and consists in revoking the corresponding abstract entity from the concrete entity. The second method is dynamic as in the above example.

In CABAC, the revocation of an abstract subject from a concrete subject is performed using the ternary relation predicate *revoke_subject* that is defined over domains $Org \times S \times AS$. If org denotes an organization, s a subject and as an abstract subject, then $revoke\_subject(org, s, as)$ means that in organization org, subject s is revoked from the abstract subject as.

Static revocation of abstract actions from concrete actions and of abstract objects from concrete objects can be specified similarly, and is left as an exercise for the reader.

However, as shown in the previous example, a modern access control model should be flexible enough to evolve according to the current status of the controlled system. This situation has motivated us to introduce the notion of dynamic revocation within the CABAC model.

The dynamic revocation of abstract entities from concrete entities can be also specified using first order logic taking advantage of contexts expression, meaning that the corresponding rules are updated according to the current context with relation to the security policy that is defined dynamically.

A dynamic revocation of an abstract subject from a concrete subject is expressed using the relation predicate $revoke\_subject(org, s, as)$ that is dynamically activated using the same relation predicate *occurs_dynamic_abs_subject* defined in Section 7. As for dynamic abstract subjects assignment, the revocation of the corresponding abstract subject das_{sc} is implicitly revoked from subject s as follows:

$$revoke\_subject(org, s, das_{sc}) \leftarrow occurs\_dynamic\_abs\_subject(org, s, sc)$$

In the same manner, we can express dynamic revocation of abstract actions (resp. abstract objects) from concrete actions (resp. concrete objects):

$$revoke\_action(org, a, daa_{ac}) \leftarrow occurs\_dynamic\_abs\_action(org, a, ac)$$

$$revoke\_object(org, o, dao_{oc}) \leftarrow occurs\_dynamic\_abs\_object(org, o, oc)$$

9 Specifying constraints and organizational policies

Constrained CABAC adds the concept of constraint to the basic model presented in the previous sections. The concept of constraint is very important in the CABAC model, as it must be in any access control model that has to specify accurate policies according to security requirements that are defined at the organizational level. For example, one can find in practice constraints on the cardinality of roles within an organization. Consider the constraint specifying that only one individual (concrete subject) within an organization is authorized to be department head. This is a cardinality constraint over a role, or an abstract subject in our terminology. It can be specified in CABAC as follows :

$$\forall s \in S, \forall s' \in S, (assign\_subject(CS\_Department, s, department\_head)$$
$$\wedge (CS\_Department, s', department\_head)) \rightarrow s = s'$$

Other constraints such as *separation of duties* and *role cumulation* are also considered in our model. Notice that these constraints are taken into account within the RBAC model [21]. However, while RBAC considers only constraints over roles, in constrained CABAC we can specify constraints over abstract actions and abstract objects. As an example of a constraint over abstract objects, we may express a policy that says that two different abstract objects such as a network device and local equipment in organization CS_Department cannot be assigned the same concrete object:

$$\forall o \in O, \forall o' \in O, (assign\_object(CS\_Department, o, net\_device)$$
$$\wedge assign\_object(CS\_Department, o', local\_equipment)) \rightarrow o \neq o'$$

Similarly, we may specify constraints over abstract subjects or abstract actions.

Also, we may express other constraints such as separation of duty policies, as shown in the following example:

$$SoD(Hospital, doctor, \{diagnose, operate\}_1, patient)$$

meaning that in organization Hospital, abstract_subject doctor is allowed to perform only one action, either diagnose or operate, on a patient.

Similarly, we may specify that in organization Hospital a doctor can consult at most two abstract objects in a set of three:

$$Limit(Hospital, doctor, consult, \{medical\_record, emergency\_record, salary\_record\}_2)$$

By using similar concepts, we can specify other organizational policies such as Chinese Wall.

Notice that constraint specifications in addition to permissions, obligations and privileges may lead to inconsistencies. These inconsistencies should be detected and then resolved. We have already discussed a method to detect similar anomalies using typing systems [2]. However, this method is limited to permissions and prohibitions. More general methods are being investigated and will be discussed in future work.

10 Comparison with RBAC and OrBAC

In this section, we complete our brief comparison of the CABAC model with the RBAC and OrBAC models. Since we know that OrBAC is an extension of RBAC, most of the comparison will be with OrBAC, although RBAC will also be mentioned. The comparison is based on some basic examples that will show the advantages of CABAC, most importantly a more synthetic style. We focus our examples on the usefulness of dynamic entity specifications and hierarchy directions. Other elements of comparison can be found in the examples of the previous sections.

10.1 Example 1: Compact specification style

Let us consider the following three policy rules:
Security technicians and security managers that are in the security datacenter are allowed to:

- perform secured configure command on all security devices
- use the emergency telephone line
- write a report about the physical status of the security devices.

In the example, Serge will be a security technician in organization CA and Bob will be a security manager.

10.1.1 Specification of the policy with the CABAC model

The above security policy is specified as follows in CABAC:

– $assign\_subject(CA, Serge, technician)$;
meaning that Serge plays the role of technician in organization CA,

– $assign\_subject(CA, Bob, manager)$;
meaning that Bob is a manager in organization CA,

– $\forall s \in S$ where S is the set of subjects in organization CA (all staff);

– $occurs\_dynamic\_abs\_subject(CA, s, SecTM\_inSecDC)$
$\leftarrow (assign\_subject(CA, s, Sec\_Manager) \vee assign\_subject(CA, s, Sec\_Technician))$
$\wedge in\_DC(CA, s, Sec\_Datacenter)$;
meaning that in organization CA, the context "SecTM_in SecDC" is true for s if either s is a security manager or s is a security technician and s is in the Security Datacenter of organization CA.

In CABAC, it is sufficient to specify the rule that assigns dynamically the dynamic abstract subject *Dynamic_configurer* to subject s as follows:

$\forall s \in S$;
$assign\_subject(CA, s, DynConfigurer\_SecTMinSecDC))$
$\leftarrow occurs\_dynamic\_abs\_subject(CA, s, SecTMinSecDC)$

After this, each of he above three rules can be directly specified in a CABAC rule as follows:

$-permission(CA, DynConfigurer\_SecTMinSecDC, SecuredConfigureCommand, sec\_device, default)$
$-permission(CA, DynConfigurer\_SecTMinSecDC, use, emergency\_line, default)$
$-permission(CA, DynConfigurer\_SecTMinSecDC, write, report, default)$

Note that "default" corresponds to the default context which is always true.

In conclusion, to specify our example we need in CABAC one rule for dynamic assignment and one rule for each permission.

10.1.2 Specification of the policy with OrBAC model

In OrBAC, the above policy can be defined as follows:

$-$ $empower(CA, Serge, technician)$,
$-$ $empower(CA, Bob, manager)$,

These two rules correspond to the first two rules in the previous section. We then need six rules for the permissions:

$-$ $permission(CA, Sec\_Technician, SecuredConfigureCommand, sec\_device, inSecDC)$
$-$ $permission(CA, Sec\_Technician, use, emergency\_line, inSecDC)$
$-$ $permission(CA, Sec\_Technician, write, report, inSecDC)$
$-$ $permission(CA, Sec\_Manager, SecuredConfigureCommand, sec\_device, inSecDC)$
$-$ $permission(CA, Sec\_Manager, use, emergency\_line, inSecDC)$
$-$ $permission(CA, Sec\_Manager, write, report, inSecDC)$

More in general, n natural language rules such as the ones defined above can be specified in CABAC with $n + m$ formal rules, where m is the number of rules that handle the dynamic assignment of abstract subjects to concrete subjects ($m = 1$ in our example). By contrast, $r \times n \times m$ rules are necessary using the ORBAC or RBAC models, where r corresponds to the number of roles in consideration. In our case $r = 2$ since two roles: security technician and security manager are considered in this policy.

Therefore, the CABAC formalism is much more synthetic and aligned with the intuitive requirements than the RBAC or OrBAC formalisms.

10.2 Example 2: UP and DOWN inheritance

Let us consider the hierarchy example presented in Section 6.2 (Figure 2) and assume that we have the following policy rules:

- Technicians are prohibited to perform unsecured configure command on firewalls in all circumstances.

- Technicians are permitted to perform secured configure command on firewalls in all circumstances.

Also this security policy specifies that all abstract actions on the paths from configure command to secured configure command within the hierarchy of Figure 2 are authorized, otherwise it would be not possible to perform the secured configure command at the bottom of the hierarchy. However, it is forbidden to perform the unsecured configure command in all circumstances. Therefore, abstract action unsecured configure command is forbidden to be performed by Technicians whereas all other abstract actions are allowed to be performed by security technicians. This is what exactly should be specified by the security policy.

10.2.1 Specification of the policy with the CABAC model

The policies of Example 2 are modeled as follows in CABAC:

H_ConfigureCommand Hierarchy specification
A sub abstract action (senior abstract action) is specified using the predicate *sub_abstract_action*:
– $sub\_abstract\_action(CA, H\_ConfigureCommand, GUIConfigureCommand, ConfigureCommand)$;
meaning that in organization CA, $GUIConfigureCommand$ is a sub abstract action of $ConfigureCommand$ within hierarchy $H\_ConfigureCommand$.

The different derivations of hierarchies of Figure 2 corresponds to the following statements:
– $sub\_abstract\_action(CA, H\_ConfigureCommand, GUIConfigureCommand, ConfigureCommand)$
– $sub\_abstract\_action(CA, H\_ConfigureCommand, ConfigureCommandLine, ConfigureCommand)$
– $sub\_abstract\_action(CA, H\_ConfigureCommand, DeviceManagerConfigureCommand, GUIConfigureCommand)$
– $sub\_abstract\_action(CA, H\_ConfigureCommand, WebInterfaceConfigureCommand, GUIConfigureCommand)$
– $sub\_abstract\_action(CA, H\_ConfigureCommand, SecuredConfigureCommand, WebInterfaceConfigureCommand)$
– $sub\_abstract\_action(CA, H\_ConfigureCommand, SecuredConfigureCommand, DeviceManagerConfigureCommand)$
– $sub\_abstract\_action(CA, H\_ConfigureCommand, SecuredConfigureCommand, ConfigureCommandLine)$
– $sub\_abstract\_action(CA, H\_ConfigureCommand, UnsecuredConfigureCommand, WebInterfaceConfigureCommand)$
– $sub\_abstract\_action(CA, H\_ConfigureCommand, UnsecuredConfigureCommand, ConfigureCommandLine)$

The exact security policy of Example 2 is defined by using the following rules:
– $prop(CA, permission, H\_ConfigureCommand, UP)$
– $prop(CA, prohibition, H\_ConfigureCommand, DOWN)$
– $permission(CA, Technician, SecuredConfigureCommand, FW, default)$
– $prohibition(CA, Technician, UnsecuredConfigureCommand, FW, default)$

10.2.2 Specification of the policy in OrBAC and RBAC

In the RBAC or OrBAC models, there is DOWN propagation. Let us see what this means for our example, concentrating first on OrBAC and its specification language. We can try to specify the policy of Example 2 as follows: First of all, we specify the hierarchy of Fig. 2 by using statements such as:
– $sub\_activity(CA, GUIConfigureCommand, ConfigureCommand)$
– $sub\_activity(CA, ConfigureCommandLine, ConfigureCommand)$
– ...
The policy rules can be then specified:
(i)$permission(CA, Technician, ConfigureCommand, FW, default)$
(ii)$prohibition(CA, Technician, UnsecuredConfigureCommand, FW, default)$
Note that, however, from rule *(i)* and from the default DOWN propagation in the configure command hierarchy, technicians will be allowed to perform the unsecured configure command since the unsecured command view derives all the privileges from its junior abstract actions ($ConfigureCommand \rightarrow ConfigureCommandLine \rightarrow UnsecuredConfigureCommand$ is such an hierarchy propagation from $ConfigureCommand$ to $UnsecuredConfigureCommand$). Hence, we derive *rule (iii)* from *rule (i)* and the hierarchy as follows:
(iii)$permission(CA, Technician, UnsecuredConfigureCommand, FW, default)$
This is not what is specified in the original security policy set and in fact rule (ii) is opposite

to rule (iii). Another solution could be to define all privileges on each view within the hierarchy but this does not solve the issue since the privileges are always propagated down and hence can generate undesired conflicts. RBAC, on the other hand, can specify hierarchies only on roles. In other words, Example 2 appears to be out of the reach of OrBAC or RBAC.

11 Conclusion

We have introduced in this paper CABAC, a new access control model that allows expressing policies at different levels of abstraction. CABAC extends previous models, in particular RBAC and OrBAC, by making it possible to describe policies in terms of combinations of abstract and concrete subjects, objects and actions. We have shown how temporal, spatial, knowledge, and historical contexts, as well as combinations of them, can be specified in CABAC (Section 5). Mechanisms for policy propagation through abstraction hierarchies are available, and are presented in Section 6. In addition, the model provides the ability to assign dynamically abstract entities to concrete entities (Section 7). This allows specifying dynamic policies that can change according to the current state of the environment. By these characteristics, CABAC generalizes and formalizes what has been implemented in at least one leading commercial access control tool, CA-EEM (also called CA-eIAM) of CA Technologies. In practical terms, CABAC offers the ability to specify individual and environment-depending policies as exceptions, thus simplifying policy management by the security officer. Furthermore, the CABAC model includes primitives to automatically revoke privileges and automatically reassign them (Section 8). This makes it possible to plan for responses to dynamically appearing and disappearing intrusions and threats, responses that are usually implemented manually in present systems. To our knowledge, no other model combining all such characteristics has been described in the literature. The formal semantics of the CABAC model is specified in first-order predicate logic.

Future work includes investigating the policy conflicts (also called inconsistencies) that can be generated in CABAC, as well as methods for their resolution. An easily seen example of such conflicts is the case where a security officer expresses a permission and a prohibition for a subject to perform an action on an object. Although obvious instances of such conflicts are probably rare, hidden conflicts can arise by means of the mechanisms provided by the model. In [2], a typing system is presented to detect inconsistencies between permissions and prohibitions within security policies that are expressed using a CABAC formalism simpler than the one we present in this paper. Propositions for solving conflicts in access control policies were also presented in [3, 6, 7, 8, 12]. Related work on managing inconsistencies in the presence of obligations may be found in [18]. However, our model with its flexible syntax has the potential of generating many types of inconsistencies particularly when dealing with dynamic assignment and revocation of entities that can create new policies or delete existing policies in real time.

Moreover, we plan to introduce a formal language for expressing recommendations and faculties. Recommendations are used for expressing policies to warn network and computer administrators about new available software patches, for example. We also plan to investigate the inclusion in CABAC of a formal model for delegation, a very useful but also dangerous mechanism for transferring privileges. Delegation may also lead to conflict situations, which in their turn will have to be detected and resolved.

12 Acknowledgments

This research was funded in part by grants of NSERC-CRD and of CA Technologies. The authors are indebted to the anonymous referees for suggestions that have led to the improvement of the paper.

References

[1] A. AbouElKalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *Proceedings of IEEE 4^{th}*

International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), pages 120–134, Lake Come, Italy, June 2003.

[2] K. Adi, Y. Bouzida, I. Hattak, L. Logrippo, and S. Mankovskii. Typing for conflict detection in access control policies. In *E-Technologies: Innovation in an Open World. Proc. of the 4th Intern. Conf. MCETECH 2009*, pages 212–226. Springer, May 2009.

[3] S. Benferhat, R. ElBaida, and F. Cuppens. A stratification-based approach for handling conflicts in access control. In *SACMAT2003*, pages 189–195, 2003.

[4] E. Bertino, P. A. Bonati, and E. Ferrari. Trbac: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3):191–223, August 2001.

[5] E. Bertino, B. Catania, M. L Damiani, and P. Perlasca. Geo-rbac: a spatially aware rbac. *ACM Transactions on Information and System Security (TISSEC)*, 10(1), February 2007.

[6] E. Bertino, S. Jajodia, and P. Samarati. Supporting Multiple Access Control Policies in Database Systems. In *IEEE Symposium on Security and Privacy*, pages 94–107, 1996.

[7] Y. Bouzida. Managing security rules conflicts. European Patent Number 07 114 047.9, August 2007.

[8] Y. Bouzida. Online security rules conflict management. European Patent Number 07 114 046.1, August 2007.

[9] Computer Associates. Computer Associates Embedded Entitlement Manager (CA-EEM). http://www.ca.com/us/products/product.aspx?id=5423, 2009.

[10] J. Crampton and H. Khambhammettu. Delegation in Role-Based Access Control. In *11th European Symposium on Research in Computer Security (ESORICS'2006)*, pages 174–191, September 2006.

[11] F. Cuppens, N. Boulahia-Cuppens, Y. Bouzida, W. Kanoun, and A. Croissant. Expression and deployment of reaction policies. In IEEE, editor, *SITIS Workshop "Web-Based Information Technologies & Distributed Systems (WITDS)*, 2008.

[12] F. Cuppens, N. Cuppens-Boulahia, and M. BenGhorbel. High Level Conflict Management Strategies in Advanced Access Control Models. *Electr. Notes Theor. Comput. Sci.*, 186:3–26, 2007.

[13] F. Cuppens and A. Miège. Modelling contexts in the Or-BAC model. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, pages 416–427, Las Vegas, Nevada, USA, December 2003.

[14] D. F. Ferraiolo and R. Kuhn. Role-Based Access Controls. In Z. Ruthberg and W. Polk, editors, *Proceedings of the 15th NIST-NSA National Computer Security Conference*, pages 554–563, Baltimore, MD, 13-16 October 1992.

[15] V. Weissman J. Y. Halpern. Using First-Order Logic to Reason about Policies. In *16th IEEE Computer Security Foundations Workshop (CSFW2003)*, 2003.

[16] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, 1997.

[17] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, January 2005.

[18] H. Kamoda, M. Yamaoka, S. Matsuda, K. Broda, and M. Sloman. Access control policy analysis using free variable tableaux. *IPSJ Digital Courier*, 2:207–221, 2006.

[19] B. Lampson. Protection. In *5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, March 1971.

[20] J. Park and R. Sandhu. The UCON-ABC Usage Control Model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1), February 2004.

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