UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

DÉTECTION DES CONFLITS DANS LES POLITIQUES DE CONTRÔLE D`ACCÈS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

Seifallah Yakine Layouni

Août 2010

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

Ce mémoire intitulé :

DÉTECTION DES CONFLITS DANS LES POLITIQUES DE CONTRÔLE D`ACCÈS

Présenté par

Seifallah Yakine Layouni

Pour l'obtention du grade de maître ès science (M.Sc.)

Évalué par un jury composé des personnes suivantes :

Dr. Luigi Logrippo . . . . . Directeur de recherche
Dr. Kamel Adi . . . . . .Co-directeur de recherche
Dr. Karim El Guemhioui .. . . . . Président du jury
Dr. Alain Charbonneau . . . . . Membre du jury

Abstract

Controlling access to resources is a very important requirement in a variety of systems. Access control mechanisms associate rights of access between resources and identities through sets of policies. As security systems become larger and more complex, the risk to create conflicts between the various policies, protecting a given resource, can be very high. For this reason, we have decided to work on methods to automate the validation process of such sets of policies. In this thesis we study the validation of access control systems using a formal verification tool: the Alloy Analyzer. We first specify access policies using the first-order logic Alloy language and then we use the Alloy Analyzer tool to detect conflicts between them.

By using this technique, we have implemented the SEMPO (SEMantic POlicies) tool to detect inconsistencies for an industrial tool, CA's EEM. The SEMPO tool extracts policies from XML files (in EEM format) and translates them into the Alloy language. Then the Alloy Analyzer (used as an API in SEMPO) validates the set of access control policies and warns of any logic violations found in the set.

Résumé

La sécurisation des accès aux ressources privées est un enjeu très important pour les entreprises. Ceci n'est faisable qu'en déployant un système de contrôle d'accès. Ces systèmes permettent d'associer des droits d'accès aux utilisateurs pour permettre l'accès aux ressources des entreprises en question. Mais la complexité croissante des systèmes de contrôle d'accès modernes augmente considérablement le risque d'avoir des incohérences dans l'ensemble des politiques d'un système. Puisque les systèmes de contrôle d'accès modernes contiennent généralement des grands nombres de politiques, alors nous avons décidé d'étudier les techniques pour automatiser le processus de validation. Dans ce mémoire, on étudie la validation des politiques de contrôle d'accès formellement en utilisant l'outil Alloy. Les politiques sont spécifiées dans le langage Alloy, et l'outil permettra de détecter les conflits via les assertions.

En utilisant cette technique, nous avons implémenté l'outil SEMPO (SEMantic POlicies) pour la détection des conflits dans les politiques d'un outil de contrôle d'accès industriel, l'EEM de la compagnie CA. Cet outil permet d'extraire les politiques de sécurité depuis EEM (sous forme XML) et de les transformer vers une spécification dans le langage Alloy. L'analyseur Alloy sera alors utilisé pour compiler la spécification et afficher aux administrateurs tout conflit détecté.

TABLE OF CONTENTS

CHAPTER 1: ACCESS CONTROL STRUCTURE

1.1. Overview

This chapter is dedicated to the motivation of our work (section 1.3) and to define the basic concepts that can be found in access control policies and their relationships. This will be useful to better understand how access control systems work. We also present different types of policies, such as delegation and obligation policies respectively in sections 1.5.5 and 1.5.6.

1.2. Introduction

An access control system consists of a set of policies that define how users may behave inside a system and how they may interact with the resources (objects). Its role is to protect the resources of the security system from unauthorized access. In general, such policies may specify that some users can, or cannot, have access to some resources.

Many older access control systems can only express policies that grant subjects access permissions to perform actions inside the system. If a subject does not have explicit permission, access is denied. These positive policies are only adequate to express simple security goals. But complex cases of real life may need more expressive policies. This is why modern security systems allow administrators more options in order to express explicit negative policies and constraints. On the other hand, if we can express both positive and negative policies in the same system, this may lead to embarrassing problems since policies might be conflicting. Some example will be discussed later in this thesis.

1.3. Motivation

Complex environments, involving large sets of access control policies, usually have many security policies that are expressed in complex ways, including positive and negative authorizations. Therefore, the more complex a security policy is, the bigger is the risk to have conflicts in the policies.

Generally the conflicts occur during the assignment of policies to subjects. For example, a new policy that assigns a new access right or a requirement added by the administrator could conflict with a previously existing policy. This is why after any modification of a policy set we have to revalidate it to be sure that the new modification does not cause conflicts. We call this a static detection of conflict.

In this thesis, we recommend a solution to detect conflicts in sets of policies. Our solution is based on validation of access control policies by using a formal language called Alloy.

There is a great number of definitions of access control methods and policies in the literature For this reason, we should define some basic terms that we will use in the rest of the thesis. If a policy declares that a user can have access to an object, it is called a "positive authorization" or "permission". If a policy states that a user cannot have access to an object, it is called a "negative authorization" or "prohibition". The word "authorization" is used to describe positive or negative authorization. Other terminology can be found in the literature such as "right", "privilege" or "role" as well as "denial" or "negative permission", but we will not use it.

In this chapter we also define briefly the main entities of access control policy.

1.4. Intended contribution

We are planning to develop a tool to assist the security administrator of an enterprise, who is responsible to manage the policy database, to detect and repair inconsistencies, incompleteness and other related problems in the database. This tool will be called SEMPO, SEMantic POlicy analyzer.

The following guidelines will be kept in mind in our work:

1. The approach used in developing this tool should be applicable in principle to access control systems of any type
2. The validation process should be automated

3. The solution should use an existing model checker, such as Alloy, to perform the verification
4. The tool should provide a detailed trace of detected conflicts

1.5. Model entities

In this section we present an access control model that is consistent with the one of EEM (Embedded Entitlement Manager), the industrial tool that has motivated our work and which will be discussed later. This tool is based on the well-known concepts of subject, action and object used in most access control systems. It builds on this basis a more expressive model using new concepts like filters, delegations and obligations policies.

1.5.1. Subject

In general, subjects are entities that can perform actions in a system (active entities). The meaning of the entity "Subject" changes from one security system to another. In EEM a subject (called Identity) can be a human or a program. It is the entity that can perform actions, delegation and more generally use the system. In other models, such as [10] any subject can operate both as a subject and as an object in a given security system. In modern access control systems, a subject can be either a user or a group of users. For example, subjects can be users such as Bob, Mary, John…

In EEM, the set of actions represents the set of actions that a subject can perform on the objects (which represent the resource of the system). Usually, the actions correspond to elementary commands. Actions can be executed according to user policies. In EEM we cannot specify the fact that a transaction is composed of several tasks. The system should allow the execution of the complete transaction or deny it. However in other systems like TBAC [21] (Task Based Authorization Control, which is widely used in commercial transactions systems) and RBAC [7, 8] (Role Based Access Control), administrators can make abstractions and use composite actions (activities). The latter contain sequences of atomic actions.

1.5.2. Group

A group is a set of users that share some characteristics. This notion may simplify the design of policies. For example Bob and Jeff may belong to the user group Doctor. If so, we can use only one policy, such as "doctors can write in the medical record of patients" instead of two, one for each doctor.

1.5.3. Object

Objects are passive entities. They are the resources that must be protected by the access control system. To perform actions on these resources, users should have policies that allow them access to these resources.

Note that in EEM we can associate attributes to subjects or objects and these attributes permit to give further information about the latter. A simple example of an access control policy based on attributes is a subject requesting access to a medical record in the emergency room (the attribute *assignement_zone* should be equal to *emergency*). In this case the subject has to provide his assignement_zone attribute to access the medical record. In this example, subject and object specific attributes are used directly to compute the access decision. Different application domains require differents set of attributes. The attributes have an important role in the design of filters (constraints) in EEM.

1.5.4. Attributes

Attributes permit to add information about the entities that play a role in a given policy. This is very important in order to design more expressive policies. EEM provides this concept to express dynamic policies and groups. For example, a policy may declare: "Doctors working in the emergency room can admit patients". To build this policy, first we have to assign an attribute "working_zone" for Doctors and determine its possible values (the name of the offices where they work). Second, we add a constraint (filter) saying that the *working_zone* attribute must be equal to emergency.

1.5.5. Delegation policy

Delegation is a special type of policy by which a user can empower other users to perform new actions. When a delegation policy is applied in a system, functions or authorizations are transferred from a user to another user, who did not have them. In this way, this latter user can act in another user's behalf. Delegation is a critical feature in access control systems. However, administrators have to be careful when they specify delegation policies, because such policies can have major side effects inside the system. For example, suppose that an administrator delegates grant/revoke actions to a technician. This implies that the technician can now revoke some authorizations from administrators, or change his own user group (technician) to administrator user group.

1.5.6. Obligation policy

An obligation policy starts the execution of a set of actions when a given event occurs inside the system. This is a very important kind of policy in the access control domain, because through it we can signal many problems, such as violation of policy conditions etc. For example, an obligation policy can state that the actions revoke/grant can be executed only by administrators, otherwise the system will send an email to the administrator.

1.5.7. Constraint

In modern security systems, security policies cannot be expressed only by sets of simple authorizations without taking into account constraint enforcement. In this way, policies become more complex but also more expressive. The concept of constraint is very important because it allows us to build policies which are activated while the information evolves inside the system. In EEM a security policy is not defined just by subject, object and action but also depends on constraints which should be satisfied to get access. Constraints are expressed as conjunctions and disjunctions of primitive constraints like $c_1 \wedge c_2 \wedge \ldots \vee c_i$ where $\wedge$ is the "and" operator

and ∨ is the "or" operator. Negation (¬) can also be used. The EEM tool provides a powerful interface to express constraints where the administrator can insert parentheses as well as operators and options. In EEM we can use simple constraints (a test on the attributes) and temporal constraints.

1.6. Conclusion

In this chapter, we have presented the basic concepts of modern access control systems. We presented the entities that administrators use to build their access control policies. These concepts simplify the management and the structure of policies and explain the relationship between the different components needed to build security policies. Other security systems may provide different structures, but each of them has some shortcomings. For example in the EEM tool it is not possible to express policies like the separation of duty requirement (SOD), while IBAC (Identity Based access control) doesn`t support constraints.

CHAPTER 2: EXPRESSING ACCESS CONTROL POLICIES IN FIRST ORDER LOGIC

2.1. Overview

In this chapter we describe a first order formal logic notation for access control policies. We also investigate some types of conflicts. With this notation we can efficiently determine if actions are permitted or prohibited by policies and requirements. We illustrate our work with examples to show different kinds of conflicts.

This chapter is structured as follows. Section 2.3 introduces the formal notation of an access control policy (permission and prohibition). In section 2.4 we discuss different types of conflicts and we show how to detect them. Direct conflict, incompleteness and violation of requirements are discussed respectively in section 2.4.1, section 2.4.2 and section 2.4.3.

2.2. Introduction

This chapter discusses the representation of policies and conflicts in a logical formalism in order to better understand their characteristics and create a formal notation for them.

The logical formalism should be sufficiently powerful to express in an easy and natural way the access control policies that are the subject of this research. We shall see that first order logic is sufficiently flexible to specify access control policies. This logic will enable us to specify these policies in the Alloy language, which is a first order language that is associated with a logic analysis tool also called Alloy (section 2.5 will be dedicated to Alloy).

2.3. Formalization

Policies describe the conditions under which subjects are allowed or prohibited to perform actions, such as reading or writing on resources. In very general terms, the policies we are interested in are of the form:

c → permission to perform operation on resource

or

c → prohibition to perform operation on resource

where c is a condition. For example, consider the following policy P:

- ✓ P: all nurses working in the emergency room can read patients' medical records inside the emergency room

This policy means that users who want to read a patient's medical record in the emergency room should satisfy the following condition c:

c: the user belongs to the Nurse group and works in the emergency room

c→ Permission to read the medical record

If c is true (satisfied) the user will have the permission to read the medical record otherwise the policy cannot be applied.

Unfortunately, in real life, policies are often expressed informally (in human language form), and in many cases their meaning and consequences are not clear. We believe that without a precise formal specification of access control policies:

- We cannot tell if the access policies are correct.
- We cannot tell if the policies we build do what we want

Precise specifications allow us to perform a rigorous analysis of the policies in order to find possible inconsistencies or conflicts. Formal specification holds the key for the detection of possible conflicts present in access control systems.

Let us see how we can provide a formal notation for access control policies. The following is the first order logic policy pattern that we will use in this chapter:

$$(P(S,A,R) \rightarrow (\neg)\ \textbf{Permitted}\ (S',A',R'))$$

Where P(S,A,R) is a first order formula (also called condition predicate) composed of conjunctions, disjunctions or negations of elementary predicates on elements of S (subjects), A (actions), and R (resources). Similarly, S',A',R' are

respectively sets of subjects, actions and resources for which permissions or prohibitions are expressed. This pattern can involve quantifiers such as ∀ on elements of the sets S, A, R.

The notation (¬) Permitted (S',A',R') indicates that the predicate Permitted can be negated to express a prohibition. Without the ¬ operator, the Permitted predicate expresses permission.

To demonstrate how we can express policies in first order logic, let us consider the following example:

Example 1:

- P1: Software programmers can`t read nor write the financial folder
- P2: Financial employees can write in the financial folder
- P3: Any employee working in Section A can read the financial folder

The first policy applies (the prohibition of read and write applies) iff the user belongs to the group of software programmers and the resource is the financial folder. The condition of P2 can be understood in a similar way. The condition of P3 is satisfied iff the work area of the employee is SectionA (named attribute = SectionA). P3 is said to be a dynamic policy because it depends on a condition which is computed when the policy is executed. So each policy includes a condition that must be satisfied to make the policy applicable. In the following we specify P1, P2 and P3 in first order logic notation:

- P1: ∀x,z (User(x) ∧ Group(x,Soft_prog) ∧ Finan_fold(z) → ¬ **Permitted** (x, read or write, z))
- P2: ∀x,z (User(x) ∧ Group(x,Fin_emp) ∧ Finan_fold (z) → **Permitted** (x, write, z))
- P3: ∀x,z (User(x) ∧ Attribute(SectionA, x.Zone) ∧ Finan_fold (z) → **Permitted** (x, read, z))

For each policy, if the conjunction of predicates is satisfied then the user is permitted or not permitted to access resource z.

In the above expressions:

- The predicate User(x) is true if x is a user,

- The predicate Group(x,Soft_prog) can be read as "x belongs to Soft_prog group"
- The predicate Finan_fold(z) represents the type of the resource on which the action will be executed.
- The Attribute(SectionA, x.Zone) predicate can be read "the named attribute Zone of the user x should be equal to SectionA".

2.4. Conflict analysis

2.4.1. Direct contradiction

At first sight, there is no contradiction in Example1, among P1, P2 and P3, because we have different conditions for these policies. However, let us suppose that we have a policy system having the following properties:

User(Bob) ∧ Group(Bob, Soft_prog) ∧ Attribute (SectionA, Bob.Zone) ∧

User(Alice) ∧ Group(Alice, fin_emp) ∧ Attribute (SectionA, Alice.Zone) ∧

User(Christine) ∧ Group(Christine, admin_staff) ∧ Attribute (SectionA, Christine.Zone) ∧

User(Tara) ∧ Group(Tara, admin_staff) ∧ Attribute (SectionA, Tara.Zone)

According to the above instantiations Alice satisfies policies 2 and 3, Christine and Tara satisfy policy 3, and there are no conflicts for these users. But conflict occurs with Bob because his profile satisfies P1 and P3. Satisfying P1 and P3 means that the user is allowed to read but cannot read or write file folder z. In the following we show the authorizations for Bob:

¬ **Permitted** (Bob, read or write, z) ∧ **Permitted** (Bob, read, z)

(¬ **Permitted** (Bob, write, z) ∨ ¬ **Permitted** (Bob, read, z)) ∧ **Permitted** (Bob, read, z)

The logical contradiction is clear, however our method requires to derive an explicit conflict. The following axiom allows to detect a conflict and determines which subjects and resources are involved in the inconsistency: ∀ x,a,z (¬ Permitted (x, a, z) ∧ Permitted (x, a, z) → Conflict).

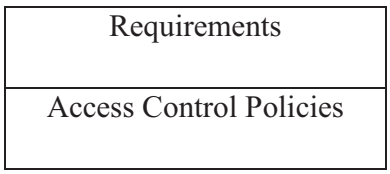
This axiom can be read as "if a user is both allowed and prohibited at the same time to execute an action **a** on resource **z**, then there is a conflict".

2.4.2. Incompleteness with respect to requirements

2.4.2.1. Requirements

Requirements have a very important role to play in access control systems. They provide general information about the system, for example, they can indicate that the only user that can sign for funds is the director. This requirement is available all the time, it doesn`t depend on the state of the system, on who is the director, etc.

Requirements should always be satisfied because they are at a higher level than ordinary policies. So if there is a conflict between requirements and policies, the policies should change and not the requirements.



2.4.2.2. Incompleteness

With the growing complexity of access control systems it becomes very difficult to cover all possibilities of user access. Consequently a security system contains incompleteness if no authorization is specified for some possible requests. As a solution, we integrate security requirements in access control policies to reduce the possibility of incompleteness.

In this section we discuss incompleteness in accordance to requirements. we introduce policies that express general requirements, and if these are not satisfied we can say that our system is incomplete with respect to the requirement.

Let us consider the following example:

Example2:

- P4: All Nurses who work in the emergency room can order new drugs.
- P5: All Nurses who work in the emergency room can update the drugs data base
- RQ1: The actions order and update should be permitted together

In the above example we see that requirement RQ1 is more general than the other policies. It provides information about the atomicity of the set of actions "order" and "update". This requirement is very important in order to keep our system coherent, since the quantity of drugs in the emergency room drug cabinet should always be reflected in the data base. Formally:

- P4:$\forall$x,z (User(x) $\wedge$ Group(x, Nurse) $\wedge$ Attribute(ER, x.Zone) $\wedge$ Data_base(z) $\rightarrow$ **Permitted**(x, order, z))
- P5:$\forall$ x,z (User(x) $\wedge$ Group(x, Nurse) $\wedge$ Attribute(ER, x.Zone) $\wedge$ Data_base(z) $\rightarrow$ **Permitted**(x, update, z))
- RQ1:$\forall$x,z (**Permitted**(x, order, z) $\rightarrow$ **Permitted**(x, update, z)) $\wedge$ $\forall$x,z (**Permitted**(x, update, z) $\rightarrow$ **Permitted**(x, order, z))

Suppose that Karen is a Nurse (User(Karen) $\wedge$ Group(Karen, Nurse)). She can make a request of ordering drugs or making an update on the data base through respectively policies P4 and P5. This can be expressed in the following table:

Policy	Order	Update
P4	T	–
P5	–	T
RQ1: requirement	T	T

The above table shows how policies P4 and P5 are not in accordance with requirement RQ1. So we should replace policies P4 and P5 by a single policy such as the following one:

- $\forall$x,z (User(x)$\wedge$ Group(x, Nurse) $\wedge$ Attribute(ER, x.Zone) $\wedge$ Data_base(z) $\rightarrow$ **Permitted**(x, order, z) $\wedge$ **Permitted**(x, update, z))

We assume that the corrections are made by the administrator, who will take into account the general context of the policies set.

2.4.3. Requirements violation

In this section we discuss conflicts between the two different levels of requirements and policies. We can show that we can detect these conflicts by applying requirements on system instances (appropriately assigning users to policies).

Let us analyze the following example:

Example 3:

- P7: Administrators can read, write, revoke, grant or delete the database
- P8: Technician can read or write the database
- P9: Administrator can delegate his rights to technician.

Until now there is no conflict even if the administrator delegates his right to the technician. But a conflict occurs when we add the following requirement:

- RQ2: The actions revoke and grant can only be performed by administrators.

If a delegation occurs we can see that the technician acquires the actions revoke and grant through policy P9 which contradicts requirement RQ2. In this simple case we can see the conflict directly, but if we had thousands of policies in our system, this could be very difficult to detect.

Let us show the contradiction:

- P7: $\forall$ x,z (User(x) $\wedge$ Group(x, Admin) $\wedge$ Data_base(z) $\rightarrow$ **Permitted**(x,read or write or delete or revoke or grant, z))
- P8: $\forall$ x,z (User(x) $\wedge$ Group(x, Tech) $\wedge$ Data_base(z) $\rightarrow$ **Permitted**(x, read or write, z))
- P9: $\forall$ x1,x2,z (User(x1) $\wedge$ User(x2) $\wedge$ Data_base(z) $\wedge$ **Permitted**(x1,read or write or delete or revoke or grant, z) $\wedge$ Deleg(x1,x2) $\rightarrow$ **Permitted**(x2,read or write or delete or revoke or grant, z))
- RQ2: $\forall$ x,z (User(x) $\wedge$ $\neg$Group(x, Admin) $\wedge$ Data_base(z) $\rightarrow$ $\neg$ **Permitted**(x, revoke or grant, z))

The predicate **Deleg(x1, x2)** can be read as "the user x1 delegates his right to user x2". Suppose that the following formulas are true in the system and the fact that Alex delegates his rights is true:

User(Alex) $\wedge$ Group(Alex, admin) $\wedge$ User(Adam) $\wedge$ Group(Adam, Tech) $\wedge$ Deleg(Alex, Adam)

In accordance to the instances and policies described in P7 and P8, Alex has the right to execute read, write, delete, revoke, and grant. And Adam has the right to read and write the database. Before delegation Adam has the following authorization:

- **Permitted**(Adam, read or write, z)

After delegation Adam is empowered by additional actions which are delete, revoke and grant:

- Adam: **Permitted**(Adam, read or write or delete or revoke or grant, z)

Now let us see if the requirement RQ2 is satisfiable by Adam:

$$\text{User(Adam)} \wedge \neg \text{Group(Adam, Admin)} \wedge \text{Data\_base(z)} \rightarrow \neg \text{Permitted(x,revoke or grant, z)}$$

True True True

Since Adam satisfies the requirement, he is not allowed to perform revoke or grant actions.

$$\underbrace{\text{Permitted(Adam, read or write or delete or revoke or grant, z)}}_{\text{By delegation}} \wedge \underbrace{\neg \text{Permitted(Adam, revoke or grant, z)}}_{\text{By requirement}}$$

After derivation and simplification we conclude that Adam is permitted to revoke and grant and not permitted to revoke and grant which makes the access control system inconsistent. If we need to resolve the problem we have to disable the delegation policy, because we can assume that the requirement RQ2 is more general than delegation and must be always satisfied.

2.4.4. Separation of duty violation

In access control systems, a situation of conflict of interest can be avoided by specifying a separation of duty constraint. Conflicts of interest arise as a result of the simultaneous assignment of two mutual exclusive permissions to the same subject. Mutual exclusive permissions result from the need of separating powerful rights or responsibilities to prevent fraud and abuse. In this section we discuss only static separation of duty.

For example it is common practice to separate the actions create and delete accounts in commercial web sites. In the following, we show an example where a conflict occurs between three types of policies: access policy, delegation policy and separation of duty policy:

Example 4:

- P11: Bob can create account for web users
- P12: Mark can delete account for web users
- P13: If Mark is sick then he can delegate his rights

There is no obvious problem in the above set of policies. However, these policies can be abused by a mischievous user. Suppose that Mark falls ill (Sick(Mark) is true) and then he delegates his rights to Bob. Now Bob is able to create and delete accounts on the commercial web site.

Suppose that a client John sends a request to buy a product online from this website, and then he sends his name, credit card number and his address. Bob will create an account for John, take all this information (of client John) for his own personal purpose and after that he deletes John`s account in the website. Now Bob can use the credit card number for himself without anyone realizing the fraud committed because Bob has deleted the account of the victim.

To avoid the fraud we have to specify separation of duty between the creation and deletion actions as follows:

- P14: Employees can`t have access to both actions create and delete

This way we avoid fraud, but now the question is: are our security policies still coherent? Let us formalize these policies to better analyze them:

- P11: $User(Bob) \wedge Account(z) \rightarrow Permitted(Bob, create, z)$
- P12: $User(Mark) \wedge Account(z) \rightarrow Permitted(Mark, delete, z)$
- P13: $\forall x, \forall z\ User(Mark) \wedge Sick(Mark) \wedge Deleg(Mark,x) \wedge Permitted(Mark, delete, z) \wedge Account(z) \rightarrow Permitted(x, delete, z)$

The separation of duty specifies mutual exclusive actions that must be never assigned to the same subject simultaneously. The nature of this relation is bidirectional that is why P14 is represented by these requirements as follows:

- RQ3:User(x) ∧ Permitted(x, create, z) →¬Permitted(x, delete, z)
- RQ4:User(x) ∧ Permitted(x, delete, z) →¬Permitted(x, create, z)

So Bob can create and delete (policy P11 and P13) and Mark can delete. Let us see if the separation of duty constraint is satisfied or violated for users Mark and Bob:

- If User(Mark): RQ3 is not applicable because the predicate Permitted(Mark, create, z) cannot be derived. RQ4 is satisfied because Mark can only delete the database, so we can say that Mark is not involved in any conflict.
- Bob: P11, P13(delegation), RQ3 and RQ4(P14) are satisfied:
  - P11&P13: Permitted(Bob, create, z) ∧ Permitted(Bob, delete, z)
  - RQ3-RQ4: ∧¬Permitted(Bob, delete, z) ∧¬Permitted(Bob, create, z)

In the above we show the conflict derived from policies described in P11, P12, P13 and P14. We don't speak about automatic correction of this type of inconsistency because this topic requires more research.

2.5. Alloy language and Analyzer

2.5.1. Overview

The Alloy language is designed for the specification of object models through graphical and textual formalisms. It is a state-based language, and invariants can be used to constraint the relationships between objects. Alloy is supported by a constraint analysis facility, which allows us to analyze policy specifications in order to detect any possible conflicts between policies.

The Alloy syntax is based on the Z language, and integrates further concepts that are used in other object modeling notations. Each state component is a set, a binary relation, or an indexed relation. For sets, the Alloy syntax offers the usual set theoretic operators (shown here in ASCII format):

- s + t : union of s and t
- s & t : intersection of s and t

There are no set constants in Alloy. However we have:

- some s : the set s is non-empty
- no s : the set s is empty
- one s : the set s has exactly one element

Elementary formulas in Alloy are made by comparing sets (which can be singletons):

- s = t (equality) : s and t have the same elements
- s in t (subset) : every element of s is an element of t

Alloy has standard quantifiers. If a variable x is part of a formula F we can write:

- all x:s | F : which means that F is true for every value of x in the set s.
- some x:s | F : which means that F is true for some value of x in the set s.

Standard logical operators are also provided, among which:

- && : and
- || : or

An important operator in Alloy is the relational image ".". The expression "s.r" denotes the set of objects that the set s maps to, through relation r. This kind of expression is often called a "navigation expression" because we navigate from s along a relation r. Applying the image operator several times yields a longer navigation.

Suppose, for example, that we have the sets Subject, Obligation, and Action denoting the components of a simple managed system. In addition suppose that the relation "holds" maps the sets of subjects to obligations.

For a given subject s1, the relation s1.holds gives us the set of obligations held by the subject. Next we introduce another relation "requires", which maps an obligation to the required actions:

s1.holds.requires

The above expression denotes the set of all actions required by the obligation of subject s1.

2.5.2. Alloy Analyzer

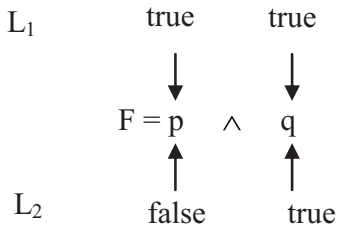
The Alloy Analyzer is a software tool which is used to analyze specifications written in the Alloy language. The Analyzer can generate test scenarios by creating instances of specifications and simulate the interactions between the signatures (objects of the specification) defined. The Alloy analyzer permits also to check whether the properties of the system are violated or not (logic checking). This feature is used to check inconsistencies in access control policies.

2.5.3. Satisfiability

In order to perform logic checking, the Alloy analyser uses satisfiability algorithms.

Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluated to *true*. If no such assignments exist, this means that the function expressed by the formula is *false* for all possible variable assignments. In this latter case, we would say that the formula is unsatisfiable; otherwise it is satisfiable. This problem is also called the Boolean or propositional satisfiability problem, or simply the SAT problem.

Let us see the following example:



We say that the interpretation L_1 satisfies the formula $p \wedge q$, if F = *true* for the assignment provided by L_1. It is necessary that the two variables p and q are *true*. And as we can see the interpretation L_2 does not satisfy the formula. Alloy, if asked to verify this formula, will provide L_2 as a counterexample.

SAT solvers are program that are designed to check satisfiability of Boolean expressions. Alloy finds counterexamples by using a SAT solver.

2.5.4. Reasoning of SAT solver

In order to compute the satisfiability of Boolean formulas, SAT solvers use a marking algorithm. This algorithm computes marks as constraints on all valuations that can make a formula *true*. All marked cases have to be true for any such valuation of the formula. We can extend this idea to general formulas by setting constraints on which its sub formulas require a certain truth value for all valuations that make it true.

SAT solvers require that the input formulas be in Conjunctive Normal Form (CNF). A formula in CNF is a conjunction of disjunctions of Boolean variables. Examples of CNF expressions are:

$A \wedge B$
$(A \vee C) \wedge B$
$(A \vee \neg C \vee B) \wedge (\neg D \vee F \vee E)$

Any Boolean formula can be translated into CNF by an appropriate algorithm, however this algorithm is internal to Alloy and does not concern us.

As mentioned, suppose we have a conjunctive CNF formula ϕ, we can say that ϕ is satisfiable if and only if it is possible to associate a logical (Boolean) value to each variable of ϕ in such a way that ϕ is logically *true*. Generally, the answer to this question also provides an example of such variable assignments.

Suppose this set of variable: $\{ p_1 , p_2, p_3 \}$

And $\phi = (p_1 \vee p_2) \wedge (\neg p_1 \vee p_3) \wedge (\neg p_2 \vee \neg p_1)$

ϕ is satisfiable and logically true if p_1=true, p_2=false and p_3=true

But the same assignment does not satisfy the following:

$\phi' = (p_1 \vee p_2) \wedge (\neg p_1 \vee p_3) \wedge (\neg p_2 \vee p_1) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$.

An obvious way to solve the SAT problem, is to traverse the truth table for the expression.

The SAT problem is NP-complete, which means that the best algorithms known are of exponential complexity [16]. However substantial research has taken place in this area, and now algorithms that are very efficient in most cases are known.

2.5.5. Different stages for validation

The detection of conflicts between access control policies, that we have presented earlier, requires a process to check for consistency of policies. Using Alloy, a possible method to perform this detection is shown in the following diagram. The validation process consists basically of five steps:

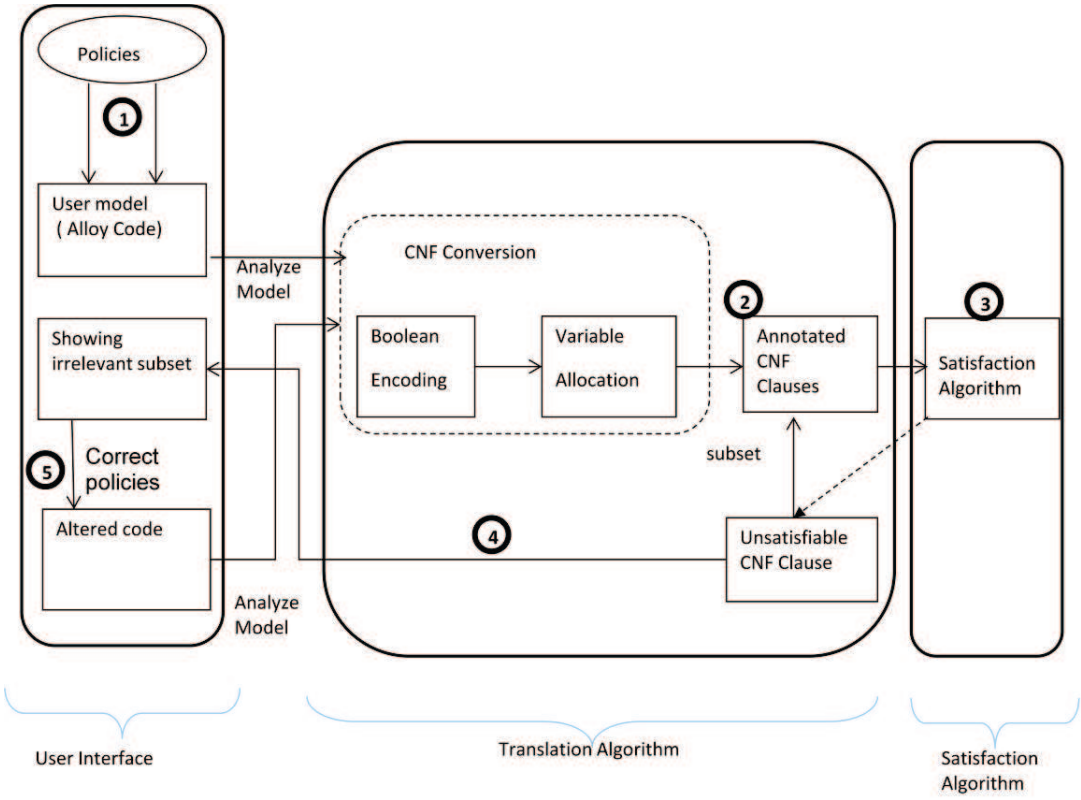


Fig 2.1: Steps of the validation model

- Step 1: A model is created in a constraint language, in our case the Alloy language.
- Step 2: The Alloy tool transforms the previous model into a set of CNF (conjunctive normal form) formula,
- Step 3: A SAT solver determines if the model is satisfiable (see previous section), and extracts the subset of the CNF clauses that is unsatisfiable, if it exists.
- Step 4: The result is mapped back to the original model.

The analysis is now complete, the remaining step is to correct the conflict found in the specification. This step is not carried out automatically by the analyzer, it is a function to be performed by the user.

- Step 5: The user should correct the irrelevant logic expression.

2.6. Conclusion

In this chapter, we have presented a first order logic notation that we consider to be sufficiently expressive for many access control policy applications. In particular, it is sufficient for expressing the policies of the tool EEM that is the main subject of our study. With this notation we can efficiently determine if actions are permitted or prohibited by policies and requirements. We have illustrated our work with examples to demonstrate the formalization of the access control policies and their combinations.

We have also discussed direct conflicts between policies, and conflicts with requirements, with delegation and separation of duty.

Finally, we have presented the essential notions necessary in order to understand our logical method for detecting conflicts and incompleteness in policies. We have discussed satisfaction algorithms, and the tool Alloy that is used in the process. The detection method was outlined.

CHAPTER 3: STATE OF THE ART Conflict Detection Techniques

3.1. Overview

This chapter is dedicated to the Literature Review, where we discuss a collection of conflict detection techniques. We begin with the Free Variable Tableau method at section 3.2, and we describe the detailed process of conflict detection (translation of access control policies to a logic notation and processing the analysis tree). Then we discuss the work done in [18] in section 3.3, which is related to the specification of properties of RBAC96 model. We show how we can specify some properties of the RBAC model in the Alloy language. We don't introduce role hierarchy and permission propagation because these features are not well developed in EEM. Then we introduce the work of [17] in section 3.4, which is related to the validation of access control policies written in the XACML language. We introduce the main architecture of XACML policy enforcement and the conflict resolution mechanism. Section 3.5 provides a formal notation for XACML access control policies, to facilitate their translation to logic expressions, which are then used as input for a SAT Solver. It shown how with this method it is possible to check whether policy properties are respected or not. Section 3.6 proposes a formal notation for access control policies and an approach to handle conflicts in access control policies based on priority assignment. In section 3.5.2 we show how authors specify their policies, section 3.5.3 will be dedicated to the conflict resolution approach.

3.2. Free variable tableaux

In this section we present a conflict detection method based on the use of Free Variable Tableaux, or FVT [14]. It is a static method that permits to detect conflicts in sets of policies and determines which policies conflict. We will give a detailed presentation of the method as described in [14], and at the end we add an extension, which is our own contribution, to support conflicts caused by delegation policies.

3.2.1. Free variable tableaux application

In this section we will describe how the FVT method operates and how it permits to detect conflicts.

In order to be able to say that we have a conflict in a set of policies P, it is necessary to prove that a contradiction $\perp$ can be derived from the set. In order to prove that a statement C results from Γ ($\Gamma \models C$) it can be proved that $\{\Gamma , \neg C\}$ is inconsistent ($\{ \Gamma , \neg C\} \models \perp$). To use this method we need to translate each access control policy into a logic expression. Then the FVT method can be applied to detect contradictions and obtain the information that explains the cause of the conflict.

A translating mapping function Z permits to convert policies from various languages (EEM, XACML...) to logical expressions:

$$Z: P \rightarrow L$$

$$p \rightarrow Z(p) \; (p \in P \text{ and } Z(p) \in L)$$

Where P is a set of policies, p is a single policy and L is a set of logic expressions. After the translation process, conflicts can be detected independently from the language in which the policies are expressed. This makes the Free Variable Tableaux method applicable to validate many access control policy systems.

3.2.1.1. Policy translation

In this section we present how access policies can be translated to logic expressions. We will apply the mapping function Z on different kinds of policies. We begin with the most basic policies which are authorization and obligation policies.

3.2.1.1.1 Access policy

A positive policy (Auth+) specifies that a subject S_1 is authorized to perform an action A_1 on a resource T_1 (called also target). A negative policy (Auth–) specifies that a subject S_1 is prohibited to perform an action A_1 on a resource T_1. Formally:

$$Auth\pm (S_1, T_1, A_1)$$

3.2.1.1.2 Obligation policy

Positive obligation policy (Obli+) specifies that if an event E_1 happens, then the subject S_1 should performs the action A_1 on the resource T_1. The negative obligation policy (Obli–) specifies that if an event E_1 happens then the subject S_1 shouldn't performs the action A_1 on the resource T_1. Formally:

$$Obli\pm (E_1, S_1, T_1, A_1)$$

3.2.1.1.3 Translation

The translation of policies is made by Z function as follows:

$$Z (Auth+ (S_1, T_1, A_1)) := \forall x(E_x \rightarrow P(S_1, T_1, A_1))$$

$$Z (Auth- (S_1, T_1, A_1)) := \forall x(E_x \rightarrow \neg P(S_1, T_1, A_1))$$

$$Z (Obli+ (E_1, S_1, T_1, A_1)) := E_1 \rightarrow O(S_1, T_1, A_1)$$

$$Z (Obli- (E_1, S_1, T_1, A_1)) := E_1 \rightarrow R(S_1, T_1, A_1)$$

In this formalization, P represents the predicate which allows subject S_1 to perform action A_1 on resource T_1. The predicate O represents the fact that subject S_1 must carry out the action A_1 on the resource T_1 and the predicate R specifies that the subject S_1 must not carry out the action A_1 on the resource T_1. E_x represents the event (or the trigger) which causes the activation of the authorization, this event happens when the user decides to send his request. E_1 represent a specific event in the system. If this event is true, the obligation policy is active and the user can exploit it.

Now we need some axioms to link the access policy, positive obligation and negative obligation, because positive and negative obligation policies need an authorization to allow their execution. The axioms are represented as follow:

$$Ax1: \forall s, t, a (O(s, t, a) \rightarrow P(s, t, a))$$

$$Ax2: \forall s, t, a (\neg(O(s, t, a) \wedge R(s, t, a)))$$

The first axiom Ax1 will be useful to detect conflicts between access policies and obligation policies. The second one (Ax2) is used to detect conflicts between positive and negative obligation policies.

3.2.1.1.4 Separation of duty

In this section we show how to translate the separation of duty to logic expressions (the Chinese wall policy and time constraints can be translated in similar ways). Note that we will discuss only the static separation of duty constraint in this chapter.

Separation of duty (soD) has the same effect as the Chinese wall policy except that separation of duty is applied on actions (not on resources). This kind of policy is used to prevent fraud in commercial systems (see chapter 2). In this document, separation of duty can be set for any two actions. In the following we specify the syntax:

$$\text{sod1: soD(all, all, } \{A_1, A_2\})$$

Policy sod1 specifies that any user (all) that can execute A_1 on any resource (all) cannot execute action A_2 on the same resource. The mapping (translation to logic expression) of separation of duty policy is as the follows:

$$Z\,(\text{sod1}) := \forall x, y(\neg(P(x, y, A_1) \wedge P(x, y, A_2)))$$

This means that the execution of actions A1 and A2 on resource y by subject x is always false.

3.2.2. Conflict detection

In this section we will show how the FVT method permits to detect anomalies between policies. We will see how this technique can show which policies are involved in a conflict.

Basically, conflicts appear as combinations of the following types of policies:

$$\{\text{Auth+/Auth–}\},\ \{\text{Obli–/Obli+}\},\ \{\text{Obli+/Auth–}\}\ldots$$

This method proceeds by analyzing the logic expressions and dividing each expression in a single branch of the tree. The divisions are made in accordance with the usual rules of first order logic.

In FVT method, if contradictory predicates can be found on the same path (on the graph, see Fig 3.1) and all paths of the graph are closed (closed paths are indicated by horizontal lines) we can say that there is inconsistency between the input policies. By default, the analysis starts from the premise that the input policies don't conflict, but a conflict can be derived in the following steps. A conflict can only be derived by studying the truth table of the input sentences (to see all Boolean combinations). For example the logic expression: $A \rightarrow B$ is true only if $\neg A$ is true or B is true. As we can see this leads to two branches which represent two possibilities. As another example, suppose that we want to study this expression: $\forall x\ (E_x \rightarrow B)$. In this case E_x can be an infinite set of event (because $E_x \rightarrow B$ is true for all values of x), so for simplicity we have to fix x to x_1, where x_1 represents an instance of x (a specific event). We replace $\forall x\ (E_x \rightarrow B)$ by $(E_{x1} \rightarrow B)$. This reduces the complexity of analysis which otherwise could never stop.

3.2.2.1. Conflict caused by contradiction

Contradiction occurs when the same subject is allowed and prohibited to execute an Action A_1 on the same resource T_1. Let us see the result of analyzing two conflicting policies (between authorization and obligation). The policies are expressed as the follows:

P17: Obli+ (E_1, S_1, T_1, A_1)
P18: Auth– (S_1, T_1, A_1)

The FVT analyzing tree is represented as follows:

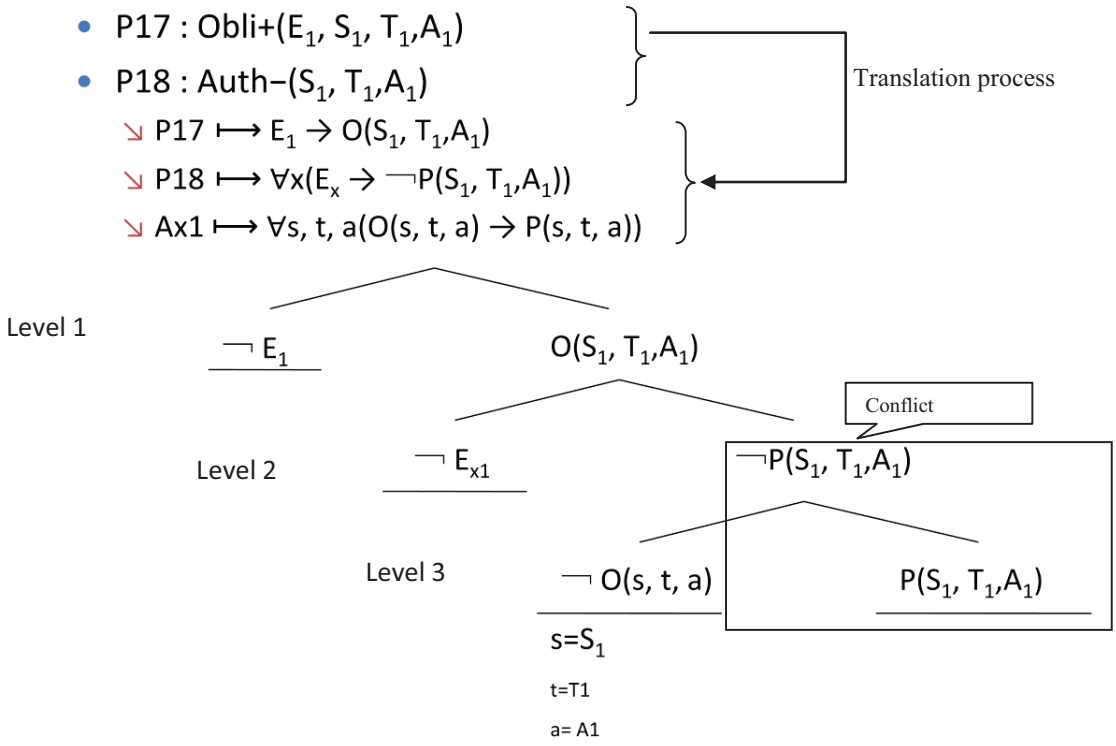


Fig 3.1 : Conflict caused by negative authorization and positive obligation

In this example, we need axiom Ax1 to show the relation between obligation policies and authorization policies (see section 2.3.1.1.3). E_1 represent the event which leads to an activation of new authorization. At the first level, the left branch is closed because the event E_1 is true (closed if E_1 occurs). This is then available as assumption to the next level of tree. At the second level, the left branch is closed, in the case where x_1 is equal to 1(this means E_{x1} and E_1 are the same event). The third level is closed by using the axiom Ax1. The conclusion of this analysis of the tree is: if an event E_1 happens, the two policies could be conflicting.

3.2.2.2. Conflict caused by delegation policy

In this section we propose an extension to the FVT method. This extension permits to support the delegation policy and to detect conflicts caused by it. As mentioned, this extension is our original contribution.

The delegation policy permits to the delegator to delegate (empower) some permissions to another subject (who is called delegatee). The notation of the delegation policy is as follows:

$$\text{Deleg } (S_2, S_1, T, A)$$

This policy specifies that S_2 delegates his right (action A) to the user S_1. Now S_1 can perform the action A on the resource T.

The translation mapping Z of the delegation policy is defined as the follows:

$$Z(\text{Deleg } (x, z, y, a)) := E_1 \rightarrow D(x, z, y, a)$$

In the above translation, the predicate D(x, z, y, a) is interpreted as "the subject x delegates the action a to the subject z, y is the resource". This delegation policy is started by the event E_1 (E_1 is the event where x decides to delegate his rights).

Finally there needs to be a new axiom that relates the delegation policy D and the authorization policy P. This axiom is described as follows:

$$\text{Ax5: } \forall \text{ x, z, y, a } (D(x, z, y, a) \rightarrow P(z, y, a))$$

Ax5 is used to detect conflicts involving both authorization and delegation policies. To see how we can detect conflicts caused by the delegation policy let us analyze the following example:

P23: $\text{Auth}^{-} (S_1, T_1, A_1)$

P24: $\text{Deleg } (S_2, S_1, T_1, A_1)$

This is the analysis tree

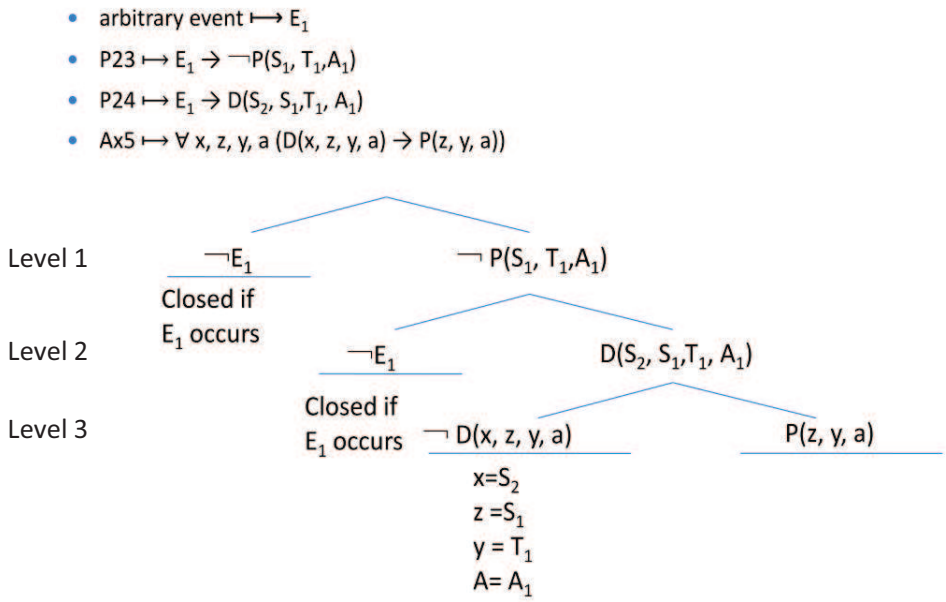


Fig 3.2: Conflict caused by delegation policy

The first level shows that $E_1 \rightarrow \neg P(S_1, T_1,A_1)$ is true if $\neg E_1$ is true or $\neg P(S_1, T_1,A_1)$ is true. The left branch is closed because E_1 is true at the second level. We divide the policy P24 ($E_1 \rightarrow D(S_2, S_1, T_1, A_1)$), the left branch is closed because E_1 is true. Finally the third level shows how the relation between delegation and authorization is true. The left branch is closed because it conflicts with the right branch at the second level. And the right branch (at the third level) conflicts with the right branch of the first level.

3.3. Formal specification of RBAC96

3.3.1. Introduction

Role-Based Access Control (RBAC) is a system of controlling which users have access to resources based on the role of the user. Access rights are grouped by role name, and access to resources is limited to users who have been authorized to assume the associated role. For example, if a RBAC system were used in a hospital domain, each person that is allowed access to the hospital's medical records has a predefined

role (doctor, nurse, lab technician, administrator, etc.). If someone is defined as having the role of doctor, then that user can access only medical records that the role of doctor has been allowed access to. Each user is assigned one or more roles, and each role is assigned one or more privileges to users in that role.

In this section we show how the Alloy language can be used to specify and analyze separation of duty in RBAC96. This section is based on [18] and we will use their terminology and notation. Let assume the following policies:

- P1: Roles r_1 and r_2 are in mutual exclusive relationship (separation of duty)
- P2: Role r_1 is assigned to u_1 and role r_2 is assigned to u_2
- P3: u_2 has being ill, he delegates r_2 to user u_1

Where r_1 and r_2 are roles; u_1 and u_2 are users.

P1 represents a constraint which expresses the separation of duty relation between roles r_1 and r_2. Users must not be assigned to two mutually exclusive roles (r_1 and r_2) at the same time. This constraint is violated by the policy P3, which would result in an unintended conflict.

3.3.2. RBAC96 specification in Alloy

In the following we discuss RBAC96 specification model properties. Let us analyze the following Alloy code:

```
1. model RBAC96{
2. domain {fixed User, fixed RegRole,
3. fixed Session, fixed Permission}
4. state {
5. ureg_assignment: User -> RegRole
6. regp_assignment: RegRole -> Permission
7. us_assignment: User! -> Session
8. sr_assignment: Session -> RegRole+
9. }}
```

The *domain* set presents the object space that will be used later in the specification. The *state* paragraph permits to describe relations between objects

(Alloy atoms). If the cardinality of a relation is not specified, this means that it is fixed to zero or more. If the cardinality is fixed to ! (1) or + (1..*) this means that the cardinality is set to one or one or more respectively. *ureg_assignment* and *regp_assignment* specify relations where the first maps users to roles (roles can be assigned to users), and the second maps roles to permissions (permissions are associated to roles). *us_assignment* specifies that the user can access the system through session and *sr_assignment* states that one or more roles can be active during a session. In the following we show the model which satisfies these relations. This model is simplified with respect to the one of [18].

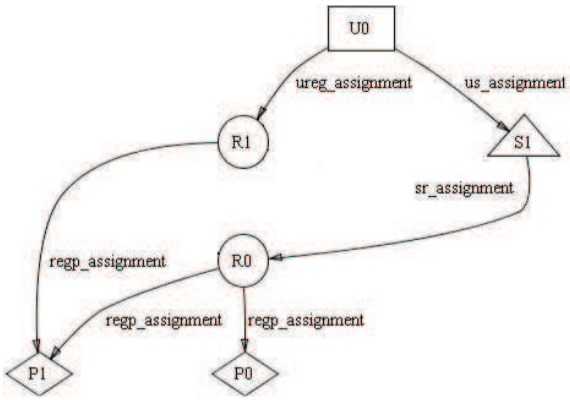


Fig 3.3 : RBAC Model

In the above we show a graphical output representation of the model that we have just described. This interface (Fig 3.3) shows clearly how users, roles and permissions interact among each others; it permits also to detect undesired properties and conflicts. The Alloy analyzer provides also a textual output (xml, text) but in this chapter we show only the graphical output.

At first sight, we can see that user U0 can reach role R0 through session S1 even if role R0 is not assigned to him. To avoid this effect we have to define assertions that must be always satisfied in our model (if it is not the case Alloy will display a counterexample).

Consider the following assertion:

```
1. all u:User | all r:Regrole |
2. r in u.us_assignment.sr_assignment ->
3. r in u.ureg_assignment
```

The assertion means that for all model users, if a role is reached via a session (line2) then that role should be already assigned to the user (line3) through *ureg_assignment* relation. In other words, a user can only activate his assigned role during his current session (*r in u.us_assignment.sr_assignment*) iff that role is assigned to him (*r in u.ureg_assignment*).

Now that we have added the above assertion we re-execute the validation process to see if the undesired behavior disappears:

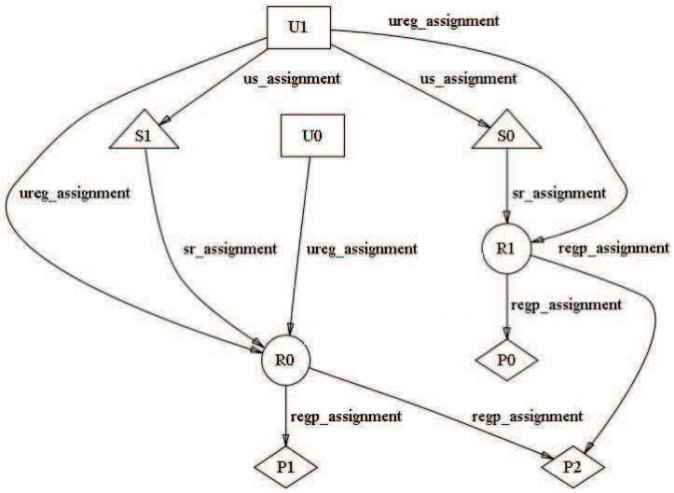


Fig 3.4: RBAC Model with assertion

We can express the rest of the RBAC properties model in the same way (hierarchy, admin range …see [18]).

3.3.3. Separation of duty specification

In this section we present an Alloy specification of static separation of duty (Ssod)[18]. Ssod permit to express a mutual exclusive relation between two roles. These two roles should be not be assigned to the same user. To represent this constraint (Ssod) we first of all add the following relation to our Model:

```
regr_exclusive: Regrole -> Regrole
```

The above relation (*regr_exclusive*) specifies a mutual exclusive relation between roles. Now we have to express an assertion which will state that the separation of duty condition must be always respected in our model.

```
1. all r1, r2:Regrole |
2. r1 in r2.regr_exclusive ->
3. no(r1.~ureg_assignment & r2.~ureg_assignment)
```

The above says that r_1 and r_2 are in mutual exclusive relation, and so no user can be assigned to both of them. The & operator in line 3 specifies the intersection of all users assigned to r_1 and r_2, and the *no* operator means that this set should be empty.

In this section we don't speak about dynamic separation of duty or operational separation of duty. For more information see [18].

3.4. XACML Formal specification

3.4.1. Introduction

In [17] the authors propose a method based on first order logic to detect conflicts and inconsistencies in a set of access control policies. The policies are expressed in XACML, and the method is to translate them into the Alloy language. The properties of the system will be defined in the form of predicates. Then the Alloy specification is analyzed with the Alloy Analyzer tool to find interactions and conflicts.

3.4.2. XACML system

XACML is an OASIS standard that permits to define policies and messages in access control systems.

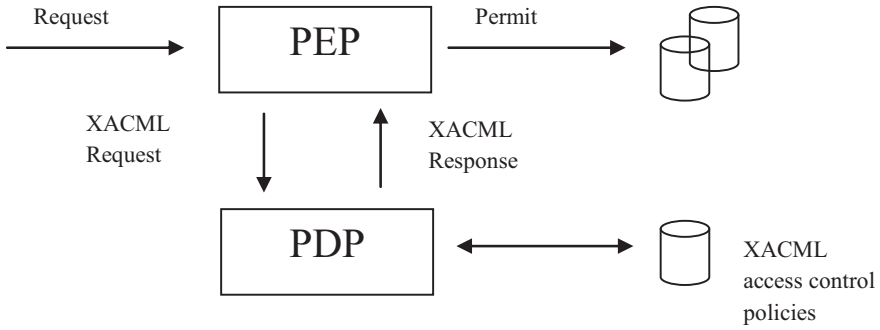


Fig 3.5 : XACML architecture

In the above figure we show the main architecture of XACML. There are two important entities in this architecture which are the policy enforcement point (PEP) and the policy decision point (PDP). The PEP is the module which protects objects (resources), it receives the access requests from subjects and forwards them to the PDP module. The PDP receives the XACML request (sent by the PEP) and computes the decision in accordance to the policies stored in the policy database and the content of the request.

3.4.3. XACML conflict resolution

The policy database can contain different policies that are applicable to a given request. A conflict occurs if according to these policies, different decisions can be provided by the PDP.

XACML provides combining algorithms to avoid conflict problems. Combining algorithms are rules which permit to select a unique and final decision that the system should return.

In the following we describe four combining algorithms defined by the XACML standard:

- Deny-overrides: In the entire set of policies, if any policy evaluates to "Deny", then the result of the algorithm combination shall be "Deny". If any policy evaluates to "Permit" and all other policies evaluate to "NotApplicable", then the result of the algorithm combination shall be "Permit". In other words, "Deny" takes precedence, regardless of the result of evaluating any of the other policies in the combination. If all policies are found to be "NotApplicable" to the decision request, then the algorithm combination shall be evaluated to "NotApplicable".
- Permit-overrides: In the entire set of policies, if any policy evaluates to "Permit", then the result of the algorithm combination shall be "Permit". If any policy evaluates to "Deny" and all other policies evaluate to "NotApplicable", then the result shall evaluates to "Deny". In other words, "Permit" takes precedence, regardless of the result of evaluating any of the other policies. If all policies are found to be "NotApplicable" to the decision request, then the result shall evaluate to "NotApplicable".
- First-applicable: Each policy shall be evaluated in the order in which it is listed in the policies set. For a particular policy, if the condition evaluates to "True", then the evaluation of the policies shall halt and the corresponding effect of the last policy shall be the result of the evaluation (i.e. "Permit" or "Deny"). For a particular policy selected in the evaluation, if the condition evaluates to "False", then the next policy in the order shall be evaluated. If no further policy in the order exists, then the result shall evaluate to "NotApplicable".
- Only-one-applicable: In the entire set of policies, if no policy is considered applicable by virtue of their conditions, then the result of the policy combination algorithm shall be "NotApplicable". If more than one policy is considered applicable by virtue of their conditions, then the result of the combination algorithm shall be "Indeterminate".

3.4.4. Model analysis

In this section we show the following example of file access Marks [17]:

- P1: a professor can read or modify the file of course marks
- P2: a student can read the file of course marks
- P3: a student cannot modify the file of course marks

To see if there is a conflict we can ask the Alloy analyzer if there exist a request that leads to two different decisions (Permit and Deny). The following predicate defines a contradiction:

```
1. pred InconsistentRules (q : Request, r1, r2 : Rule){
2. ruleResponse(r1,q) = Permit
3. and ruleResponse(r2,q) = Deny}
```

This predicate selects requests which lead to permit and deny access at the same time. When we execute this predicate with the Alloy analyzer, the latter will show the following figure:

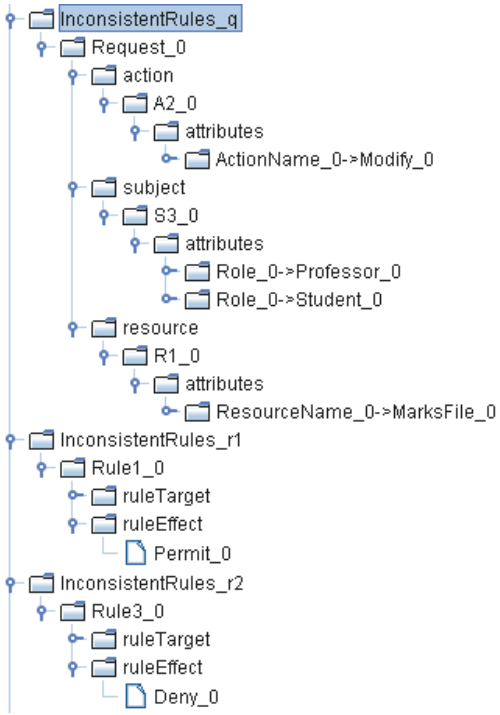


Fig 3.6: Rule conflict in XACML

In the above tree representation (Fig 3.6) we can see that the request (Request_0) to modify the file marks (R1_0) can be performed by a specific subject (S3_0) that has two roles: professor and student. The Alloy analyzer tries to generate all possible use case for these policies, and we can deduce that for this use case they lead to conflict.

3.5. Access control policy verification using SAT solvers

3.5.1. Introduction

In [11] the authors show how to validate access control policies for web-based applications. For this purpose, they propose a formal model for access control policies. This model allows enforcing access control within a single or multiple security system, each with its own access control language. In this section, we focus on the OASIS XACML standard.

3.5.2. Formal Model

Let R= {Permit,Deny,NotApp, Indet} be a primitive set of XACML valid results (decisions) and P be a set of valid policies:

$$
\begin{aligned}
Permit &\in P \\
Deny &\in P \\
\forall p \in P\ \forall S \subseteq E\ Scope(p,S) &\in P \\
\forall p \in P\ \forall S \subseteq E\ Err(p,S) &\in P \\
\forall p,q \in P\ p \oplus q &\in P \\
\forall p,q \in P\ p \ominus q &\in P \\
\forall p,q \in P\ p \otimes q &\in P \\
\forall p,q \in P\ p \oslash q &\in P
\end{aligned}
$$

Where Permit and Deny are primitive policies that give respectively permit and deny permission, E represents the environment. The Scope function is used to enclose policies and environment. The Err function is used to attach error to policies. $\oplus$,$\ominus$, $\otimes$, $\oslash$ operators are used to express combining algorithms on policies and represent respectively permit overrides, deny overrides, only one applicable and first applicable.

Now we can define a semantic for these policies, let consider the following function:

$$eff : E \times P \rightarrow R$$

The above function takes an environment and a policy as input and returns a valid result as output.

In the following we present the policy semantics; the latter is slightly different from the XACML standard because Permit does not override Indet using $\oplus$ operator and Deny does not override Indet using $\ominus$ operator, since Indet represents an error and not a normal condition. The model properties are defined as follows:

$$
\begin{aligned}
\text{eff}(e, Permit) &= Permit \\
\text{eff}(e, Deny) &= Deny \\
\text{eff}(e, Scope(p, S)) &= \begin{cases} \text{eff}(e,p) & \text{if } e \in S \\ NotApp & \text{otherwise} \end{cases} \\
\text{eff}(e, Err(p, S)) &= \begin{cases} Indet & \text{if } e \in S \\ \text{eff}(e,p) & \text{otherwise} \end{cases} \\
\text{eff}(e, p \oplus q) &= \begin{cases} Indet & \text{if } \text{eff}(e,p) = Indet \\ & \quad \vee \text{eff}(e,q) = Indet \\ Permit & \text{if } (\text{eff}(e,p) = Permit \\ & \quad \wedge \text{eff}(e,q) \neq Indet) \\ & \vee (\text{eff}(e,q) = Permit \\ & \quad \wedge \text{eff}(e,p) \neq Indet) \\ Deny & \text{if } (\text{eff}(e,p) = Deny \\ & \quad \wedge \text{eff}(e,q) \neq Permit \\ & \quad \wedge \text{eff}(e,q) \neq Indet) \\ & \vee (\text{eff}(e,q) = Deny \\ & \quad \wedge \text{eff}(e,p) \neq Permit \\ & \quad \wedge \text{eff}(e,p) \neq Indet) \\ NotApp & \text{otherwise} \end{cases} \\
\text{eff}(e, p \ominus q) &= \begin{cases} Indet & \text{if } \text{eff}(e,p) = Indet \\ & \quad \vee \text{eff}(e,q) = Indet \\ Deny & \text{if } (\text{eff}(e,p) = Deny \\ & \quad \wedge \text{eff}(e,q) \neq Indet) \\ & \vee (\text{eff}(e,q) = Deny \\ & \quad \wedge \text{eff}(e,p) \neq Indet) \\ Permit & \text{if } (\text{eff}(e,p) = Permit \\ & \quad \wedge \text{eff}(e,q) \neq Deny \\ & \quad \wedge \text{eff}(e,q) \neq Indet) \\ & \vee (\text{eff}(e,q) = Permit \\ & \quad \wedge \text{eff}(e,p) \neq Deny \\ & \quad \wedge \text{eff}(e,p) \neq Indet) \\ NotApp & \text{otherwise} \end{cases} \\
\text{eff}(e, p \otimes q) &= \begin{cases} \text{eff}(e,p) & \text{if } \text{eff}(e,q) = NotApp \\ \text{eff}(e,q) & \text{if } \text{eff}(e,p) = NotApp \\ Indet & \text{otherwise} \end{cases} \\
\text{eff}(e, p \oslash q) &= \begin{cases} \text{eff}(e,p) & \text{if } \text{eff}(e,p) \neq NotApp \\ \text{eff}(e,q) & \text{otherwise} \end{cases}
\end{aligned}
$$

In the above the result of eff(e, p $\oplus$ q) can be Indet if p or q brought an error, eff(e, p $\oplus$ q) result can be a Permit if p is a permission and q is not undetermined or q is a permission and p does not bring an error. We can read the rest of the semantic in the same manner.

As we can see this semantic is quite complete for a coherent access control system. Any violated requirement will result in a warning.

3.6. Stratification-based for handling conflicts

3.6.1. Introduction

[4, 6] propose a formal notation for access control policies and an approach to handle conflicts in access control policies. In section 3.6.2 we show how the authors specify their policies, section 3.6.3 will be dedicated to the conflict resolution approach.

3.6.2. Logic notation

In this section we present the logic used in [4, 6] for access control policy representation. In this logic different roles are represented by different predicates, e.g., RDoctor(x) (x is assigned to role Doctor), RPatient(x) (x is assigned to role Patient) and RFamilly(x) (x plays the role Family). Objects are also defined in the same manner: MedicalRecord(x), Nurse_Report(x)...

The assignment of roles and separation of roles is done by using logic connective operators: conjunction, disjunction, negation and implication. The separation of roles saying that a user cannot be assigned simultaneously to roles Doctor and Nurse is written as:

$$\forall x, \neg \text{RDoctor}(x) \vee \neg \text{RNurse}(x)$$

Now let us see the norm base of access control policies:

If Conditions on (users, objects, contexts) are satisfied then a user is permitted/prohibited/obliged to do some actions on (objects).

Contexts are logic formulas composed of disjunctions of conjunctions of predicates that can be used to restrict the applicability of policies. In the following we present three modalities of policies: Permission, Prohibition and Obligation:

- Permission: in this case, two predicates can be used, A (e.g., Read(x)) which expresses the fact that A is achieved (A is true) and PA (e.g., PRead(x)) which

expresses the fact that the permission to execute A is true. NotA expresses the fact that A is not achieved.

- Prohibition: here it is better to express prohibition as a negation of permission than to add a new predicates (e.g., ¬PRead(x) expresses the fact that the permission to execute action read is false):

  Prohibition to execute an action A = ¬ permission to execute A.

- Obligation: with obligation we have two approaches to take in consideration. First if a subject should (obligation) execute an action then he has an implicit permission to do that action. Second, an obligation means that: it is prohibited to not execute an action. So if we say "it is not permitted not execute to A" this implies that A is permitted.

$$\neg PNotA \rightarrow PA$$

In the following we discuss some examples of access control policies [4]:

- A user playing a role of ”non staff member” is forbidden (i.e., not permitted) to read a patient’s record.

$$\forall x, \forall y, \neg RStaff(x) \wedge Patient(y) \rightarrow \neg PReadRecord(x, y)$$

The above formula says that subject x cannot read the medical record of patient y if x doesn’t have the role RStaff and y is a patient.

- in the case of fatal disease, a user playing a doctor’s role must (i.e., is not permitted not to) inform the patient’s family:

$$\forall x, y, z, RDoctor(x) \wedge Patient(y) \wedge Family(z, y) \wedge FatalDisease(y) \rightarrow \neg PNotInform(x, z)$$

In this case (obligation) we have to add the relation between obligation and permission (this will be useful to analyse access control policies):

$$\forall x, z, \neg PNotInform(x, z) \rightarrow PInform(x, z).$$

3.6.3. Conflict analysis and resolution

In [6] the authors associate permission rules with priorities in order to evaluate their importance (that will be useful to compare conflicting policies). Let us suppose that we have two conflicting policies P_1 and P_2, where P_1 has a higher priority than P_2, then P_2 will be ignored and the system will execute the policy P_1. In the following we show the general format of permissions:

Permission(subject, action, object, priority)

Prohibition(subject, action, object, priority).

The authors introduce also a new predicates Higher-Permission and Higher-Prohibition. Higher-Permission(s, a, o, p) means that there exists a permission for subject s, action a and object o having a priority higher than p. Higher-Prohibition(s, a, o, p) means that there exists prohibition over s, a and o having a priority higher than p. The < operator is used to compare policy priority.

$\text{Permission}(s, a, o, p_2) \wedge p_2 < p_1 \rightarrow \text{Higher-Permission}(s, a, o, p_1)$

$\text{Prohibition}(s, a, o, p_2) \wedge p_2 < p_1 \rightarrow \text{Higher-Prohibition}(s, a, o, p_1)$

$(p_1 < p_2 \wedge p_2 < p_3) \rightarrow p_1 < p_3$ (transitivity of <)

$(p_1 < p_2 \wedge p_2 < p_1) \rightarrow \text{error}()$ (anti-symmetry of <)

The resolution strategy is applied as follows:

$\text{Permission}(s, a, o, p) \wedge \neg\text{Higher-Prohibition}(s, a, o, p) \rightarrow \text{A-Permission}(s, a, o)$

$\text{Prohibition}(s, a, o, p) \wedge \neg\text{Higher-Permission}(s, a, o, p) \rightarrow \text{A-Prohibition}(s, a, o)$

The first formula states that permission can be derived if there is no higher priority prohibition in the access control policy. The second one is similar but for deriving a prohibitions.

To solve conflicts we have to detect them first and for this purpose we use the following formulas:

$$\text{Permission}(org, r_i, a_i, v_i, c_i, p_i) \wedge \text{Prohibition}(org, r_j, a_j, v_j, c_j, p_j) \wedge$$
$$\neg(\text{separated-role}(org, r_i, r_j)) \wedge \neg(\text{separated-activity}(org, a_i, a_j)) \wedge$$
$$\neg(\text{separated-view}(org, v_i, v_j)) \wedge \neg(\text{separated-context}(org, c_i, c_j)) \wedge$$
$$\neg\text{Solved-Conflict}(org, r_i, a_i, v_i, c_i, p_i, r_j, a_j, v_j, c_j, p_j)$$

The above formula means that a potential conflict occurs if there is a permission for the role r_i of the organization org to execute the action a_i on the view v_i with condition c_i with priority p_i and a prohibition for the role r_j of the organization org to execute the action a_j on the view v_j with condition c_j with priority p_j and there is no separation between roles, actions, views (resources), contexts and the conflict is not solved.

Where Solved-Conflict $(org, r_i, a_i, v_i, c_i, p_i, r_j, a_j, v_j, c_j, p_j)$ predicates means that if there is a conflict between roles r_i and r_j this conflict can be solved. The Solved-Conflict predicate is defined as follows:

$$\text{Permission}(org, r_i, a_i, v_i, c_i, p_i) \wedge \text{Prohibition}(org, r_j, a_j, v_j, c_j, p_j) \wedge$$
$$((\text{Prohibition}(org, r, a, v, c, p) \wedge (p_i < p))$$
$$\vee (\text{Permission}(org, r, a, v, c, p) \wedge (p_j < p))) \wedge$$
$$(r=r_i \vee r=r_j) \wedge (a=a_i \vee a=a_j) \wedge (v=v_i \vee v=v_j)\ (c=c_i \vee c=c_j)$$
$$\rightarrow \text{Solved-Conflict}(org, r_i, a_i, v_i, c_i, p_i, r_j, a_j, v_j, c_j, p_j)$$

As we can deduce from [3, 4, 6] the detection process is made at the organizational level, which means that the approach permits to detect only potential conflicts. By contrast, our purpose is to detect existent conflicts, and our work will focus on the detection of conflicts at the concrete level. For example, let see the following policies:

- P1: Professors can write in rates file
- P2: Students can't write in rates file

As we can see there is no direct conflict between roles Professors and Students (P1 and P2), but we can have a concrete scenario where there exists a student (Bob) who gives a lab. In this case, the student Bob can be assumed to be a professor

(because he gives a lab), so he can execute both of policies P1 and P2, leading to a conflict. In other words if we look at policies as they are (high level) we can't be sure that our access control system is free of conflicts, that is why we have consider the concrete level also (concrete users, resources and actions).

3.7. Other related work

Other static conflict detection approaches have been presented in the literature. For example in [5], the authors use a tool called PCV (Policy Consistency Verifier) to detect inconsistencies between security policies written in the SPL language and workflow specifications. In [15], the authors have implemented a prototype role framework to detect conflicts which may appear when the subject and the resource domains overlap. They also introduce a priority relationship between these domains to resolve the conflicts. [13] Specify a logic language to define access control policies. This language supports both concepts of groups and roles. They propose a method to detect conflicts by using derivation rules. [20] Propose a method for conflict checking of separation of duty in the RBAC model as well as a method to detect static separation of duty conflicts caused by propagation policy. [1] propose a new access control model which permits to express negative and positive authorization and contexts at the abstract and concrete levels. This model introduces a typing system in order to detect potential conflicts.

3.8. Conclusion

[14] have proposed the Free Variable tableaux method which allows administrators to statically detect conflicts in set of policies. This method was tried on several real life use cases and FVT appears to be a complete theorem prover. Access control policies are translated to logic expressions. This can be very useful to detect inconsistency in any security system independently from the used language.

This method not only detects direct conflicts (conflicts between positive and negative authorization) but also allows to detect conflicts caused by propagation, by violation of Chinese wall, separation of duty, time constraint and others. In addition,

this method permits to show which policies are part of the problem. The knowledge of conflicting policies permits administrators to resolve the problem in further steps.

We have introduce also an extension to this work (FVT) to detect conflicts caused by the delegation policy. For this we have added the axiom Ax5 to build a link between access control policies and delegation policy. However this method doesn't permit to analyze constraints.

[11] provides a formal notation for access control policies, to facilitate their translation to logic expressions, which are then used as input for a SAT Solver. The authors show how with their method it is possible to check whether policy properties are respected and if the policies are related in the desired manner.

The work presented in [18] is about specifying separation of duties inside the RBAC96 model. They begin by specifying the general model then they propose some assertions to detect undesired properties. They don't specify conditions based on attributes and they don't specify negative policies either.

[17] showed how Alloy can be used to detect conflicts and inconsistencies. However they used Alloy predicates to detect conflicts, while we will use assertions in our work. We use assertion rather than predicates because assertions are always active in the specification; they are always checked by the compiler. In an Alloy specification, we can have several predicates and several assertions. But in the case of the predicates, only one at the time holds.

We have reviewed in this chapter several of the most interesting detection techniques that have been presented in the literature. Several of these methods are very well developed and have influenced our work in various ways. Our main contributions with respect to the existing work are three:

1. Consideration of access control constraints (filters in EEM)
2. Consideration of high and concrete level policies
3. Application of the method to the industrial policy tool EEM

CHAPTER 4: EEM TOOL OVERVIEW

4.1. Overview

In this chapter, we describe an industry tool called EEM (Embedded Entitlement Manager). EEM is an access control application, which allows administrators to manage their security policies. In next sections we address the way we build policies in EEM. Section 4.3 is dedicated to present EEM`s main features. In section 4.4 we show how to build identities, resources, groups and policies. Section 4.4.5 provides information about the resolution algorithm used in EEM.

4.2. Introduction

eTrust EEM (Embedded Entitlement Manager) is a tool that helps organizations to securely manage identities, resources and their associated access control policies. It is a web based toolkit that allows administrators to easily embed authentication, authorization and security auditing in all their enterprise applications. Authentication and authorization can be handled by using simple API calls. Applications can share a common web interface for policy definition and identity management.

An older name for this tool was eIAM, for embedded Identity and Access Manager, and the name eIAM is still found in the toolkit as we will see.

This chapter provides an overview of the main EEM features and shows how to build basic policies for EEM. We won't discuss all EEM options and configurations.

4.3. EEM features and functionalities

4.3.1. Identities management

EEM allows administrators to see who is in their application environment and to control what they can access and what actions they can do. In the following we show some identities management functionalities:

- Embedded authentication and authorization: Identities can use common authentication and authorization services using the eTrust Embedded

IAM SDK (the EEM runtime environment). Only legitimate users (identities) can access EEM applications.

- External identity repository: The single authoritative source of user identities for eTrust Embedded IAM access is very compatible with external systems that can be Microsoft Active Directory, Novell eDirectory or eTrust Admin. However, user identities can be managed directly by eTrust Embedded IAM and stored internally or externally (in other databases types).

4.3.2. Access management

EEM tool provides a graphical user interface for policy management. This interface will provide integrity and privacy for administrator data and applications. Some of the access management functionalities are:

- Access policy enforcement: here the tool allows administrators to define and enforce policies to control who may access applications and to define under which conditions the access can be granted.

- Administrative scoping: administrator permissions can be limited to specific users, resources and policies (this feature is used to define delegation policies).

- Delegation of permission: this functionality allows a user to grant to another user the use of some of his rights.

- Calendar: calendar permits to add temporal constraints on policies. For example, visitors can visit patients only during visiting hours (e.g. 3 pm to 5 pm…).

- Policies checks: this functionality permits administrators to enhance and modify their policies by evaluating these locally. This will avoid the online test calls.

4.3.3. EEM integration

There exist several ways to integrate EEM in a development environment already in use by enterprises:

- Development kits in C, C++ and JAVA: these API permit administrators to embed their security functions into applications and develop authentication, build authorizations and event management. The API`s include also XML scripts, the XML syntax to add new application. The tool Safex which is used to parse XML files.

- Web user interface: EEM provides a web interface to manage users, groups and access control policies. Administrators need only a web browser to build their policies for access control (we suppose that the EEM server is already configured).

- Identity and access management server: eTrust Embedded IAM provides a run time utility service hosted on an Embedded IAM Server that interfaces between application client and the eTrust Embedded IAM data store. In addition, it makes authentication calls and delivers secure events to the audit system.

4.4. EEM access control policies design

In this chapter, we don't discuss how to install EEM but we present briefly the required services without which EEM can't be active and responsive. There are six services in EEM:

- eTrust Directory - iTechPoz -
- eTrust Directory - iTechPoz - - Router
- eTrust Directory Administration Daemon - master
- eTrust Directory SSL daemon - iTechPoz-Server
- iTechnology iGateway
- Ingres Intelligent Database [EI]

As long as the above services are running correctly EEM should be active constantly. The web browser must be pointed to

https://ip_address:5250/spin/eiam/eiam.csp to start EEM. This is the ip address of the server where EEM is installed, and the requests will be exchanged on the 5250 port. We show now the login page of EEM:

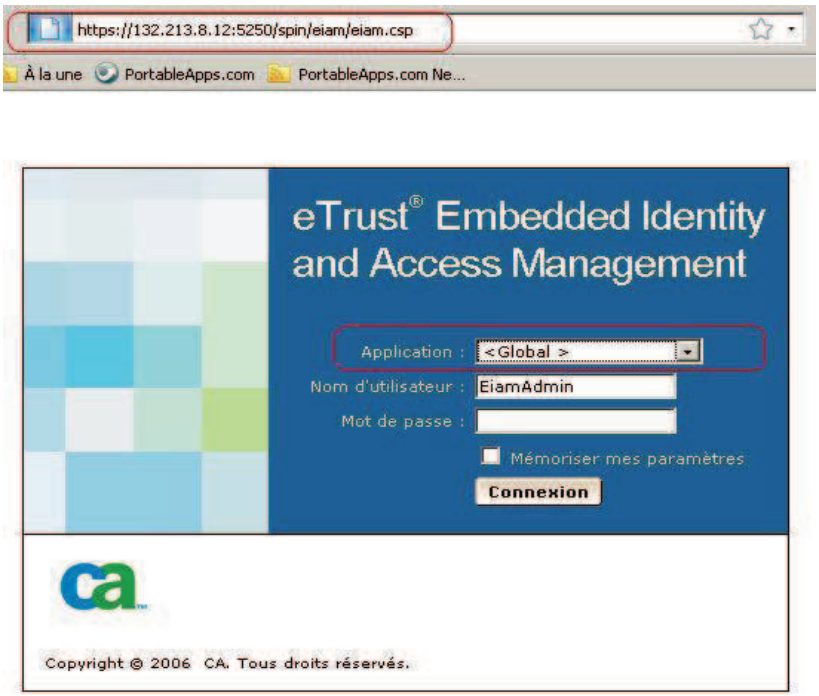


Fig 4.1: EEM login page

In the above page, we select which application we will work on (circled). Each application that EEM provides is registered under a specific name space. This is done by the Safex tool (section 4.4.6). To see all EEM applications we have to use a drop down box list, but we have at least one name space, which is the default name space of EEM: Global. Any policy built in Global application will be applied in all EEM applications. Each application in EEM has its own separate set of policies as well as pre-defined policies (policies of EEM system) for different objects of the application.

EEM security policies are based on an Object Oriented approach. Three objects need to be taken into account:

- Identities (users, groups)
- Resources
- Actions

To build an access control policy we have to link the above objects together. One of these objects, identities, appears after login interface (see Fig 4.2). By clicking on the link "Gestion des identités", one will be redirected to the identities screen (Fig 4.3) making it possible to search through the user list. By clicking on the "Go/Aller" button EEM will display the list of all users, which were created through different applications. By default, EEM uses the MDB database as its user store. In this chapter, we do not discuss the external active directory integration; we just show how to create an actual policy.

4.4.1. Identities

When the user is logged in, a default page will be displayed, which contains a quick start link for identities and policies creation.

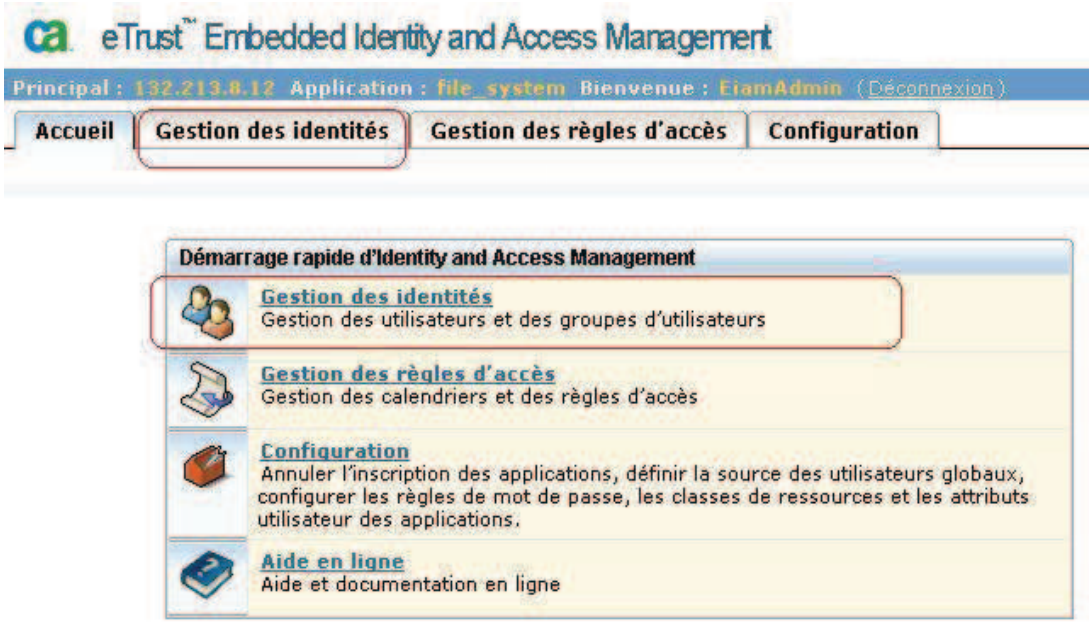


Fig 4.2: EEM home page

From the above page we can see what application we are working on (in the top information bar: file_system application), which user is logged in (in our case is EiamAdmin) and which server name or ip we are connected to (132.213.8.12).

To create new identities we choose "Gestion des identités" link in quick start or from tab under the top information bar. By default EEM will display the following identities page:

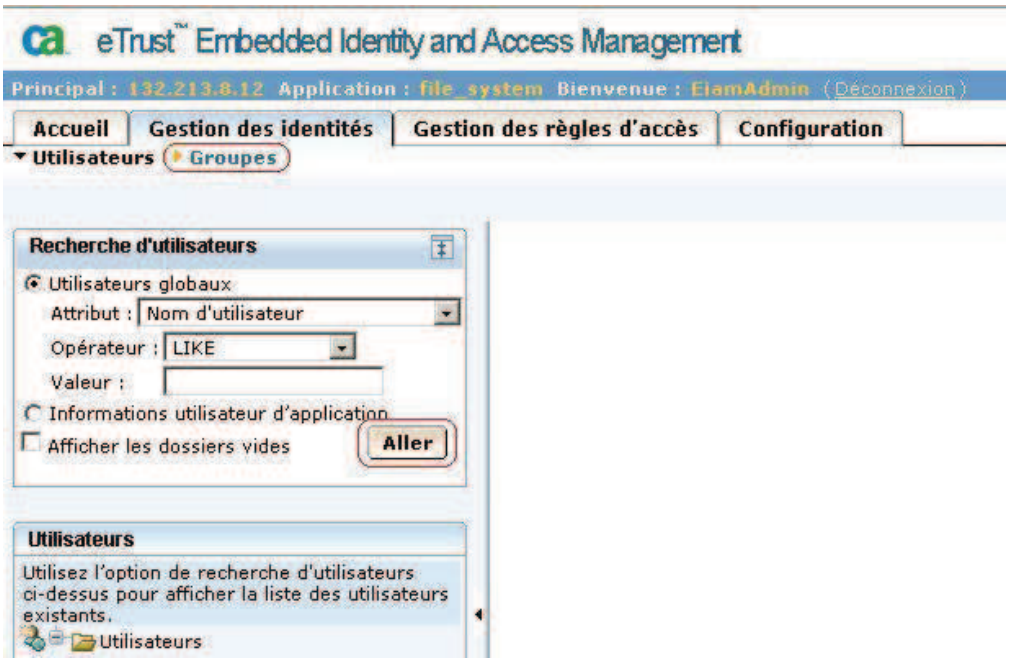


Fig 4.3: EEM Identities page

In the same page, we can also create groups by clicking on the "Groupes" link (circled). By clicking on "Aller" button, EEM will display all users created by the administrator. We can also select different attributes to search users by selecting from the drop down box named "Attribut". We can use the "*" character to display all users. To see user properties we have to select them from the left panel.

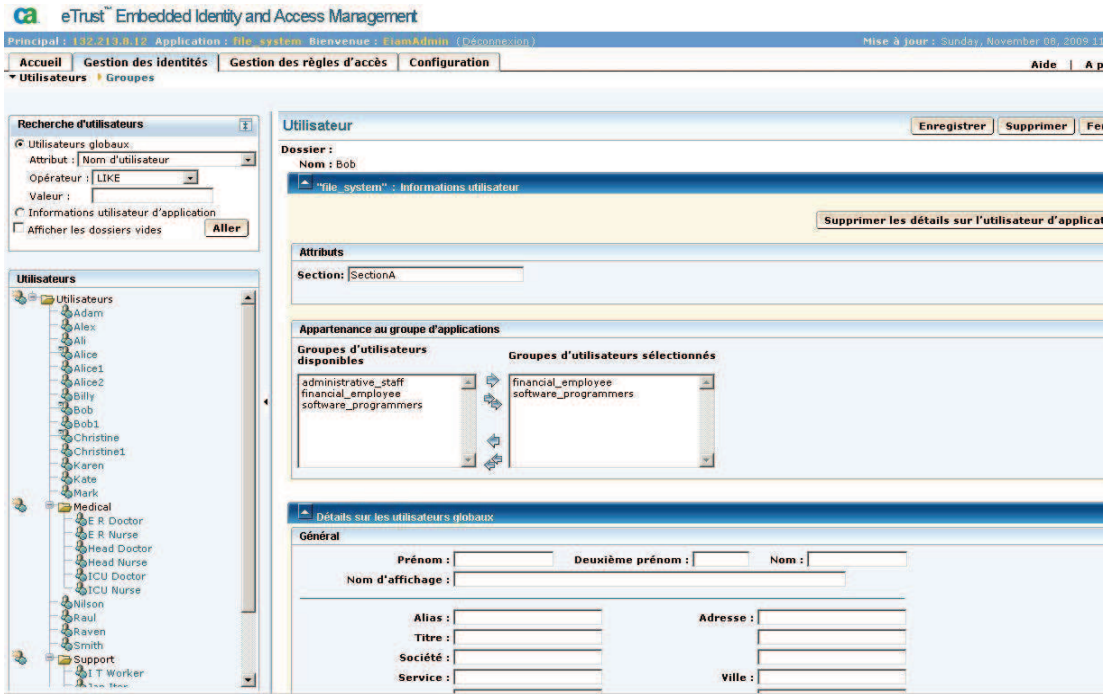


Fig 4.4: User property form

The above interface shows which application group the user belongs to, attribute values and name. We can see Bob's information, he belongs to two groups: financial_employees and software_programmers. To assign the group to which the user belongs, we have to press the arrow to move the group from the available list to the selected list. We can also setup various user option details such as password, address, etc.

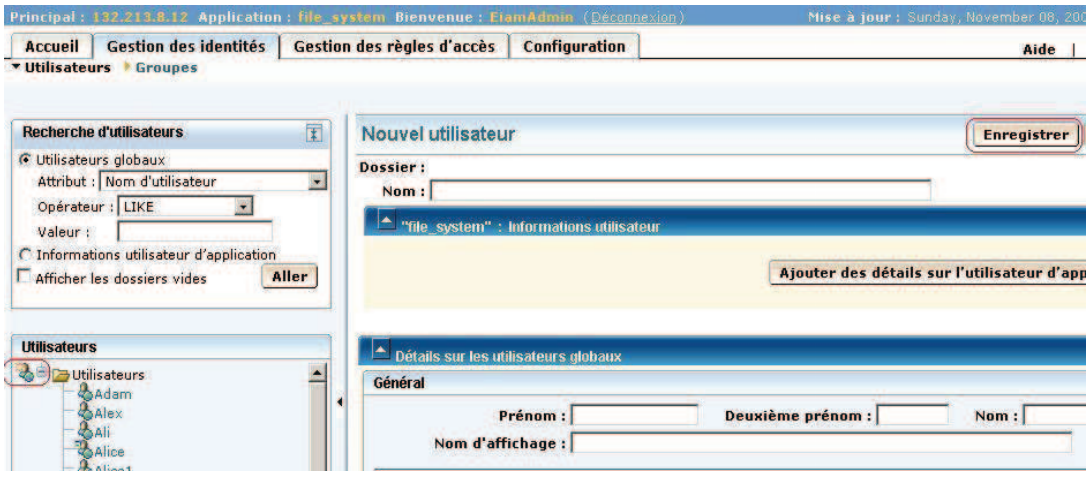


Fig 4.5: Creation of new user

To add new users we have to press the "Add user" icon in the left user panel. After filling all the new user's properties we should press the save button "Enregistrer". A message will be displayed at the top of the page saying "User Created Successfully". Note that the user name we choose should not already exist in the EEM database. We need to press the "Aller" button to update the left user panel to see the updates (new user).

Now let us see how we can create a new group for our user. We have to press the "Groupes" link below the "Gestion des identiés" tab. The interface will change as follows:

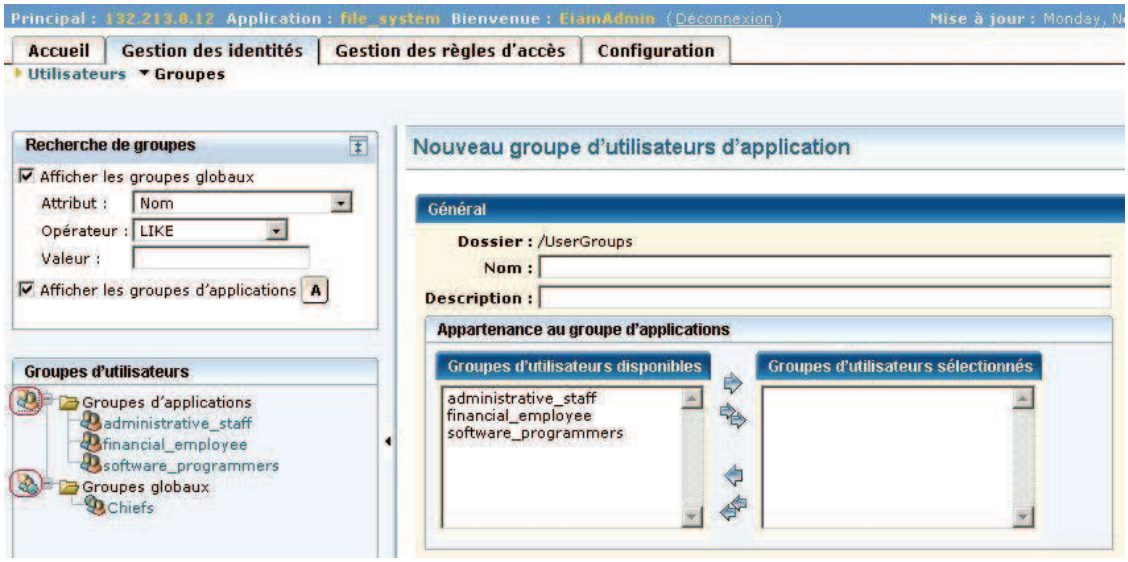


Fig 4.6: EEM Group interface

In the above interface, we see two kinds of folders in the left panel: "Groupe d'applications" and "Groupes globaux". If the group is created as global, this means that all application policies apply to all users assigned to the group. If the group is created locally (in a given application) then only local policies can apply to that group. Note that Users are created globally in EEM to be able to access all applications (they are always stored at "/Utilisateurs" folder). Press the "Add group" icon next to the "Group d'application" (circled). Once we finish filling the group name and description form we have to press the Save button, a confirmation will be displayed on the top of the screen. Note that it will be easier to manage access control

security if we apply policies to groups rather than to users. This will reduce the number of policies that we need to build, test, edit and change.

Now we have to go back to the user`s page to assign users to the appropriate groups (see Fig 4.4). Select the user that we have created from the left panel, his information will be displayed in the right panel, we look at the list of groups "Groupes d'utilisateurs disponibles" under the user detail section. To assign a user to a global group we just scroll down the page, under the global user details, we can see the "Groupes d'utilisateurs globaux disponibles" box

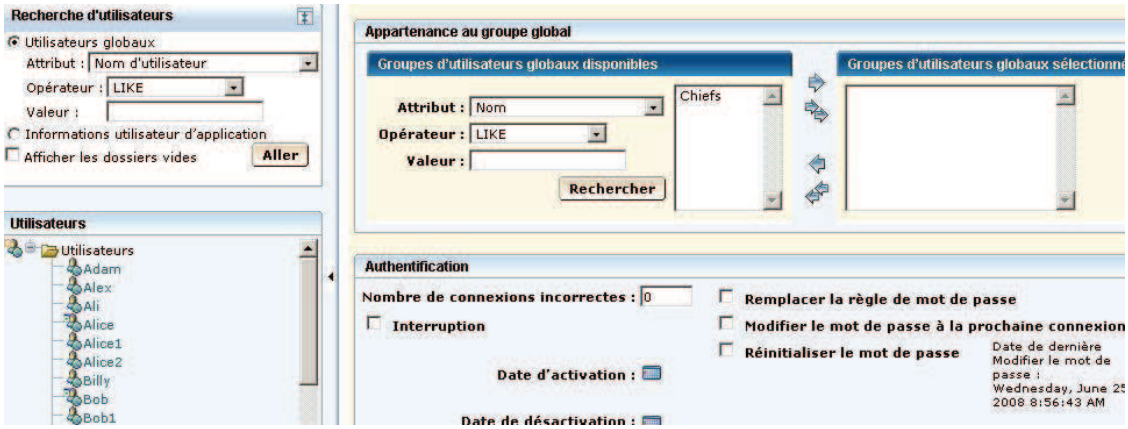


Fig 4.7: EEM Global group interface

In the above interface, we can see that we have a global group called "Chiefs" (we don't select a global group). Through these two group lists (seen in Fig 4.6 and Fig 4.7) we can determine whether the user belongs to a local application group or to a Global group. For instance, we join the user Bob to the local groups: financial_employee and software_programmers. We also fix his attribute "Section" to "SectionA" (see Fig 4.4). If we scroll down the page, we can see a summary of user group's information as follows:

Fig 4.8: EEM Group information

4.4.2. Resources

To create a new resource we have to select the "Configure" tab at the top of the screen, on the left panel (called "Applications", see Fig 4.9). We can see two applications: our application to which we are connected and "the Global" application. In our case, we choose "file_system" application, now we can see our resources on the right panel. To add new resources we press on the "Ajouter une classe de resource" button, and then we fill the template (resources name, actions and attributes of resource). Once we finish filling the template we have to press the "Enregistrer" button to save the new resource in the EEM database.

Fig 4.9: EEM resource interface

4.4.3. Calendar

EEM allows to specify conditions or filters, that constrain the execution of policies. A policy is executed only if the condition specified by the filter is true. The most important type of filter is the calendar.

In this section, we show how to add time constraints using the custom EEM calendar interface. Access policies can specify time-based criteria like standard working hours through calendar.

We are about to build a calendar which represents the working hours. The working hours are split in two time slots: morning and afternoon.

- Morning time: 08am to 12am
- Afternoon time: 1pm to 4:15 pm

Press the "Calendrier" link under the "gestion des Règles d'accès" tab, a panel will be displayed in the left side of the screen (see Fig 4.10). Press the calendar icon in the left panel near the "Calendriers" folder, next we have to press the "ajouter un bloc en horaire d'inclusion". The interface permits to specify the calendar name (calendrier 1), days and months masks. The mask is simply the collection of days and month selected. We can see in Fig 4.10 that the day mask selects Monday, Tuesday, Wednesday, Thursday and Friday and the time of the day (08am to 12 and 1pm to 4:15 pm).

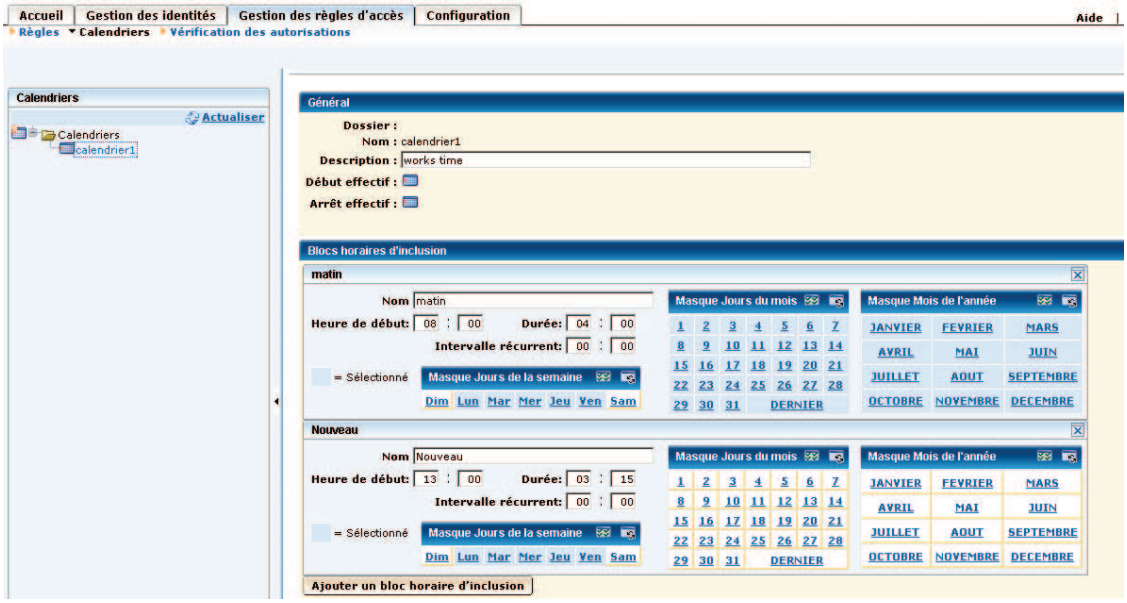


Fig 4.10: EEM calendar interface

4.4.4. Policies

In the three previous sections, we showed how to create identities, groups, resources, actions and filters (temporary constrains and attributes based constraints). All these elements are linked together in policies. In this section, we will see how to build policies by linking all these elements together. So let us build the following policies:

- P1: all Section A employees can read the financial folder file during working hours.
- P2: group software_programmers can`t read/write the financial folder file.

We suppose that we have created already a user named Bob (with named attribute Section), a group software_programmers and a resource file named File. Note that P1 is a policy that takes a user's dynamic group as identity. A Dynamic group permits to add identities to groups through a filter. The latter is computed at the execution of the policy. To build policy P1 click on "règle de groupe dynamique" (see Fig 4.8).

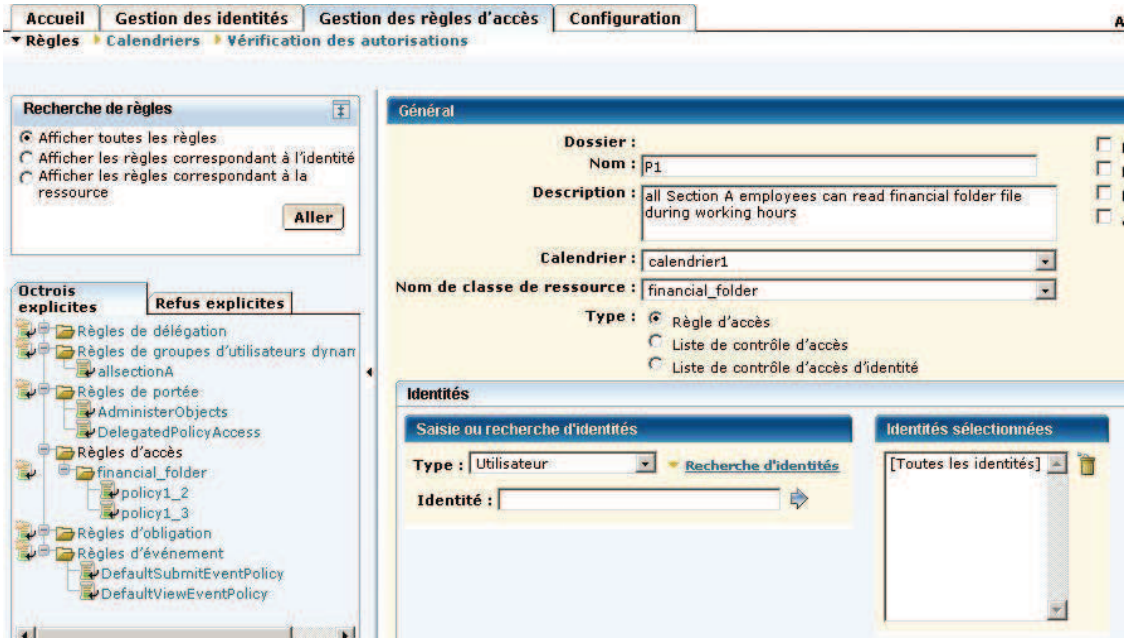


Fig 4.11: Dynamic group policy interface

We have to name our policy, the name should be unique (P1) and the calendar defined in the previous section (calendrier 1) should be selected. We don't have to specify an identity because it is a dynamic group (by default all users are selected, the identity will be selected in the filter). Now we scroll down the page to specify the resource and the dynamic group (See Fig 4.12).

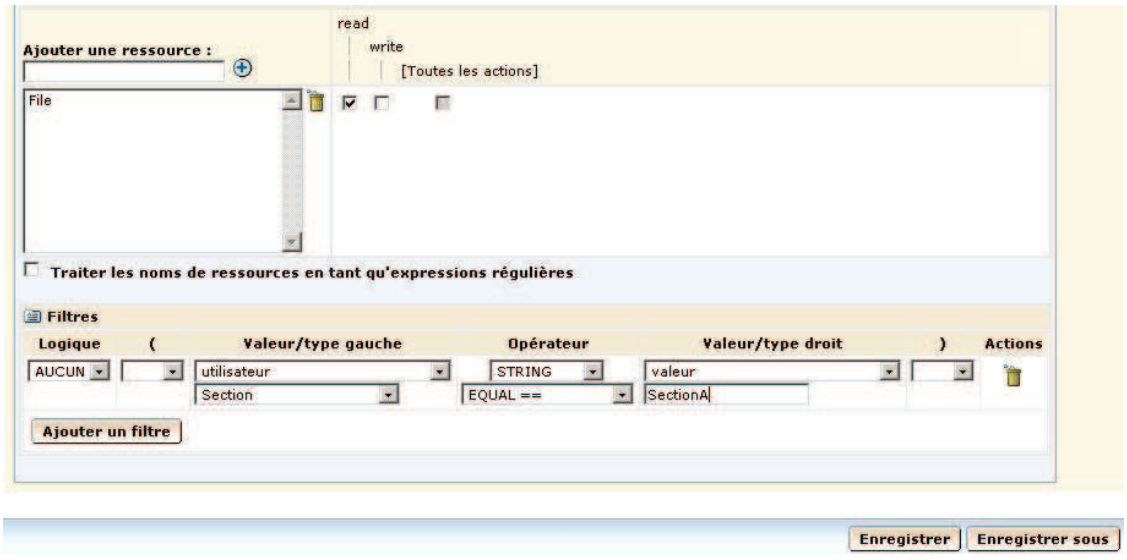


Fig 4.12: EEM filters interface

As we see in the above interface, we have specified the resources (File), and we have checked the action read. We click on "filtres" icon, and a form will be displayed. We just fix the attribute value called Section to SectionA value. We can filter users in dynamic groups by name, group membership or attributes etc.

For the policy P2 we don't need a filter, we can just select the group "software_programmers" as identity (Fig 4.11). This is done by pressing the "Gestion des régles" button. Now we can see two tabs in the left panel: "Octrois explicites" and "refus explicites", they respectively permit to create permission and prohibition policies.

Note that policies are grouped into folders (folders are generated by EEM and cannot be changed manually). This helps administrators to find easily their policies, since the folder name corresponds to resources. We know from looking at the folder application access, that it will contain policies of the resource identified by its name.

4.4.5. Policy evaluation algorithm

The policy evaluation interface is a very interesting feature for administrators since they can check user accessibility without sending a real request to the EEM server. It allows to see if what is defined in access control policies is conform to what is expected. For this, EEM uses a resolution algorithm. We describe the different steps used to evaluate a given request (policy):

1. Gather the identity's attributes from the GlobalUser, User, GlobalUserGroup, and UserGroup objects
2. Assemble any Environment information
3. Check for explicit denials
   - "Matching" appropriate policies
   - Evaluate the matched policy filters
4. If no explicit denial was "hit":
   - check for explicit grants
     - "Match" appropriate policies
     - Evaluate the matched policy filters
   - if no explicit grant was "hit":
     - Evaluate any delegated authority
5. calculate Obligations for this access check

The first step of the evaluation algorithm is to collect all identity information necessary to process access requests. This information will be stored in the local cache (because the gathering operation is relatively expensive in terms of time). The second step is to link the identity information with resource information.

In the third step the algorithm will look for explicit denial policies that match the access request (EEM searches the policies that apply to the specific request). If there are any such policies, the algorithm evaluates the filters, using the environment information. The fourth step is executed if the algorithm does not find any denial policy which matches the access request. In this step, the algorithm will look for explicit grant policies. If there is no explicit grant policy, the algorithm looks for delegation policies which match the access request.

Finally, the algorithm will look for obligation policies. If an obligation policy is found, then EEM will execute the latter otherwise it will stop.

Let us try the EEM evaluation interface on the policy P1 with the identity Bob (which has the attribute sectionA, see Fig 4.4).

Fig 4.13: EEM evaluation interface

In the above figure we show two parts: the first part (on the top) permits to build the administrator's requests, the second (on the bottom) shows the decision computed by EEM. So we have tried two requests: in the first one, the user Bob asks to read in the financial folder the file "File" (resource) at 8.55am (see last column of the table, Fig 4.13), and in this case of course the decision is "grant". We can see also that the decision is based on policy P1. In other words, user Bob is allowed to read File through policy P1. In the second test, the user Bob tries to read File at 12.55pm. The access is denied for him because the request was sent out of working hours, which are: 8am to 12.00pm and from 13pm to 16.15pm. In this case the decision is not based on any policy, but is deduced by the resolution algorithm: if no policy matches a request, the decision is "deny".

4.4.6. Safex

The EEM command line interface (called CLI), is named Safex and can be invoked at the command prompt with several options and an XML file specification. The purpose of the CLI is to allow products to generate XML input and output files which can then be used to help setup policies or migrate them to other tools in preparation for product deployment. Product registration details, policies, users and calendars are all examples of the types of objects that can be pre-configured using this tool. The syntax of the Safex tool is shown as follows (we mention only the important options):

Safex [-h backend] [-u user -p password] [-f xml_file]

4.5. Conclusion

We have shown in this chapter how EEM permits to build access control policies through user interfaces. We have discussed also EEM's principal features, and how the resolution algorithm orders the different steps of policy execution to deduce the access control decisions. However, if administrators specify conflicting policies, EEM is not able to detect inconsistency. Detecting conflicts will be our

challenge, together with finding a method to give suggestions to administrators for the correction process.

We have only mentioned briefly the main services offered by this tool. Those that are necessary to understand our work.

Note that the concepts of role hierarchy and permission propagation are not well developed in EEM. For this reason, our work does not deal with them.

CHAPTER 5 : CONVERSION OF EEM POLICIES TO ALLOY SPECIFICATION

5.1. Overview

The verification and testing processes are very important steps in order to build consistent access control systems. However, such essential tasks are not part of the EEM application. In this chapter we propose a solution which supports the validation of access control policies formally (using the Alloy language) and we present a method to automate the process of validation. In other words our solution attempts to verify the EEM access control policies through their corresponding formal specifications in Alloy. We systematically run formal specifications and we generate some suggestions if conflicts are detected. These suggestions will be useful for the correction of policies.

We demonstrate the feasibility and effectiveness of our solution by using the Alloy tool, which in its own turn uses a SAT algorithm.

5.2. Introduction

As discussed in the preceding chapters, organizing large sets of access control policies is a complex task for administrators. Any addition, deletion or modification of policies may cause many potential and unknown side effects ranging from policy conflicts to requirement violations. However checking this manually is hard, it takes time and cannot be exhaustive. Thus automatic and effective validation techniques should be deployed to detect conflicts.

We will demonstrate the use of formal verification techniques to identify conflicts and inconsistencies in EEM access control policies, the result will be improved quality of the policy sets.

5.3. Translation of EEM access control policies into Alloy

In this section we show our approach using examples. We start with sets to show how to compute intersection and union. We will present a complete example later on.

Suppose we have three groups of employees in a hospital, namely: *Doctors, Nurses* and *Surgeons*. Each user is allowed to access data (medical records) under their group membership. In addition there are *Administrators* who may or may not belong to these three groups, as shown in Fig 5.1:

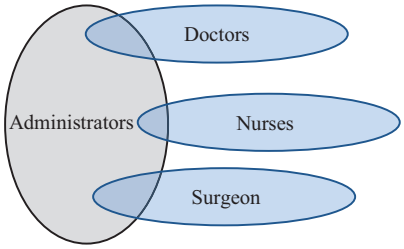


Fig 5.1: Groups relationships

Let us see how to represent *Objects*, *Groups*, *Actions* and *Policies* in Alloy. In the following we show the EEM representation on the left and the associated Alloy code on the right side (note that we don't discuss all EEM's XML tags, many of which are not useful for us):

```
1.  <ResourceClass>
2.    <Name>Form</Name>
3.    ……….
4.    <Action>Read</Action>
5.    <Action>Write</Action>
6.  </ResourceClass>
7.  <ResourceClass>
8.    <Name>File</Name>
9.    ……….
10.   <Action>Read</Action>
11.   <Action>Write</Action>
12. </ResourceClass>
13. . . . . .
14. <UserGroup Folder="/UserGroups" name= "Doctors">
15.     <Name>Doctors</Name>
16.     <Attribute Name="Name"> Doctors </Attribute>
17. </UserGroup>
    . . . . . .
```

```
1.  sig Group {}
2.  sig Doctors, Surgeons, Nurses extends Group{}
3.  sig Administrators in Group {}
4.  sig Object
5.  {
6.  }
7.  sig File, Form extends Object {}
8.  sig Action {}
9.  sig Read, Write, Execute extends Action {}
10. sig Authorization{
11. group: Group,
12. object: Object,
13. action: Action
14. }
15. sig Policy
16. {
17. auth: set Authorization
18. }
```

In the above (right side) Alloy code groups, objects and actions are represented as signatures. The authorization is represented as a signature which is composed by the triplet of *Group*, *Object* and *Action* (lines 10, 11, 12, 13).

On the left side (XML) we represent two resources (*Form* and *File*) and a group of users (*Doctors*). We can see in lines 1, 2, 3, 4, 5, 6 that actions (*Read* and *Write*)

are represented with the resource itself (inside the tag *ResourceClass*). In this case the tag *Name* represents the name of the resource, which is *Form* and actions are represented by the tag *Action*. Lines 14,15,16,17 represent the definition of group in EEM internal representation which is represented as a signature in Alloy (line 1).

Consider now the following user definition in XML:

```
1.  <User folder="/" name="Bob">
2.      <GroupMembership>Administrators</GroupMembership>
3.      <Attribute name="Section">SectionA</Attribute>
4.  </User>
```

```
1.  abstract sig user {
2.  belong : one group,
3.  pol :set policy,
4.  Section : one Section_ ,
5.  }
```

This is translated into Alloy in two steps: first we define the user signature (general definition) and second we define its attributes (line 4 in Alloy specification).

Next step, we have to say that user *Bob* belongs to the group *Administrators* (line 2 in XML code) and has an attribute called *Section* which has value *SectionA* (line 3 in XML code). The associated Alloy code is described in lines 1 and 4. So far we have introduced the generic definition of user, now we have to define the concrete relations in the *Fact* paragraph, as follows (note that we can define several *Facts* in the same Alloy specification):

```
1.  sig  Bob extends user {}
2.  Fact  { ……..
3.  Bob.belong=Administrators
4.  Bob.Section = SectionA
5.  ……}
```

In Line 3 and 4 we assign *Bob* to the group *Administrators* and *SectionA* as his working location.

5.4. Defining the properties to be checked

Once we have defined the entities (objects, actions, groups, and users) that represent a policy, we can go on to define the properties that our sets of policies should satisfy.

These rules should be defined in a *Fact* paragraph. Let us see some examples:

```
19.  Fact
20.  {
21.  // Specify the elements in the universe
22.  Group = Doctors + Nurses + Surgeons + Administrators
23.  Object = File + Form
24.  Action = Read + Write + Execute
25.  // Every Group have administrators
26.  Doctors & Administrators != none
27.  Surgeons & Administrators != none
28.  Nurses & Administrators != none
29.  // Each Group must be nonempty
30.  Doctors != none
31.  Surgeons != none
32.  Nurses!= none
33.  Administrators != none
}
```

- The set *Group* is the union of the sets *Doctors, Surgeons, Nurses* and *Administrators* (line 22).
- The set *Object* is the union of the sets *File* and *Form* (line 23).
- The set *Action* is the union of the sets *Read*, *Write* and *Execut*e (line 24).
- Every group has its own *Administrators* (lines 26,27,28), some of which are Doctors, Surgeons and Nurses (Fig 5.1).
- There must be at least one member in each group (lines 30,31,32,33).

Now we need to build our generic policies, to do this we have to define a predicate called *Scope* as follows:

```
1.  pred Scope (p: Policy, i:Identity, o: Object, a: Action)
2.  {
3.  p.auth.Identity = i
4.  p.auth.object = o
5.  p.auth.action = a
6.  }
```

In the above code we define a predicate that builds a generic policy by passing three arguments where *p, i, o, a* represent respectively the name of the policy, its identity (it can be a user or a group), the resource name and the action to perform on the resource. This predicate will help us to generate policies in further steps.

We show now how to build an assertion to check for separation of duties. In this case, we want to check whether it is possible for a subject to be involved in two

different permissions (*o->a*). In order to do this, we first build a more general assertion for two arbitrary policies *p1* and *p2* and we do the *Scope* restriction based on the first set *(Scope[p1, i1, o1, a1])*, the result of scoping must not be the same as the second set. In other words the relation *i1 -> o1 -> a1* (which represent the policy *p1*) should be different from the relation *i2 -> o2 -> a2* (policy *p2*).

```
1.  assert intersection
2.  {
3.  all i1, i2: Group, o1, o2: Object, a1, a2: Action, p1: Policy|
4.  (i1 = i2) && (o1 = o2) && (a1 != a2)  && Scope[p1, i1, o1, a1] =>
5.  ((p1.auth.Identity -> p1.auth.object -> p1.auth.action) !=
6.  (i2 -> o2 -> a2))
7.  }
8.  check intersection
```

To build such assertion we need specific information (a_1 and a_2 are in separation of duty) about the actions involved in the separation of duty relation since EEM tool does not support such feature. So in our case we insert the separation of duty constraint in the SEMPO tool.

This assertion could also be useful in other cases, for example if we need to specify a policy such as Chinese Wall. However, in order to specify separation of duty from the above code we derive an assertion to check it between two policies. To do this, we change lines 4, 5, 6 by the following code:

```
1.  (o1 = o2) && (a1 != a2) && Scope[p1, i1, o1, a1] && Scope[p2, i2, o1, a2]
2.  =>p1.auth.Identity!= i2)
```

Note that in the examples below the anomalies are easy to see because the rules that cause them are written together. In reality, these rules may be scattered among dozens of rules, and can be added or modified over time.

In the following we give a brief description of how we detect conflicts by using the Alloy analyzer. We don't show the signatures. We only show the *Fact* paragraph and assertions.

Example 1: Conflict between dynamic group and static group

P1: Static group of Programmers can't read/write in financial folder

P2: Static group of Financial employees can write the financial folder

P3: Dynamic group for Section A employees can read financial folder

In this case the SEMPO tool warns of an inconsistency scenario: Bob, a programmer, can be assigned to Section A. Bob cannot read the financial folder according to P1, can according to P3.

Possible resolution:

Either modify P1 allowing programmers to read the financial folder, or add a condition in the filter of P3 as follows: *and usergroup != software_programmers*.

To deal with this example, we begin with the specification of policies which is done in two steps: first we define which actions are invoked by the policies and second we associate identities to policies.

```
1.  <Policy folder="/" name="policy1_1">
2.  <Description>Static group of software_programmers can`t read/write the financial folder</Description>
3.  <ResourceClassName>financial_folder</ResourceClassName>
4.  ……
5.  ……
6.  <ExplicitDeny>True</ExplicitDeny>
7.  ……
8.  ……
9.  <Resource>File</Resource>
10. <Action>read</Action>
11. <Action>write</Action>
12. <Identity>ug:software_programmers</Identity>
13. </Policy>
14. . . . . .
```

This XML code expresses *policy1_1* which forbids (line 6: *ExplicitDeny is true*) actions *read* and *write* on resource *File*. This is shown in lines 1 to 3 below:

```
1. sig policy1_1 extends policy {
2. no_read : one File ,
3. no_write : one File }
4. sig policy1_2 extends policy {
5. write : one File }
6. sig policy1_3 extends policy {
7. read : one File}
8. ……
```

Policy1_1 represents the permission of *P1* which is a prohibition to execute the *read* and *write* actions on the resource *File*. Lines 2 and 3 illustrate two relations called *no_read* and *no_write* between policy signature and resource *File*. These relations represent the prohibition of *read* and *write* in the *File*. In addition, the Alloy code above also shows two other examples of policies:

1. *policy1_2* represents the permission of *P2* which is a permission to execute the *write* action on resource *File*.
2. *policy1_3* illustrates a *read* permission of *P3*.

Below is the associated XML format for lines 4, 5, 6, 7 in the above Alloy code

```
1.  <Policy folder="/" name="policy1_2">
2.  <Description>Static group of financial_employee can write the financial
    folder</Description>
3.  <ResourceClassName>financial_folder</ResourceClassName>
4.  ..........
5.  ..........
6.  <ExplicitDeny>False</ExplicitDeny>
7.  ..........
8.  ..........
9.  <Resource>File</Resource>
10. <Action>write</Action>
11. <Identity>ug:financial_employee</Identity>
12. </Policy>
```

The above XML code can be read as: the group *financial_employee* (line11) can (line 6) *write* (line 10) on the resource named *File* (line 9). Note that when explicit deny is false the policy expresses permission.

```
1.  <Policy folder="/" name="policy1_3">
2.  <Description>SectionA employees can read financial folder</Description>
3.  <ResourceClassName>financial_folder</ResourceClassName>
4.  ..........
5.  ..........
6.  <ExplicitDeny>False</ExplicitDeny>
7.  ..........
8.  ..........
9.  <Resource>File</Resource>
10. <Action>read</Action>
11. <Filter logic="nologic" lparens="0" col=" u:Section " optype="STRING" oper="EQUAL"
    val=" val:SectionA" rparens="0" />
12. </Policy>
```

In the XML representation line 1 defines the name of the policies, line 6 represents the type of the policy which can be a prohibition or permission, and line 10 represents the actions invoked by *Policies*. The resource is represented in line 9 by the specified tag *Resource*. The tag *Filter* in line 11 represents constraints in EEM (the named attribute *Section* should be equal to the value *SectionA*). We can interpret these attributes as follows:

- The attribute *logic* is used to build a conjunction or disjunction of constraints (it can be fixed to values *and, or*…).
- The attributes *lparens* and *rparens* represent respectively the number of parenthesis at the left and at the right of the constraint.
- The attribute *optype* and *oper* are used to respectively express the type of values (*SectionA*) and the operator. In our case we test equality on strings.
- The attribute *col* represents the user attributes (in our case we have only one attribute: *Section*)

The associated Alloy code below (line 5) states that if a given user u has his attribute *Section* fixed to *SectionA* value, so that user will be assigned to policy1_3 (P3).

The second step for defining access control policies is to associate identities to policies. To do this, we can use the Fact paragraph as follows (see the above XML code: line 12 *<Identity>ug:software_programmers</Identity>* and it associated Alloy code in line 2):

```
1. Fact{…….
2. all u:user | u.belong= software_programmers => u.pol=u.pol + policy1_1
3. // policy P2 and P3
4. all u:user | u.belong= financial_employee => u.pol=u.pol + policy1_2
5. all u:user | ( u.Section = SectionA )  => u.pol=u.pol + policy1_3
6. …….
```

Where *u* is a user, *u.pol*, *u.belong* and *u.Section* represent respectively the policies assigned to user *u*, the group to which *u* belongs and the *section* where the

user works (the "." operator is used for navigation between objects, see chapter 2). Line 1 defines the beginning of the *Fact* paragraph. Line 2 states that all users created in the Alloy environment, who belong to the *software_programmers* group, can use the permissions of *policy1_1* (if a user belongs to the *software_programmers* group then (=>) he can use the permissions of *policy1_1*). The fourth and the fifth lines can be interpreted in the same manner. As we can deduce, the attribution of permissions in *P1* and *P2* (lines 2 and 4) are based on groups. In policy *P3* the permissions are based on attributes (*Section*). Line 5 illustrates the following: if the *section* attribute of *user* is equal to *SectionA*, he can use the permissions of *policy1_3*.

Until now we have only described policies, now we have to build an assertion which allows us to detect conflicts. The assertion is illustrated as follows:

```
assert contradiction {
no u:user, disjoint x:u.pol, x1:u.pol ,p1:policy, p2: policy|
p1 + p2 in x + x1 && (p1.read = p2.no_read || p1.write = p2.no_write)
}
```

This assertion asks the system to check if *p1* and *p2*, which are two different policies (*disjoint*), are assigned to user *u*. *x* and *x1* represents the set of policies assigned to the user *u*. The system will detect a contradiction if *p1* and *p2* belong in the set of policies of user *u* (*p1 + p2 in x + x1*).

The above Alloy code asserts that for no user there are two different policies where the first provides permission (*read or write*) and the second provides a prohibition (*no_read or no_write*) for the same resource. If this condition is violated the analyzer will flag an inconsistency.

Example 2: Violation of Separation of Duty

P1: Billy can Create Accounts for web users

P2: Mark can Delete Accounts for web users

P3: If Mark is sick, and Billy is in good health, he delegates his rights to Billy

Requirement: No employee can have simultaneous access to both actions: *Create Account* and *Delete Account*. SEMPO tool warns of violation of requirement for *Bob* after delegation.

Possible resolution: Since probably the Requirement cannot be removed, *P3* must be changed to delegate Mark's rights to someone who does not have deletion rights.

To represent the delegation relation in Alloy, the *Fact* paragraph is as follows:

```
1. Fact{…….
2. Billy.health = good
3. Mark.health = sick
4. no u:user, x:u.Delegatee | u=x
5. no u:user, x:u.Delegator | u=x
6. Mark.health==sick -> Mark.Delegatee = Billy&&
   Billy.Delegator = Mark
7. ……}
```

In the above code, line 2 and 3, *Billy's* and *Mark's* attribute *health* is set respectively to *good* and *sick*. These lines represent the attributes (XML tag *Attribute*) and they are extracted from EEM application:

```
1.  <User folder="/" name="Billy">
2.    <Attribute name="health">good</Attribute>
3.  </User>
4.  <User folder="/" name="Mark">
5.    <Attribute name="health">sick</Attribute>
6.  </User>
```

This means that we have a scenario that activates the delegation policy *P3* (because *Mark* is *sick*). We have to exclude the possibility that a user can delegate actions to himself (line 4 and 5 in Alloy code), because this doesn't make sense in an access control systems. So we define two relations: *Delegatee* and *Delegator*. *Delegatee* is a user to whom the rights are delegated, and *Delegator* is the user who gives his rights to another user. Line 4 means: if a user *u* receives rights (is *delegatee*) from user *x*, then *u* and *x* should be different users (users can't delegate to themselves). The same interpretation holds for line 5: if a user *u* (*delegator*) delegates

his rights to user *x*, then *u* and *x* should not be the same user. Line 6 says: if *Mark* is *sick* so he delegates his rights to *Billy* and *Billy's* delegator is *Mark* (policy *P3*):

```
<Policy folder="/" name="policy3">
  <Description>if Mark is sick, he can delegate his rights to Billy</Description>
  ……

  <Delegator>Mark</Delegator>
  <ExplicitDeny>False</ExplicitDeny>
  ……
  <Identity>Billy</Identity>
  <Filter logic="nologic" lparens="0" col="u:health" optype="STRING"
oper="EQUAL" val="val:sick" rparens="0" />
  </Policy>
```

Now we need to build the assertion to be able to detect conflicts:

```
……..
assert deleg_separation_create_delete{
 no u:user, u1:u.Delegate, p:u1.pol, p1:u.pol|
p.create = p1.delete
}……
```

The above assertion states that if *u1* is a *delegator* and *u* is a *delegatee*, policies *u* and *u1* should never provide the permission of *creation* and *deletion* on the same resource. If this happens, Alloy flags a violation.

5.5. Programming techniques

In this project we have used two main technologies: JDOM and Alloy4 engine. JDOM is an open source API designed to represent an XML document and its contents to the typical Java developer in an intuitive way. JDOM was created to be Java-specific and thereby takes advantage of Java's features, including method overloading, collections, reflection, and familiar programming idioms. JDOM API most recent version is available as a jar file in: http://www.jdom.org/dist/binary/ (the JDOM.jar file should be accessible from the *classpath*). JDOM was used to parse EEM XML files and extract useful information to generate Alloy specifications. We also used JDOM to create XML reports (see next chapter). Following is a brief summary of what JDOM supports:

- org.jdom: defines the basic model of an XML file
- org.jdom.input: defines a stream for reading XML documents
- org.jdom.output: defines a stream for writing XML documents
- org.jdom.filters: classes for filtering nodes (very useful for parsing)
- org.jdom.xpath: XPath support (permits to define XPath requests)

The Alloy4 Java source code was designed to be modular and extensible, so it was easy for us to incorporate it in the SEMPO tool project and use it by calling class methods. Note that Alloy4 is used independently from other software components that are included into Alloy4 at /edu/mit/csail/sdg directory (we find a collection of components like: Kodkod, CUP Parser Generator for Java, JFlex scanner generator, the zChaff solver, the MiniSat solver, the SAT4J solver, NanoXML). Here we show a brief summary of the Alloy4 classes that we have used:

- alloy4: contains fundamental data structures and helper classes.
- alloy4compiler.ast: contains the definition for AST nodes.
- alloy4compiler.parser: contains the compiler.
- alloy4compiler.translator: contains the translator from Alloy4 syntax code to CNF (using kodkod)
- alloy4graph: is a self-contained library for doing graph layout
- alloy4viz: reads and displays Alloy4 instances and the relationship between entities of the specification.

The Alloy4 last version is available in: http://alloy.mit.edu/alloy4/alloy4.jar. The jar file contains also the source code (.java and .class files).

5.6. Conclusion

In this chapter we have presented the implementation of our formal method to detect inconsistencies in sets of access control policies. We have encoded a tool to translate EEM's XML access control policies into an Alloy specification. As well, we have demonstrated that this automated analysis is feasible and can detect inconsistencies. We have automated the process of validation and we have analyzed interesting properties such as direct contradictions, separation of duty, and contradictions caused by intersection between dynamic and static groups.

From the examples given, it should be clear that the process of translating from XML to Alloy is far from being straightforward, and in fact its implementation was the most time consuming component of our work.

CHAPTER 6 : SEMPO TOOL

6.1. Overview

The SEMPO tool was developed to help administrators to check access control policies built in CA`s EEM tool. It permits to extract policies from EEM and to analyze them through the Alloy engine. It allows also to detect conflicts that may exist in an EEM policy set and to help administrators to eliminate them.

In this chapter we discuss the SEMPO user interface by introducing some examples of policies. We show also how SEMPO detects inconsistencies, suggestions and produces graphic diagnostics.

6.2. Introduction

As mentioned in the previous chapters, organizing large sets of access-control policies is a complex task for administrators. Any addition, deletion or modification of any policy may cause unknown side effects such as policy conflicts or requirement violation.

We attempt to solve this problem by proposing a tool, called SEMPO, which analyzes sets of policies and provides administrators with the information that they need to avoid inconsistencies and to better manage the policies. This tool provides an interface that visualizes the result of the policy analysis process (diagnostics).

This chapter will be dedicated to the presentation of the SEMPO tool. In section 6.3 we discuss the main architecture of our tool and in section 6.4 we present the SEMPO user interface.

6.3. Main architecture

The SEMPO tool works in connection with CA`s EEM tool (Embedded Entitlement Manager, formerly known as eIAM, Embedded Identity and Access Manager). The latter tool allows specifying access control policies but does not analyze them. SEMPO extracts policies in EEM`s internal representation (which is in CA XML file format) and translates them to first order logic language, in the syntax needed for the following step. A logic analyzer is then used to identify anomalies. We

use MIT`s Alloy logic analyzer and its SAT solver. Alloy was integrated in SEMPO since it is encoded in the Java programming language, which makes it easy to introduce it as an API.

The tool interface provides clear-text diagnostics to administrators if anomalies are found. The administrator can then return to the EEM tool to correct these anomalies, and run the tool again to ensure that the set of policies is now free of anomalies.

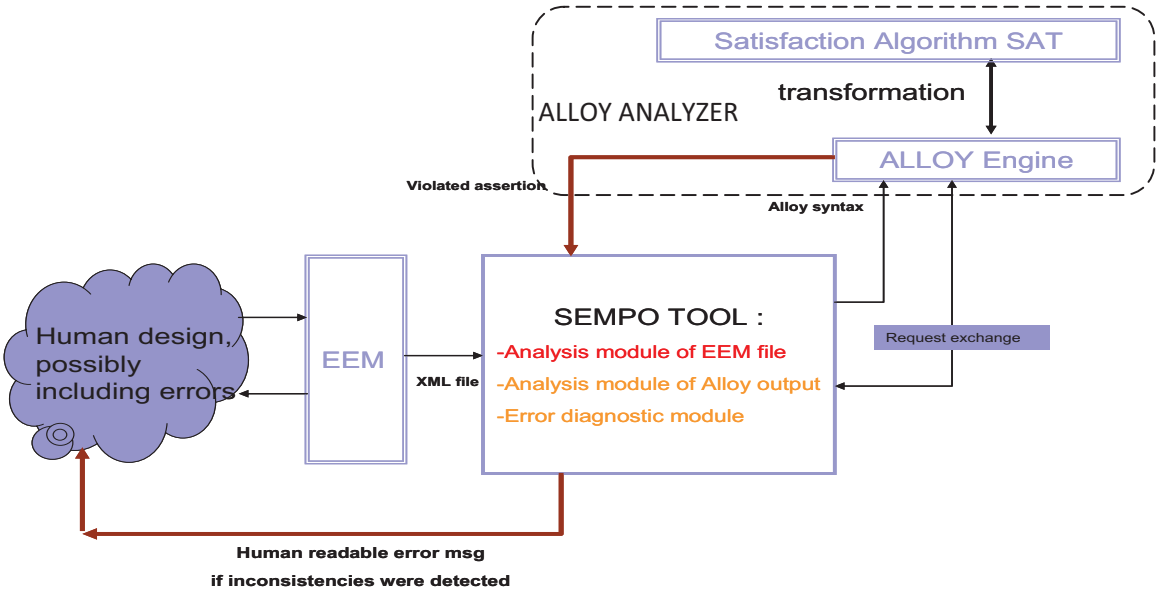


Fig 6.1: SEMPO main architecture

To represent access control policies at the concrete level, we should introduce concrete EEM entities (concrete EEM policies) in the SEMPO tool. For this purpose we use an object oriented approach. All entities (users, groups, policies…) are considered as objects (object oriented approach). Each object has an identifier and a set of attributes used to describe the object.

6.4. SEMPO user interface

We have designed the SEMPO user interface to be as user-friendly as possible to permit administrators to easily handle conflicts. We show the main interface in Fig 6.2 as follows:

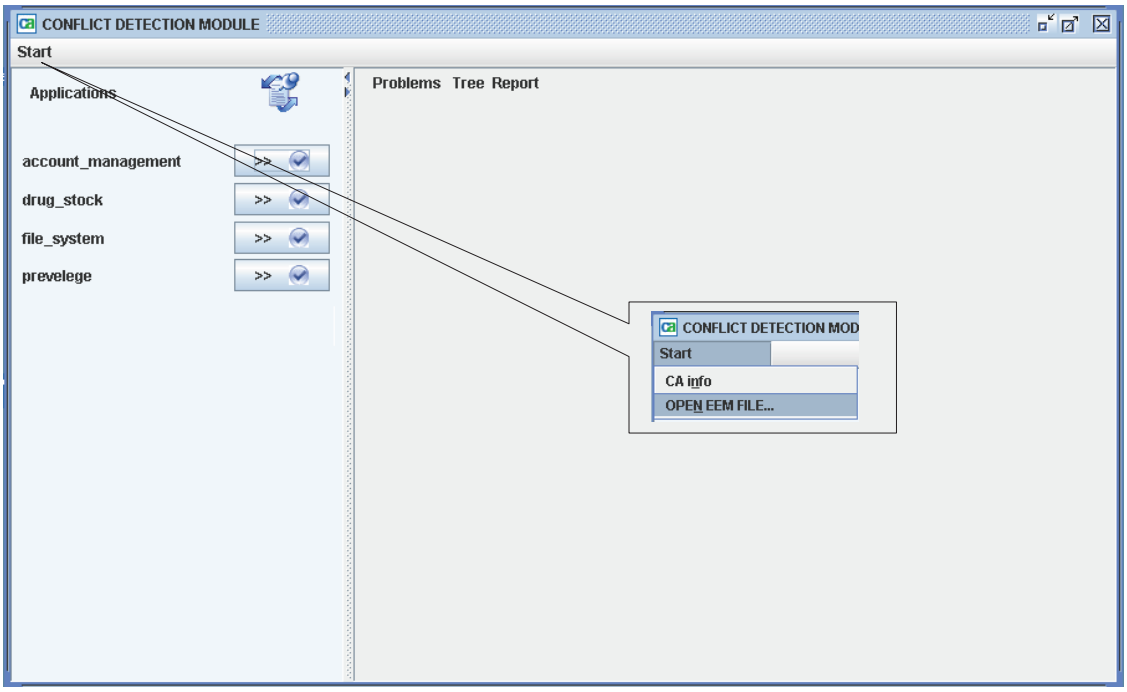


Fig 6.2: SEMPO main interface

In the above figure, the left side shows different applications, where each application represents a set of policies (the same applications are built in EEM, see Fig 6.3). These sets of policies exist in the policy database in XML format. The right side is dedicated to display conflicts if detected.

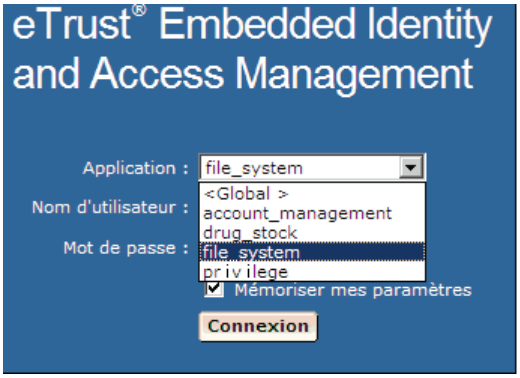


Fig 6.3: EEM list of applications

Fig 6.3 shows all the example files that we have built in EEM. The SEMPO tool will extract (through Safex tool) these applications for validation. The default application *Global* is not extracted; it is just used by the administrator to visualize all applications contained in EEM.

To load applications in SEMPO for the first time we have to go to the *Start* menu, then click on *OPEN EEM FILE*, and select the applications we want to analyze (at first execution we need to export the applications through the EEM user interface: *Configuration* > *Serveur Embedded IAM* > *Exporter*):

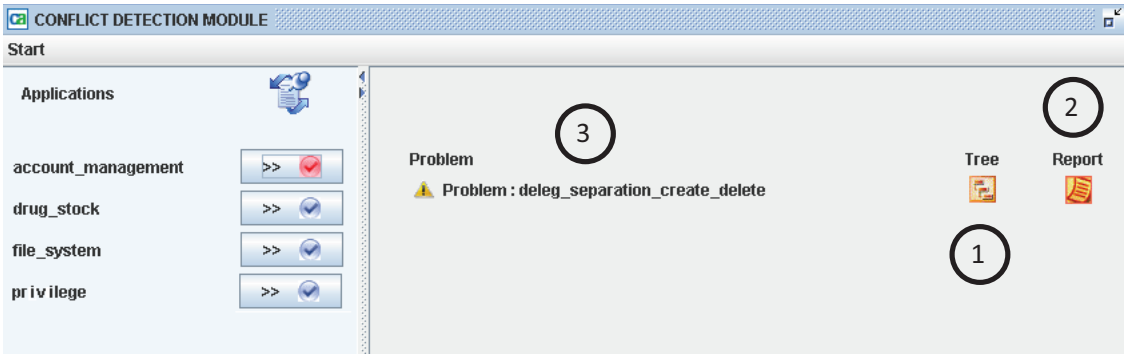


Fig 6.4: Loaded EEM applications

6.5. Example of inconsistency due to delegation

Suppose that we select the application named *account_management* (see Example number 4 in chapter 2 section 2.4.4), we can see all inconsistencies related to this application in the right side of the window. The application contains the following policies:

- P1: Bob can create account for web users
- P2: Mark can delete account for web users
- P3: If Mark is sick then he can delegate his rights to Bob
- ***Requirement: No employee can have simultaneous access to both actions:***
    - ***"create" and "delete "***

From the above policies we can deduce that if *Mark* falls *sick* and he delegates his rights to *Billy*, this will violate the requirement because *Billy* will be able to execute both actions *delete* and *create*.

In Fig 6.4 we can see the name of the example in the left side of the main interface (circle number 3). *deleg_separation_create_delete* represents the name of the violated Alloy assertion, which will be useful for experts to trace and have a detailed idea about the conflict detected. In addition SEMPO provides two representations (1 and 2 circled in the above snapshot). The first has the format of a simplified tree where it displays *identities* and *policies* descriptions. The second provides information about runtime execution of the whole validation process, human readable suggestions, ALLOY engine version used, EEM policies … In the following we show a different representation:

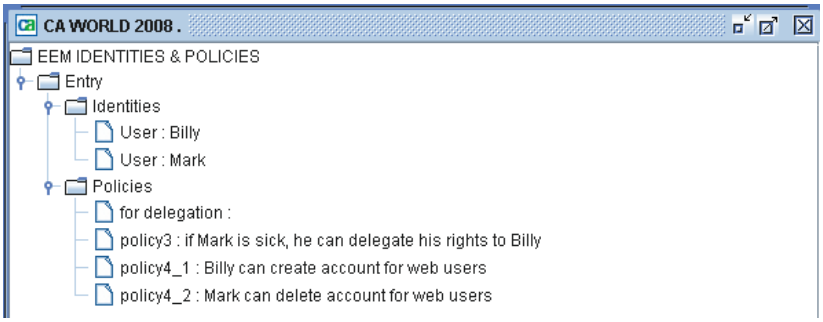


Fig 6.5: Tree representation

The tree presentation permits to have a quick idea on policies and users. The leaves under *Identities* level are represented as follows:

User: Name of the user

The leaves under *Policies* level are represented as follows:

Name of the Policy: description

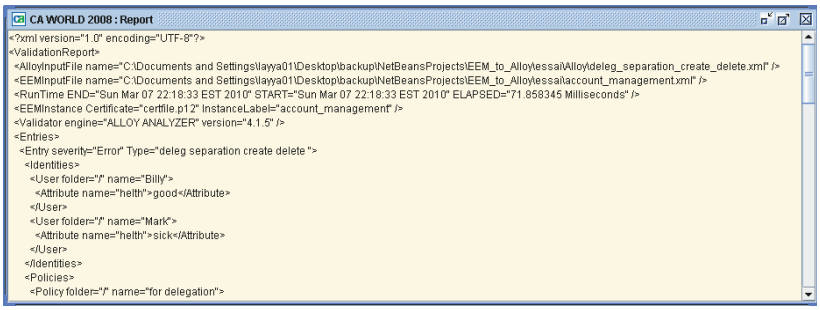


Fig 6.6: XML report

The XML report presents a detailed view of the inconsistencies and the policies. Here some important tags:

- AlloyInputFile: represents the output of Alloy violated assertion (this will be given to SEMPO for analysis).
- EEMInputFile: represents the EEM application (the set of policies).
- RunTime: represents the execution time.
- Suggestion: give some suggestions for correction.

Also SEMPO provides a human readable message for administrators, for this example (Fig 6.4) we get the following dialog box:

Fig 6.7: SEMPO suggestion message

The above message shows the users (*Billy, Mark*) and Policies (*policy4_1 and policy4_2* represented by P1 and P2 in the example above) that are involved in the conflict. So policies P1 and P2 are conflicting with the requirement. We can also see the relation between these entities to better understand the conflict:

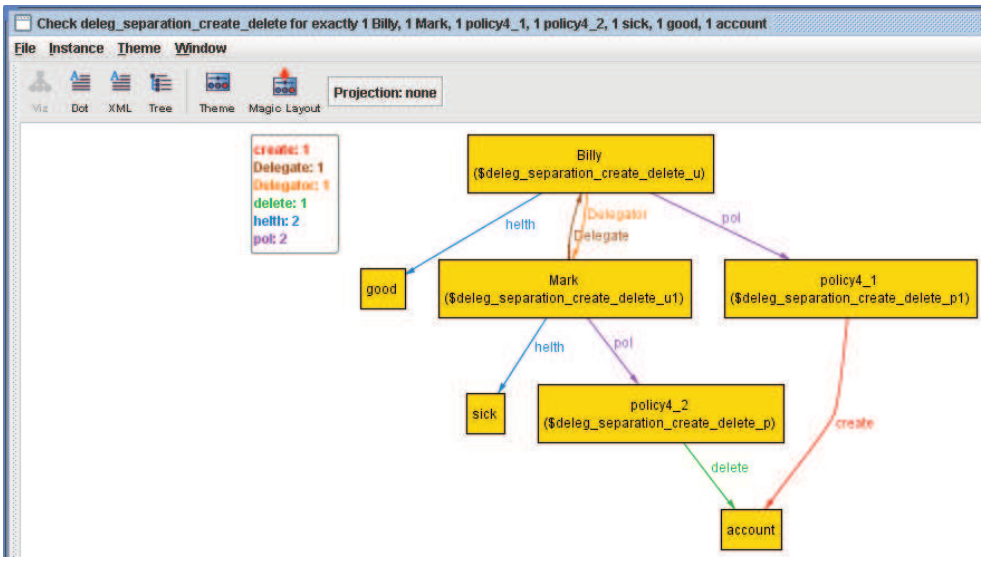


Fig 6.8: Entities relationships

This diagram provides information about access for both users *Billy* and *Mark*. *Mark* can *delete account* through policy *policy4_2* (the *health* status of *Mark* is *Sick*) and *Billy* can create account through *policy4_1*. Since *Mark* was *sick* so it is possible for him to delegates his rights (see relation *Delegate* and *Delegator* between *Mark* and Billy). Now *Billy* is able to *create an account* (*policy4_1*) and *delete account* through delegation. In this schema the entities part of the conflict are tagged by *$deleg_separation_create_delete* (it represent the assertion violated in Alloy code)

which means that the conflict was due to the violation of the separation of duty between actions *create* and *delete*.

To avoid this problem we can simply delete the delegation policy P3, so *Mark* can`t delegates his right to *Billy* even if he falls ill.

6.6. Example of inconsistency due to Incompleteness

In this section we will present an example which shows an inconsistency due to incompleteness (see section 2.4.2.2 in chapter 2). Suppose that we have the following policies in our system:

- P1: Group "Nurse" (named attribute ER), can "update" Resource "database"
- P2: Group "Nurse" (named attribute ER), can "Order" Resource "drug stock"
- ***Requirement: Group "Nurse" named attribute ER, must "update" and "order" (Atomicity)***

In this case the key policy is the requirement; it provides information about atomicity for actions *order* and *update*. Such requirement is useful to keep the system coherent, since the quantity of drugs in the drug stock should always be the same as in the *data base*.

Let see how the SEMPO tool reacts for these policies.

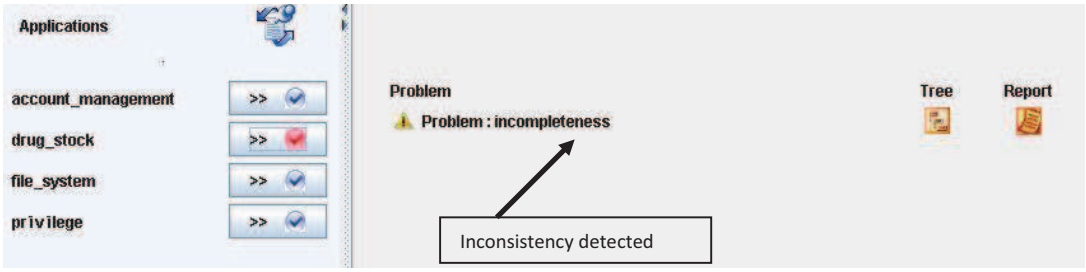


Fig 6.9: Validation of EEM *drug_stock* application

We can see that the SEMPO tool is capable of detecting this inconsistency (incompleteness in accordance to requirements). We can see that SEMPO displays a message warning that an incompleteness has been detected in the right side of the

interface. Now we want to know more about this, so we click on the icon. The following message will be displayed:

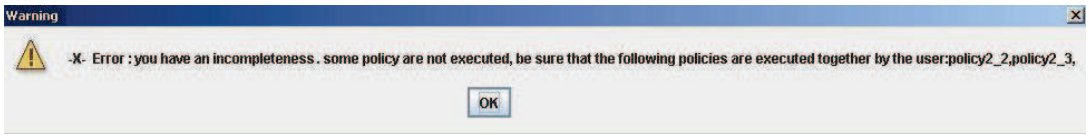


Fig 6.10: SEMPO suggestion message

The above message was generated because the Alloy engine finds scenarios where actions *update* and *order* are not executed together. In graphical form, this is shown as follows:

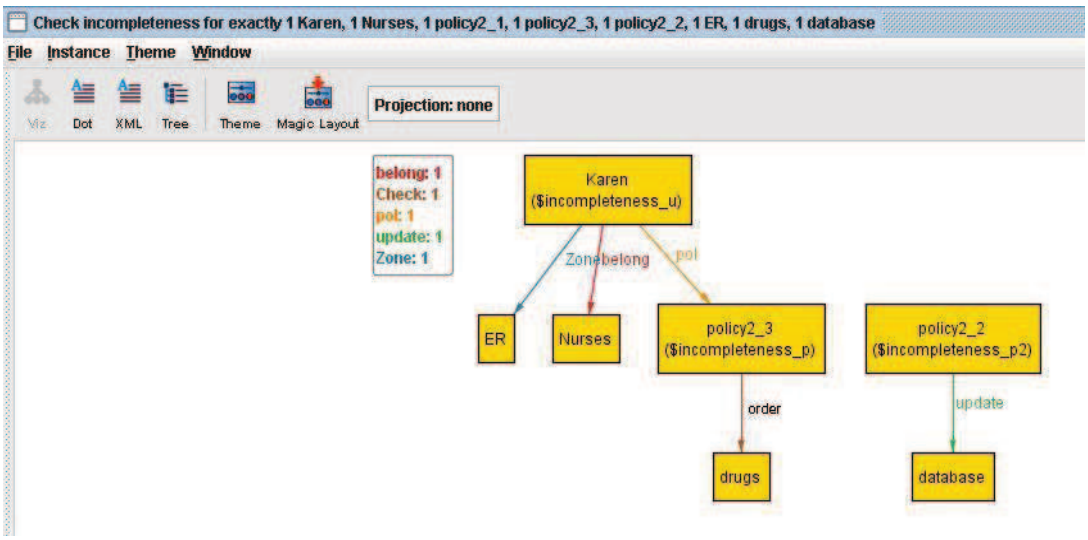


Fig 6.11: Incompleteness detected

In the above diagram *Karen* is a *Nurse* because of the relation *belong* and the fact that she works in the emergency room (*ER*). This user used only the action *order* through *policy2_3* (which represents policy P2) but not the action *update*. So the Alloy engine tags the user *Karen* by *$incompleteness,* showing which *Identity* is involved in this conflict.

In this case the correction is not feasible because the EEM tool does not support the atomicity feature.

6.7. Conflict between dynamic and static group

In this section we show an example (see example 1 in chapter 2, section 2.4.1) where the conflict is due to intersection between groups. As mentioned in chapter 4, in EEM we can define dynamic groups, these groups are based on constraints. Let us see the following example:

- P1:Static group of *software_programmers* can`t read/write the financial folder
- P2:Static group of *financial_employee* can write the financial folder
- P3:Dynamic group for *Section A employees* can read financial folder

At first sight there is no conflict between the above policies. However, if we take a look at the concrete level we can infer a conflict between policies P1 and P3. Here the concrete scenario:

- Bob : software_programmers => SectionA

As a consequence, *Bob* can`t *read* or *write* in *financial folder* (policy P1) since he belongs to *software_programmers* group. But also Bob can read in *financial folder* because his named attribute *Section* is equal to *SectionA*.

- Bob: $\underbrace{\neg Read \text{ and } \neg write}_{P1} \text{ and } \underbrace{Read}_{P3}$

Let us see what the SEMPO tool flags for the administrator:

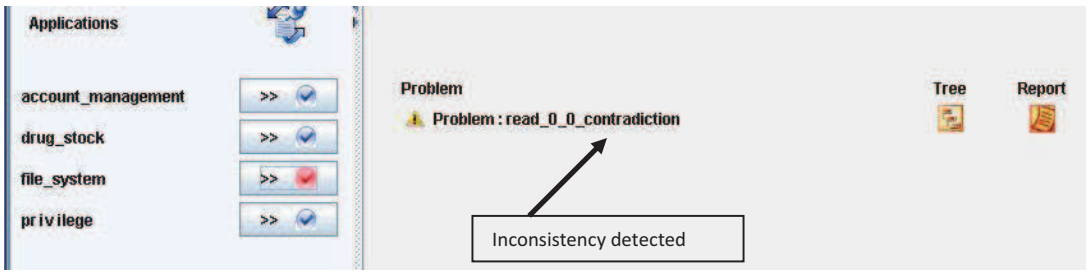


Fig 6.12: Validation of EEM *file_system* application

In the above figure we can see that SEMPO flags a conflict called *read_0_0_contradiction*. This is the name of the Alloy violated assertion which is generated automatically during the conversion from EEM policies into Alloy specification. This assertion was violated so intuitively by seeing the assertion name we can deduce that the conflict is about a contradiction caused by the action read. If we want to have a detailed view we have to click on the assertion ( ) to see a suggestion message which will be useful for the administrator to have an idea of the conflict and which users are involved in this conflict:

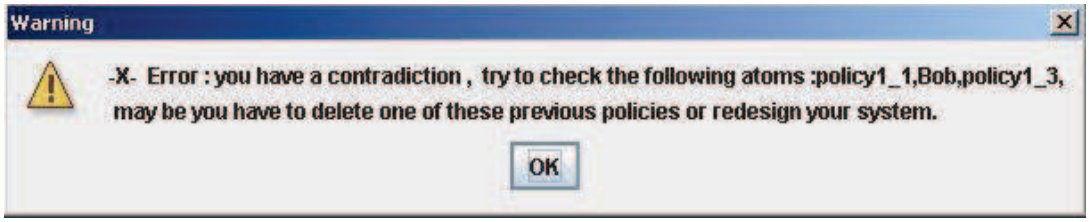


Fig 6.13: SEMPO suggestion message

The above message shows the different entities part of the conflict. The fact that *Bob* uses the *policy1_1* (P1) and *policy1_3* (P3) causes an inconsistency in our access control policies sets. Because through these policies, Bob is allowed and denied at the same time to read in the financial Folder. We can see it clearly in the following figure:

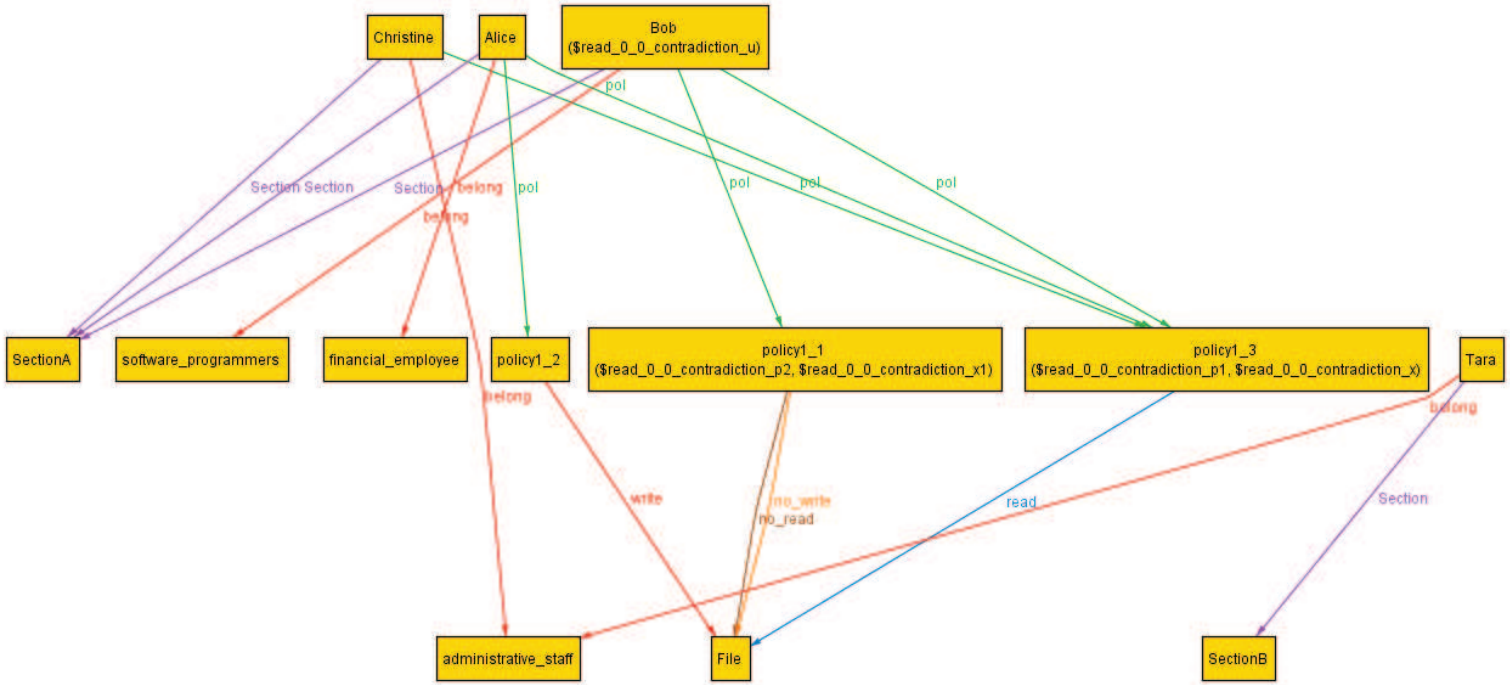


Fig 6.14: Conflict detected

Here, we can see a scenario where both policies *policy1_1* and *policy1_3* refer to *Bob*. *Policy1_1* denies him to read or write (*no_read*, *no_write*) on the resource *File* but *policy1_3* permits him to *read*.

To avoid this conflict we can add a constraint for policy P3, where *user group membership* should be different from *software_programmers*. So we have to go back to the EEM tool and modify the policy P3 as follows:

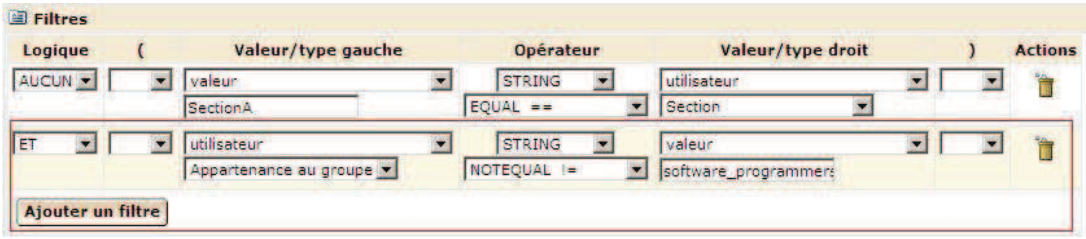


Fig 6.15: EEM filter user interface

Once we have done the corrections we have to go back to the SEMPO tool and press the button refresh  located at the top in the left side window shown in Fig 6.4 to re-export the application under EEM XML format and revalidate it once again in the SEMPO tool.

6.8. Conclusion

In this chapter, we have presented the architecture of the SEMPO tool, and we have shown how it can be used for the validation of access control policies. We showed also all the functionalities of SEMPO (relationship draw, suggestion dialogue,…).

The SEMPO tool was developed to help administrators in their access control policies design. It permits to detect inconsistencies and displays easy understanding diagnostics in the form of diagram, tree and XML reports. It also displays clear diagnostic messages for the administrators.

Some enhancements of our tool can be envisaged. First, our tool could be integrated in the EEM tool, so that the process of checking and revising policy sets

would not require going back and forth between the two tools; however this could be done only if we had available the source code of the EEM tool. If this could be done, then the second step would be to allow the SEMPO tool to analyze policies immediately as they are changed in EEM.

CHAPTER 7: CONCLUSION

7.1. Conclusion

Access control is one of the most popular and frequently used security mechanisms in today`s modern information systems. Enterprises are always looking for better access control systems to safeguard critical data and resources. On the other hand these systems can be complex since they can express multiple kinds of constraints (related to the location, user group, user ID etc). Hence, using a robust access control system is not enough to secure enterprise data. We have to be sure that our access control policies are not conflicting.

We had the chance to work in collaboration with CA Technologies and experiment with their software called EEM (Embedded Entitlement Manager). This tool permits to build, manage and deploy access control policies for enterprises. But it does not detect inconsistencies.

In this work our mission consisted in finding solutions to detect conflicts in EEM`s access control policy sets. Also, the solutions should be easy to use because they may be used by non experts. Thus we need to automate the validation process of access control policies. We chose Alloy as a formal language to validate and detect conflicts in EEM access control policies. We built a program called SEMPO which interfaces between EEM and the Alloy analyzer. SEMPO has the main task of converting EEM XML policies into Alloy specifications and call the Alloy engine for validation. Once the validation is finished, SEMPO interprets the output of the Alloy engine and generates human readable messages for administrators.

Chapter 1 was dedicated to give brief definitions of concepts related to access control policies (subject, object, constraints…). We also explained what motivates our work and how we can solve the inconsistency detection problem.

Chapter 2 discussed the representation of policies and conflicts in a logical formalism in order to better understand their characteristics. We chose first order logic because it is sufficiently powerful to express in an easy and natural way the access control policies. We also presented an overview of the Alloy language.

Chapter 3 was dedicated to the Literature Review, where we discussed a collection of conflict detection techniques. We began with the Free Variable Tableau method [14], where the detailed process of conflict detection was described (translation of access control policies to a logic notation and processing the analysis tree). Then we discussed the work done in [18], which is related to the specification of properties of RBAC96 model. We have shown how we can specify some properties of the RBAC model in the Alloy language. Then we introduced the work of [17], which is related to the validation of access control policies written in the XACML language. We introduced the main architecture of XACML policies enforcement and the conflict resolution mechanism. [11] provides a formal notation for XACML access control policies, to facilitate their translation to logic expressions, which are then used as input for a SAT Solver. This work showed how with this method it is possible to check whether policy properties are respected or not. However some shortcomings of these techniques were identifies, especially the fact that they don't deal with constraints.

Chapter 4 was dedicated to describe an industrial tool called EEM (Embedded Entitlement Manager). We presented the main features of this tool and we showed how access control policies can be built on it. We described also the conflict resolution algorithm used in EEM.

Chapter 5 demonstrated the use of formal verification techniques to identify conflicts and inconsistencies in EEM access control policies, including constraints (EEM's filters). We showed also how we translate CA`s XML files (EEM access control policies) to an Alloy specification.

In Chapter 6 we have proposed a tool, called SEMPO, which analyzes sets of policies and provides administrators with the information that they need to avoid inconsistencies and to better manage policies. We discussed SEMPO`s main architecture and user interface by introducing some examples of policies. We showed also how SEMPO detects inconsistencies, generates suggestions and displays graphic diagnostics (workflow of policies).

7.2. Contributions

The related work presented in chapter 3 is very interesting and can be implemented to detect conflicts but none of it treats the constraints or takes into account complex scenarios such as conflicts between access control policies and delegation policies, or conflicts between requirements (high level policies) and access control policies. Our work does this and detects inconsistencies by using the Alloy engine which in its own turn uses a SAT solver.

In this thesis, we recommend a solution to detect conflicts in sets of policies. Our solution is based on validation of access control policies by using a formal language called Alloy and Boolean satisfaction methods (SAT).

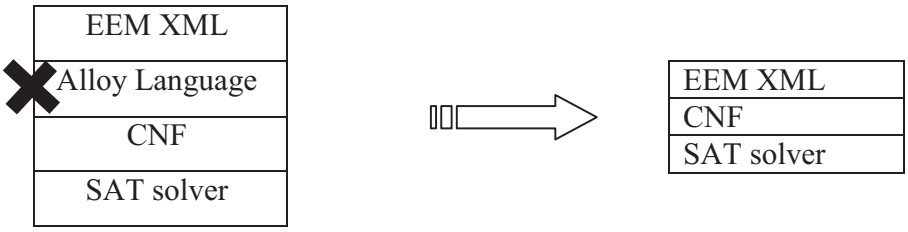
In the following we mention our original contribution:

1. Automated validation process
2. General solution, which can be used with access control system other than EEM
3. A generic formal representation which can express many types of access control policies
4. Support of access control policy constraints (EEM's filters)
5. SEMPO user-friendly interface appropriate for non-experts
6. Precise information about conflicts is provided (entities involved in a conflict are well identified)
7. The tool provides clear schematic diagrams which give a general view of the conflict situation

In our approach, access control policies are expressed through first order logic and are compiled for a SAT solver. We chose this approach because it permits us to compute all possible scenarios for a given configuration and check violation of assertions.

7.3. Future work

As a future perspective, it would be interesting to investigate the direct utilization of SAT solvers. The translation would be made directly from EEM XML policies to logic formulas in the format accepted by a SAT solver rather than going through the two intermediate translations of EEM policies into the Alloy language (implemented by us) and from the Alloy language to CNF formulas (implemented in the Alloy system).



By eliminating this intermediate translation phase, two important parameters would be improved:

- Complexity of the logic formulas: clearly, directly translated formulas would be simpler.
- Run time execution: the solution would then be computed faster.
- Complexity of the translation algorithm: the translation program would be simpler. As mentioned, the particularities of the Alloy language necessitate a complex translation program.

Bibliography

[1] Adi, K., Bouzida, Y., Hattak, I., Logrippo, L., Mankovskii, S. Typing for Conflict Detection in Access Control Policies. Proc. of the 4th Intern. Conf. MCETECH 2009. Ottawa, Canada May 2009, 212-226.

[2] Anderson, S. de O. Rewriting-Based Access Control Policies, Electronic Notes in Theoretical Computer Science, Volume 171, Issue 4, Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006). 10 July 2007. 59-72.

[3] Autrel, F., Cuppens, F., Cuppens, N., Coma-Brebel. C. MotOrBAC 2: a security policy tool. SARSSI'08 : 3ème conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information. Loctudy, France, octobre 2008. 13-17.

[4] Benferhat, S., El Baida, R., Cuppens, F. A stratification-based approach for handling conflicts in access control. In Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies. Como, Italy, June 02/03/2003. 189-195.

[5] Carlos, R., Andre, Z., Paulo, F., Paulo, G. Security Policy Consistency CoRR cs.LO/0006045. 2000.

[6] Cuppens, F., Cuppens-Boulahia, N., Ghorbel, M. B. 2007. High Level Conflict Management Strategies in Advanced Access Control Models. Electron. Comput. Sci. 186. July 2007. 3-26.

[7] El Kalam, A. B., El Baida, R., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C., Trouessin, G. Or-BAC : un modèle de contrôle d'accès basé sur les organisations. Cahiers francophones de la recherche en sécurité de l'information. 2003.30-43.

[8] Ferraiolo, D. F., Kuhn, D. R. Role-Based Access Controls. Reprinted from 15th National Computer Security Conference (1992). 1992. 554 - 563.

[9] forum. http://alloy.mit.edu/community. [Online Link] http://alloy.mit.edu/community/forum.

[10] Harrison, M. A, Ruzzo, W. L, Ullman, J. D. Protection in operating systems. Commun. august 1976. 461-47.

[11] Hughes, G., Bultan, T. Automated verification of access control policies using a SAT solver. Int. J. Softw. Tools Technol. Transf. Oct. 2008. 503-520.

[12] Ishakian, Vatche. ALLOY. 03 24, 2008.

[13] Jajodia, S., Samarati, P., Subrahmanian, V. S. A Logical Language for Expressing Authorizations. In Proceedings of the 1997 IEEE Symposium on Security and Privacy. May 04/07/1997.

[14] Kamoda, H., Yamaoka, M., Matsuda, S., Broda, K., Sloman, M. Access Control Policy Analysis Using Free Variable Tableaux. Transactions of Information Processing Society of Japan and Information Processing Society of Japan (IPSJ). Japan May 2006.

[15] Lupu, E. C, Sloman, M. 1999. Conflicts in Policy-Based Distributed Systems Management. IEEE Trans. Softw. Eng. 25, 6. Nov 1999. 852-869.

[16] Malik, S., Zhang, L. Boolean satisfiability from theoretical hardness to practical success. Commun. ACM 52, 8 (Aug. 2009), 76-82.

[17] Mankai. M, Logrippo, L. Access Control Policies: Modeling and Validation. Proceedings of NOTERE 2005. Gatineau, August 2005. 85-91.

[18] Schaad, A., Moffett, J. D. A lightweight approach to specification and analysis of role-based access control extensions. In Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies. Monterey, California, USA, June 03/04/2002.13-22.

[19] Seater, R., Greg, D. tutorial4/. http://alloy.mit.edu. [Online Link] http://alloy.mit.edu/alloy4/tutorial4/.

[20] Strembeck, M. Conflict Checking of Separation of Duty Constraints in RBAC - Implementation Experiences. in: Proc. of the Conference on Software Engineering. Innsbruck, Austria, February, 2004.

[21] Thomas, R. K., Sandhu, R. S. Task-Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-Oriented Autorization Management. In Proceedings of the IFIP Tc11 Wg11.3 Eleventh international Conference on Database Securty Xi: Status and Prospects (August 10 - 13, 1997). T. Y. Lin and S. Qian, Eds. IFIP Conference Proceedings, vol. 113. Chapman & Hall Ltd., London, UK, August 1997. 166-181.