MotOrBAC 2: a security policy tool

Fabien Autrel, Frédéric Cuppens, Nora Cuppens-Boulahia, Céline Coma-Brebel

▶ **To cite this version:**

Fabien Autrel, Frédéric Cuppens, Nora Cuppens-Boulahia, Céline Coma-Brebel. MotOrBAC 2: a security policy tool. SARSSI'08 : 3ème conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information, Oct 2008, Loctudy, France. hal-03093665

HAL Id: hal-03093665
https://hal.science/hal-03093665v1

Submitted on 4 Jan 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MotOrBAC 2: a security policy tool

Fabien Autrel (fabien.autrel@telecom-bretagne.eu)*
Frédéric Cuppens (frederic.cuppens@telecom-bretagne.eu)*
Nora Cuppens (nora.cuppens@telecom-bretagne.eu)*
Céline Coma (celine.coma@telecom-bretagne.eu) *

Abstract: Given the growing complexity of information systems and the difficult task of security policy enforcement, system administrators need simple and powerful security management tools. This article presents the second version of MotOrBAC, a tool that embeds policy specification and administration in a same model. MotOrBAC is able to simulate and analyze a security policy specified using the *OrBAC* model. It also implements the *AdOrBAC* model which is used to administrate *OrBAC*.
This article presents why the *OrBAC* model has been chosen to implement such tool. We also present the *OrBAC* application programming interface used by the new version of MotOrBAC and how it can be integrated into other applications to enforce an *OrBAC* policy.

Keywords: acces control, security policy model, security policy administration, implementation, *OrBAC*

1 Introduction

As information systems are getting more and more complex, security administrators must face multiple problems related to configuration and security. Those systems generally use wired and wireless connectivity and various hardware which run multiple OSes on which several applications are ran.

In such a context the specification and enforcement of a security policy is a very tedious, complex and error-prone task. Eventually each hardware (a firewall for instance) and software (SELinux for instance) security component configuration are managed manually. This requires learning several configuration languages and writing configuration files for each component.

Some administration tools exist to help in the configuration of components but they are almost only targeted at network components. For example Firewall Builder[Firb] or Firestarter[Fira] help the administrator by making easier the specification of security rules. However when it comes to system and application security, the list of tools is drastically shortened.

This article presents MotOrBAC version 2, a tool which as been developed to write security policies. MotOrBAC provides multiple functionalities such as (1) policy specification based on the *OrBAC* model [KBB+03], (2) potential and effective conflict detection, (3) policy simulation and (4) administration policy specification. MotOrBAC has been developed on top of the *OrBAC* application programming interface (API), a java API

* GET-Télécom-Bretagne, 35576 Cesson Sévigné (France)

we have developed to ease the integration of our *OrBAC* implementation. The previous version of MotOrBAC was developped using java and prolog whereas the new version is written in pure java. The new version is also more modular as it uses the *OrBAC* API.

MotOrBAC aims at giving the user the possibility to specify all his/her security requirements independently of its enforcement. To do so MotOrBAC implements the *OrBAC* model which specifies the security requirements at the organizational level. Each security component can be represented as a sub-organization of the organization representing the information system that manages the sub-part of the global policy associated with this component.

Once the policy specification at the organizational level is done, concrete entities corresponding to the information system users, actions and objects can be introduced. This way the policy designer can simulate the security policy by checking the concrete security rules inferred by MotOrBAC.

Centralizing the expression of the security policy offers a framework to analyze its consistency. Since the *OrBAC* model allows the expression of positive (permission) and negative (prohibition) privileges, conflicting security rules can be introduced. MotOrBAC can detect those conflicts and help the policy designer solve them.

Most of current security models make the hypothesis that only one administrator will write and maintain the information system security policy. As those systems become more and more distributed, this hypothesis is no longer adapted. MotOrBAC implements the *AdOrBAC* [CM03a] administration model which uses the concepts developed in *OrBAC* (thus making *OrBAC* a self-administrated model). Using *AdOrBAC*, policy administration rights can be distributed over several roles. When the *AdOrBAC* mode is activated, the current policy designer must authenticate himself/herself and then the corresponding administration policy is applied.

This article is organized as follows: Section 2 motivates the need for a centralized specification and management tool. Section 2.2 briefly introduces the *OrBAC* model and presents the main functionalities of MotOrBAC. Section 3 explains how the tool can be used to write a security policy. Section 4 presents the concrete policy simulation tool. Section 5 explains how the conflict detection functionality can be used to solve conflicts. Section 6 presents the *AdOrBAC* implementation. Section 7 gives more details on the MotOrBAC architecture and explains how the *OrBAC* API can be integrated into an application to enforce an *OrBAC* policy. Finally section 8 concludes this paper and presents some future evolutions.

2 The need for an administration tool

2.1 Context

Nowadays private and public organizations face several problems when they try to specify and enforce the security policy of their information systems. To illustrate some of those problems, let us take a look at an hypothetical organization. Let *WorldCompany* be a company having several subsidiary companies (*FranceCompany*, *EnglandCompany*, etc...) and subcontractors (*TaiwanSubContractor*). Several administrators (John, Peter and Rayan) are in charge of the information systems. We suppose that all the subsidiary companies work on the same product and that their hierarchies and workmanship are similar.

WorldCompany would like to define a consistent security policy which applies to all its subsidiary companies. However the task is complicated by the different countries' legislations, the policy must be adapted for each subsidiary company. As this adaptation is going on, *WorldCompany* realizes that one of its administrators, namely Peter, has designed weak security rules which result in a potential information leak. Since *WorldCompany* does not know if the not very good work done by *Peter* is due to a malicious behavior or a lack of information, decision is taken to limit his administrative privileges. Moreover subcontractors need access to the information system of *WorldCompany* and the associated privileges may be modified as time passes. Besides those aspects, the security policy of *WorldCompany* shall be modified as new people are hired, old employees retire or the law is changed (for example in France the weekly working time as been modified from 39 hours to 35 hours).

Those problems are not specific to our example and can be encountered in any company that wants to enforce a security policy. Some of those problems are related to a company infrastructure change, others are linked to errors in the policy specification, others may be related to the centralized information and some are related to administrative rights specification. As a matter of fact it is very difficult for a system administrator to have a global view of the security policy in order to manage it correctly. The use of a single software to manage the security policy would simplify a lot their task and solve many problems. However such a tool would be useless if it does not implement a model which allows to express the security problems and administrative needs of a company such as *WorldCompany*. Such a model should make the management of several entities having various administrative modes (multiple administrators, contextual, centralized, etc...) simple and should allow to verify the policy consistency.

After reviewing many policy models, we came to the conclusion that the *OrBAC* model fits the specifications of the aforementioned tool.

2.2 The *OrBAC* model

OrBAC aims at modelling a security policy centered on the organization which defines it or manages it. Hence a company is an organization but security components such as firewall or the java virtual machine security manager can also be modelled as organizations. An *OrBAC* policy specification is done at the organizational level, also called the abstract level, and is implementation-independent. The enforced policy, called the concrete policy, is inferred from the abstract policy. This approach makes all the policies expressed in the OrBAC model reproducible and scalable. Actually once the concrete policy is inferred, no modification or tuning has to be done on the inferred policy since it would possibly introduce inconsistencies. Everything is done at the abstract policy specification level. The inferred concrete policy expresses security rules using subject, actions and objects. The abstract policy, specified at the organizational level, is specified using *roles*, *activities* and *views*.

The OrBAC model uses a first order logic formalism with negation. However since first order logic is generally undecidable, we have restricted our model in order to be compatible with a stratified Datalog program [Ull89]. A Datalog program must not use any functional terms and must only include range restricted variables (i.e variables that are in the conclusion of a rule must also appear, not negated, in the rule premise). Negated literals can appear in a rule premise if the rule can be stratified. A stratified Datalog

program has all its rules ranked: if some rules contain negative literals then the rules defining those literals are evaluated first. A stratified Datalog program can be evaluated in polynomial time.

In the rest of this article all the security rules defining a security policy must correspond to a stratified Datalog program. We use a Prolog-like notation[1] where terms beginning with an upper case are variables and terms beginning with a lower case are constants. The fact $parent(john, jessica)$. says that *john* is a parent of *jessica*. A rule such as $grandparent(X, Z) : -parent(X, Y), parent(Y, Z)$. means that X is a grandparent of Z if Y exists such that X is a parent of Y and Y is a parent of Z.

Using this formalism, each organization specifies its own security rules. Some *role* may have the permission, prohibition or obligation to do some *activity* on some *view* given an associated *context* is true. The *context* concept[CM03b] has been introduced in $OrBAC$ in order to express dynamic rules. Those security rules are represented using 5-ary predicates:

- $permission(org, role, activity, view, context)$ means that in organization org, role $role$ is authorized to perform activity *activity* on view *view* if context *context* is true.
- the *prohibition* and *obligation* predicates are similarly defined but express different security requirements.
- $permission(hospital, nurse, consult, medical_record, urgency)$ means that nurses can access the patients medical records in the context of an emergency.

Security rules can be hierarchically structured so that they are inherited in the organization, role, activity and view hierarchies (see [CCBM04]). Since a security policy can be inconsistent because of conflicting security rules (for example a permission can be in conflict with a prohibition), it is necessary to define strategies to solve those conflicts. Section 4 presents the way we solve this problem.

Once the security policy has been specified at the organizational level, it is possible to test it by assigning concrete entities to abstract entities. To do so, three ternary predicates have been defined to assign a subject to a role, an action to an activity and an object to a view:

- $empower(Org, Subject, Role)$: specifies that in organization Org, subject $Subject$ is empowered in role $Role$.
- $consider(Org, Action, Activity)$: specifies that in organization Org, action $Action$ implements activity $Activity$.
- $use(Org, Object, View)$: specifies that in organization Org, object $Object$ is used in view $View$.

For example, the fact $empower(hospital, john, surgeon)$ states that *john* is empowered in the role *surgeon* in the *hospital* organization.

Contexts are defined through logical rules which express the condition that must be true in order for the context to be active. In the OrBAC model such rules haveaffected to the predicate *hold* in their conclusion:

[1] Note that Motorbac users do not have to write such rules thanks to the GUI

- $hold(Org, Subject, Action, Object, Context)$: specifies that in organization Org, subject $Subject$ does action $Action$ on object $Object$ in context $Context$.

Using this model, concrete security rules applying to subject, actions and objects can be inferred. The rules and principles used to infer the concrete security policy are explained in section 5.

2.3 Motorbac

In order to allow administrators to use the $OrBAC$ model, we have developed the MotOrBAC prototype[2]. This tool aims at making easy the use of the $OrBAC$ model to express a security policy. Its architecture is presented on figure 1. The architecture and specification of the first MotOrBAC implementation are presented in [CCBC06]. MotOrBAC uses the

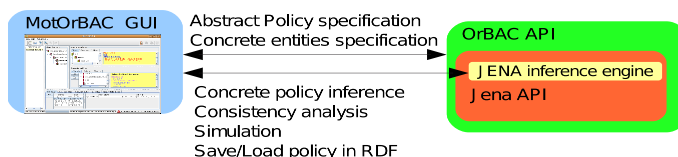

Fig. 1: The MotOrBAC tool architecture which is based on the java $OrBAC$ API

$OrBAC$ application programming interface (API) to manage the policies displayed in the graphical user interface (GUI). The $OrBAC$ API can be used to programmatically create $OrBAC$ policies. More details about the $OrBAC$ API are given in section 7. MotOrBAC can be used to do several tasks on $OrBAC$ security policies:

- Edit policies: the administrator can create the abstract entities he/she needs (organizations, sub-organizations, roles, activities, views, contexts) and the abstract security policies (see section 3).
- Policy simulation: after having specified concrete entities (subjects, actions and objects), the concrete policy can be inferred. Subjects, actions and objects can have attributes (see section 4).
- Policy consistency verification: abstract conflicts between abstract rules can be detected (see section 5).
- Once abstract conflicts have been detected, MotOrBAC is able to suggest the administrator some solutions to solve them (see section 5).
- Administrative rights management: the administrative rights of a subject or a role can be specified in order to decentralize the policy administration (see section 6).

3 Security policy specification

Specifying an $OrBAC$ security policy requires defining several abstract entities and their hierarchical relationships.

---
[2] http://motorbac.sourceforge.net

3.1 *OrBAC* organizational entities in MotOrBAC

When the administrator in charge for the policy specification wants to specify the organization hierarchy, he/she can either specify all the organizations then specify the hierarchical links between them, or he can specify the hierarchical links as he enters them. A hierarchical link between two organizations org_1 and org_2 is recorded into the policy by inserting the following fact:

$sub_organization(org1, org2)$

The organization hierarchy is displayed as a tree control in MotOrBAC, clicking on an organization in this tree displays the policy specified for this organization (roles, activities and views, as well as their hierarchies, and the security rules). Clicking the root of this tree displays the entire security policy. Note that it is possible to specify that an organization inherits from several organizations.

After the organizations have been defined, the administrator can define the roles, activities, views and contexts for each organization. For example if the role $head_nurse$ is defined in organization $hospital$ as a super-role of $nurse$, the following objects are inserted into specific views [CBCC07]:

- The role $head_nurse$ is inserted into the $role_view$ view of organization $hospital$, which is represented by the fact $use(hospital, head_nurse, role_view)$
- The role hierarchy object $RH_hospital_nurse_head_nurse$, instance of class $role_hierarchy_class$ (see section 4 for more details on classes in MotOrBAC), is inserted into the view $role_hierarchy$ of organization $hospital$: $use(hospital, RH_hospital_nurse_head_nurse, role_hierarchy)$. This object has three attributes to define the hierarchy:
  - $RH_hospital_nurse_head_nurse.authority = hospital$
  - $RH_hospital_nurse_head_nurse.senior_role = head_nurse$
  - $RH_hospital_nurse_head_nurse.junior_role = nurse$

Note that since the role hierarchy object is inserted into the $role_hierarchy$ view of an organization, the created role hierarchy is only defined in this organization. This way it is possible to create different hierarchies with the same roles in different organizations. Note that multiple inheritance can be specified as for organizations.

The $activity_hierarchy$ and $view_hierarchy$ views as well as the $activity_hierarchy_class$ and $view_hierarchy_class$ classes are similarly defined to create activities and views hierarchies.

As abstract entities are defined by the policy designer, he/she can introduce contraints that some entities must respect. If a modification which violates one or more constraints is attempted on the policy, the modification is discarded. In the $OrBAC$ model, constraints are expressed by rules which infer the nullary $error$ predicate. For example a role separation constraint, which states that a subject cannot be affected to two roles at the same time in two organizations, is defined as follows:

$$\begin{aligned} error :- \quad & separated_role(Org1, Role1, Org2, Role2), \\ & empower(Org1, Subject, Role1), \\ & empower(Org2, Subject, Role2). \end{aligned}$$

Similarly we define activity, view and context separation constraints. Separation constraints define symetric and anti-reflexive relationships between abstract entities. They

can be simply specified by the administrator in the interface without knowledge of its internal representation as inference rules. Section 5 explains how separation constraints can be used to solve policy inconsistences. MotOrBAC implements those separation constraints but does not allow the user to define its own rules inferring the *error* predicate. More details about the inference engine used by MotOrBAC are given in section 7.

Once the organizations, roles, activities, views, contexts have been defined, the administrator can specify the abstract permissions, prohibitions and obligations in the corresponding parts of the GUI. When adding a rule, it is defined in the currently selected organization. For example the following rule states that the *student* role is permitted to access some teaching resources in the context of a economic class project:

$$permission(hospital, student, consult, teaching_resource, eco_projetc_ctx)$$

Security rules are represented by objects inserted into a specific view called *license*. Three classes are defined and represent the three different security rule types: *license_class* for permissions, *inhibition_class* for prohibitions and *commitment_class* for obligations. When an object is inserted into this view, it is interpreted according to its type. Those classes are also used by the *AdOrBAC* implementation when specifying the administration policy.

3.2 Inheritance

The *OrBAC* model specifies the automatic inference of privileges given entities hierarchies. An entity inherits the security requirements expressed on its super entities. Several inheritance mechanisms exist in *OrBAC*:

- **Abstract entities hierarchy inheritance:** given an organization, if a security rule applies to an abstract entity e (role, activity or view), then the sub-entities of e inherits the security rule. For example the following rule expresses how permissions are inherited through role hierarchies:
  $permission(Org, Sub_role, Activity, View, Context) :-$
  $\quad senior_role(Org, Super_role, Sub_role),$
  $\quad permission(Org, Super_role, Activity, View, Context).$

  Similar rules exist for activities and views hierarchies. The *senior_role* relationship is anti-symetric, reflexive and transitive (the same applies for the *senior_activity* and *senior_view* relationships).

- **Separation constraints inheritance:** separation constraints are inherited the same way as security rules:

  $separated_role(Org1, Sub_role, Org2, Role2) :-$
  $\quad senior_role(Org, Super_role, Sub_role),$
  $\quad separated_role(Org1, Super_role, Org2, Role2).$

  Separation constraints are symetric and anti-reflexive relationships:
  $separated_role(Org1, Role1, Org2, Role2) :-$
  $\quad separated_role(Org2, Role2, Org1, Role1).$

- **Organization hierarchy inheritance:** security rules are inherited through the organization hierarchy. The following rule defines how permissions are inherited, the same applying for prohibitions and obligations:

$permission(Sorg, Role, Activity, View, Context) :-$
$sub_organization(Sorg, Org),$
$use(Sorg, Role, role_view),$
$use(Sorg, Activity, activity_view),$
$use(Sorg, View, view_view),$
$use(Sorg, Context, context_view),$
$permission(Org, Role, Activity, View, Context).$

This rule states that if a (1) *Sorg* is a sub-organization of *Org*, (2) *Role* is a role in organization *Sorg*, (3) *Activity* is an activity in organization *Sorg*, (4) *View* is a view in organization *Sorg*, (5) *Context* is a context in organization *Sorg* and (6) *Role* is permitted to do *Activity* on *View* when *Context* is valid in organization *Org*, then the permission is also true in organization *Sorg*.

- **Context definition inheritance:** Let *org* and *subOrg* be two organizations, *subOrg* being a sub-organization of *org*, *ctx* a context and ctx_def_org its definition in organization *org*. *subOrg* inherits the definition ctx_def_org if no definition exists for context *ctx* in *subOrg*.

4 Simulation

MotOrBAC can infer a concrete policy given an *OrBAC* abstract policy *P* specification and the set of concrete entities assigned to the abstract entities of *P*. Concrete entities are represented as objects being instances of classes. Each instance has an identifier and a list of attributes and values (see figure 3).

Attributes are represented using binary predicates in the *OrBAC* model. For example the predicate $diploma(s_1, doctor)$ expresses the fact that s_1 has an attribute called *diploma* which value is *doctor*. The fact that some object *o* is an instance of some class *c* is represented by $class(o, c)$. For instance $class(peter, doctor_class)$ expresses that peter is an instance of the $doctor_class$ class.

The MotOrBAC GUI includes a class editor (figure 2). The administrator can create classes and their attributes as well as class hierarchies. Class attributes are inherited in the class hierarchies. After concrete entities have been created the designer can easily modify their properties so that they are instances of some classes (it is possible to specify multiple class inheritance). When a concrete entity has been set as an instance of some classes, its inherited attributes values can be modified (figure 3). The administrator can then affect concrete entities to abstract entities. This can be done manually through a contextual menu or by specifying entity definitions. Entity definitions are conditions evaluated on a concrete entity attributes. Three different types of entity definitions exist: role definition, activity definition and view definition. In the *OrBAC* model, defining an entity definition implicitly creates the abstract entity associated to the entity definition. In MotOrBAC, abstract entities must be defined before creating any entity definition. Entity definitions can have several conditions, at maximum one per organization, which are inherited through the organization hierarchy the same way context definitions are inherited. A concrete entity can be affected to several abstract entity. For example to automatically assign some subjects to a role *doctor* in an organization *hospital*, the administrator can create a role definition. This definition would state that all subjects

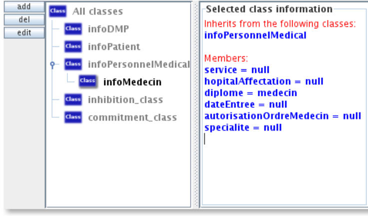

Fig. 2: The MotOrBAC class editor

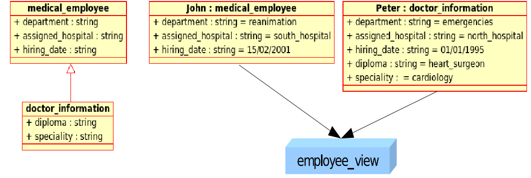

Fig. 3: A class ontology on the left and two $OrBAC$ objects instantiating those classes on the right, inserted into view *employee_view*

which have an attribute *diploma* set to *doctor*. Using the language used in MotOrBAC, the administrator would write $diploma = doctor$ as the role definition for organization *hospital*. In section 3.1, we introduced the separation constraints. Those constraint are evaluated each time a concrete entity is affected to an abstract entity, either manually or through an entity definition.

4.1 Derivation of concrete security rules

Once concrete entities have been defined and assigned to abstract entities, the organizational policy can be applied to infer the concrete policy. The following rule is used to infer concrete permissions, similar rules are defined for prohibitions and obligations:
$is_permitted(Subject, Action, Object, Priority) :-$
$\quad permission(Org, Role, Activity, View, Context, Priority),$
$\quad empower(Org, Subject, Role),$
$\quad consider(Org, Action, Activity),$
$\quad use(Org, Object, View),$
$\quad hold(Org, Subject, Action, Object, Context).$

Note that we introduce a priority in the abstract permission. This priority is used to rank order abstract rules when conflicts are detected. This rule states that if (1) a permission exists in an organization for a role, an activity and a view in some context, (2) a subject is empowered in the role, (3) an action implements the activity, (4) an object is used in the view and (5) the context is active in the organization for the triple $\{Subject, Action, Object\}$, then the subject is authorized to do the action on the object. Note that the concrete security rule has the same priority than the abstract rule.

MotOrBAC can infer the concrete policy and show it in a simulation window to help the administrator in his/her task. Figure 4 shows the simulation window. This window lists the concrete security rules in the upper table (a different color is used for each rule type) and the contexts states in the lower table. Light-colored entries in the concrete policy table shows concrete security rules for which the associated context is inactive. Light-colored entries in the context table represent inactive contexts.

The following context types are currently implemented in MotOrBAC (see [CM03b] for more details on the different types of contexts defined in $OrBAC$): temporal contexts (expressing temporal conditions), user defined contexts (contexts which definitions are set to *true* or *false*) and prerequisite contexts (expressing conditions on the concrete entities' attributes). MotOrBAC also implements context composition, allowing the administrator to express complex contexts. The upper part of the simulation window can be used to

Fig. 4: The MotOrBAC concrete policy simulation window

Fig. 5: Concrete conflicts tab. The abstract rule name from which each concrete is inferred is displayed

set the current simulation date. The main interface can be used to modify the concrete entities attributes so that some contexts state may change.

If the abstract security policy contains some incoherences, conflicting concrete security rules specifying that some subject is at the same time permitted and prohibited from doing the same action on an object may be inferred. MotOrBAC can display the concrete conflicts (figure 5) but does not allow the administrator to solve them at the concrete level. The next section explains how such conflicts can be avoided at the abstract level so that no conflict can exist at the concrete level.

5 Analysis

5.1 Managing conflicts

The simulation function presented in the previous section can infer all the concrete conflicts. This section presents how abstract conflicts can be managed with MotOrBAC (see [CCBG07] for a more in-depth view of abstract conflict management). The following rule is used to infer conflict at the organizational level:

$conflict :-$
$permission(Org1, Role1, Activity1, View1, Context1, Priority1),$
$prohibition(Org2, Role2, Activity2, View2, Context2, Priority2),$
$not(separated_roles(Org1, Role1, Org2, Role2)),$
$not(separated_activities(Org1, Activity1, Org2, Activity2)),$
$not(separated_views(Org1, View1, Org2, View2)),$
$not(separated_contexts(Org1, Context1, Org2, Context2)),$
$not(Priority1 < Priority2),$
$not(Priority1 > Priority2).$

This rule states that if (1) a permission and a prohibition exist at the organizational level, (2) the roles $Role1$ and $Role2$ or activities $Activity1$ and $Activity2$ or views $View1$ and $View2$ or contexts $Context1$ and $Context2$ are not separated or (3) the rules priorities cannot be compared, then potentially concrete conflicts can exist. Actually if the same concrete entities $subject$ is empowered in $Role1$ and $Role2$, $activity$ implements $Activity1$ and $Activity2$, $object$ is used in $View1$ and $View2$, then two concrete conflicting rules can be inferred.

If the $conflict$ predicate cannot be inferred, the policy is said to be *consistent*. If the policy is consistent, the administrator can assign concrete entities to abstract entities

without worrying about concrete conflicts since they cannot exist[CCBG07].

5.2 Solving conflicts with MotOrBAC

MotOrBAC can help the administrator solve the abstract conflicts. The detected abstract conflicts are listed in the GUI and separation constraints as well as rule priorities can easily be added through a contextual menu (figure 6). The contextual menu gives the administrator several choices:

- **Add a separation constraint:** depending on the two conflicting rules parameters, the administrator can choose to separate at most the rules roles, activities, views and contexts. Figure 6 shows an example of conflict where the administrator cannot separate the contexts since they are the same in the two conflicting rules.
- **Order rules:** the administrator as always the choice to modify the priorities between rules when processing a conflict. However some rules might become redundant or inapplicable. This article does not tackle the complex problem of detecting those anomalies, see [GACC06] for an example of algorithm given for a network security policy.

The administrator has two other choices not listed in the contextual menu:

- **Modify the conflicting rules:** the conflict might be caused by an error in the rules specifications.
- **Ignore the conflict:** the administrator can deliberately ignore the conflict but might introduce concrete conflicts when assigning concrete entities to abstract entities.

Fig. 6: Abstract conflicts tab. Each couple of conflicting rules is displayed. The contextual menu shows the choices the administrator has to solve the conflict

6 AdOrBAC

6.1 View-based administration

Generally access control models are constituted of two distincts models: a model to express the security policy and another model to express its administration policy. The administration policy expresses which users have privileges to update the security policy and under which conditions they have those privileges. For instance the *ARBAC* model [SBM97, SM99] has been specified to manage the *RBAC* [FSSGC01, SJLE96] access control model.

The *OrBAC* model also has its administration model, called *AdOrBAC* [CM03a]. *AdOrBAC* has been designed to express an administration policy using the same concepts introduced in the *OrBAC* model. This makes the *OrBAC* model a self-administrated model. *AdOrBAC* has been designed to be more expressive than *ARBAC*. For instance *AdOrBAC* can be used to express that in some company *global.com*, each department leader has the permission to choose his/her team leader. Such administration rule is difficult to express in *ARBAC* since it would require to specify as many permissions as the number of teams. The base idea is to consider all administrative operations as insertions of objects into specific views (such as the *role_hierarchy_view* introduced in section 3) using the *use* predicate. Figure 7 shows the *AdOrBAC* administrative views. The following activities can be administrated using these views (see [CBCC07] for more details):

- Permissions management (*license* view, in which instances of the *license_class* are used). The *inhibition_class* and *commitment_class* are used respectively to specify administrative prohibitions and obligations.
- Concrete entities assignment (*role_assignment* and *activity_assignment* views)
- Abstract entities hierarchy management (*role_hierarchy*, *activity_hierarchy*, *view_hierarchy* and *context_hierarchy* views)

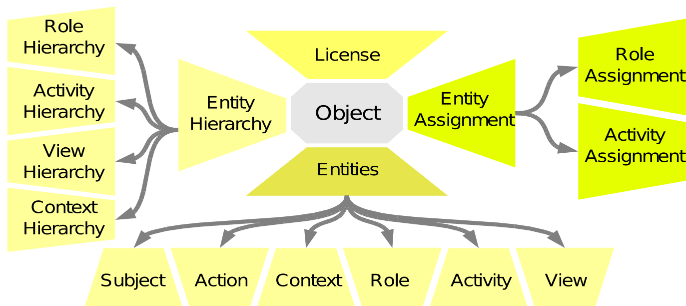

Fig. 7: The *AdOrBAC* views. The administrative policy specifies the operations permitted on those views

6.2 *AdOrBAC* in MotOrBAC

MotOrBAC includes an administration function implementing the *AdOrBAC* model [GCCBB07]. When the *AdOrBAC* function is activated in MotOrBAC, after the user has authenticated himself/herself, he/she can edit the *OrBAC* security policy accordingly to the *AdOrBAC* policy. If an unauthorized operation is attempted, the policy is not modified and the user is informed. In [CBCC07] we have shown that *AdOrBAC* can be used to manage a decentralized administration policy with several administrators having restricted administrative privileges.

The delegation example given in section 6.1 can be specified in *AdOrBAC* as follows: first a sub-view of the *role* view must be created, for example the *role_leader* sub-view.

Then we put a constraint on this view so that the *role_assignment* instances used in this view assign subjects to the *team_leader* role:

$use(global.com, RA, role_leader) :-$
$assignment(RA, role_leader),$
$authority(RA, global.com).$

then we can give the *departement_leader* role the right to insert role assignment objects in this view: $permission(global.com, departement_leader, insert, role_leader, default_context)$ where *insert* is an administrative operation consisting in inserting an object into a view. Hence this rule means that the role *departement_leader* can always (the default context is used in this rule) insert instances of the *role_assignment* class in the *role_leader* view.

Since the *OrBAC* model is self-administrated, a MotOrBAC policy file contains both the *OrBAC* policy and its associated *AdOrBAC* policy.

7 Integration

The concrete security policy inferred by MotOrBAC can be translated to configure a security component such as a firewall for example [CCBSM04]. Another solution to enforce an *OrBAC* security policy is to use the *OrBAC* Java API, on top of which MotOrBAC is built. This API uses the *Jena*[3] java library to represent an *OrBAC* policy as a RDF graph. It can be used to load MotOrBAC RDF policies and interpret them, i.e access control requests can be made on a loaded policy. *Jena* Features an inference engine which is used by the *OrBAC* API to infer the concrete policies and the conflicts. When an *OrBAC* RDF policy is loaded by the API, the concrete policy can be inferred and stored in memory. An instance of the *OrbacPolicy* java class which encapsulates an *OrBAC* policy uses a cache of concrete security rules to enhance the performances when the policy is queried. Contexts are evaluated upon a query, this feature is actually used in the MotOrBAC simulation tool to show the contexts state. The contexts implementation can be easily extended in order to interface the API with other applications and add new types of contexts.

Integrating the *OrBAC* Java API[4] into a Java application can be done without modifying the application source code. Aspect Oriented Programming (AOP) can be used to separate security concerns from other concerns relative to the application. We have developped an application which simulates a medical form where entries must be filed by subjects having specific roles and rights. This little application does not include any security related code. Using *AspectJ* we weaved the security concerns with the *OrBAC* API.

The API can also be used to create *OrBAC* policies, it can be for instance integrated into a web server to communicate with a web browser which could run a web interface to create and/or administrate *OrBAC* policies.

8 Conclusion

In this article we showed that the expressivity of MotOrBAC can be easily used by system administrators to specify dynamic security policies and administrate them. The tool

[3] http://jena.sourceforge.net/

[4] The full specification of the *OrBAC* API can be found on http://orbac.org

implements the *OrBAC* access control model and features (1) a security model based on organizational entities (organization, role, activity, view, context), (2) an explicit separation between the organizational level and the concrete level (subject, action, object), (3) the possibility to model permissions, prohibitions and obligations, (4) a simulation and analysis tool, (5) a policy conflict detection tool along with resolution strategies, (6) a decentralized policy administration.

The administrator can directly specify a policy with MotOrBAC from its expression in the *OrBAC* model. The GUI is structured around the organization tree control to allow the administrator to easily specify the abstract policy. The concrete entities are specified using an object-oriented approach. The policy is internally represented as a RDF graph to which the administrator has no direct access, all the editing operation being done through the GUI. The underlying RDF graph can be saved as an XML file or as a N-Triple[5] file so the policy can be easily read by other applications. The policy administration, which is known to be a very complicated task, is included in MotOrBAC. Using the *OrBAC* concept of view and object, administrative privileges are specified by inserting administrative objects into specific views. This model being self-administrated, specifying the administration policy is done using the same concepts involved in the specification of the security policy, which does not require the administrator to learn a separate administration model. Moreover several administrators can use MotOrBAC to manage the same security policy with more expressiveness than the *ARBAC* model. The privileges given to a subject assigned to an administrative role are activated once the subject has authenticated himself/herself using the GUI.

The security policy enforcement can be done by translating the concrete security policy to other languages used to configure security components. The work did in [CCBSM04] to translate a concrete security policy to a firewall configuration file has been implemented in MotOrBAC.

The problem of delegation [GCCBB07] was not tackled in this article as it is currently under implementation in MotOrBAC.

Several extensions to the current MotOrBAC implementation are currently under development, namely other types of contexts [CM03b], additional policy translation modules to configures other kinds of security components than firewalls, an interoperability module, a management and inforcement of obligations module and a module implementing usage control.

References

[CBCC07] N. Cuppens-Boulahia, F. Cuppens, and C. Coma. Multi-granular licences to decentralize security administration. In *First International Workshop on Reliability, Availability, and Security (WRAS). Paris, France*, 2007.

[CCBC06] F. Cuppens, N. Cuppens-Boulahia, and C. Coma. MotOrBAC : un outil d'administration et de simulation de politiques de sécurité. In *First Joint Conference on Security in Networks Architectures (SAR) and Security of Information Systems (SSI). Seignosse, France*, 2006.

[5] http://www.w3.org/TR/rdf-testcases/

[CCBG07] F. Cuppens, N. Cuppens-Boulahia, and M. Ben Ghorbel. High-level conflict management strategies in advanced access control models. In *Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 186, pp. 3-26*, 2007.

[CCBM04] F. Cuppens, N. Cuppens-Boulahia, and A. Miège. Inheritance hierarchies in the Or-BAC model and application in a network environment. In *Second Foundations of Computer Security Workshop (FCS'04). Turku, Finland*, 2004.

[CCBSM04] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège. A formal approach to specify and deploy a network security policy. In *Second Workshop on Formal Aspects in Security and Trust (FAST)*, 2004.

[CM03a] F. Cuppens and A. Miège. Administration model for Or-BAC. In *International Federated Conferences (OTM'03), Workshop on Metadata for Security. Catania, Sicily, Italy, November 3-7*, 2003.

[CM03b] F. Cuppens and A. Miège. Modelling contexts in the Or-BAC model. In *19th Annual Computer Security Applications Conference, Las Vegas*, 2003.

[Fira] Firestarter. http://www.fs-security.com/.

[Firb] FirewallBuilder. http://www.fwbuilder.org.

[FSSGC01] David F.Ferrailo, Ravi Sandhu, D.Richard Kuhn Serban Gavrila, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. In *ACM Transactions on Information and System Security*, 2001.

[GACC06] J. Garcia-Alfaro, F. Cuppens, and N. Cuppens. Towards filtering and alerting rule rewriting on single-component policies. In *Computer Science, 4166, Conference on Computer Safety, Reliability, and Security (Safecomp), Gdansk, Poland*, 2006.

[GCCBB07] M. Ben Ghorbel, F. Cuppens, N. Cuppens-Boulahia, and A. Bouhoula. Managing delegation in access control models. In *15th International Conference on Advanced Computing and Communication (ADCOM), Guwahati, India*, 2007.

[KBB+03] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarteand A. Miège, C. Saurel, and G. Trouessin. Organization based access control. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks (Policy 2003), Lake Come, Italy, June 4-6*, 2003.

[SBM97] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. In *ACM Trans. Inf. Syst. Secur.*, 1997.

[SJLE96] Ravi S.Sandu, Edward J.Coyne, Hal L.Feinstein, and Charles E.Youman. Role-based access control models. In *Computer*, 1996.

[SM99] Ravi Sandhu and Qamar Munawer. The ARBAC99 model for administration of roles. In *15th Annual Computer Security Applications Conference (ACSAC99)*, 1999.

[Ull89] Jeffrey D. Ullman. Principles of database and knowledge-base systems. In *Computer Science Press*, 1989.