UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

VÉRIFICATION ET ANALYSE DES POLITIQUES DE CONTRÔLE D'ACCÈS : APPLICATION AU LANGAGE XACML

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

MAHDI MANKAI

JANVIER 2005

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

Ce mémoire intitulé :

VÉRIFICATION ET ANALYSE DES POLITIQUES DE CONTRÔLE D'ACCÈS :

APPLICATION AU LANGAGE XACML

présenté par
Mahdi Mankai

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

a été évalué par un jury composé des personnes suivantes :

Dr. Luigi Logrippo .... Directeur de recherche
Dr. Kamel Adi .... President du jury
Dr. Michal Iglewski .... Membre du jury

Mémoire accepté le : 13 janvier 2005

Remerciements

Je remercie tout d'abord Professeur Luigi LOGRIPPO pour son aide et ses conseils précieux. Les activités décrites dans ce mémoire sont, pour beaucoup, issues de directions de travail qu'il a initiées.

Je remercie également Professeur Kamel ADI et Professeur Michal IGLEWSKI, les membres du jury, pour leurs enrichissants conseils et suggestions.

Je tiens aussi à remercier le Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG) pour avoir subventionné ce travail.

Enfin, je remercie mon père qui a financé la majeure partie de mes études maîtrise. Sans son soutien, ce mémoire n'aurait pas eu lieu.

Table des matières

Remerciements **i**

Liste des figures **vi**

Liste des tableaux **viii**

Résumé **ix**

1 Introduction **1**

2 Le langage XACML **4**

2.1 Introduction 4

2.2 Principes 4

2.2.1 Requête 4

2.2.2 Règles de contrôle d'accès 6

2.2.3 Politiques de contrôle d'accès 8

2.2.4 Ensemble de politiques 10

2.2.5 Réponse 10

2.3 Architecture de XACML 11

2.3.1 Point d'administration des politiques 11

2.3.2 Point d'application des politiques 12

2.3.3 Point de décision des politiques 12

2.3.4 Source d'information de politique 12

2.4 Diagramme de flux XACML 12

3 Analyseur de modèle ALLOY **15**

3.1 Introduction 15

3.2 Notions de base 16
3.2.1 Micro-modèles 16
3.2.2 Atomes et relations 16
3.2.3 Structures dans ALLOY 17
3.2.4 Logique dans ALLOY 17
3.3 Le langage ALLOY 18
3.3.1 Signatures 18
3.3.2 Faits 20
3.3.3 Prédicats 21
3.3.4 Fonctions 21
3.3.5 Assertions 22
3.4 Analyse de modèle avec ALLOY 22
3.4.1 Fonctionnalités 22
3.4.2 limite de vérification 23
3.4.3 Exemple 23

4 Travaux connexes **25**
4.1 Travaux avec le formalisme RW 25
4.1.1 Le formalisme RW 25
4.1.2 Transformation vers XACML 29
4.1.3 Discussion 32
4.2 Travaux sur *Ponder* 32
4.2.1 Le langage de spécification de politiques *Ponder* 32
4.2.2 Composition des politiques 34
4.2.3 Contraintes dans *Ponder* 35
4.2.4 Conflits dans *Ponder* 36
4.2.5 Outils pour la détection de conflits 37
4.2.6 Discussion 38
4.3 Transformations sur les graphes 38
4.3.1 Spécification des politiques de contrôle d'accès 39
4.3.2 Discussion 40
4.4 L'outil *Margrave* 40
4.4.1 Représentation des politiques avec les MTBDD 40
4.4.2 Vérification et analyse 42

4.4.3 Discussion 43
4.5 Autres travaux 43

5 Modèle logique de XACML **45**
5.1 Structures dans XACML 46
5.1.1 Sujets, ressources et actions 46
5.1.2 Requête 47
5.1.3 Cibles 48
5.1.4 Effets 48
5.1.5 Règles 49
5.1.6 Politiques 49
5.1.7 Ensembles de politiques 50
5.1.8 Algorithmes de combinaison 50
5.1.9 Méta-modèle de XACML 51
5.2 Contraintes logiques 54
5.2.1 Évaluation des cibles par rapport aux requêtes 54
5.2.2 Réponse des règles 55
5.2.3 Réponse des politiques 55
5.2.4 Réponse des ensembles de politiques 57
5.2.5 Algorithmes de combinaison 58

6 Vérifications et analyses **61**
6.1 Relations entre cibles 61
6.1.1 Inclusion 62
6.1.2 Intersection non vide 62
6.1.3 Disjonction 63
6.1.4 Égalité 64
6.2 Interactions entre cibles 64
6.2.1 Chevauchement des cibles 64
6.2.2 Incohérence des cibles 66
6.3 Analyses et vérifications supplémentaires 68
6.3.1 Requêtes conflictuelles 68
6.3.2 Politiques positives et négatives 69
6.3.3 Tests de conformité 69

7 Transformation de XACML vers ALLOY **71**

7.1 Approche générale 71

7.2 Exemple d'application 73

7.3 Extraction des attributs 74

7.3.1 Extraction des attributs 74

7.3.2 Structuration des attributs 76

7.3.3 Génération des définitions d'attributs en ALLOY 77

7.4 Construction des politiques 78

7.4.1 Construction d'objets *JAVA* 79

7.4.2 Transformation des objets en spécifications ALLOY 81

7.5 Intégration 83

7.5.1 Intégration 83

7.5.2 Contraintes du domaine 85

7.6 Analyses et vérifications 86

7.6.1 Visualisation 87

7.6.2 Transformation vers ALLOY 87

7.6.3 Analyses et vérifications 89

8 Conclusion **93**

A Modèle *ALLOY* : Famille **95**

B Politique de contrôle d'accès en XACML **96**

C Modèle ALLOY complet **99**

Bibliographie **104**

Liste des figures

2.1 Requête XACML 6
2.2 Évaluation des requêtes dans XACML 7
2.3 Structure de XACML 11
2.4 Diagramme de flux dans XACML 13

3.1 Simulation dans ALLOY 23

4.1 Analyse du contrôle d'accès avec *Prolog* 30
4.2 Génération de XACML à partir de RW 32
4.3 Politiques de délégation dans *Ponder* 34
4.4 Détection de conflits dans *Ponder* 37
4.5 Exemple de conflits dans *Ponder* 38
4.6 Hiérarchie des domaines avec les graphes 39
4.7 Politique d'autorisation avec les graphes 40
4.8 Politiques de contrôle d'accès avec les MTBDD 42

5.1 Sujets, ressources et actions 51
5.2 Ensembles de politiques, politiques et règles de contrôle d'accès 52
5.3 Requêtes 53
5.4 Algorithmes de combinaison 53
5.5 Effets des règles de contrôle d'accès 53

7.1 Approche de vérification des politiques de contrôle d'accès 72
7.2 Extraction des attributs 75
7.3 La classe Attribute 76
7.4 Génération des définitions d'attributs en ALLOY 77
7.5 Attributs formatés 77

7.6 Diagramme de classes 80
7.7 Eléments structurés 82
7.8 Cibles structurées 82
7.9 Outil intégré d'analyse 87
7.10 Visualisation de XACML 88
7.11 Règles redondantes 89
7.12 Règles conflictuelles 90
7.13 Règle inutile 91
7.14 Requête conflictuelle 92

Liste des tableaux

2.1 Requête XACML au format tabulaire 5

3.1 Relation binaire Parent-Enfant 17

7.1 Attributs et valeurs 74

Résumé

Le contrôle d'accès définit des contraintes et des règles d'autorisation. Pour exprimer les politiques de contrôle d'accès, plusieurs langages tels que XACML, EPAL ou PONDER, sont utilisés. Ces langages spécifient quels sujets sont (ou ne sont pas) autorisés à accéder à un ensemble de ressources ou services pour effectuer des actions spécifiques. Ces langages peuvent définir plusieurs règles et politiques de contrôle d'accès, mais ils n'offrent pas de mécanisme pour éviter les conflits et les incohérences. Par exemple, il est possible d'avoir plusieurs règles ou politiques appliquées à un contexte donné et aboutissant à des décisions contradictoires.

Nous proposons dans ce mémoire, une méthode basée sur la modélisation en logique de premier ordre pour analyser et détecter les interactions ainsi que les conflits présents dans un ensemble de politiques de contrôle d'accès exprimées en XACML. Le modèle logique obtenu est traduit vers le langage ALLOY. ALLOY permet de spécifier des ensembles de prédicats et d'assertions définissant les propriétés d'un système. Nous pouvons ainsi analyser les interactions et les conflits au sein des politiques de contrôle d'accès en utilisant un outil spécifique. Nous avons développé cet outil afin de permettre de visualiser les politiques XACML dans un format plus lisible et de les transformer automatiquement en un modèle ALLOY. De plus, cet outil utilise *ALLOY Analyzer* pour analyser et détecter les différents conflits et incohérences potentiels.

Abstract

Access control requires authorization rules and constraints. To express access control policies, several languages, such as XACML, EPAL or PONDER, are used. These languages specify which subjects can (or cannot) access sets of resources or services to perform specific actions. These languages can define several access control policies and rules, but they do not offer any mechanism to avoid conflicts and inconsistencies among them. In fact, it can happen that more than a rule or a policy, with opposite decisions, is applicable in a given context.

We propose a method based on first order logic modeling to analyze and detect possible conflicts and interactions within sets of access control policies expressed in XACML. We translate the model into a relational first order logic language called ALLOY. ALLOY allows to specify sets of predicates and assertions defining the properties of a system. We can then analyze interactions and conflicts among access control policies by using a dedicated tool. We developed this tool to visualize XACML policies and to translate them into ALLOY. In addition, this tool uses *ALLOY Analyzer* to verify and analyse potential interactions and inconsistencies.

Chapitre 1

Introduction

Le contrôle d'accès est une composante importante dans la sécurité des systèmes. Il permet de définir pour un ensemble d'entités (utilisateurs, programmes ou machines) les permissions pour accéder à une ressource ou un service [48]. Pour pouvoir définir ces autorisations, les matrices d'accès et les listes de contrôles d'accès *ACL* (*Access Control Lists*) ont été utilisées dans les systèmes d'exploitation. Ces matrices ont montré leurs limites, notamment dans le cas des applications distribuées. Des politiques et des règles de contrôle d'accès ont aussi été utilisées pour décrire les permissions d'une manière plus flexible qui répond aux exigences et aux contraintes du monde réel. Nous pouvons citer comme modèles de contrôle d'accès classiques :

- le modèle de contrôle d'accès discrétionnaire [37, 10] ou *DAC* (*Discretionary Access Control*)
- le modèle de contrôle d'accès obligatoire [46] ou *MAC* (*Mandatory Access Control*)
- le modèle de contrôle d'accès basé sur les rôles [12] ou *RBAC* (*Role-Based Access Control*)

Avec l'évolution rapide des réseaux de communication tels que Internet, les systèmes ont tendance à devenir de plus en plus répartis et donc accessibles à un nombre important d'utilisateurs. De plus, les applications communiquent entre elles et partagent leurs ressources y compris les politiques de contrôle d'accès. Différents langages sont utilisés pour exprimer le contrôle d'accès, chacun ayant sa propre finalité. Dans la littérature, il existe des langages formels pour permettre la vérification et l'analyse comme *ASL* [30], *RW* [19] etc. Ces langages sont souvent complexes et difficiles à exploiter. D'autres technologies comme XACML (*eXtensible Access Control Markup Language*) [16, 38] et EPAL (*Enterprise Privacy Authorization Language*) [50] commencent à gagner du terrain. Ces

technologies offrent des langages et des environnements fonctionnels permettant de gérer les autorisations d'accès. Toutefois, ils n'offrent aucun moyen de valider les politiques et les règles.

En effet, plusieurs facteurs peuvent justifier la présence de plusieurs politiques dans un système réparti, ou même dans un système centralisé : différentes politiques peuvent correspondre à différentes stratégies, différents groupes d'utilisateurs, différents types de ressources etc. Dès lors, des conflits peuvent exister dans ces politiques.

D'abord, deux règles ou deux politiques peuvent se contredire. Par exemple, si l'accès aux locaux d'une entreprise est régi par les deux règles suivantes :

1. Aucun employé ne peut accéder aux locaux de l'entreprise entre 20h et 6h.
2. Le gardien peut toujours accéder aux locaux.

Dans ce cas, il y aura un conflit possible puisque le gardien peut être considéré comme employé et risque donc de se faire refuser l'accès entre 20h et 6h.

Les langages de contrôle d'accès offrent généralement des mécanismes qui résolvent ce genre de conflits. Cependant, ces langages n'assurent pas que la décision finale soit la décision souhaitée. Nous croyons que l'administrateur et les utilisateurs doivent être avertis de ces conflits et donc disposer de moyens permettant de les détecter.

En plus, chaque système doit satisfaire à des propriétés et à des contraintes. Établir un ensemble de politiques de contrôle d'accès ne garantit pas que ces exigences soient respectées. Le non-respect de ces exigences peut engendrer des failles de sécurité dont les conséquences peuvent être particulièrement néfastes.

Enfin, les langages de contrôle d'accès sont souvent complexes et difficiles à interpréter pour des utilisateurs ordinaires. Ceci est un facteur supplémentaire qui peut favoriser la présence de conflits et d'incohérences dans les politiques de contrôle d'accès.

Dans ce mémoire, nous proposons de traiter cette problématique de vérification et d'analyse de conflits dans les politiques de contrôle d'accès. Nous avons choisi comme langage d'application XACML dans sa version 1.0. Le choix de XACML est justifié par les considérations suivantes :

- XACML est un standard *OASIS* (*Organization for the Advancement of Structured Information Standards*) [44], il commence à être largement utilisé par les entreprises et fait l'objet de nombreuses recherches.
- XACML est un langage généraliste qui n'est pas limité à un modèle particulier. De plus, il utilise des technologies standards et libres.

Nous proposons dans ce mémoire d'étudier les moyens et les méthodes permettant de détecter les conflits dans les politiques de contrôle d'accès ainsi que de proposer et d'implémenter certaines solutions à ces fins. Nous avons choisi d'utiliser le langage de modélisation et l'outil de vérification et d'analyse ALLOY [22]. C'est un langage formel qui permet de définir des modèles et de les valider automatiquement.

Comme deuxième but de ce projet, nous proposons de développer un outil jouant le rôle d'interface entre les utilisateurs et les politiques en XACML. Cet outil permettra d'afficher les politiques avec un format textuel plus lisible, de transformer les politiques de contrôle d'accès en modèle ALLOY et d'effectuer certaines analyses et vérifications. Nous utiliserons, pour ce faire, le langage Java et les transformations XSL [20, 55].

Dans le présent document, nous introduisons d'abord le langage XACML et l'outil ALLOY. Ensuite, nous présentons une revue de la littérature qui concerne les travaux semblables. Finalement, nous détaillons notre approche qui permet d'aboutir aux objectifs cités ci-dessus. Cette approche consiste à :

- construire un modèle du langage XACML basé sur la logique de premier ordre
- définir un ensemble d'interactions à étudier et à analyser
- transformer les politiques exprimées en XACML vers ALLOY et effectuer ces analyses

Chapitre 2

Le langage XACML

2.1 Introduction

XACML [44, 16, 42] est un ensemble de schémas *XML* [45, 43] qui définissent les spécifications d'un langage de politiques de contrôle d'accès. Avec XACML, il est possible de définir des règles de contrôle d'accès structurées en politiques et ensembles de politiques. Ces règles permettent de répondre aux requêtes qui demandent d'effectuer des opérations sur des ressources. La réponse peut être soit positive (*permit*) soit négative (*deny*). XACML fournit également un environnement pour concevoir et réaliser un système de contrôle d'accès.

Dans la suite, nous allons définir les principes du langage. Puis, nous décrirons l'architecture de l'environnement XACML. Enfin, nous introduirons le diagramme de flux de XACML. Nous présenterons également la syntaxe du langage au travers d'un exemple présent en annexe B.

2.2 Principes

2.2.1 Requête

Dans un environnement réparti tel que Internet, il y a un ensemble de ressources et de services à partager. Pour pouvoir contrôler l'accès à une ressource partagée, il est possible de définir un ensemble de règles. Une demande d'accès est une *requête* (*Request*) qui spécifie les éléments suivants :

		Attributs	**Valeurs**
Requête	Sujet	Identificateur	manm08
		Rôle	étudiant
		Institution	UQO
	Ressource	Identificateur	fichier de notes
		Propriétaire	manm08
	Action	Identificateur	lecture

TAB. 2.1 – Requête XACML au format tabulaire

- L'identité des demandeurs qui seront appelés les *sujets* (*Subjects*). Une requête dans XACML peut être initiée par plusieurs sujets notamment dans le cadre particulier d'un travail collaboratif.
- La *ressource* à accéder (*Resource*)
- L'*action* à effectuer (*Action*).

Ces éléments peuvent posséder des propriétés. Un sujet peut être défini par un identificateur, une institution à laquelle il appartient, un rôle etc. ; une ressource peut être caractérisée par un identificateur, un contenu structuré et un type ; idem pour l'action qui peut être définie par un identificateur. Comme exemple, nous pouvons considérer la requête suivante :

> Un sujet, avec l'identificateur *manm08*, le rôle *étudiant* et l'institution *UQO*, veut accéder à la ressource *fichier de notes*, dont il est le propriétaire, et effectuer une action de *lecture*.

Dans cette requête nous pouvons distinguer :
- Le sujet : manm08, étudiant de l'UQO
- La ressource : fichier de notes de manm08
- L'action : lecture

Les propriétés des sujet, ressource et action sont appelées attributs. Chaque attribut possède une valeur. La requête précédente peut être définie sous le format tabulaire présenté par le tableau 2.1.

Une requête peut contenir, en plus des informations concernant le sujet, la ressource et l'action, des informations optionnelles concernant l'environnement. Ces informations sont les propriétés qui ne sont associées ni au sujet, ni à la ressource, ni même à l'action, par exemple des informations sur un horaire ou une date.

Nous modélisons ainsi la requête dans XACML par le diagramme d'objets de la figure 2.1.

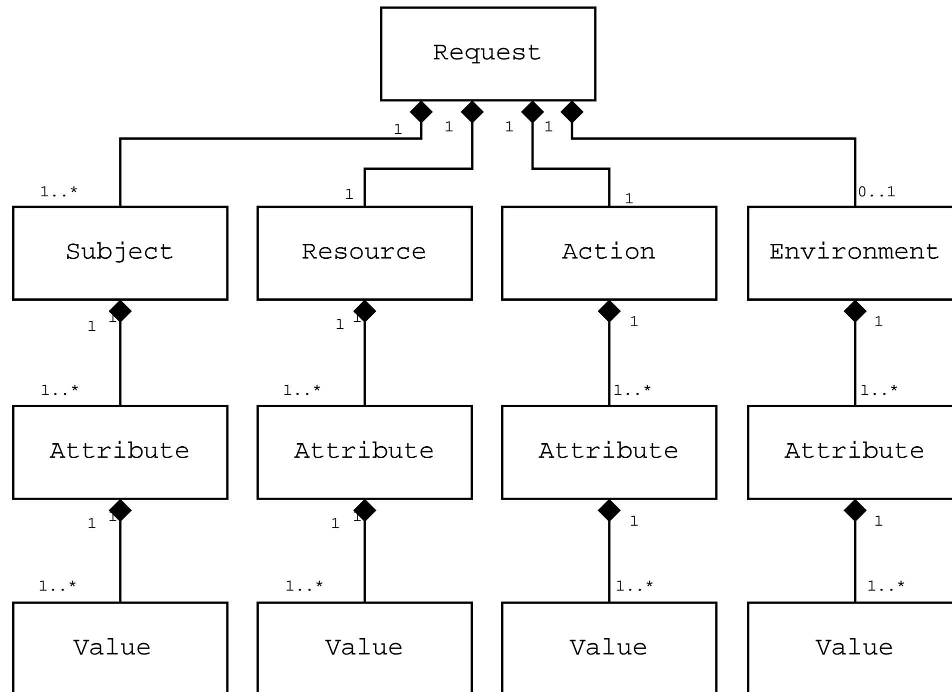

FIG. 2.1 – Requête XACML

Un système de contrôle d'accès tient compte de la requête et des informations qu'elle contient pour prendre une décision. Une décision est générée à partir d'un ensemble de règles (*rule*). Les règles sont regroupées en politiques (*policy*). Les politiques peuvent être regroupées en ensembles de politiques (*policySet*).

2.2.2 Règles de contrôle d'accès

Nous pouvons définir une *règle* de contrôle d'accès comme étant un ensemble de spécifications qui répondent aux questions suivantes :

- Quels sont les sujets concernés ?
- Quelles sont les ressources accédées ?
- Quelles actions demandées ?
- Y a-t-il d'autres conditions à satisfaire ?
- Quelle décision renvoyer ?

Une règle de contrôle d'accès définit un ensemble de conditions et une décision. La décision peut être soit positive, pour permettre l'accès à la ressource (*permit*), soit

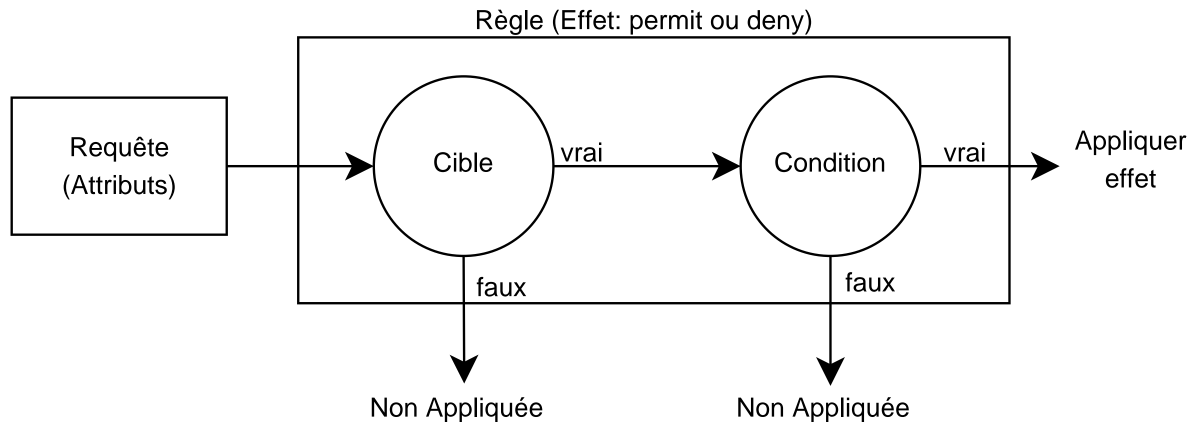

Fig. 2.2 – Évaluation des requêtes dans XACML

négative, pour en refuser l'accès (*deny*). La décision constitue la réponse d'une règle dans le cas où les conditions imposées sont satisfaites.

Une demande d'accès spécifie le sujet, la ressource à accéder, l'action à exécuter et les propriétés de l'environnement. Le système de contrôle d'accès doit évaluer ces paramètres et selon leurs propriétés générer une réponse *permit* ou *deny*. La manière la plus directe d'aboutir à une décision est de vérifier si les attributs des sujet, ressource et action correspondent à des valeurs particulières. Nous parlons dans ce cas de cible (*target*), qui représente une première étape pour savoir si une règle peut être appliquée à une requête ou non. Ensuite, une règle peut spécifier un ensemble de conditions supplémentaires et plus complexes à vérifier. Ainsi, si une requête correspond à la cible d'une règle, alors les conditions de cette règle seront évaluées, et si elles sont satisfaites, la réponse sera l'effet spécifié. Sinon, si la cible d'une règle ne correspond pas à la requête, ou bien si ses conditions ne sont pas satisfaites alors cette règle ne sera pas appliquée (voir figure 2.2).

Prenons l'exemple de règle suivant :

- Cible :
  - Le rôle du sujet est étudiant
  - La ressource est le fichier de notes
  - L'action est lecture
- Conditions :
  - L'identificateur du propriétaire du fichier de notes est égal à l'identificateur du sujet

- Effet :
  - Permettre l'accès

Si un étudiant demande de lire son propre fichier de notes, cette règle sera appliquée et retournera un *permit*.

Si un étudiant demande de modifier son propre fichier de notes, cette règle ne sera pas appliquée car elle cible seulement l'action de lecture.

Si un étudiant demande de lire le fichier de notes d'un autre étudiant, cette règle ne sera pas appliquée car sa condition n'est pas satisfaite.

Ainsi, une règle n'est appliquée que dans les situations spécifiées par sa cible et ses conditions. Pour faire face à plusieurs situations, il est nécessaire de définir plusieurs règles de contrôle d'accès.

2.2.3 Politiques de contrôle d'accès

Une *politique* (*Policy*) est une entité qui regroupe plusieurs règles de contrôle d'accès. En effet, l'objectif des langages de contrôle d'accès est de pouvoir fédérer plusieurs politiques, relatives à plusieurs systèmes répartis sur un réseau de communication. Il est clair qu'il faut grouper l'ensemble des règles d'autorisation en des ensembles, voir même des sous-ensembles, comme nous allons le voir plus loin, et ce, pour optimiser leur exploitation. Donc, si nous avons des règles de contrôle d'accès qui concernent les fichiers de notes dans une université et d'autres qui concernent les états de compte, il sera plus judicieux de les séparer en deux ensembles distincts. Ainsi, quand une demande de lecture du fichier de notes parviendra au système de contrôle d'accès, seules les règles concernées seront utilisées.

Comme une règle, une politique doit avoir une cible qui restreint son champ d'application à un ensemble limité de requêtes qui satisfont des conditions bien particulières.

Considérons la politique que nous appelons *Opérations sur les fichiers de notes*. Elle est appliquée dans le cas où une requête demande un accès aux fichiers de notes et contient les règles suivantes :

1. Un professeur peut lire et modifier les fichiers de notes de tous les cours qu'il enseigne. Mais il ne peut ni lire ni modifier les fichiers de notes des autres cours.
2. Un étudiant peut lire les fichiers de notes dont il est le propriétaire, mais ne peut pas les modifier.
3. Un étudiant ne peut ni lire ni modifier les fichiers de notes des autres étudiants.

4. Le personnel d'administration peut lire les fichiers de note de tous les étudiants.

Dans cet exemple, nous constatons d'abord que le champ d'application de cette politique est limité aux actions de lecture et modification sur la ressource fichier de notes, et aux sujets professeurs, étudiants et personnel administratif. Ainsi, nous pouvons spécifier la cible de la politique comme suit :

- Le rôle du sujet est étudiant, professeur ou personnel administratif
- Le nom de la ressource est Fichier de notes
- Le nom de l'action à effectuer est lecture ou modification.

Une fois une politique est appliquée à un contexte de requête, toutes les règles qui sont contenues dans la politique sont appliquées. Une sélection plus fine est alors obtenue. Les cibles des règles limitent le nombre de règles appliquées dans une politique. Ainsi, si un étudiant veut lire son fichier de notes, seules les règles 2 et 3 seront appliquées.

Puisque plusieurs règles peuvent être contenues dans une politique, et comme à chaque règle est associée une décision, alors nous pouvons avoir plusieurs règles qui s'appliquent à un contexte de requête particulier. Par conséquent, plusieurs décisions peuvent être prises. La façon la plus simple de faire face à de telles situations est de spécifier au niveau de chaque politique la manière de combiner les différentes règles et leurs décisions. Ces comportements sont appelés en XACML *algorithmes de combinaisons des règles* (*Rule Combining Algorithms*). Quatre comportements standards sont définis :

- *Permit-overrides* : si au moins une règle appliquée retourne un *permit*, alors la réponse de la politique sera *permit*. Si ce n'est pas le cas et il y a au moins une règle qui retourne *deny*, alors la réponse de la politique sera *deny*. Autrement, la politique n'est pas appliquée.
- *Deny-overrides* : si au moins une règle appliquée retourne un *deny*, alors la réponse de la politique sera *deny*. Si ce n'est pas le cas et il y a au moins une règle qui retourne *permit*, alors la réponse de la politique sera *permit*. Autrement, la politique n'est pas appliquée.
- *First-applicable* : appliquer la première règle qui s'applique
- *Only-one-applicable* : il faut qu'il y ait une seule règle applicable dans la politique. Si ce n'est pas le cas, la politique ne génère pas de réponse.

Le système de contrôle d'accès peut également demander des actions supplémentaires à exécuter en conjonction avec la décision générée. Ces actions sont appelées des obligations. Comme exemple, nous pouvons considérer l'envoi d'un message électronique de

notification à l'étudiant propriétaire comme étant une obligation en plus des opérations à effectuer sur le fichier de notes.

2.2.4 Ensemble de politiques

Il est encore possible de regrouper les politiques de contrôle d'accès en des ensembles de politiques (*PolicySet*) et ce, pour structurer les politiques et les règles. Nous aurons alors une structure arborescente d'ensembles de politiques puis de politiques et enfin de règles. Il est même possible de regrouper les ensembles de politiques (voir figure 2.3).

De la même manière, les ensembles de politiques possèdent une cible qui détermine leur applicabilité face à des demandes d'autorisation.

Aussi, et du fait que dans un même ensemble de politiques, plusieurs politiques peuvent s'appliquer et générer des réponses différentes, les algorithmes de combinaisons sont encore utilisés par les ensembles de politiques, mais cette fois-ci nous parlons de combinaison des politiques (*Policy Combining Algorithms*).

Enfin, nous pouvons associer des obligations à exécuter en plus de la décision d'un ensemble de politiques.

La figure 2.3 présente la hiérarchie entre les ensembles de politiques, les politiques et les règles de contrôle d'accès. Un ensemble de politiques doit spécifier une cible en plus d'un algorithme de combinaison qui permet de combiner les décisions issues des politiques et des ensembles de politiques enfants. Une politique doit aussi avoir une cible ainsi qu'un algorithme de combinaison, qui permet de choisir une décision parmi celles des règles filles. Enfin, une règle spécifie sa cible, sa condition et son effet (*permit* ou *deny*).

2.2.5 Réponse

La réponse d'un système de contrôle d'accès peut être soit positive (*permit*) soit négative (*deny*). Par contre, nous pouvons faire face à des situations exceptionnelles. Quand une erreur inattendue survient ou que le système n'arrive pas à répondre, la réponse est indéterminée (*Indetermined*). Par contre, si tout se déroule bien, mais qu'aucune règle (politique ou ensemble de politiques) n'est appliquée, car les cibles ou les conditions ne sont pas vérifiées, alors la réponse sera non-appliquée (*Not Applicable*). Notons que la seule réponse qui permette l'accès à une ressource est *permit*.

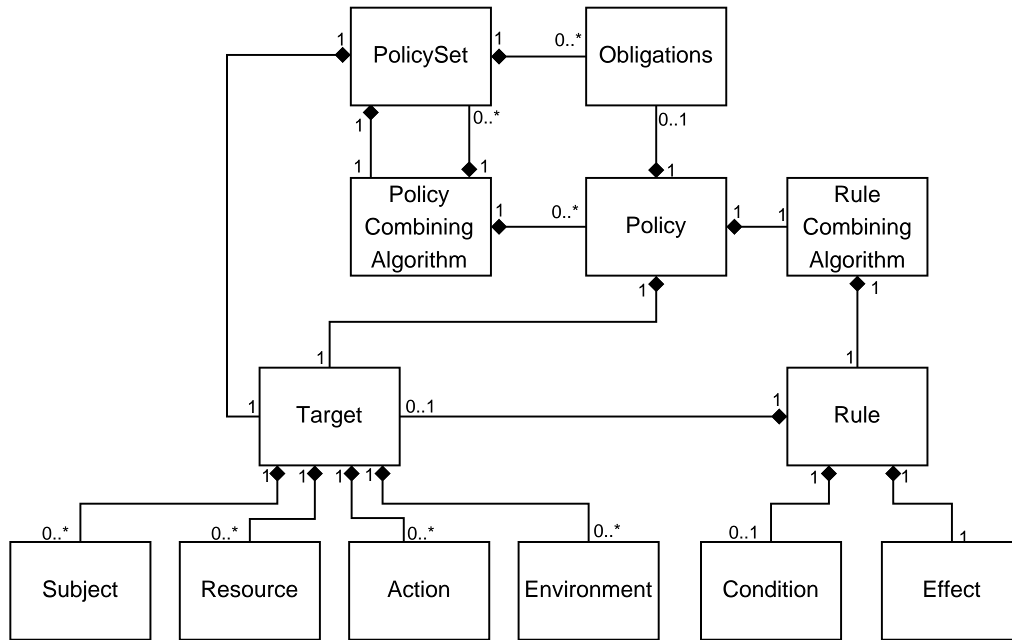

FIG. 2.3 – Structure de XACML

2.3 Architecture de XACML

Le langage XACML n'a pas seulement apporté une syntaxe, il a aussi apporté une architecture et des concepts qui fournissent les grandes lignes pour concevoir un système de contrôle d'accès. Cette architecture vise à atteindre plusieurs objectifs :

- Assurer une protection efficace des ressources et ce, du point de vue du contrôle d'accès.
- Permettre de concevoir un système indépendant de la plate-forme utilisée.
- Permettre d'intégrer le système de contrôle d'accès dans des applications déjà existantes.

L'architecture globale du langage consiste en plusieurs composantes collaborant entre elles. Nous détaillerons dans la suite ces composantes et les interactions qui les animent.

2.3.1 Point d'administration des politiques

Le point d'administration ou *PAP* pour *Policy Administration Point*, est l'entité qui crée les règles, les politiques et les ensembles de politiques de contrôle d'accès.

2.3.2 Point d'application des politiques

Le *point d'application des politiques* ou *PEP* (*Policy Enforcement Point*) est l'entité qui protège les ressources. Le *PEP* interagit avec des entités extérieures (applications ou utilisateurs) via des requêtes d'accès. Le *PEP* se charge d'envoyer la requête aux différentes autres entités pour avoir une réponse à la requête. Selon la réponse, le *PEP* accorde ou refuse l'accès. Le *PEP* travaille en collaboration avec le gestionnaire de contexte (*Context Handler*). Ce dernier a le rôle de transformer les requêtes initiales dans un format spécifique appelé le contexte XACML. Il contient les spécifications (attributs et valeurs) du sujet, de la ressource et de l'action. Le gestionnaire de contexte prend en charge également la transformation de la réponse en un format compréhensible par l'entité qui a généré initialement la requête.

2.3.3 Point de décision des politiques

Le *centre de décision des politiques* ou *PDP* (*Policy Decision Point*) est l'entité en charge de sélectionner les règles, les politiques ou les ensembles de politiques qui sont applicables à une requête donnée. Il évalue les cibles et les conditions afin d'aboutir à une décision.

2.3.4 Source d'information de politique

La source d'information de politique ou *PIP* (*Policy Information Point*) a pour rôle d'extraire les informations supplémentaires qui ne sont pas présentes dans la demande d'accès. Le *PIP* peut lui-même chercher les informations dans des sources externes. Ces sources externes peuvent être une base de données, un annuaire d'utilisateurs etc.

2.4 Diagramme de flux XACML

La figure 2.4 montre le flux de données dans un environnement XACML. Les étapes de traitement d'une requête XACML sont :

1. Le PAP génère des politiques ou des ensembles de politiques et les rend disponibles au *PDP* pour évaluation.
2. Une demande d'accès parvient au *PEP*.
3. Le *PEP* envoie la requête dans son format d'origine au gestionnaire de contexte.

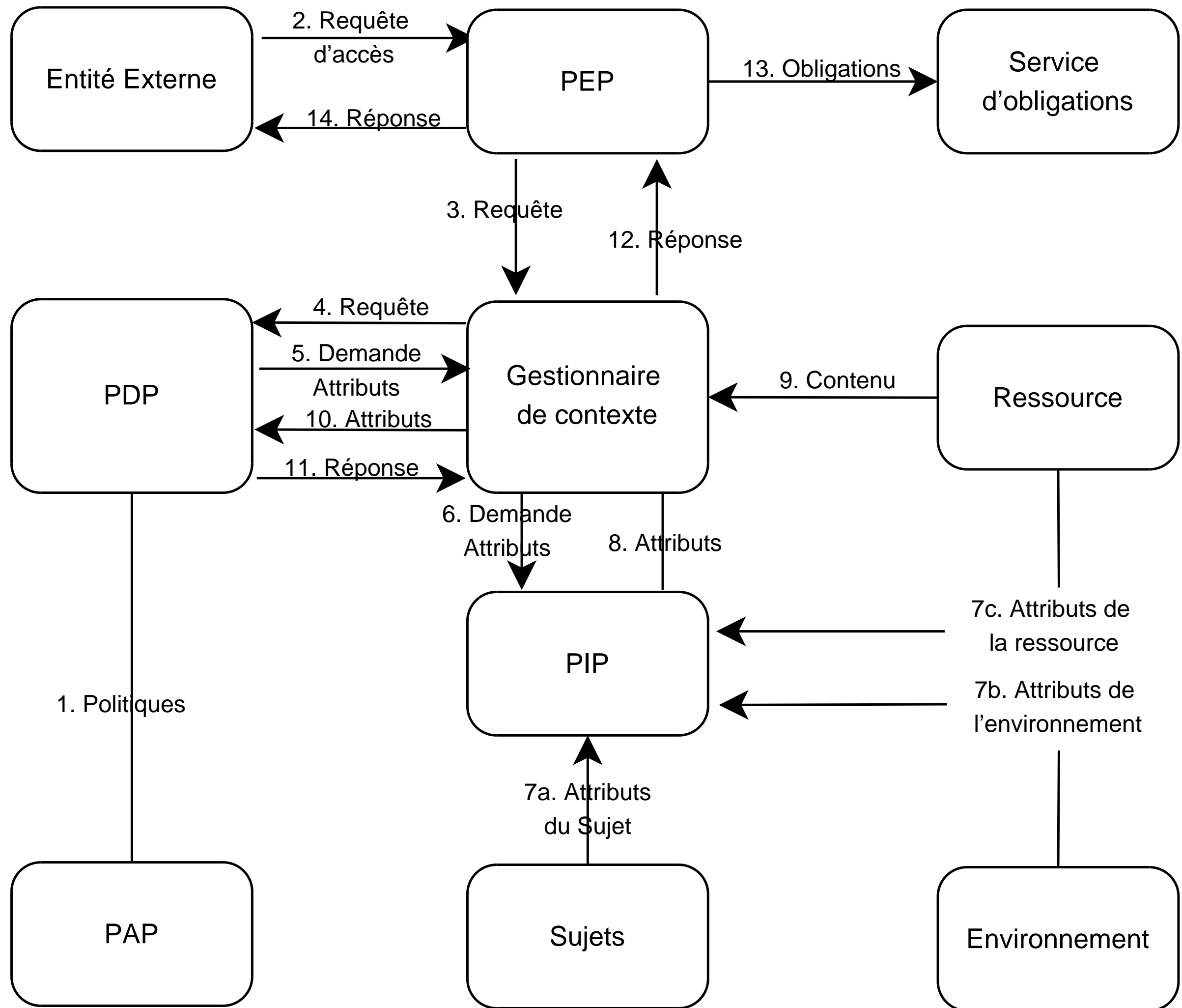

Fig. 2.4 – Diagramme de flux dans XACML

4. Le gestionnaire de contexte extrait les attributs des sujet, ressource et action. Il génère ainsi une requête au format XACML et il l'envoie au *PDP*.
5. Le *PDP* analyse la requête et au besoin envoie une demande d'attributs supplémentaires non contenus dans la requête. En effet, pour pouvoir évaluer les cibles et les conditions des politiques et des règles de contrôle d'accès, il faut les valeurs de certains attributs. Par exemple, si une requête spécifie juste la ressource à accéder, le *PDP* a besoin du propriétaire de la ressource.
6. Le gestionnaire de contexte demande au *PIP* les attributs manquants.
7. a, b et c : Le *PIP* extrait les informations nécessaires sur le sujet et la ressource à partir des sources externes qui contiennent toutes les informations sur les sujets, ressources.
8. Le *PIP* renvoie les valeurs des attributs demandés.
9. Au besoin, et dans le cas où la ressource contient des données structurées (fichier XML), elle envoie des informations concernant son contenu au gestionnaire de contexte.
10. Les valeurs des attributs sont envoyées au *PDP*.
11. Le *PDP* évalue les politiques disponibles par rapport à la requête et génère une réponse au format XACML. Cette réponse est envoyée au gestionnaire de contexte.
12. Le gestionnaire de contexte transforme la réponse XACML en un format compatible avec la requête initiale émise par une entité externe.
13. Le *PEP* vérifie les obligations et essaie de les satisfaire.
14. Le *PEP* envoie sa réponse à l'entité qui a demandé l'accès.

Chapitre 3

Analyseur de modèle ALLOY

3.1 Introduction

ALLOY [23] est un outil qui offre à la fois un langage [27] et un outil de vérification et de validation de modèles formels. Il a été développé par le groupe de recherche *Software Design Group* dirigé par *Daniel Jackson* du *MIT* (Massachusetts Institute of Technology).

ALLOY permet de modéliser les systèmes afin de les simuler, les vérifier et valider certaines propriétés. ALLOY permet de présenter une vue simplifiée des systèmes en faisant abstraction des détails d'implémentation et en mettant l'accent sur les propriétés et les contraintes. Le langage possède une syntaxe simple basée sur le langage Z [51]. Il possède les caractéristiques suivantes :

- ALLOY est un langage structurel puisqu'il permet de modéliser des structures complexes avec des hiérarchies et des relations.
- ALLOY est un langage déclaratif : il n'est pas un langage opérationnel, il ne permet pas de réaliser des traitements, mais il définit des entités avec des propriétés et des contraintes qui permettent de décrire les systèmes.
- ALLOY est un langage analysable. Les propriétés d'un modèle ALLOY peuvent être vérifiées, et le modèle peut être simulé avec l'analyseur ALLOY (*ALLOY Analyser*) [28].

Dans la suite, nous introduirons les notions de base d'ALLOY [26]. Puis, nous présenterons la structure du langage. Enfin, nous présenterons l'outil *ALLOY Analyzer*.

3.2 Notions de base

3.2.1 Micro-modèles

La philosophie d'ALLOY est de faire abstraction des détails complexes pour se concentrer sur les fonctionnalités importantes. Un modèle ALLOY est à la fois un modèle analysable et testable de plus, c'est un modèle formel et abstrait. Souvent, il n'est pas nécessaire de construire un modèle complet avec toutes les fonctionnalités puisque nous n'avons pas besoin de toutes les vérifier et valider.

Grâce à une syntaxe simple basée sur la logique relationnelle de premier ordre, nous pouvons créer des micro-modèles [29] contrôlables et analysables. Pour ce faire, ALLOY utilise deux notions principales : les *atomes* et les *relations*.

3.2.2 Atomes et relations

Les atomes sont dans ALLOY des entités élémentaires de base. C'est un concept abstrait qui sert à modéliser les aspects du monde réel.

Les relations représentent un concept qui sert à définir des corrélations entre les atomes. Les relations et les atomes coopèrent pour représenter différents aspects des systèmes.

Dans ALLOY les atomes sont structurés en types et chaque relation est définie par un ensemble de types. Nous pouvons considérer les relations comme un tableau à deux dimensions, où chaque colonne contient un ensemble d'atomes de même type. Ainsi, la relation relie les atomes des différentes colonnes entre eux. Les colonnes ne peuvent contenir que des atomes, elles ne peuvent pas contenir des relations (logique relationnelle de premier ordre [24]).

Prenons comme exemple l'ensemble des atomes (*Paul*, *Jean*, *Emma* et *Anna*) de type *Personne*. Le tableau 3.1 montre la relation binaire Parent-Enfant de type (Personne, Personne).

Une relation peut avoir une arité de N. Mais ALLOY se limite aux relations ternaires (trois colonnes). La relation présentée par le tableau 3.1 est une relation binaire, alors que les relations unaires modélisent les ensembles d'atomes.

Parent	Enfant
Paul	Jean
Anna	Jean
Paul	Emma
Anna	Emma

TAB. 3.1 – Relation binaire Parent-Enfant

3.2.3 Structures dans ALLOY

Le concept de relation permet de modéliser les structures. En effet, les relations peuvent être interprétées de différentes manières.

Les relations peuvent exprimer l'appartenance. Ainsi, si nous considérons un type *Pays* et la relation de type (*Personne*, *Pays*) qui associe chaque personne au pays auquel il appartient, la relation inverse de type (*Pays*,*Personne*) exprime le sens contraire.

Les relations peuvent aussi modéliser les hiérarchies comme dans le cas du tableau 3.1, et donc les liens entre atomes.

Nous pouvons également considérer les relations pour grouper des atomes partageant les mêmes propriétés.

Les relations peuvent être utilisées pour modéliser des attributs. Ainsi la relation de type (*Personne*, *Entier*) attribue à chaque personne son âge.

ALLOY fournit l'opérateur « . » (point), comme opérateur de navigation. Il est utilisé de la manière suivante :

```
atome.relation
```

Cette expression aura comme résultat l'ensemble des atomes qui sont associés à *atome* via la relation *relation*. Ainsi, et si la relation représentée par le tableau 3.1 s'appelle *enfant*, nous aurons :

```
Paul.enfant = {Jean,Emma}
```

3.2.4 Logique dans ALLOY

ALLOY supporte la logique propositionnelle et la logique de premier ordre (logique de prédicats) [21]. Par conséquent, les opérateurs logiques et les quantificateurs sont offerts en ALLOY, mais sous format ASCII.

Opérateurs logiques

Les opérateurs logiques dans ALLOY sont :
- **!** ou **not** pour la négation
- **&&** ou **and** pour le ET logique
- || ou **or** pour le OU logique

Les opérateurs d'implication sont :
- F => G : Si F alors G (**implies** est aussi utilisé)
- F <=> G : F si et seulement si G (**iff** est aussi utilisé)
- F => G,H : Si F alors G sinon H

F, G, H sont des formules logiques.

Quantificateurs logiques

ALLOY utilise plusieurs quantificateurs logiques de la manière suivante :
- **all** x : e | F : pour tous les éléments x de l'ensemble e, la formule logique F est vraie (quantificateur universel)
- **some** x : e | F : il existe un ou plusieurs éléments x de l'ensemble e pour lesquels la formule F est vraie
- **no** x : e | F : il n'existe aucun élément x de l'ensemble e pour lequel la formule F est vraie
- **lone** x : e | F : il existe au plus un élément de l'ensemble e pour lequel la formule F est vraie
- **one** x : e | F : il existe exactement un élément de l'ensemble e pour lequel la formule F est vraie
- **let** x = y | F : la formule F est vraie pour un élément x égal à l'élément y

3.3 Le langage ALLOY

Nous allons, dans cette section, présenter les principaux éléments du langage ALLOY. La syntaxe complète du langage est détaillée dans [27].

3.3.1 Signatures

Dans Alloy, toutes les entités sont appelées *Signatures*. La signature est l'unique élément permettant de représenter les types et les atomes dans un modèle ALLOY.

Chaque signature possède des attributs pouvant appartenir à d'autres signatures. La notion de signature est analogue à la notion de classe dans le paradigme objet [25] [11]. La signature représente, par conséquent, un gabarit pour ses instances.

Considérons la signature suivante :

```
abstract sig Personne {
  pere : lone Personne,
  mere : lone Personne
}
```

Nous avons donc défini une signature *Personne*. La signature *Personne* est définie par deux attributs : *pere* et *mere*.

Nous pouvons remarquer la présence de l'opérateur de multiplicité *lone* qui indique que chaque instance de *Personne* possède zéro ou un seul père et zéro ou une seule mère. Les opérateurs de multiplicité fournis par ALLOY sont :

- *lone* : pour indiquer une ou zéro occurrence
- *one* : Exactement une occurrence
- *set* : zéro ou plusieurs occurrences
- *some* : une ou plusieurs occurrences

Pour accéder à un attribut d'une signature, l'opérateur de navigation « . », appelé aussi opérateur de jointure, est utilisé.

ALLOY utilise également la notion d'héritage. Pour créer une signature qui hérite d'une autre, le mot-clé *extends* est utilisé. L'exemple qui suit montre que deux signatures *Homme* et *Femme* héritent de la signature *Personne*.

```
abstract sig  Homme, Femme extends Personne {}
```

La sous-signature est aussi appelée sous-type (*subtype*) et elle hérite de toutes les propriétés de la signature mère. Ainsi chaque homme (respectivement femme) possède un père et une mère.

Dans la définition de la signature *Personne*, nous remarquons la présence du mot-clé *abstract* qui signifie que *Personne* est une signature abstraite. Une signature abstraite est une signature qui n'a comme instances que celles de ses sous-signatures. Les signatures représentent donc des partitions de la signature héritée. Ainsi, une personne ne peut être qu'un homme ou une femme.

Les spécifications suivantes définissent deux hommes (*Paul* et *Jean*) et deux femmes (*Anna* et *Emma*). Le mot clé *one* impose la présence d'un seul *Paul* (respectivement *Jean*, *Anna* et *Emma*).

```
one sig Paul, Jean extends Homme {}
one sig Anna, Emma extends Femme {}
```

Grâce aux notions de signature et de relation, il est alors possible de définir des structures particulièrement complexes. Cependant, pour pouvoir modéliser un système, il faut ajouter des propriétés et des contraintes que le système doit satisfaire. C'est pourquoi nous allons introduire la notion de contrainte qui englobe les faits, les prédicats, les fonctions et les assertions.

3.3.2 Faits

Un *fait* (*fact*) dans ALLOY est une contrainte toujours vraie. Il y a deux types de fait dans ALLOY, les faits implicites et les faits explicites.

Les faits implicites sont les contraintes imposées par les définitions de signatures et d'attributs. L'héritage, l'utilisation de *abstract* et des opérateurs de multiplicité, les sous-ensembles et les sous-types sont des exemples de contraintes triviales.

Les faits explicites sont des contraintes moins évidentes exprimés directement par des formules logiques utilisant les opérateurs logiques et les quantificateurs de la logique de premier ordre. Un fait peut être introduit à la suite d'une définition de signature comme l'exemple ci-dessous, qui impose au père d'être un homme et à la mère d'être une femme.

```
abstract sig Personne {
  pere : one Personne,
  mere : one Personne
}{pere in Homme
mere in Femme}
```

Dans l'exemple précédent, les attributs *pere* et *mere* ont été introduits directement sans mentionner la signature à laquelle ils appartiennent puisque c'est un fait propre à la signature *Personne*.

Un fait peut aussi être défini par le mot-clé *fact*. Dans ce cas, les contraintes concernent toutes les signatures. Donc, chaque attribut introduit dans une formule logique sera précédé par la signature à laquelle il appartient. Ainsi, le fait précédent peut également s'écrire de la manière suivante :

```
fact {Personne.pere in Homme
      Personne.mere in Femme}
```

3.3.3 Prédicats

Les faits introduisent des contraintes toujours vraies, mais nous avons besoin aussi de définir des contraintes ou des propriétés pour lesquelles nous ne savons pas si elles sont vraies ou non. C'est pourquoi ALLOY définit la notion de *prédicats* (*Predicates*). Les prédicats servent à remplacer des formules logiques pour pouvoir les réutiliser. Un prédicat peut être défini avec des paramètres utilisés dans la formule logique du corps du prédicat. Il peut retourner *vrai* si sa formule est valide, ou *faux* sinon. Donc, si nous considérons l'exemple de la sous-section précédente, nous pourrons définir un prédicat pour indiquer que deux personnes sont frère et sœur, c'est-à-dire, qu'ils ont le même père et la même mère.

```
pred FrereSoeur(f:Homme, s:Femme) {
f.pere = s.pere or f.mere = s.mere }
```

En considérant les faits suivants, le prédicat *FrereSoeur (Jean, Emma)* retourne *vrai* puisque *Jean* et *Emma* ont le même père et la même mère qui sont respectivement *Paul* et *Anna*.

```
fact {
Jean.pere = Paul
Jean.mere = Anna
Emma.pere = Paul
Emma.mere = Anna }
```

3.3.4 Fonctions

Un prédicat remplace une expression qui retourne une valeur booléenne. Dans ALLOY, il est possible de définir des *fonctions* (*functions*) qui retournent des valeurs typées comme suit :

```
fun Mere ( p:Personne) : Femme { P.mere }
```

Ainsi, l'expression *Mere(Jean)* sera évaluée à *Anna*.

3.3.5 Assertions

Une *assertion* est aussi une formule logique évaluée à *vrai* ou *faux* mais qui ne peut pas être réutilisée comme les prédicats. Une assertion ne prend pas de paramètre, son rôle est de définir des propriétés que le système doit satisfaire et qu'ALLOY doit vérifier. L'exemple suivant montre la propriété qui consiste à dire que le père et la mère d'une même personne ne peuvent pas être la même personne.

```
Assert { no p : Personne | p.mere = p.pere }
```

3.4 Analyse de modèle avec ALLOY

3.4.1 Fonctionnalités

L'outil ALLOY offre deux fonctionnalités : la simulation et la vérification. D'une part, la simulation consiste à générer une instance d'un modèle pour laquelle une propriété donnée est vraie. Cette propriété est définie par un prédicat logique. Ainsi, la commande suivante va essayer de trouver une instance du modèle contenant deux personnes frère et sœur :

```
run FrereSoeur for 1
```

La réponse d'ALLOY est soit une instance de modèle sous format textuel ou graphique (diagramme ou arbre), soit un message indiquant qu'aucune instance n'est possible. Ceci veut dire que la propriété formulée par le prédicat n'est pas valide.

D'autre part, ALLOY peut vérifier des assertions sur un modèle avec la commande *check* suivie de l'assertion à vérifier.

```
check PereMere for 1
```

La réponse à cette commande est soit un contre exemple, ce qui implique que la propriété n'est pas valide, soit que la propriété est valide. Mais, cette validité n'est pas absolue puisque, l'analyseur ALLOY génère des instances dans une limite finie qui constitue la limite de vérification introduite dans la section suivante.

3.4.2 limite de vérification

Une commande ALLOY (*run* ou *check*) spécifie une limite (*scope*) d'instanciation. En effet, lors de la simulation ou la vérification, ALLOY génère un ensemble d'instances pour chacune des signatures présentes dans le modèle. C'est cette limite qui est spécifiée dans les commandes. Ainsi, dans les exemples cités dans la section 3.4.1, nous avons spécifier la limite 1 pour ne générer qu'une seule instance de *Paul*, *Jean*, *Anna* et *Emma*. Il faut rappeler aussi que les signatures abstraites ne sont jamais instanciées.

Cette limite a des conséquences sur la manière d'interpréter les résultats d'ALLOY. Quand ALLOY ne trouve pas de contre-exemple à une assertion, cela ne signifie pas qu'elle est toujours vraie, mais qu'elle l'est juste pour cette limite. De même, si aucune instance d'un prédicat n'est générée, cela veut dire que ce prédicat n'est pas vrai pour un certain nombre d'instance. Pour tirer profit des résultats d'ALLOY, il faut bien définir le contexte du modèle.

3.4.3 Exemple

Le modèle ALLOY incluant les spécifications complètes de l'exemple traité le long de tout ce chapitre, est défini en annexe A.

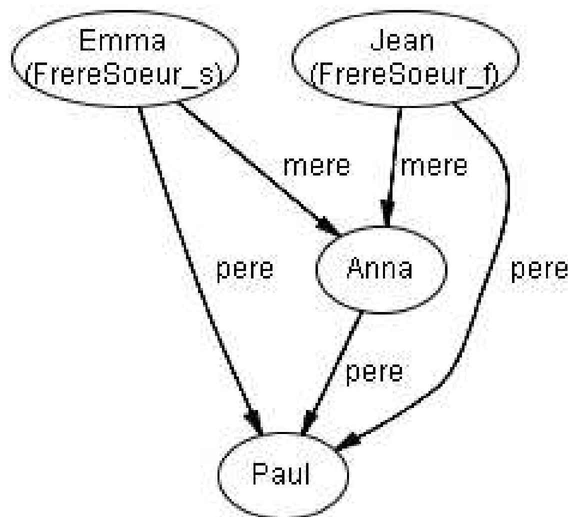

Fig. 3.1 – Simulation dans ALLOY

La figure 3.1 montre l'instance générée par la simulation du modèle pour le prédicat *FrereSoeur*. ALLOY a montré les deux instances concernées par le prédicat, ainsi que les instances paramètres. Donc, *Emma* est marquée par *FrereSoeur_s* (*FrereSoeur* est le nom du prédicat et *s* est le paramètre correspondant à la sœur). Le même principe est appliqué à *Jean*.

Chapitre 4

Travaux connexes

La vérification et la validation des politiques de contrôle d'accès ont suscité beaucoup d'intérêt dans la communauté de l'informatique. Cette problématique a motivé le développement d'une panoplie de formalismes et langages pour exprimer d'abord le contrôle d'accès et ensuite l'analyser et le valider. Dans ce chapitre, nous proposons une revue de la littérature dans ce domaine. Nous allons mettre l'accent sur quelques travaux similaires au nôtre et nous finirons par citer quelques autres travaux reliés.

4.1 Travaux avec le formalisme RW

Nous allons introduire dans cette section les travaux de Zhang, Ryan et Guelev sur la vérification des politiques de contrôle d'accès en XACML [57]. L'approche adoptée dans ce travail est différente de celle que nous proposons mais la finalité demeure la même. En effet, ces travaux consistent à considérer des spécifications de contrôle d'accès exprimées dans un formalisme spécifique appelé RW [19]. Les spécifications en RW peuvent être analysées et vérifiées à l'aide d'un programme *Prolog* [18]. Les politiques de contrôle d'accès en RW sont compilées et transformées en politiques XACML. Nous allons donner un bref aperçu simplifié du formalisme RW utilisé et de la transformation vers XACML.

4.1.1 Le formalisme RW

Systèmes de contrôle d'accès

Le formalisme RW pour le contrôle d'accès se base sur les définitions suivantes :

- P est un ensemble de variables propositionnelles.

- $\mathbf{L}(P)$ est un ensemble de formules logiques utilisant l'ensemble de variables P.
- Σ un ensemble d'agents (ou utilisateurs).
- $\mathcal{P}_{fin}(\Sigma)$ est l'ensemble des partitions finies de Σ.
- r et w sont deux relations de type $P \times \mathcal{P}_{fin}(\Sigma) \rightarrow \mathbf{L}(P)$. Les relations r et w indiquent, respectivement, l'ensemble des conditions nécessaires pour la lecture et l'écriture d'une variable par un ensemble d'agents.
- Un système de contrôle d'accès S est un uplet (P, Σ, r, w).
- Un état s d'un système de contrôle d'accès S est un ensemble de valeurs prises par les variables appartenant à P.

Considérons un système de contrôle d'accès $S = (P, \Sigma, r, w)$, une ressource modélisée par la variable $p \in P$ et un ensemble d'agents $A \subset \Sigma$. À un état s de S, les agents de A sont autorisés à lire ou modifier la ressource p si $r(p, A) = true$, respectivement $w(p, A) = true$. r et w sont définies par des formules logiques exprimées en fonction de p et A.

Programmes

Le formalisme RW introduit également la notion de *programme* [19]. Un programme est une séquence d'instructions qui permettent de manipuler les différentes variables propositionnelles d'un système de contrôle d'accès S en modifiant leurs valeurs. Un programme peut donc changer l'état d'un système. RW définit un ensemble de programmes $\mathbf{P}$. Un programme α possède la structure générale suivante :

$$\alpha \ ::= \ \text{skip} \mid p := \varphi \mid \text{if}\ \varphi\ \text{then}\ \alpha\ \text{else}\ \alpha \mid (\alpha; \alpha)$$

- *skip* : indique que le programme ne modifie aucune variable. Donc, le système reste dans le même état après l'exécution du programme.
- $p := \varphi$: indique que la valeur de la variable p prend la valeur de la variable φ.
- if φ then α else α : indique que si une variable φ est vraie alors un programme est exécuté sinon un autre peut être exécuté.
- $(\alpha; \alpha)$: indique qu'un programme peut être une séquence de plusieurs programmes.

RW définit aussi la fonction $[\![.]\!] : \mathbf{P} \rightarrow (P \rightarrow \mathbf{L}(P))$. Dire qu'un système S accorde toutes les permissions à un programme α pour s'exécuter à partir d'un état s et de manipuler une variable p signifie que $([\![\alpha]\!](p))(s) = true$.

Mais pour indiquer qu'un ensemble d'agents A peut exécuter un programme α, RW définit la fonction $[\![.,.]\!] : 2^{\Sigma} \times \mathbf{P} \rightarrow \mathbf{L}(P)$. Donc un système de contrôle d'accès S autorise les agents $A \subseteq \Sigma$ à exécuter le programme α à partir d'un état s si et seulement si $[\![A, \alpha]\!](s) = true$.

Finalement, RW définit $R_{A\varphi}$, $W_{A\varphi} \in \mathbf{L}(P)$ qui indiquent que $A \subseteq \Sigma$ peut finalement lire, respectivement écrire, les variables présentes dans la formule $\varphi \in \mathbf{L}(P)$. En effet, à un état s, les agents A peuvent ne pas avoir la permission d'effectuer une opération sur une variable $p \in P$, mais ils peuvent avoir la permission d'exécuter des programmes pouvant changer l'état du système et que le nouvel état permet l'accès à p.

Exemple

Considérons un système de contrôle d'accès aux fichiers. Supposons que les fichiers systèmes ne sont accessibles qu'aux administrateurs du système et que les autres fichiers ne sont accessibles aux utilisateurs ordinaires qu'en lecture. En termes de RW, nous pouvons définir $Agents$ un ensemble d'agents, $Files$ un ensemble de fichiers et P l'ensemble des variables propositionnelles. $\forall a \in Agents$ et $\forall f \in Files$, P contient :

- $admin(a)$: a est un administrateur
- $user(a)$: a est un utilisateur ordinaire
- $system(f)$: f est un fichier système
- $other(f)$: f n'est pas est un fichier système

Nous pouvons définir les relations r et w comme suit (le symbole « $\rightleftharpoons$ » remplace « est défini par ») :

$$r(system(f), a) \rightleftharpoons admin(a)$$

$$r(other(f), a) \rightleftharpoons true$$

$$w(system(f), a) \rightleftharpoons admin(a)$$

$$w(other(f), a) \rightleftharpoons admin(a)$$

Nous pouvons ajouter qu'un utilisateur peut changer son profil pour devenir administrateur, nous obtenons donc les définitions suivantes :

$$w(admin(x), y) \rightleftharpoons true$$

$$w(user(x), y) \rightleftharpoons true$$

Considérons le programme α qui permet d'affecter à la variable $admin(x)$ la valeur $true$. Nous pouvons dire que à partir de n'importe quel état du système s, le programme α possède les permissions de s'exécuter et de changer la valeur de $admin(x)$:

$$([\![\alpha]\!](admin(x)))(s) = true$$

Ceci est valable également pour l'ensemble des agents $Agents$:

$$[\![Agents, \alpha]\!](s) = true$$

Cette règle que nous venons d'ajouter, a pour conséquence de permettre à n'importe quel agent de A de modifier un fichier en modifiant d'abord son profil donc $W_{Agents\,admin(x)} = W_{Agents\,system(f)} = true$.

Opérateur d'accessibilité

Un système de contrôle d'accès peut changer d'état, donc l'applicabilité des règles de contrôle d'accès peut être modifiée. Ainsi, RW définit un langage pour exprimer les objectifs en termes d'états. L'opérateur d'accessibilité dont la forme est $A(\Phi, \psi)$ où $A \in \Sigma$ est un ensemble d'agents, $\Phi \subseteq \mathbf{L}(P)$ une liste de variables propositionnelles et $\psi \in \mathbf{L}(P)$ est une formule logique, indique que l'ensemble d'agents A possède la permission d'exécuter un programme qui peut lire les variables Φ tout en respectant la validité d'un objectif ψ. Cet objectif ψ a la syntaxe suivante :

$$\psi \ ::= \ \bot \,|\, \top |\, \mathbf{p} \,|\, \psi \,\wedge\, \psi \,|\, \psi \,\vee\, \psi$$

Où **p** désigne un objectif atomique de la forme :

- $\overline{\varphi}'$ où $\varphi \in \mathbf{L}(P)$: la valeur finale de φ est $true$
- $\overline{\varphi}$ où $\varphi \in \mathbf{L}(P)$: la valeur initiale de φ est $true$

$\top$ et $\bot$ désignent les buts triviaux, le but toujours vrai et le but impossible à atteindre.

Exemple : L'expression $A(\langle p \rangle, (\overline{q} \wedge \overline{q}') \vee (\neg\overline{q} \wedge \neg\overline{q}'))$ indique que les agents de A veulent lire la variable p sans changer la valeur initiale de q.

Les buts $R_{A\varphi}$ et $W_{A\varphi}$ peuvent être exprimés à l'aide des expressions suivantes :

$$R_{A\varphi} \Longleftrightarrow A(\{\varphi\}\,, \top) \tag{4.1}$$

$$W_{A\varphi} \Longleftrightarrow A(\varnothing, \overline{\varphi}') \wedge A(\varnothing, \overline{\neg\varphi}') \tag{4.2}$$

Dans l'expression (4.1) les agents de A peuvent exécuter un programme capable de lire les variables φ sans changer l'état du système.

Dans l'expression (4.2) les agents de A peuvent exécuter un programme capable de garder ou modifier la valeur initiale de φ.

Analyse et vérification dans RW

L'opérateur d'accessibilité $A(\Phi, \Psi)$ peut être analysé afin de savoir si l'objectif souhaité peut être atteint. Guelev a développé un programme en *Prolog* [18] qui permet de trouver un programme RW capable d'atteindre un objectif donné.

Avec RW nous pouvons ainsi vérifier si un ensemble d'agents peut atteindre une situation particulière du système en combinant des opérations d'écriture et de lecture. Ceci permettra de valider un système de contrôle d'accès par rapport aux spécifications initialement établies et de détecter les failles possibles.

4.1.2 Transformation vers XACML

À partir des spécifications en RW, *Zhang*, *Ryan* et *Guelev* [57] ont abouti à des politiques de contrôle d'accès en XACML (voir chapitre 2). Ainsi ces politiques sont indirectement analysées et vérifiées (voir figure 4.1). En effet, la vérification et la validation se font sur le modèle RW, la forme XACML n'est qu'une représentation différente. La possibilité de faire l'analyse à partir du format XACML n'est pas évoquée. De plus, XACML offre plus de fonctionnalités et il est un langage plus général que RW. Donc cette méthode permet de générer des politiques de contrôle d'accès en XACML valides.

La transformation des politiques RW vers XACML est réalisée en deux étapes. La traduction des règles en RW en un format lisible par les ordinateurs et la transformation de ce format en XACML.

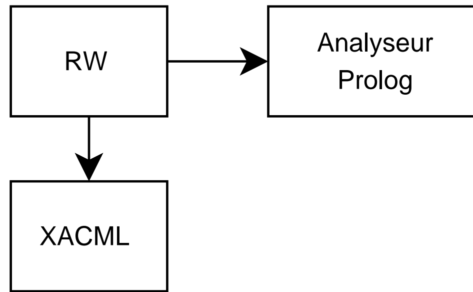

FIG. 4.1 – Analyse du contrôle d'accès avec *Prolog*

Une syntaxe de RW lisible par les ordinateurs

La forme intermédiaire proposée pour RW sert à décrire un système de contrôle d'accès. Cette syntaxe commence par le mot `AccessControlSystem` suivi par le nom attribué au système. Le corps de cette syntaxe est composé de deux parties : une partie pour les déclarations et une autre pour définir les politiques de contrôle d'accès.

La partie déclarative contient la définition des différentes variables propositionnelles et l'ensemble des agents du système. Les objets et les agents sont définis en tant que classes. Les variables propositionnelles sont sous forme de prédicats définissants les relations entre objets et agents.

La définition des politiques de contrôle d'accès contient pour chaque variable propositionnelle les conditions de lecture et d'écriture à l'aide des formules logiques.

L'exemple de la sous-section précédente est donné ci-dessous avec le format lisible par ordinateur.

```
AccessControlSystem File_Sysytem
  /** Partie déclarative **/
  /* Déclaration des objets */
  /* L'objet Agent est un objet prédéfini */
  class File;
  /* Définition des variables propositionnelles */
  Predicate  admin(a : Agent),
             user(a : Agent),
             system(f : File),
             other(f : File);

  /** Définition des politiques de contrôle d'accès **/
```

```
  /* Les rôles sont toujours accessibles en
     lecture-écriture */
  admin(a){
    read : true;
    write : true;
  }
  user(a){
    read : true;
    write : true;
  }
  /* Les fichiers systèmes ne sont accessibles
     que par les administrateurs                */
  system(f){
    read : admin(a);
    write : admin(a);
  }
  /* Les fichiers ordinaires sont toujours
     accessibles en lecture */
  /* Seul l'administrateur peut modifier les fichiers */
  other(f){
    read : true;
    write : admin(a);
  }
End
```

Transformation vers XACML

La transformation vers XACML se fait à l'aide d'un outil écrit en Java [56]. Le compilateur lit un fichier RW sous la forme décrite ci-dessus, ensuite il convertit les formules logiques définies par *read* et *write* (r et w) en des conditions et cibles XACML et des requêtes SQL. La requête SQL permet d'interroger une base de données contenant les informations spécifiques à chaque contexte. Une fonction externe *evaluate-sql* permet de remplacer les requêtes SQL par leurs résultats (Figure 4.2).

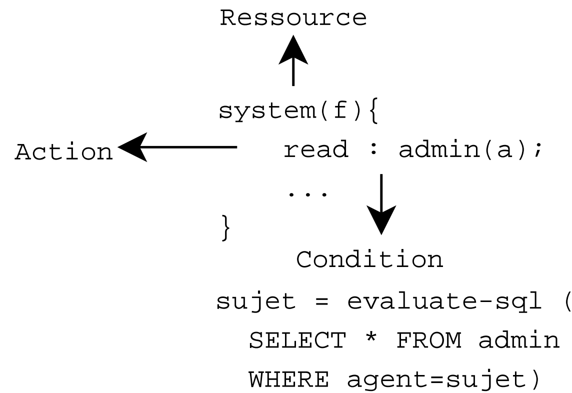

FIG. 4.2 – Génération de XACML à partir de RW

4.1.3 Discussion

Le langage RW est un langage qui a été développé dans la perspective d'analyser et de vérifier les conflits qu'il peut générer. Mais il reste un langage spécifique peu utilisé. La transformation RW-XACML permet d'aboutir à des spécifications avec un langage standard et en pleine expansion, mais il aurait été utile de pouvoir vérifier directement des politiques déjà existantes.

4.2 Travaux sur *Ponder*

4.2.1 Le langage de spécification de politiques *Ponder*

Comme XACML, *Ponder* [6] [9] est un langage de spécification de politiques de contrôle d'accès pour les systèmes distribués. Ce langage se base sur les notions suivantes :

- Le *Sujet* (*Subject*) fait référence aux utilisateurs, programmes ou toute autre entité qui peut accéder à une ressource ou un service.
- La *Cible* (*Target*) fait référence aux ressources et aux services partagés dans un réseaux de communication ou dans un système. Les cibles sont accessibles par les sujets selon les politiques de *Ponder*.
- Les sujets et les cibles sont regroupés en *Domaines* (*Domains*).

Le langage *Ponder* considère plusieurs types de politiques de contrôle d'accès. Nous allons introduire ces types dans les paragraphes qui suivent.

Politiques d'autorisation

Les politiques d'autorisation définissent l'ensemble des actions permises et défendues pour un ensemble de sujets et sur un ensemble de cibles. Elles peuvent également être validées par un ensemble de contraintes. Ces contraintes vont être présentées dans la sous-section suivante.

Politiques de filtrage de l'information

Les politiques de filtrage servent à sélectionner l'information accessible aux sujets. Ces politiques sont associées aux actions dans les politiques d'autorisations. Comme exemple, nous pouvons considérer la politique d'autorisation suivante (extraite de [7]) :

inst auth+ VideoConference {
subject /Agroup + /Bgroup ;
target /USAgroup -NYgroup ;
action VideoConf(BW,Priority)
{ **in** BW=2 ; **in** Priority=3 ;}

Une politique de filtrage de l'information est associée à la politique d'autorisation nommée *VideoConference*. Cette politique autorise aux membres des groupes */Agroup* et */Bgroup* d'initialiser une vidéoconférence avec les membres du groupe *USAgroup* excepté ceux de *NYgroup*. L'initialisation d'une vidéoconférence prend comme paramètres la bande passante (BW) et la priorité (*Priority*). Une politique de filtrage est associée à cette action, elle limite la bande passante à 2MB/s et établit sa priorité à 3.

Politiques de délégation

Les politiques de délégation autorisent un ensemble d'utilisateurs à transférer les autorisations d'accès, en entier ou en partie, qu'ils possèdent à un autre ensemble d'utilisateurs. Un sujet peut donc déléguer à un autre utilisateur les droits qu'il possède.

Dans l'exemple de la figure 4.3 le groupe *Agroup* délègue à *Cgroup* la politique d'autorisation *VideoConference*. La politique de délégation est appelée *DelegVC*.

Politique de retenue

Les politiques de retenue (*Restrain Policies*) sont des politiques qui empêchent un groupe de sujets d'exécuter un ensemble d'actions sur un ensemble de cibles. Les poli-

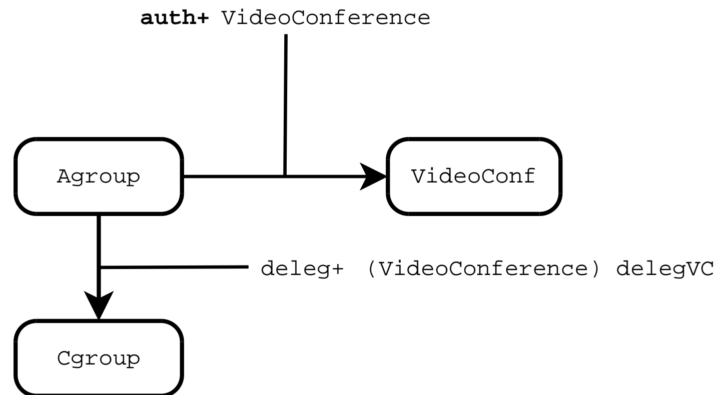

Fig. 4.3 – Politiques de délégation dans *Ponder*

tiques de retenue sont similaires aux politiques d'autorisation négatives, mais ces dernières sont gérées par les cibles alors que les politiques de retenues sont gérées par les sujets.

Politiques d'obligation

Les politiques d'obligation spécifient les actions à entreprendre à la suite d'un événement.

inst oblig	LoginFailure {
on	3*loginfail(userid)
subject	s = /NRegion/SecAdmin ;
target	t = /NRegion/users ;
do	t.disable() -> s.log(userid) ;

La politique de l'exemple ci dessus (adapté de [7]) est déclenchée à la suite de trois échecs d'authentification successifs par les sujets du domaine */NRegion/SecAdmin* accédant au domaine */NRegion/users*. La politique désactive le sujet et écrit son identificateur dans le journal.

4.2.2 Composition des politiques

Le langage *Ponder* offre la possibilité de structurer les politiques de contrôle d'accès en des groupes de politiques. Un groupe peut contenir des politiques ou des sous-groupes. Le regroupement des politiques peut avoir deux objectifs :

- structurer et organiser les politiques pour faciliter l'accès et la réutilisation et là on parle de *groupe* (*group*)
- structurer les politiques selon leur sémantique et les associer à un domaine de sujets et là on parle de *rôle*

4.2.3 Contraintes dans *Ponder*

Les contraintes des politiques de contrôle d'accès servent à spécifier les conditions de validité et d'applicabilité des politiques. Dans *Ponder*, il y a deux types de contraintes : les contraintes relatives à une seule politique et les contraintes relatives à un groupe de politiques : *métapolitiques*.

Contraintes relative à une seule politique

Les contraintes relatives à une seule politique sont spécifiées au niveau des politiques d'autorisation avec des prédicats. Ces contraintes peuvent être basées sur :
- les sujets et les cibles
- les paramètres des actions et des événements
- le temps (date et heure)

```
inst auth+ VideoConference {
subject   /Agroup + /Bgroup;
target    /USAgroup -NYgroup;
action    VideoConf(BW,Priority)
when      time.between("1600","1800");
```

L'exemple ci-dessus montre la même politique d'autorisation de la sous-section précédente mais qui possède une contrainte sur le temps. Ainsi, cette politique n'est appliquée qu'entre 16h00 et 18h00.

Métapolitiques

Les métapolitiques ont pour objectif de définir des contraintes sur un ensemble de politiques. Une méta-politique utilise un sous-ensemble de OCL (*Object Constraint Language* [17]). Une contrainte est exprimée avec des d'expressions OCL qui peuvent déclencher l'exécution d'actions.

Les métapolitiques jouent un rôle important dans la détection des conflits dans *Ponder*. En effet, il est possible de définir des conditions qui caractérisent les conflits à l'aide d'expressions OCL.

4.2.4 Conflits dans *Ponder*

Dans la mesure où plusieurs politiques peuvent s'appliquer à un contexte particulier, des conflits peuvent exister entre elles. Dans *Ponder* deux types de conflits peuvent être détectés et analysés [39] [40] : les conflits de modalité et les conflits sémantiques.

Conflits de modalité

Si des politiques positives et négatives s'appliquent pour les mêmes sujets, cibles et actions, alors il y aura un conflit de modalité. Ce type de conflits survient lorsque les domaines des politiques chevauchent (domaines des sujets, cibles et actions).

Pour résoudre ce genre de conflits, des règles de priorité et de préséance peuvent être établies afin d'aboutir à une seule décision. Cette technique est similaire aux algorithmes de combinaison dans XACML. Comme relations de préséance nous pouvons citer :

- Les politiques négatives ont toujours la priorité
- Des priorités explicites sont assignées aux politiques
- Considérer les politiques les plus spécifiques

Ce genre de conflits est détecté par l'outil *Ponder Toolkit* (voir sous-section suivante).

Conflits sémantiques

Les conflits sémantiques sont des situations non conformes aux spécifications d'un système. Ces conflits sont spécifiques aux applications et aux contextes associés. Ces conflits dépendent des facteurs externes tels que la disponibilité des ressources ou bien des politiques externes telles que les politiques globales dans une entreprise. Nous pouvons citer les exemples suivants :

- Séparation des tâches : une même personne ne peut pas autoriser et initier le même paiement.
- Conflits des ressources : une ressource n'est disponible qu'en quantité limitée.
- Un administrateur a toujours la priorité d'accès.

Ces situations peuvent générer des conflits sémantiques. En effet, les conflits sémantiques sont le résultat de la contradiction entre les politiques et les propriétés du sys-

tème. Pour traiter ce genre de conflits les métapolitiques sont utilisées pour caractériser ce genre de situation (OCL) et effectuer les actions nécessaires.

4.2.5 Outils pour la détection de conflits

Ponder Toolkit est une suite d'outils [8] permettant de gérer et d'analyser les politiques de contrôle d'accès dans un environnement distribué. Cette suite intègre un outil d'analyse des conflits de modalité (*Conflict Analyzer*). Cet outil est implémenté en Java et détecte le chevauchement des domaines et les présente sous un format graphique.

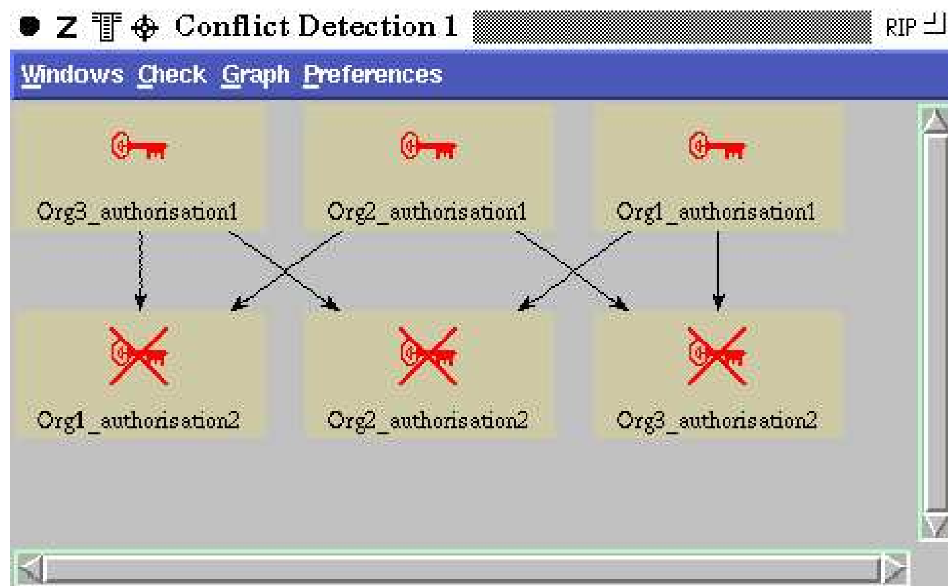

FIG. 4.4 – Détection de conflits dans *Ponder*

La figure 4.4 montre le résultat obtenu pour la détection de conflits de modalité pour six politiques d'autorisations, trois politiques positives (les clés) et trois politiques négatives (les clés barrées). La relation de chevauchement est représentée par un arc orienté de la politique la plus spécifique vers la politique la plus générale. Nous pouvons bien remarquer que les politiques positives chevauchent avec les politiques négatives, ce qui peut être une source de conflit. Par exemple la politique positive *Org1_ authorisation1* annule la politique négative *Org3_ authorisation2* car leur domaines chevauchent.

La figure 4.5 montre un exemple de conflit. Les sujets des domaines *O2_ m1* et *O2_ m2* peuvent effectuer les opérations *retract()*, *delete()* et *disable()* sur les cibles du domaine *SharedPolicies* puisque la politiques *Org2_ authorisation1* l'autorise. Mais, les politiques *Org1_ authorisation2* et *Org3_ authorisation2* ne l'autorisent pas.

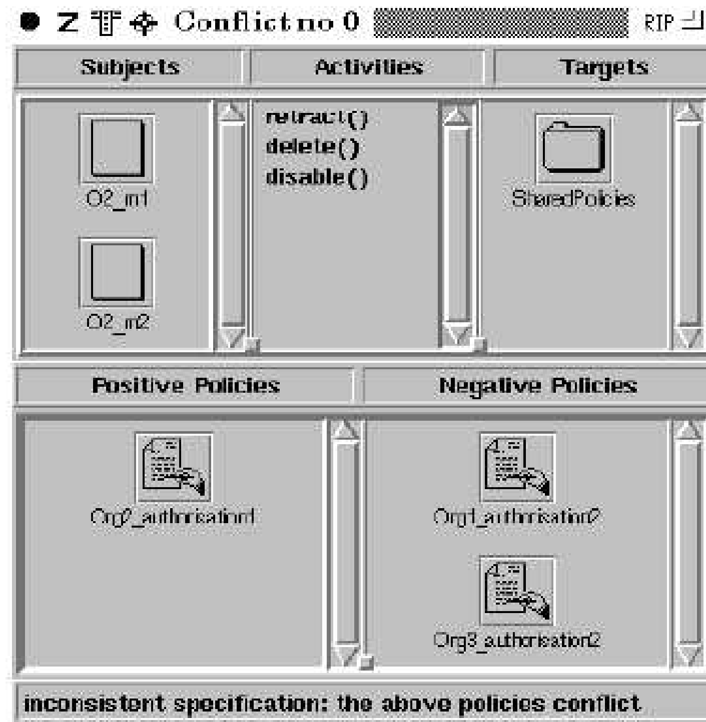

FIG. 4.5 – Exemple de conflits dans *Ponder*

L'outil permet également de transformer les politiques et les métapolitiques en des prédicats en *Prolog* afin de détecter les conflits sémantiques.

4.2.6 Discussion

Les travaux réalisés sur *Ponder* restent spécifiques à ce langage. La vérification et l'analyse se font avec deux techniques différentes pour les conflits de modalité et les conflits sémantiques. Ces deux types de conflits sont étroitement liés mais il n'y a pas un seul outil qui les traite. De plus, malgré que les métapolitiques représentent une technique efficace pour traiter les conflits, il existe encore un risque que les métapolitiques soient elles mêmes incohérentes.

4.3 Transformations sur les graphes

Dans [32] et [33], on évoque l'utilisation des graphes pour spécifier les politiques de contrôle d'accès. Dans [58] les transformations sur les graphes ont été utilisées afin d'analyser et vérifier les politiques de contrôle d'accès exprimées en *Ponder*. Ces travaux

se sont inspirés des travaux plus généraux portant sur la détection et l'analyse des conflits dans les politiques de contrôle d'accès [35] telles que celles respectant le modèle *RBAC* (Role Based Access Control) [34].

L'idée générale est de décrire la structure d'un système de contrôle d'accès (objets et relations) à l'aide des graphes. Ensuite, considérer les règles et les politiques de contrôle d'accès comme des morphismes. Nous donnons dans la suite un bref aperçu simplifié de ces méthodes basées sur les graphes et appliquées à *Ponder*.

4.3.1 Spécification des politiques de contrôle d'accès

Hiérarchie des domaines

La hiérarchie des domaines est représentée par un graphe où les noeuds sont les domaines et les arcs modélisent la relation d'inclusion (sous-domaine). L'exemple de la figure 4.6 montre que le domaine global *UQO* contient trois sous domaines *Administration*, *Professeurs* et *Étudiants*. Le sous-domaine *Chargés de cours* est inclus à la fois dans *Professeurs* et *Étudiants*.

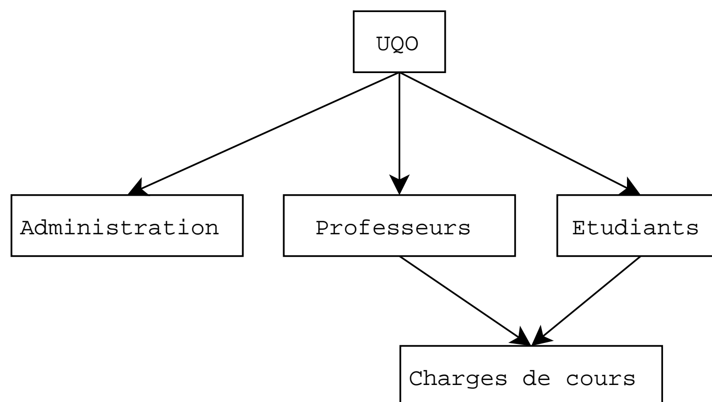

FIG. 4.6 – Hiérarchie des domaines avec les graphes

Représentation des politiques d'autorisation

Une politique d'autorisation est décrite par une règle de production avec un morphisme partiel montré par la figure 4.7. Dans cette figure l'effet de l'action sur le système est montré par la règle de production qui transforme le graphe de gauche vers celui de droite. L'action est représentée par un arc, entre le sujet et la cible, valué par 0 ou 1 (politique positive ou négative) et par le nom de l'action.

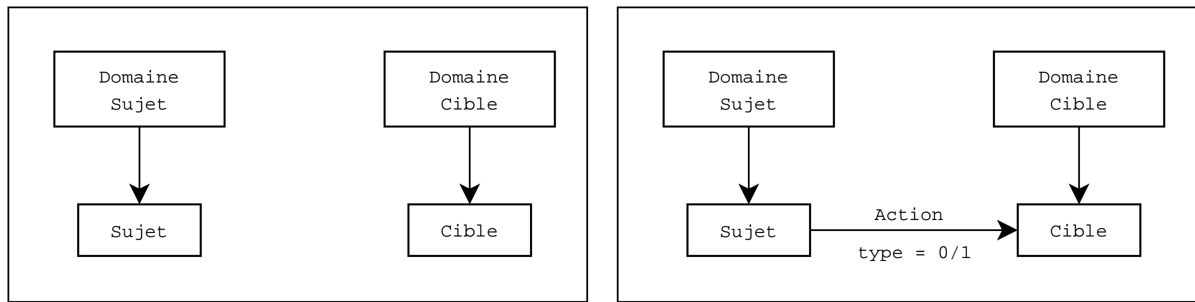

FIG. 4.7 – Politique d'autorisation avec les graphes

Vérification et analyse des politiques

L'analyse des graphes des politiques de contrôle d'accès peuvent être réalisée à l'aide d'outils tels que AGG [52] et GROOVE [47]. Certains résultats sont montrés dans [58].

4.3.2 Discussion

L'utilisation des graphes semble être une technique efficace pour traiter et analyser les politiques de contrôle d'accès. Toutefois c'est une technique qui pèche par sa complexité qui augmente avec la complexité des systèmes.

4.4 L'outil *Margrave*

Des travaux ont été entrepris à l'université Brown et l'institut polytechnique de Worcester pour vérifier et analyser les politiques de contrôle d'accès [14]. Le langage XACML a été pris comme langage d'application. Ces travaux se sont intéressés à la vérification des propriétés des systèmes et à l'analyse de l'impact des changements sur les politiques de contrôle d'accès. Les arbres binaires de décision à terminaux multiples : MTBDD (*Multi-Terminal Binary Decision Diagrams*) ont été utilisés et un outil de vérification (*Margrave*) [13] a été développé. Dans les sous-sections suivantes, nous donnons une brève description de cette technique et cet outil.

4.4.1 Représentation des politiques avec les MTBDD

Les politiques de contrôle d'accès en XACML sont représentées par des MTBDD. Les MTBDD sont un cas plus général des arbres binaires de décision BDD (*Binary Decision*

Diagrams). Dans les BDD les seuls terminaux possibles sont 0 ou 1 alors que dans les MTBDD plusieurs terminaux sont autorisés. Dans le cas des politiques de contrôle d'accès, les décisions possibles ont été considérées comme terminaux (***D**eny*, ***P**ermit* et ***N**otApplicable*). De plus, chaque valeur possible des attributs des sujets, ressources et actions est considérée comme une variable représentée par un nœud. Considérons l'exemple suivant extrait de [14] :

- Il y a deux profils possibles pour les sujets {**f**aculty, **s**tudent}.
- Il y a une seule ressource {**g**rades}.
- Les actions possibles sont {**a**ssign, **r**eceive}.
- Un sujet dont le profil est ***s**tudent* peut effectuer l'action ***r**eceive* sur la ressource ***g**rades*.
- Un sujet dont le profil est ***f**aculty* peut effectuer l'action ***a**ssign* sur la ressource ***g**rades*.

Cette politique est représentée par l'arbre de la figure 4.8. Dans cet arbre, chaque variable est représentée par un nœud. Les arcs sortants de chaque nœud représentent les valeurs possibles de cette variable. Le nœud f représente la variable *faculty* qui désigne un profil du sujet. L'arc sortant de f avec la valeur 1 mène à un sous-arbre qui correspond au cas où le profil du sujet est *faculty*, l'arc dont la valeur est 0 mène vers le cas où le profil du sujet ne correspond pas à *faculty*. Parcourir ainsi l'arbre permet de mener à une décision (terminal **N** pour *NotApplicable*, terminal **P** pour *Permit* et terminal **D** pour *Deny*).

Une politique contient plusieurs règles, c'est pourquoi plusieurs fonctions sur les arbres et les sous arbres ont été définies afin de combiner les règles et les synthétiser en un seul arbre représentant la politique de contrôle d'accès. Parmi ces fonctions nous pouvons citer :

- *Apply* : permet de combiner deux arbres en un seul
- *Combine* : permet de combiner les arbres de plusieurs règles en un seul en tenant compte de l'algorithme de combinaison (*permit-overrides*, *deny-overrides* ou *first-applicable*).

Les contraintes du domaine ainsi que les propriétés du système peuvent être représentées sous la forme d'arbres. Grâce aux fonctions citées ci-dessus il possible d'intégrer ces arbres avec les politiques de contrôle d'accès et ce en un seul arbre. Cet arbre peut ensuite être analysé et vérifié avec l'outil *Margrave*.

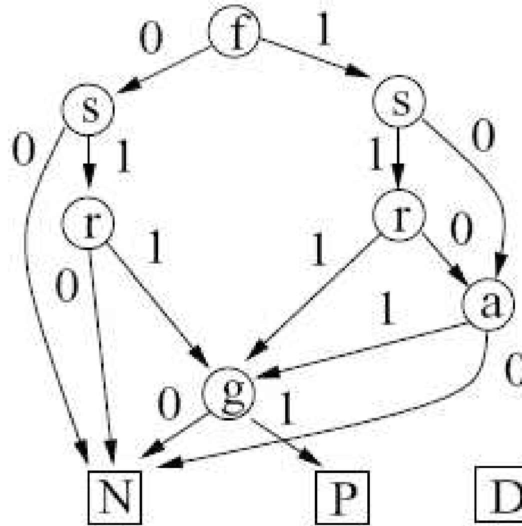

FIG. 4.8 – Politiques de contrôle d'accès avec les MTBDD

4.4.2 Vérification et analyse

L'outil *Margrave* permet d'interroger les politiques de contrôle d'accès à l'aide d'un ensemble de primitives et d'opérateurs listés dans [14] . L'outil permet également d'effectuer une analyse de l'impact des modifications apportées aux politiques en établissant un compte rendu sur les changements de décision générés. La sortie générée par *Margrave* possède le format textuel suivant :

```
1:/Resource, resource-class, ExternalGrades/
2:/Resource, resource-class, InternalGrades/
3:/Action, command, Assign/
4:/Action, command, View/
5:/Subject, role, Faculty/
6:/Action, command, Receive/
7:/Subject, role, Student/
8:/Subject, role, TA/
12345678
{
10100011
}
```

La première partie de la sortie (les lignes numérotées) montre toutes les variables avec leurs valeurs possibles. La deuxième partie montre la requête générant un conflit (1,3,7,8) et qui correspond à un sujet, dont le rôle est à la fois *Student* et *TA*, qui veut effectuer l'action *Assign* sur la ressource *ExternalGrades*.

4.4.3 Discussion

L'utilisation des arbres binaires peut être avantageuse dans la vérification et la détection de conflits grâce à la rapidité de leur parcours. Toutefois, cette technique présente l'inconvénient de ne pas pouvoir modéliser les structures, fortement employées dans XACML et n'importe quel autre langage contrôle d'accès. De plus, les arbres binaires ne sont pas expressifs et sont difficiles à interpréter : la sortie de *Margrave* n'est pas très significative et ne montre pas explicitement les contre-exemples.

4.5 Autres travaux

Plusieurs autres travaux ont porté sur la validation et la vérification des politiques de contrôles d'accès.

Kamoda et Al. [31] ont proposé une approche logique basée sur les *free variable Tableaux* [15] pour détecter les conflits au seins des politiques de contrôle d'accès. Cette approche consiste à traduire les autorisations d'accès en formules logiques et de les analyser par la suite.

Les travaux de *Jajodia et Al.* [30] ont abouti au langage *ASL* (*Authorisation Specification Language*) et à l'environnement *FAF* (*Flexible Authorisation Framework*). *ASL* est un langage qui offre des mécanismes pour résoudre les conflits basés sur la logique.

Bertino et Al. [5] ont réalisé des travaux dans la même optique en définissant un langage logique et un environnement de gestion du contrôle d'accès.

Benfarhat, El Baida et Cuppens [3, 4] ont utilisé la logique classique de premier ordre et le langage Z [51] pour représenter les politiques de contrôle d'accès, les vérifier et les analyser.

Barth, Mitchell et Rosenstein [2] ont étudié les conflits des politiques de confidentialité avec le langage EPAL. Une extension d'EPAL a été développée : DPAL. DEPAL traite surtout la combinaison de plusieurs politiques. Cette approche résout les conflits mais ne les détecte pas. La détection est réalisée par des moyens algorithmiques.

Des vérificateurs de modèles tels que SPIN [36] ont été également utilisés pour valider les politiques de contrôle d'accès notamment le modèle $RBAC$ [1]. L'outil ALLOY présenté dans le chapitre 3 a aussi été utilisé dans la modélisation et l'analyse des politiques de contrôle d'accès avec le modèle $RBAC$ dans [49].

Chapitre 5

Modèle logique de XACML

Afin d'analyser les interactions et de détecter les conflits, nous allons procéder à l'utilisation d'un langage formel pour exprimer les règles de contrôle d'accès. Nous allons utiliser à la fois la logique des prédicats et le langage ALLOY (lui même basé sur la logique des prédicats) pour exprimer les principes avancés par XACML. Ainsi, ALLOY permet de construire un modèle qui, de plus, peut être validé et analysé automatiquement.

Dans le cas particulier de XACML, il faut tenir compte de deux aspects : la structure et la sémantique du langage.

Des politiques de contrôle d'accès en XACML définissent un ensemble de structures introduites dans la section 2.2 et détaillées dans [16]. Nous définissons d'abord les structures générales du langage, c'est-à-dire, des gabarits pour les différents éléments. En fait, nous allons utiliser les mêmes techniques utilisées dans la modélisation orientée objet pour définir un ensemble de structures abstraites servant de modèles pour les politiques de contrôle d'accès en XACML à analyser. Comme montré dans le chapitre 3, le langage ALLOY est adéquat pour ce genre d'approches .

Les politiques de contrôle d'accès en XACML définissent également des contraintes et des fonctions logiques. Les règles de contrôle d'accès définissent des contraintes auxquelles doivent obéir les requêtes pour pouvoir obtenir l'autorisation d'accès aux ressources. Nous allons, pour ce faire, développer des fonctions logiques permettant de renseigner sur les réponses des règles, des politiques et des ensembles de politiques de contrôle d'accès.

5.1 Structures dans XACML

Le premier objectif est de construire le modèle du langage XACML. Nous allons considérer tous les éléments de la structure définie par le modèle objet de la figure 2.3. Nous allons donc, définir des ensembles d'objets et des relations entre eux. En ALLOY, chaque élément du langage est associé à une signature abstraite (sous-section 3.3.1) et différentes relations et fonctions peuvent être associées aux signatures.

5.1.1 Sujets, ressources et actions

Les sujets, ressources et actions sont des éléments du langage XACML qui ont des propriétés identiques. En effet, chacun de ces éléments est défini par ses attributs ainsi que leurs valeurs. Nous définissons donc un ensemble d'éléments *Element*. Nous définissons également trois sous-ensembles *Subject*, *Resource* et *Action*. La relation ternaire *attributes* est définie comme suit :

$$attributes : Element \rightarrow Attribute \rightarrow Value$$

Cette relation définit pour chaque élément (Sujet, ressource ou action) ses attributs et leurs valeurs.

Nous définissons également les ensembles suivants :

- *Value* : ensemble de valeurs pouvant être prises par les attributs.
- *Attribute* : ensemble d'attributs d'éléments

La fonction $values : Attribute \rightarrow Value$ définit pour chaque attribut l'ensemble des valeurs possibles qu'il peut avoir. En ALLOY, la définition des ensembles *Value* et *Attribute* sont :

```
abstract sig Value {}
abstract sig Attribute {values : set Value}
```

La définition des éléments est donnée par :

```
abstract sig Element {
  attributes :Attribute -> Value
}{attributes in values}
sig Subject, Resource, Action extends Element{}
```

5.1.2 Requête

Soit *Request* un ensemble de requêtes dans XACML. Chaque requête est définie par un ou plusieurs sujets, une seule ressource et une seule action. Le cas d'une requête initiée par plusieurs sujets est un cas particulier, c'est pourquoi nous allons le traiter à part.

Soient les ensembles suivants :

– $Subject$: un ensemble des sujets
– $Resource$: un ensemble des ressources
– $Action$: un ensemble d'actions

Soient les fonctions suivantes :

– $subject : Request \rightarrow Subject$
  une fonction qui retourne pour une requête donnée le sujet associé
– $resource : Request \rightarrow Resource$
  une fonction qui retourne pour une requête donnée la ressource associée
– $action : Request \rightarrow Action$
  une fonction qui retourne pour une requête donnée l'action associée

En ALLOY, nous définissons la signature *Request* pour l'ensemble des requêtes. Les relations *subject*, *resource* et *action* renseignent respectivement sur les sujet, ressource et action de chaque requête.

```
one sig Request {
  subject : one Subject,
  resource : one Resource,
  action : one Action}
```

Dans le cas d'une requête à sujets multiples, la fonction *subject* devient une relation. La définition de la signature *Request* en ALLOY devient :

```
one sig Request {
  subject : some Subject,
  resource : one Resource,
  action : one Action}
```

Comme déjà mentionné, ce cas est un cas particulier. Dans notre modèle, nous allons considérer seulement la cas d'une requête qui définit un seul sujet.

5.1.3 Cibles

Soit *Target* un ensemble de cibles dans XACML. Une cible définit un ensemble de sujets, de ressources et d'actions qui sont des éléments définis dans la sous-section 5.1.1. Cet ensemble est défini avec ALLOY comme suit :

```
abstract sig Target {
  subjects : set Subject,
  resources : set Resource,
  actions : set Action}
```

Où

- $subjects : Target \rightarrow Subject$ est une relation qui définit l'ensemble des sujets permis par chaque cible.
- $resources : Target \rightarrow Resource$ est une relation qui définit l'ensemble des ressources permises par chaque cible.
- $actions : Target \rightarrow Action$ est une relation qui définit l'ensemble des actions permises par chaque cible.

Les cibles sont associées aux règles, politiques et ensemble de politiques, c'est pourquoi nous avons défini une seule structure de cible qui sera évaluée de la même manière.

5.1.4 Effets

Dans XACML les effets possibles d'une règle de contrôle d'accès sont : *Permit*, *Deny*, *NotApplicable* et *Indeterminate*. Donc, soit l'ensemble :

$$Effect = \{Permit, Deny, NotApplicable, Indeterminate\}$$

En ALLOY, nous définissons la signature abstraite *Effect*, c'est-à-dire, ses seules instances sont celles définies comme sous-signatures. Les types d'effet possibles sont définis par des signatures étendant la signature *Effect* comme suit :

```
abstract sig Effect {}
one sig Permit, Deny, NotApplicable, Indeterminate extends Effect {}
```

5.1.5 Règles

Soit *Rule* un ensemble de règles dans XACML. Une règle est définie par un effet, une cible et une condition. Les conditions n'ont pas été traitées de façon automatique dans ce mémoire, toutefois, nous allons proposer dans la sous-section 7.5.2 des méthodes permettant de les considérer dans notre analyse.

Les cibles ont été définies dans la sous-section 5.1.3 et nous définissons la fonction $ruleTarget : Rule \rightarrow Target$ qui permet d'associer à chaque règle sa cible.

Une règle est également associée à un effet qui peut être *permit* ou *deny*. Donc la fonction $ruleEffect : Rule \rightarrow Effect$ permet de spécifier l'effet de chaque règle.

En ALLOY, nous définissons respectivement la cible et l'effet d'une règle par les attributs *ruleTarget* et *ruleEffect* :

```
abstract sig Rule {
  ruleTarget : one Target,
  ruleEffect : one Effect}
```

5.1.6 Politiques

Soit *Policy* un ensemble de politiques. Chaque élément de *Policy* est défini par une cible, un ensemble de règles et un algorithme de combinaison des règles (voir sous-section 5.2.5). Donc, si nous définissons *CombiningAlgo* comme un ensemble d'algorithmes de combinaison et si nous considérons l'ensemble *Rule* défini dans la sous-section précédente, nous pourrons définir :

- $policyTarget : Policy \rightarrow Target$
  une fonction qui définit pour chaque politique sa cible
- $rules : Policy \rightarrow Rule$
  une relation qui associe à chaque politique un ensemble de règles
- $combiningAlgo : Policy \rightarrow CombiningAlgo$
  une fonction qui définit pour chaque politique un algorithme de combinaison des règles

En ALLOY, nous définissons la signature *Policy* pour un ensemble de politiques, elle possède trois attributs pour la cible, les règles et l'algorithme de combinaison des règles.

```
abstract sig Policy {
  policyTarget : one Target,
```

```
  rules : set Rule,
  ruleCombiningAlgo : one CombiningAlgo
}
```

5.1.7 Ensembles de politiques

Soit *PolicySet* un ensemble d'ensembles de politiques. Chaque élément de *PolicySet* possède une cible, un algorithme de combinaison et une ou plusieurs politiques (ou un ou plusieurs ensembles de politiques). Nous définissons donc les fonctions et les relations suivantes :

- $policySetTarget : Policyset \rightarrow Target$
  une fonction qui définit pour chaque ensemble de politiques sa cible
- $policies : PolicySet \rightarrow Policy$
  une relation qui permet d'associer pour chaque élément de *PolicySet* les politiques qui y sont incluses
- $policySets : PolicySet \rightarrow PolicySet$
  une relation qui permet d'associer pour chaque élément de *PolicySet* les ensembles de politiques qui y sont inclus
- $combiningAlgo : PolicySet \rightarrow CombiningAlgo$
  une fonction qui permet de définir l'algorithme de combinaison des politiques pour chaque élément de *PolicySet*

Avec ALLOY, nous définissons une signature *PolicySet* comme suit :

```
abstract sig PolicySet {
  policySetTarget : one Target,
  policies : set Policy,
  policySets : set PolicySet,
  combiningAlgo : one CombiningAlgo}
```

Dans la sous-section suivante, nous allons expliquer l'approche adoptée pour simuler les algorithmes de combinaison.

5.1.8 Algorithmes de combinaison

Puisqu'une politique de contrôle d'accès peut inclure plusieurs règles, et puisqu'un ensemble de politiques peut inclure plusieurs politiques ou plusieurs ensembles de poli-

tiques, il faut faire appel aux algorithmes de combinaison des règles et des politiques pour sélectionner une seule réponse. Il existe plusieurs algorithmes prédéfinis dans XACML (sous-section 2.2.3), mais nous avons choisi d'implémenter et d'utiliser que les deux les plus utilisés qui sont :

- *PermitOverrides* qui permet de retourner un *permit* dans le cas ou une politique ou règle appliquée retourne *permit*.
- *DenyOverrides* qui permet de retourner un *deny* dans le cas ou une politique ou règle appliquée retourne *deny*.

Soit *CombiningAlgo* un ensemble de règles de combinaison. Si nous considérons les deux algorithmes cités ci-dessus, cet ensemble sera constitué comme suit :

$CombiningAlgo = \{PermitOverrides, DenyOverrides\}$

Le comportement de ces algorithmes est analogue pour les règles et les politiques, c'est pourquoi l'ensemble *CombiningAlgo* sera utilisé pour les algorithmes de combinaison des règles et des politiques. Nous avons ainsi créé une seule signature abstraite en ALLOY définie par :

```
abstract sig CombiningAlgo {}
```

Nous avons défini les types d'algorithmes par les sous-types suivants en ALLOY :

```
one sig PermitOverrides, DenyOverrides extends CombiningAlgo {}
```

5.1.9 Méta-modèle de XACML

Nous obtenons, d'après les spécifications en ALLOY décrites en annexe C, un modèle sous forme graphique appelé *méta-modèle*. Il est généré automatiquement par l'outil ALLOY et montré par les figures 5.1, 5.2, 5.3, 5.4 et 5.5.

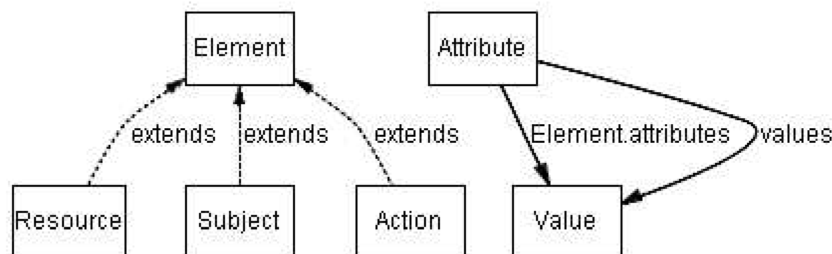

FIG. 5.1 – Sujets, ressources et actions

La figure 5.1 montre les structures des sujets, ressources et actions qui sont des sous-structures de la structure globale *Element*. De plus, nous pouvons distinguer la relation *values* qui lie un attribut à ses valeurs possible. La relation *attributes* définit pour chaque élément ses attributs et leurs valeurs.

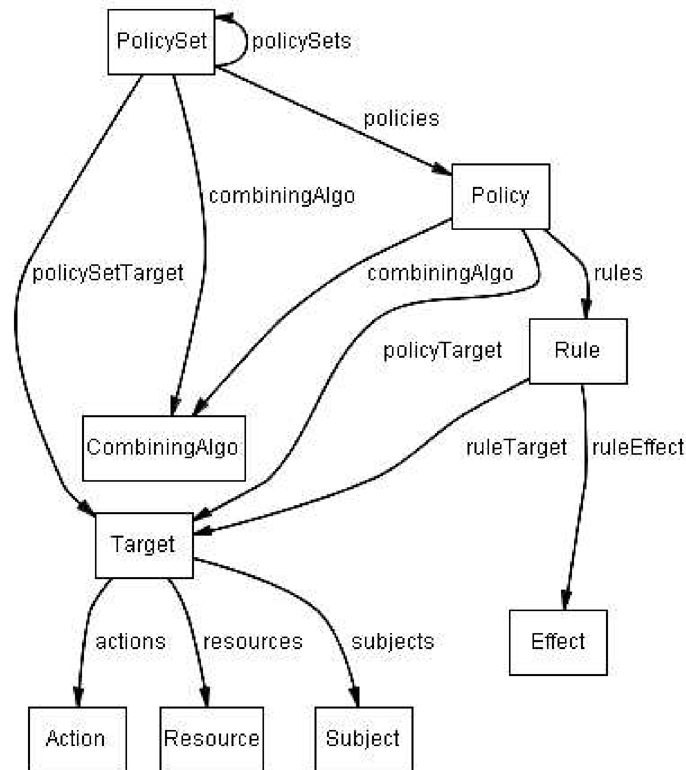

FIG. 5.2 – Ensembles de politiques, politiques et règles de contrôle d'accès

La figure 5.2 montre les structures des ensembles de politiques, des politiques, des règles et des cibles. Cette figure met également en valeur les différentes relations entre ces entités. La figure 5.3 montre la structure d'une requête.

Les figures 5.4 et 5.5 mettent en évidence la notion de sous-structure (ou sous-signature en terme de ALLOY) qui définit les différents algorithmes de combinaisons et les différents effets d'une règle.

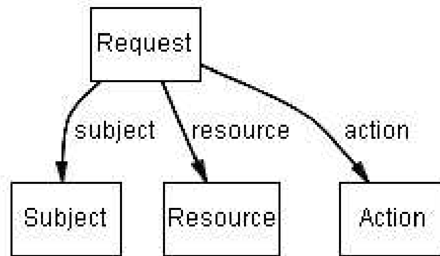

FIG. 5.3 – Requêtes

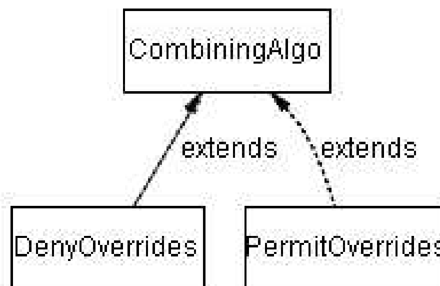

FIG. 5.4 – Algorithmes de combinaison

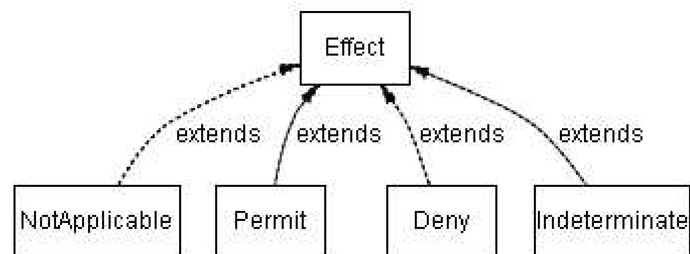

FIG. 5.5 – Effets des règles de contrôle d'accès

5.2 Contraintes logiques

5.2.1 Évaluation des cibles par rapport aux requêtes

Pour déterminer si un ensemble de politiques, une politique ou une règle sont appliqués pour une requête donnée, il faut vérifier si cette dernière correspond aux cibles. C'est pourquoi nous définissons un prédicat appelé *targetMatch* qui permet de déterminer si le sujet (respectivement la ressource et l'action) défini par la requête, correspond à un des sujets (respectivement ressources et actions) de la cible. Mais, si la cible ne défini aucun sujet (respectivement ressource et action), alors tous les sujets (respectivement ressources et actions) dans la requête seront permis. En logique de premier ordre nous écrivons : $\forall t \in Target$ et $q \in Request$

$$
\begin{array}{ll}
\text{si} & (\neg\exists s \in Subject/subjects(t,s) \vee \\
& \exists s \in Subject/subjects(t,s) \wedge elementMatch(subject(q),s)) \\
\wedge & (\neg\exists r \in Resource/resources(t,r) \vee \\
& \exists r \in Resource/resources(t,r) \wedge elementMatch(resource(q),r)) \\
\wedge & (\neg\exists a \in Action/actions(t,a) \vee \\
& \exists a \in Action/actions(t,a) \wedge elementMatch(action(q),a)) \\
\text{alors} & targetMatch(t,q) = vrai \\
\text{sinon} & targetMatch(t,q) = faux
\end{array}
$$

Avec ALLOY nous définissons ce prédicat de la manière suivante :

```
pred targetMatch (t : Target, r : Request) {
  {t.subjects=none || some e:t.subjects | elementMatch(r.subject,e)}
  {t.resources=none || some e:t.resources | elementMatch(r.resource,e)}
  {t.actions=none || some e:t.actions | elementMatch(r.action,e)}
}
```

Le prédicat *elementMatch* sert, à son tour, à vérifier si un élément (sujet, ressource ou action) d'une requête correspond à celui de la cible. Il faut donc vérifier que tous les attributs définis par la cible soient présents dans la requête, c'est-à-dire, toutes les propriétés exigées par la cible soient satisfaites par la requête. Donc, nous vérifions d'abord que tous les attributs spécifiés par la cible le sont par la requête et que leurs valeurs sont égales.

Ceci s'écrit en ALLOY de la façon suivante :

```
pred elementMatch(e1: Element, e2 : Element){
  e2 = none ||
  all a2 : e2.attributes.Value {
    some a1 : e1.attributes.Value
       | a1=a2 and a2.(e2.attributes) in a1.(e1.attributes)
  }
}
```

5.2.2 Réponse des règles

La réponse des règles de contrôle d'accès dépend en premier lieu de la cible. En deuxième lieu, elle dépend de la condition. Nous avons mentionné plutôt que nous allons nous contenter de la cible. Nous définissons donc la fonction *ruleResponse* qui permet de définir la réponse d'une règle dans un contexte de requête particulier. Si la cible est vérifiée, alors la réponse sera donnée par la fonction *ruleEffect*. Sinon, la réponse sera *NotApplicable*.

$$ruleResponse : Rule \times Request \rightarrow Effect$$
$$\forall r \in Rule,\ q \in Request$$
$$ruleResponse(r,q) = \begin{cases} ruleEffect(r) & \text{si } targetMatch(ruleTarget(r), q) \\ NotApplicable & \text{sinon} \end{cases}$$

Nous définissons la même fonction en ALLOY comme suit :

```
fun ruleResponse (r : Rule, q : Request) : Effect {
  if targetMatch(r.ruleTarget, q) then r.ruleEffect
  else NotApplicable
}
```

5.2.3 Réponse des politiques

La réponse de la politique dépend de trois paramètres : la cible, les règles appliquées et l'algorithme de combinaison utilisé. Si la cible est vérifiée, alors les règles de la politique seront évaluées et leurs réponses seront combinées grâce à l'algorithme de combinaison de la politique. Nous définissons la fonction *policyResponse* qui permet de générer la réponse d'une politique de contrôle d'accès. Cette fonction est définie comme suit :

$policyResponse : Policy \times Request \rightarrow Effect$
$\forall p \in Policy,\ q \in Request :$

$$policyResponse(p,q) = \begin{cases} ruleCombinedResponse(p,q) \\ \quad \text{si } targetMatch(policyTarget(p),q) \\ NotApplicable \\ \quad \text{sinon} \end{cases}$$

De même avec ALLOY, nous définissons la fonction *policyResponse* comme suit :

```
fun policyResponse(p:Policy,q:Request):Effect{
  if targetMatch(p.policyTarget, q) then ruleCombinedResponse(p,q)
  else NotApplicable}
```

La fonction *ruleCombinedResponse* permet de retourner pour chaque politique de contrôle d'accès une réponse unique déduite de celles des règles qui y sont incluses et ce, selon l'algorithme de combinaison des règles associé à la politique. Nous avons choisi de traiter deux algorithmes : *PermitOverrides* et *DenyOverrides*. Dans ce cas, la fonction *ruleCombinedResponse* sera définie comme suit :

$ruleCombinedResponse : Policy \times Request \rightarrow Effect$
$\forall p \in Policy,\ q \in Request :$

$$ruleCombinedResponse(p,q) = \begin{cases} rulePermitOverrides(p,q) \\ \ \text{si } combiningAlgo(p) = PermitOverrides \\ ruleDenyOverrides(p,q) \\ \ \text{si } combiningAlgo(p) = DenyOverrides \\ Indeterminate \text{ sinon} \end{cases}$$

Nous écrivons en ALLOY :

```
fun ruleCombinedResponse(p:Policy,req:Request):Effect {
  if p.combiningAlgo = PermitOverrides then rulePermitOverrides(p,req)
  else if p.combiningAlgo = DenyOverrides then ruleDenyOverrides(p,req)
  else Indeterminate}
```

Nous avons évoqué ci-dessus deux fonctions qui reproduisent le comportement des algorithmes de combinaison des règles *rulePermitOverrides* et *ruleDenyOverrides* sur lesquelles nous allons revenir dans la sous-section 5.2.5.

5.2.4 Réponse des ensembles de politiques

De la même manière que les politiques, les ensembles de politiques fournissent une réponse unique pour chaque contexte de requête. Cette réponse est donnée par la fonction *PolicySetResponse* définie comme suit :

$$policySetResponse : PolicySet \times Request \rightarrow Effect$$
$$\forall s \in PolicySet,\ q \in Request :$$
$$policySetResponse(s,q) = \begin{cases} policyCombinedResponse(s,q) \\ \quad \text{si } targetMatch(policySetTarget(s),q) \\ NotApplicable \\ \quad \text{sinon} \end{cases}$$

En ALLOY, nous définissons la même fonction :

```
fun policySetResponse(p:PolicySet,q:Request):Effect{
  if targetMatch(p.policySetTarget,req)
    then policyCombinedResponse(p,req)
  else NotApplicable}
```

La fonction *policyCombinedResponse* est définie d'une manière analogue à *ruleCombinedResponse*. Ainsi nous obtenons :

$$PolicyCombinedResponse : PolicySet \times Request \rightarrow Effect$$
$$\forall s \in PolicySet,\ q \in Request :$$
$$PolicyCombinedResponse(s,q) = \begin{cases} PolicyPermitOverrides(s,q) \\ \quad \text{si } combiningAlgo(s) = PermitOverrides \\ PolicyDenyOverrides(s,q) \\ \quad \text{si } combiningAlgo(s) = DenyOverrides \\ Indeterminate \text{ sinon} \end{cases}$$

En ALLOY, la fonction *policyCombinedResponse* est définie comme suit :

```
fun policyCombinedResponse(p:PolicySet,q:Request):Effect{
  if p.combiningAlgo = PermitOverrides then policyPermitOverrides(p,q)
  else if p.combiningAlgo = DenyOverrides then policyDenyOverrides(p,q)
  else Indeterminate}
```

5.2.5 Algorithmes de combinaison

Pour reproduire le comportement des algorithmes de combinaison dans le modèle logique de XACML, des fonctions ont été définies. Nous avons pris comme exemple le cas de l'algorithme de combinaison *PermitOverrides* pour les règles et pour les politiques. La définition complète du modèle XACML en ALLOY est donnée en annexe C.

La fonction *rulePermitOverrides* permet de retourner une seule réponse pour une politique qui contient plusieurs règles. Elle vérifie d'abord si la politique contient des règles qui retournent *permit*. Si c'est le cas, la réponse de la politique sera *permit*. Sinon, si la politique contient des règles qui retournent *deny*, alors la réponse sera *deny*. Sinon, la réponse sera *NotApplicable* ce qui signifie que la politique ne contient aucune règle qui s'applique au contexte courant.

$rulePermitOverrides : Policy \times Request \rightarrow Effect$

$\forall p \in Policy,\ q \in Request :$

$$rulePermitOverrides(p,q) = \begin{cases} Permit & \text{si } existsPermit(p,q) \\ Deny & \text{si } existsDeny(p,q) \\ & \wedge \neg existsPermit(p,q) \\ NotApplicable & \text{sinon} \end{cases}$$

Où, *existsPermit* et *existsDeny* sont des prédicats définis comme suit :

$\forall p \in Policy,\ q \in Request :$

$$existsPermit(p,q) = \begin{cases} vrai & \text{si } \exists r \in Rule \\ & (rules(p,r) \text{ et } ruleResponse(r) = Permit) \\ faux & \text{sinon} \end{cases}$$

$$existsDeny(p,q) = \begin{cases} vrai & \text{si } \exists r \in Rule \\ & (rules(p,r) \text{ et } ruleResponse(r) = Deny) \\ faux & \text{sinon} \end{cases}$$

En ALLOY, nous définissons les mêmes fonctions et prédicats par :

```
pred existsDeny(p : Policy, q : Request) {
  some r : p.rules | ruleResponse(r, q) = Deny}

pred existsPermit(p : Policy, q : Request) {
  some r : p.rules | ruleResponse(r, q) = Permit}
```

```
fun rulePermitOverrides ( p : Policy, q : Request) : Effect {
  if existsPermit(p,q) then Permit
  else if existsDeny(p,q) then Deny
  else NotApplicable}
```

En considérant l'algorithme de combinaisons des politiques *PermitOverrides*, la fonction *policyPermitOverrides* permet à un ensemble de politiques de retourner une seule réponse. Cette réponse est déduite des réponses des politiques et des ensembles de politiques subordonnés. La réponse d'un ensemble de politiques sera *Permit*, s'il existe une politique ou un ensemble de politiques parmi ceux qui y sont contenus et dont la réponse est *Permit*. Sinon, la réponse sera *Deny*, si une politique ou un ensemble de politiques retourne *Deny*. Sinon, la réponse sera *NotApplicable*.

En ALLOY, la fonction *policyPermitOverrides* est définie comme suit :

```
fun policyPermitOverrides(p:PolicySet,q:Request):Effect{
  if{some pol:p.policies | policyResponse(pol,q)=Permit}
    or
    {some ps:p.^policySets.policies |
          policyResponse(ps,q)=Permit}
  then Permit
  else if {some pol:p.policies | policyResponse(pol,q)=Deny}
          or
          {some ps:p.^policySets.policies |
                policyResponse(ps,q) = Deny}
       then Deny
  else NotApplicable}
```

Dans cette fonction, nous remarquons l'utilisation du « ˆ », le symbole de la fermeture transitive dans la portion suivante :

```
some ps : p.^policySets.policies | policyResponse(ps,q) = Permit
```

Ceci signifie que la relation *policySets* est appliquée récursivement une ou plusieurs fois à l'ensemble de politiques *p*. Donc `p.^policySets.policies` est utilisé à la place de

```
p.policySets.policies,
p.policySets.policySets.policies,
```

`...,`
`p.policySets...policySets.policies`

Ou :

$$\hat{}policySets(s,s') = \underbrace{policySets \circ policySets \circ \ldots \circ policySets(s,s')}_{n\ fois,\ n \in \mathbb{N}}$$

Le but est de vérifier toutes les politiques appartenant à un ensemble de politiques directement et indirectement. Par exemple, il est possible d'avoir un ensemble de politiques S0 qui contient un autre S1 qui contient lui aussi un autre ensemble de politiques S2 et S2 contient la politique P0. La politique P0 peut être accessible par l'expression :

`S0.policySets.PolicySets.policies`

En effet :

- S1 est contenu dans `S0.policySets`
- S2 est contenu dans `S1.policySets`
- P est contenue dans `S2.policies`.

Chapitre 6

Vérifications et analyses

Dans le chapitre précédent, nous avons construit un modèle de XACML basé entièrement sur la logique relationnelle de premier ordre d'ALLOY. L'outil *ALLOY Analyzer* offre la possibilité de vérifier et d'analyser le modèle ainsi obtenu. Nous allons, encore une fois, définir, en termes de logique de premier ordre, un ensemble de propriétés et d'interactions que nous proposons d'analyser et de vérifier. Nous allons nous intéresser, plus précisément, aux propriétés qui représentent des conflits ou incohérences potentiels. Nous allons, d'abord, définir les différentes relations qui caractérisent les cibles dans XACML. Ensuite, nous allons étudier l'impact de ces relations sur les interactions entre règles, politiques et ensembles de politiques. Enfin, nous allons proposer certaines propriétés supplémentaires que notre approche permet d'analyser et de vérifier et qui sont liées aux contextes de requêtes particuliers.

6.1 Relations entre cibles

Une cible définit un ensemble de sujets, ressources et actions qui spécifient les attributs et leurs valeurs que doit satisfaire une requête pour appliquer une règle, une politique ou un ensemble de politiques. Nous pouvons considérer les cibles comme des domaines d'application. Il est donc possible de définir un ensemble de relations qui caractérisent les cibles. Ces relations sont :

- inclusion
- intersection non vide
- disjonction
- égalité

6.1.1 Inclusion

Une cible T1 est incluse dans une autre T2 si tous les sujets (respectivement toutes les ressources et actions) de T1 sont définis dans T2. Un élément S1 est défini dans une cible T si cette dernière contient un élément S2 qui correspond à S1. La fonction de correspondance *elementMatch* a déjà été définie dans la sous-section 5.2.1.

$$T1 \subseteq T2 \Leftrightarrow \begin{pmatrix} \forall s1,\ subjects(T1, s1),\ \exists s2, subjects(T2, s2) \wedge elementMatch(s1, s2) \\ \wedge \\ \forall r1,\ resources(T1, r1),\ \exists r2, resources(T2, r2) \wedge elementMatch(r1, r2) \\ \wedge \\ \forall a1,\ actions(T1, a1),\ \exists a2, actions(T2, a2) \wedge elementMatch(a1, a2) \end{pmatrix}$$

Dans ALLOY, nous définissons le prédicat *targetIsIncluded* comme suit :

```
pred targetIsIncluded(t1, t2:Target){
  all s1 : t1.subjects {some s2 : t2.subjects | elementMatch(s1,s2)}
  all r1 : t1.resources {some r2 : t2.resources | elementMatch(r1,r2)}
  all a1 : t1.actions {some a2 : t2.actions | elementMatch(a1,a2)}
}
```

Soient T1 et T2 deux cibles définies comme suit :

- T1 cible les sujets employés
- T2 cible les sujets employés en vacances

La cible T2 est incluse dans la cible T1 puisque les employés qui sont en vacances sont couverts par la cible T1.

6.1.2 Intersection non vide

L'intersection de deux cibles T1 et T2 est non vide si certains sujets (respectivement ressources et actions) de T1 et T2 définis par les couples (attribut, valeur) chevauchent. Deux éléments E1 et E2 se chevauchent si les couples (attribut, valeur) de E1 correspondent à ceux définis par E2 en termes de la fonction *elementMatch*.

$$T1 \cap T2 \neq \oslash \Leftrightarrow \left(\begin{array}{l} \exists s1,\, s2,\, subjects(T1, s1) \wedge subjects(T2, s2) \wedge inter(s1, s2) \\ \wedge \\ \exists r1,\, r2,\, resources(T1, r1) \wedge resources(T2, r2) \wedge inter(r1, r2) \\ \wedge \\ \exists a1,\, a2,\, actions(T1, a1) actions(T2, a2) \wedge inter(a1, a2) \end{array} \right)$$

Où le prédicat *inter* est défini comme suit :
$inter(e1, e2) = elementMatch(e1, e2) \vee elementMatch(e2, e1)$

Avec ALLOY nous définissons les prédicats *inter* et *targetIntersect* comme suit :

```
pred inter(e1, e2 : Element){
  elementMatch(e1, e2) or elementMatch(e2, e1)
}
pred targetIntersect(t1, t2 : Target){
  some s1 : t1.subjects, s2 : t2.subjects | inter(s1,s2)
  some r1 : t1.resources, r2 : t2.resources | inter(r1,r2)
  some a1 : t1.actions, a2 : t2.actions | inter(a1,a2)
}
```

Soient les deux cibles T1 et T2 suivantes :

- T1 cible les sujets employés
- T2 cible les sujets en vacances

L'intersection de T1 et T2 est non vide et elle contient l'ensemble des sujets employés en vacances.

6.1.3 Disjonction

Deux cibles sont disjointes si leur intersection est vide. Le prédicat *targetDisj* vérifie si deux cibles sont disjointes ou non. Il est défini en Alloy comme suit :

```
pred targetDisj(t1, t2 : Target){
  no s1 : t1.subjects, s2 : t2.subjects | inter(s1,s2)
  no r1 : t1.resources, r2 : t2.resources | inter(r1,r2)
  no a1 : t1.actions, a2 : t2.actions | inter(a1,a2)
}
```

6.1.4 Égalité

Deux cibles sont égales si tous les sujets (respectivement ressources et actions) des deux cibles sont égaux. Deux éléments E1 et E2 sont égaux signifie que tous les couples (attribut, valeur) de E1 figurent dans E2 et vice versa.

```
pred eq(e1, e2 : Element){
  elementMatch(e1, e2) and elementMatch(e2, e1)
}

pred tagetEqual(t1, t2 : Target){
  all s1 : t1.subjects, s2 : t2.subjects | eq(s1,s2)
  all r1 : t1.resources, r2 : t2.resources | eq(r1,r2)
  all a1 : t1.actions, a2 : t2.actions | eq(a1,a2)
}
```

6.2 Interactions entre cibles

Dans cette section nous allons étudier deux types d'interactions. D'abord, nous allons spécifier les conséquences des chevauchements entre cibles des règles de contrôles d'accès (respectivement politiques et ensembles de politiques). Ensuite, nous allons étudier les situations d'incohérence au sein des politiques et des ensembles de politiques.

6.2.1 Chevauchement des cibles

Deux cibles T1 et T2 chevauchent dans les cas suivants :

- $T1 \cap T2 \neq \oslash$
- $T1 \subseteq T2$
- $T2 \subseteq T1$

Ces chevauchements peuvent concerner :

- deux règles de la même politique de contrôles d'accès
- deux politiques du même ensemble de politiques
- deux ensembles de politiques

Le chevauchement de cibles induit le chevauchement des domaines d'application pour les règles, les politiques et les ensembles de politiques. Ceci a pour conséquence de

générer des conflits potentiels dans le cas où les réponses des règles qui chevauchent sont contradictoires. Par contre, si les réponses sont identiques, alors il n'y aura pas de conflits mais il y aura des redondances potentielles.

Conflits entre règles

Prenons l'exemple des deux règles suivantes :

1. Accorder l'accès aux locaux de l'entreprise (effet = *Permit*) aux employés.
2. Refuser l'accès aux locaux de l'entreprise (effet = *Deny*) aux sujets qui sont en vacances.

L'intersection des deux cibles est l'ensemble des employés qui sont en vacances, c'est un ensemble non vide qui peut présenter un conflit potentiel. Certes, une politique de contrôle d'accès définit un algorithme de combinaison de règles qui permet de résoudre ce conflit et de générer une réponse unique. Cependant, cet algorithme est appliqué à toutes les règles présentes dans la politique. Il y a la possibilité que la réponse retournée ne soit pas celle souhaitée. De plus, les politiques qui ont comme algorithme de combinaison *Only-one-applicable*, retournent dans ce cas la réponse *Indeterminate*. L'administrateur doit être au courant de ces situations particulières.

Pour détecter ce genre de situations, potentiellement conflictuelles, nous définissons, en ALLOY, le prédicat *conflictingRules* . Ce prédicat vérifie si une politique de contrôle d'accès contient des règles dont les cibles chevauchent et dont les effets sont contradictoires.

```
pred conflictingRules(p : Policy){
  some disj r1, r2 : p.rules {
    (targetIsIncluded(r1.ruleTarget, r2.ruleTarget) ||
    targetIntersect(r1.ruleTarget, r2.ruleTarget) ||
    targetEqual(r1.ruleTarget, r2.ruleTarget))
    r1.ruleEffect != r2.ruleEffect
  }
}
```

Redondance entre règles

Considérons les deux règles de contrôle d'accès suivantes :

1. Tous les sujets sont autorisés à visiter les locaux de l'entreprise (effet = *Permit*).
2. Les visiteurs sont autorisés à visiter les locaux de l'entreprise (effet = *Permit*).

Dans ce cas, nous pouvons remarquer que la cible de la règle numéro 2 est incluse dans celle de la règle numéro 1. Les deux règles ont le même effet. La deuxième règle couvre la première et cette dernière peut même causer un conflit dans le cas ou la politique de contrôle d'accès qui contient ces deux règles a comme algorithme de combinaison *Only-one-applicable* qui exige l'application d'une seule règle.

Donc, des règles, avec le même effet et dont les cibles chevauchent, présentent des possibilités de redondance. Les administrateurs des politiques doivent en être informés.

Pour détecter de telle situations, nous définissons avec ALLOY le prédicat *redundantRules* qui vérifie si une politique risque de contenir des règles qui chevauchent et dont les effets sont identiques.

```
pred redundantRules(p : Policy){
  some disj r1, r2 : p.rules {
    (targetIsIncluded(r1.ruleTarget, r2.ruleTarget) ||
    targetIntersect(r1.ruleTarget, r2.ruleTarget) ||
    targetEqual(r1.ruleTarget, r2.ruleTarget))
    r1.ruleEffect = r2.ruleEffect
  }
}
```

Chevauchements entre politiques et ensembles de politiques

Les cibles des politiques et des ensembles de politiques peuvent également chevaucher. Mais, distinguer entre les situations de conflits et de redondances est moins évident. En effet, la réponse d'une politique de contrôle d'accès dépend du contexte de requête, des règles contenues dans la politique et de l'algorithme de combinaison des règles. Donc c'est les contextes de requête qui peuvent causer des conflits ou redondances. Nous allons détailler ces situations dans la section 6.3.

6.2.2 Incohérence des cibles

Dans la sous-section précédente, nous avons étudié les chevauchements entre cibles. Nous allons étudier, maintenant, le cas où les cibles ne chevauchent pas.

Si deux règles (respectivement politiques et ensembles de politiques) de même niveau, c'est-à-dire, appartenant à la même politique (respectivement ensemble de politique), ne chevauchent pas (cibles disjointes), alors les deux règles décriront deux contextes différents. Cette situation est une situation normale puisqu'il n'y a ni risque de conflits ni risque de redondance.

Par contre, si les cibles d'une politique et d'une règle qu'elle contient sont disjointes, alors il y aura une règle qui ne sera jamais utilisée. L'exemple suivant montre ce cas de figure :

Soit une politique T appliquée à l'ensemble des sujets qui sont employés dans l'entreprise (employés, directeurs et gardiens). T contient trois règles R1, R2 et R3 :

- R1 est appliquée aux directeurs et employés
- R2 est appliquée aux gardiens
- R3 est appliquée aux visiteurs non employés dans l'entreprise

Dans ce cas, R3 ne sera jamais utilisée car elle concerne seulement les visiteurs non employés dans l'entreprise. Dans ce cas les cibles de T et de R3 sont disjointes et il y a une incohérence entre une politique et une règle de contrôle d'accès.

Pour détecter ces situations, nous définissons les trois prédicats :

- *uselessRule* pour vérifier si une politique contient une règle ne pouvant pas être utilisée car sa cible et celle de la politique sont disjointes
- *uselessPolicy* pour vérifier si un ensemble de politiques contient une politique ne pouvant pas être utilisée car sa cible et celle de l'ensemble de politiques sont disjointes
- *uselessPolicySet* pour vérifier si un ensemble de politiques contient un autre ensemble de politiques ne pouvant pas être utilisé car leurs deux cibles sont disjointes

Ces prédicats sont définis en ALLOY comme suit :

```
pred uselessRule(p : Policy){
  some r : p.rules | targetDisj(p.policyTarget, r.ruleTarget)}

pred uselessPolicy(s : PolicySet){
  some p:s.policies | targetDisj(s.policySetTarget,p.policyTarget)}

pred uselessPolicySet(s : PolicySet){
  some p:s.policySets | targetDisj(s.policySetTarget,p.policySetTarget)}
```

6.3 Analyses et vérifications supplémentaires

Dans les sections précédentes, nous avons étudié des propriétés indépendantes du contexte de requête. Ce dernier est un facteur important dans la prise de décision dans un système basé sur le langage XACML. Dans cette section, nous proposons d'étudier des propriétés additionnelles que notre approche permet d'analyser et de vérifier.

6.3.1 Requêtes conflictuelles

La réponse d'une politique de contrôle d'accès dépend des règles qu'elle intègre, de son algorithme de combinaison et de la requête d'accès. La disponibilité de plusieurs politiques ou même ensembles de politiques implique la possibilité d'avoir plusieurs réponses différentes qui peuvent se contredire. Le rôle des algorithmes de combinaison des politiques est de résoudre ce genre de conflits, mais en être au courant peut s'avérer utile pour détecter des failles de sécurité potentielles.

Nous proposons dans la suite un ensemble de prédicats permettant de définir certains contextes conflictuels.

Le prédicat *InconsistentRules* permet de vérifier si une politique contient deux règles contradictoires et ce pour un contexte de requête particulier.

```
pred InconsistentRules (p : Policy, req : Request) {
some r : p.rules | ruleResponse(r, req) = Permit
some r : p.rules | ruleResponse(r, req) = Deny
}
```

Le prédicat *InconsistentPolicies* permet, à son tour, de déterminer si un ensemble de politiques contient deux politiques contradictoires pour un contexte de requête particulier.

```
pred InconsistentPolicies (s : PolicySet, req : Request) {
some p : s.policies | policyResponse(p, req) = Permit
some p : s.policies | policyResponse(p, req) = Deny
}
```

Les règles conflictuelles sont également définies par le chevauchement des cibles (voir section 6.2.1). Le prédicat *InconsistentRules* associe cette situation de conflits à un

contexte de requête particulier. L'outil *ALLOY Analyzer* permet même de montrer un exemple de telle requête comme montré dans le paragraphe 7.6.3.

Par contre, une situation de conflits entre politiques ou ensembles de politiques doit être spécifiée par une requête. Les relations entre cibles ne peuvent pas seules renseigner sur de telles situations puisqu'une politique ou un ensemble de politiques ne fournit pas de réponse directe à l'instar des règles.

6.3.2 Politiques positives et négatives

Nous appelons une politique positive (respectivement négative) toute politique qui retourne un *Permit* (respectivement *Deny*) dans tous les cas, c'est-à-dire, pour n'importe quel contexte de requête. Le prédicat *positivePolicy* et *ngativePolicy* permettent de déterminer, respectivement, si une politique est positive ou négative.

```
pred positivePolicy(p : Policy){
  all q : Request | policyResponse(p, q) = Permit}

pred negativePolicy(p : Policy){
  all q : Request | policyResponse(p, q) = Deny}
```

Ce genre de situation peut être causé par un algorithme de combinaison non approprié et une règle de contrôle d'accès toujours appliquée et retournant toujours un *Permit* ou un *Deny*. Par exemple si une politique de contrôle d'accès a pour algorithme de combinaison *Permit-overrides* et une règle toujours appliquée et retournant un *Permit*, la réponse de cette politique sera toujours positive. Une autre cause possible est la présence pour chaque contexte de requête d'au moins une règle retournant *Permit*.

Ces deux propriétés peuvent aussi s'appliquer aux règles ainsi qu'aux ensembles de politiques.

6.3.3 Tests de conformité

Généralement, un système implémentant les politiques de contrôle d'accès en XACML est soumis à un ensemble de tests de conformité qui servent à vérifier certaines fonctionnalités.

Il est également possible de soumettre le modèle ALLOY des politiques de contrôle d'accès à ces mêmes tests. Mais, il faut d'abord les traduire en ALLOY comme c'est le

cas par exemple pour le langage TTCN [53]. Donc il faut préparer un certain nombre de prédicats et d'assertions logiques que nous pouvons vérifier et tester.

Un exemple de fonctionnalité pouvant être testée sur le modèle est :

L'accès doit être toujours accordé pour un sujet du profil professeur

Nous définissons pour ce faire l'assertion *PermitForProfessor* qui traduit cette propriété en termes de logique d'ALLOY.

```
assert PermitForProfessor {
  all q : Request {
    {~(q.subject.attributes).Role = Professor}
    => policyResponse(P,q) = Permit }}
```

L'outil d'analyse d'ALLOY permet de valider cette assertion ou d'en trouver un contre exemple dans le cas où elle est invalide.

Chapitre 7

Transformation de XACML vers ALLOY

7.1 Approche générale

Les politiques de contrôle d'accès en XACML seront traitées afin d'en extraire les différentes informations nécessaires qui permettent de construire de nouvelles spécifications en ALLOY. Ces spécifications seront basées sur le modèle global de XACML défini dans le chapitre 5.

Les propriétés définies dans le chapitre 6, seront analysées et vérifiées sur le modèle final en ALLOY des politiques de contrôle d'accès. Les vérifications peuvent être réalisées avec l'outil *ALLOY Analyzer*.

L'approche générale de cette transformation peut se résumer aux étapes suivantes illustrées par la figure 7.1 :

1. Extraction automatique des attributs des sujets, ressources et actions, détaillée dans la section 7.3
2. Construction des règles, politiques et ensembles de politiques, détaillée dans la section 7.4
3. Intégration automatique des modèles obtenus avec le modèle global de XACML ainsi que l'intégration manuelle des contraintes du domaine (contraintes non logiques). Dans cette étape nous pouvons intégrer également les contraintes imposées par les conditions des règles. Cette étape est détaillée dans la section 7.5.

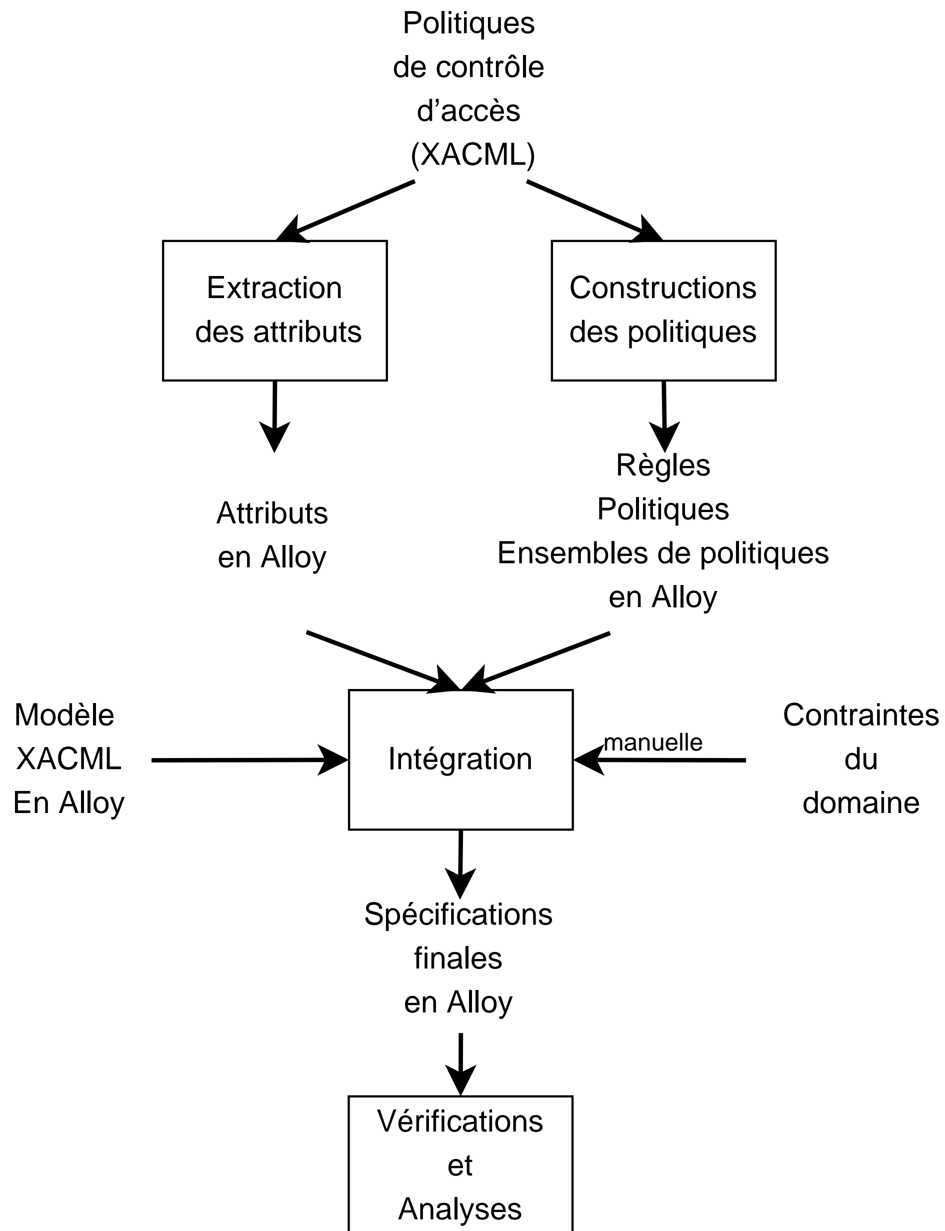

Fig. 7.1 – Approche de vérification des politiques de contrôle d'accès

4. Analyse des interactions dans les politiques de contrôle d'accès, détaillée dans la section 7.6

Avant de détailler ces transformations, nous allons présenter un exemple d'application qui sera utilisé le long de ce chapitre.

7.2 Exemple d'application

Considérons les locaux d'une entreprise comme ressources auxquelles peuvent accéder des sujets. Un sujet peut visiter un local (*Visit*), le verrouiller (*Lock*) ou le déverrouiller (*unlock*).

Ces locaux sont définis par un identificateur *Room* et peuvent être des locaux publics (*Public*) ou privés (*Private*).

Chaque sujet est caractérisé par un identificateur et un profil. Les profils possibles d'un sujet sont : directeur (*Manager*), employé (*Employee*), gardien (*Doorman*) et visiteur (*Visitor*). De plus, un sujet dont le profil est directeur, employé ou gardien peut être actif (*Active*) ou en vacance (*Vacation*).

Le contrôle d'accès est régi par des règles organisées en deux politiques. Une politique générale appelée *Default*. Une autre politique, appelée *VacationPolicy*, est spécifique aux employés de l'entreprise en vacances, y compris les gardiens et les directeurs.

La politique *Default* est composée des règles suivantes :

1. Un gardien peut visiter, verrouiller ou déverrouiller tous les locaux
2. Toute personne peut visiter les locaux publics
3. Un directeur peut visiter ou déverrouiller tous les locaux
4. Un employé peut visiter tous les locaux
5. Un employé peut déverrouiller les locaux publics
6. Un visiteur peut visiter les locaux publics
7. Toute personne autre que le gardien ne peut verrouiller aucun local
8. Un employé ne peut déverrouiller les locaux privés
9. Un visiteur ne peut pas visiter les locaux privés
10. Un visiteur ne peut verrouiller ni déverrouiller aucun local
11. La personne dont l'identificateur est "EMP01" ne peut pas accéder aux locaux

	Attributs possibles	**Valeurs possibles**
Sujet	SubjectId	EMP01
	Profile	Manager Employee Doorman Visitor
	State	Active Vacation
Ressource	ResourceId	Room
	Privacy	Private Public
Action	ActionId	Visit Lock Unlock

TAB. 7.1 – Attributs et valeurs

La politique *Default* est composée des règles suivantes :

12. Aucun gardien ni employé en vacance ne peut accéder aux locaux publics
13. Aucun gardien ni employé en vacance ne peut verrouiller ni déverrouiller les locaux
14. Un visiteur peut visiter les locaux publics

Le tableau 7.1 résume les attributs et leurs valeurs possibles pour les sujets, ressources et actions et ce, pour le contexte que nous venons de décrire.

7.3 Extraction des attributs

Cette étape permet d'extraire les informations concernant les attributs des sujets, ressources et actions à partir des politiques de contrôle d'accès définies en XACML. Cette extraction est automatique et se fait au moyen d'un outil intégré.

Cette étape est composée de trois étapes intermédiaires montrées par la figure 7.2 et décrites dans les sous-sections suivantes.

7.3.1 Extraction des attributs

Cette étape prend en entrée le fichier au format XML contenant les politiques de contrôle d'accès en XACML. En sortie, cette étape génère un fichier XML contenant l'ensemble des attributs tels que utilisés dans les politiques.

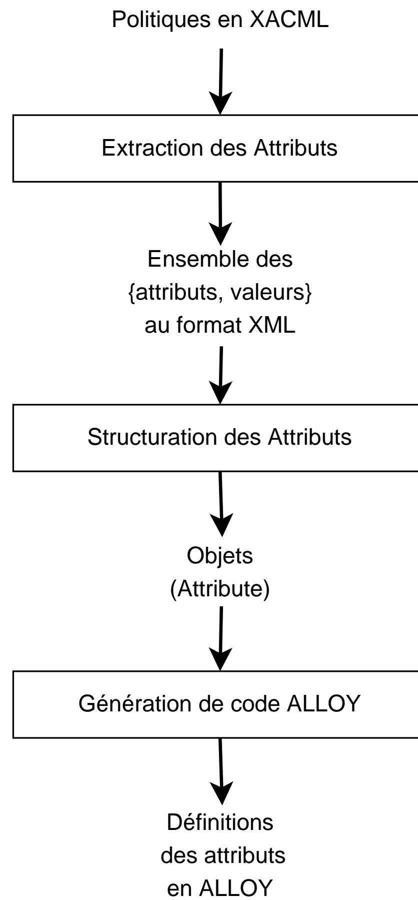

Fig. 7.2 – Extraction des attributs

Nous avons utilisé les transformations *XSL* [55, 20] sur le fichier XACML d'entrée. Pour chaque attribut rencontré, l'élément auquel il appartient, son type et sa valeur sont extraits. Ainsi, un nouveau fichier XML est obtenu. Ce fichier contient plusieurs entrées redondantes puisqu'un attribut peut être utilisé par plusieurs règles ou politiques. L'étape qui suit consiste à structurer les attributs extraits et éliminer ainsi les redondances.

7.3.2 Structuration des attributs

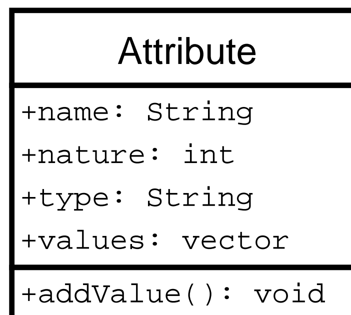

FIG. 7.3 – La classe Attribute

Dans cette étape le fichier XML obtenu comme résultat de l'étape précédente est analysé, les redondances des attributs sont éliminées et un ensemble d'objets de classe *Attribute* est généré. La classe *Attribute*, illustrée par la figure 7.3, est définie par les attributs de classe suivants :

- *name* : indique le nom de l'attribut
- *nature* : indique si l'attribut est un attribut de sujet, ressource ou action
- *type* : indique le type de l'attribut c'est-à-dire le type associé aux valeurs de l'attribut. Cet attribut sert à différencier entre deux attributs qui peuvent avoir le même nom mais deux types de valeurs différents. En général, cet attribut n'aura pas une influence sur notre approche.
- *values* : contient une liste de valeurs possibles

Cette transformation est réalisée à l'aide de l'implémentation des objets *DOM* [54] en *JAVA* [41].

7.3.3 Génération des définitions d'attributs en ALLOY

À partir d'un ensemble d'objets structurés représentant les attributs des sujets, ressources et actions, nous pouvons générer les spécifications dans le langage ALLOY. Nous utilisons pour ce faire les objets *DOM* et les transformations *XSL* (voir la figure 7.4).

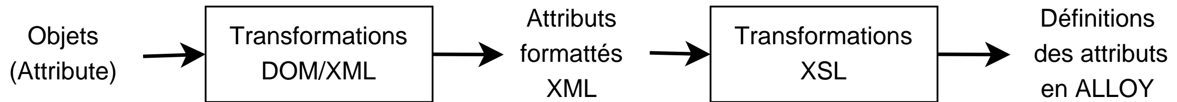

FIG. 7.4 – Génération des définitions d'attributs en ALLOY

D'abord les objets de classe *Attribute* sont transformés en objets *DOM* qui peuvent être ensuite transformés automatiquement en fichier XML. Le fichier XML contient les attributs structurés d'une manière proche des spécifications souhaitées en ALLOY. La figure 7.5 montre une partie de ce fichier XML sous la forme d'un arbre. Cette figure montre l'exemple de l'attribut *Profile*. La balise *<nature>* indique que cet attribut est un attribut de sujet. La balise *<values>* contient les valeurs possibles de cet attributs qui sont *Doorman*, *Manager*, *Employee* et *Visitor*.

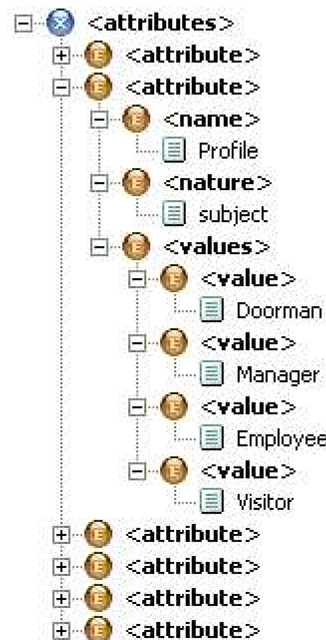

FIG. 7.5 – Attributs formatés

Finalement, en utilisant les transformations *XSL*, les spécifications des attributs en ALLOY sont générées. Chaque attribut extrait forme une signature qui étend la signature

Attribute définie dans la sous-section 5.1.1. De même les valeurs des attributs définissent des signatures qui étendent la signature *Value*. De plus, nous définissons les associations entre attributs et leurs valeurs possibles avec la relation *values*.

La partie de code suivante en ALLOY est générée automatiquement à partir des politiques en XACML. La signature *Profile* définit le profil d'un sujet qui peut être un gardien, un employé, un directeur ou un visiteur.

```
one sig Profile extends Attribute {}{
  values = Doorman + Manager + Employee + Visitor
}
one sig Doorman, Manager, Employee, Visitor extends Value{}
```

La partie de code suivante représente une contrainte logique qui limite les attributs des sujets à *Profile*, *State* et *SubjectId*, les attributs des ressources à *ResourceId* et *Privacy* et les attributs des actions à *ActionId*.

```
fact {
Subject.attributes.Value =Profile + State + SubjectId
Resource.attributes.Value =ResourceId + Privacy
Action.attributes.Value =ActionId
}
```

7.4 Construction des politiques

Cette étape consiste à extraire les règles, les politiques et les ensembles de politiques de contrôle d'accès. Le fichier initial contenant les politiques de contrôle d'accès en XACML est analysé et les contraintes de contrôle d'accès sont construites. Au cours de cette étape, nous allons extraire les informations suivantes et construire leur définition en ALLOY :

- sujets, ressources et actions
- cibles
- règles
- politiques
- ensembles de politiques

Pour aboutir à des spécifications finales en ALLOY, deux étapes sont nécessaires :

- Analyse du fichier XACML et construction d'un ensemble d'objets Java
- Transformation de ces objets en spécifications ALLOY

7.4.1 Construction d'objets *JAVA*

Cette étape consiste à analyser les informations contenues dans les politiques de contrôle d'accès en XACML pour construire un ensemble d'objets structurés pour les sujets, ressources et actions. Chacun de ces éléments est défini par un ensemble d'attributs et valeurs. Nous avons donc créé une seule classe *Elem* pour représenter ces éléments. Chaque objet de la classe *Elem* est défini par son nom et par une liste contenant des objets de la classe *ElementAttribute*. La classe *ElementAttribute* est caractérisée par le nom de l'attribut, sa valeur et le type de correspondance entre l'attribut et sa valeur (seul l'opérateur d'égalité a été traité dans ce mémoire). Les objets de la classe *ElementAttribute* servent à stocker les couples (attribut, valeur).

Ensuite, à partir des objets définissant les sujets, ressources et actions, il est possible de construire les objets cibles. La classe *Target* est donc définie et chaque objet de cette classe est défini par un nom, un ensemble de sujets, un ensemble de ressources et un ensemble d'actions.

Nous définissons donc la classe *Rule* qui représente l'ensemble des règles. Chaque objet de la classe *Rule* est défini par un nom, un effet et un objet de classe *Target* qui représente la cible.

Dès lors, il devient possible de construire les politiques et aussi les ensembles de politiques. Chaque objet de la classe *Policy* est défini par un nom de la politique, une cible, un algorithme de combinaison des règles et une liste de règles. Chaque objet de la classe *PolicySet* est défini aussi par un nom, une cible et un algorithme de combinaison des politiques. De plus, cette classe définit un ensemble de politiques. Nous nous sommes contentés des ensembles de politiques qui contiennent seulement des politiques et non des ensembles de politiques. Mais la même approche peut s'appliquer à ce cas.

La figure 7.6 montre le diagramme de classes des objets utilisés dans notre méthode de transformation. Ce diagramme reprend la même structure définie dans la figure 2.3 du chapitre 2.

Pour montrer comment ces objets sont construits nous proposons l'exemple de règle suivant :

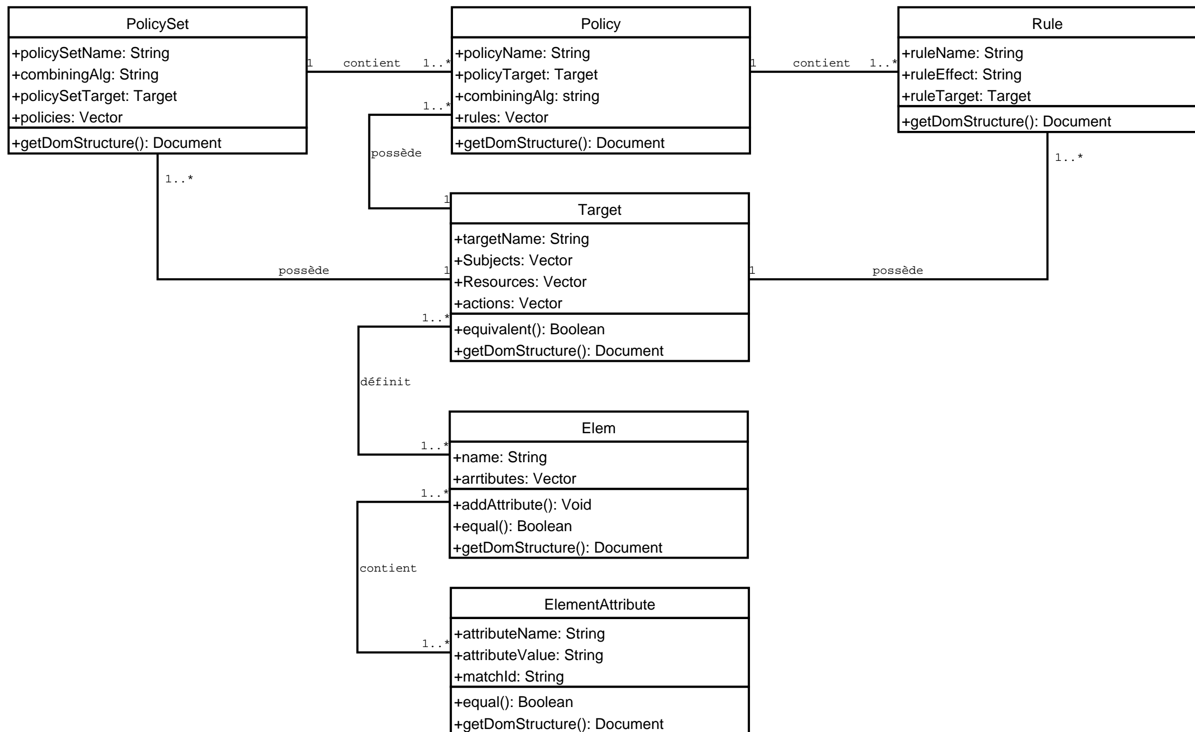

FIG. 7.6 – Diagramme de classes

> Aucun employé ni gardien en vacance ne peut verrouiller ni déverrouiller les locaux

Deux sujets sont définis par cette règle :
- les employés que nous appelons *Subject1* définis par le couple (*Profile*, *Employee*)
- les gardiens que nous appelons *Subject2* définis par le couple (*Profile*, *Doorman*)

De plus, une ressource *Resource1* est définie par le couple (*ResourceId*, *Room*).
Enfin, la règle ci-dessus définit deux actions :
- le verrouillage que nous appelons *Action1* définie par le couple (*ActionId*, *Lock*)
- le déverrouillage que nous appelons *Action2* définie par le couple (*ActionId*, *Unlock*)

Une cible *Target1* est donc définie comme suit :

$$Target1: \quad subjects = \{Subject1, Subject2\}$$
$$resources = \{Resource1\}$$
$$actions = \{Action1, Action2\}$$

Et la règle sera définie comme suit :

$$\begin{aligned} Rule1: \; & ruleName = Rule1 \\ & ruleEffect = Deny \\ & ruleTarget = Target1 \end{aligned}$$

À la fin de cette étape, des listes d'ensembles de politiques, de politiques, de règles, de cibles et d'éléments (sujets, ressources et actions) sont obtenues. Nous avons également éliminé les redondances dans ces listes, c'est-à-dire, si le même sujet, par exemple, est utilisé par plusieurs cibles, alors un seul sera créé et non plusieurs.

7.4.2 Transformation des objets en spécifications ALLOY

Le résultat de l'étape précédente est un ensemble d'objets structurés. Dans cette étape nous transformons d'abord ces objets en des spécifications XML en utilisant les objets DOM. Ensuite, nous utilisons les transformations XSL pour traduire vers ALLOY. La figure 7.7 montre le format XML pour les éléments et la Figure 7.8 montre celui des cibles.

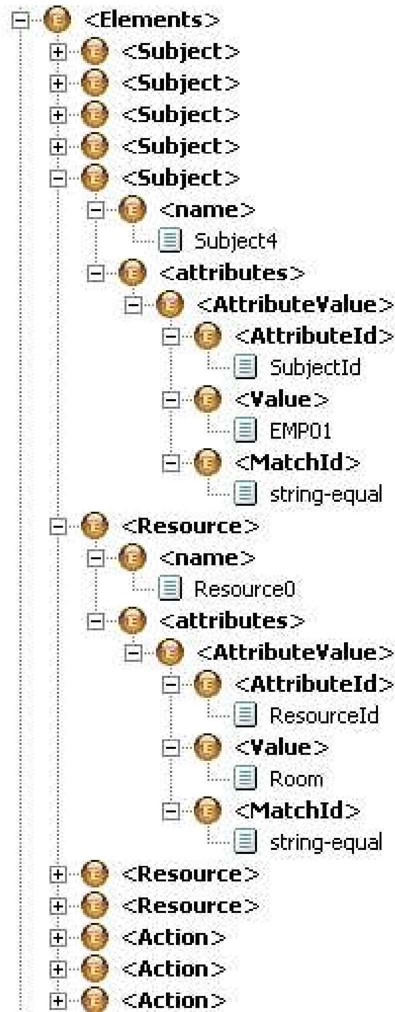

Fig. 7.7 – Eléments structurés

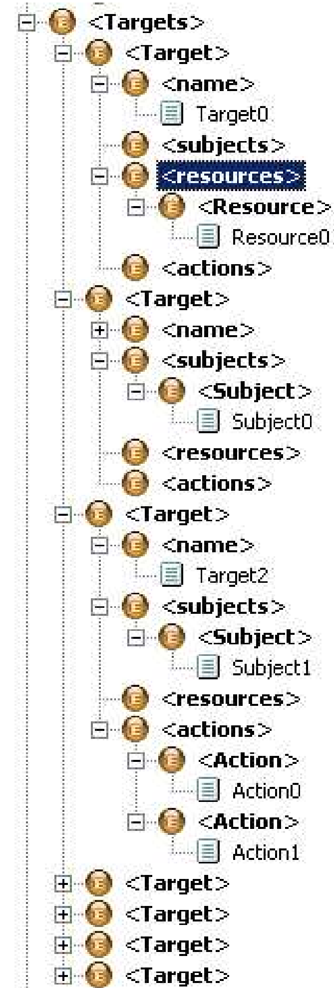

Fig. 7.8 – Cibles structurées

Finalement, nous obtenons des spécifications en ALLOY sous la forme suivante :

```
one sig Subject0 extends Subject{}{
  attributes =  Profile->Doorman}
one sig Target1 extends Target {}{
  subjects =  Subject0
  resources =  none
  actions =  none}
one sig Rule1 extends Rule {}{
  ruleTarget = Target1
  ruleEffect = Permit}
```

Ainsi, à chaque objet extrait est associée la signature correspondante dans ALLOY. Chaque attribut d'objet définit un attribut de signature.

7.5 Intégration

La dernière étape de la transformation des politiques de contrôle d'accès en XACML vers ALLOY consiste à rassembler les différentes parties obtenues à l'issu des étapes précédentes et les enrichir avec d'autres contraintes du domaine.

7.5.1 Intégration

Afin d'obtenir des spécifications complètes en ALLOY des politiques de contrôle d'accès nous devons rassembler :

– le modèle général de XACML en ALLOY défini dans le chapitre 5
– les définitions des attributs et leurs valeurs (section 7.3)
– les définitions des éléments, des règles, des politiques et des ensembles de politiques (section 7.4)

Il faut également définir les différentes propriétés à vérifier. Nous proposons dans les spécifications finales de vérifier toutes les propriétés définies dans le chapitre 6.

De plus, nous devons spécifier le nombre d'instances (voir sous-section 3.4.2) parmi lesquelles l'outil *Alloy Analyzer* va rechercher des solutions possibles (nous l'avons également appelé limite de vérification).

Nombre d'instances

Pour les propriétés relatives aux interactions entre cibles, nous n'avons pas à créer d'autres instances de signatures à part celles automatiquement extraites et qui ont un nombre fini.

Par contre, pour l'étude des requêtes conflictuelles, l'analyseur de modèle ALLOY est censé trouver une instance de requête qui génère un conflit. Cette requête contient un sujet, une ressource et une action qui ne sont peut être pas définis, c'est-à-dire, qui ne sont pas inclus dans une cible. En effet, un élément est caractérisé par une combinaison d'attributs et valeurs. Quelques éléments sont présents dans les cibles. Les éléments, qui participent à satisfaire une propriété donnée, ne sont peut être pas parmi ceux qui sont définis par les cibles. Nous souhaitons qu'ALLOY trouve un exemple de ces éléments.

Prenons l'exemple simple des deux règles :

1. Aucune personne en vacance ne peut visiter les locaux
2. Un directeur peut toujours visiter les locaux

La première règle définit le sujet, la ressource, l'action et la cible suivants :

$Subject1$: $\{(state, vacation)\}$

$Resource1$: $\{(ResourceId, Room)\}$

$Action1$: $\{(ActionId, Visit)\}$

$Target1$: $subjects = \{Subject1\}$

$resources = \{Resource1\}$

$actions = \{Action1\}$

La deuxième règle définit le sujet et la cible suivants :

$Subject2 = \{(profile, Manager)\}$

$Target2 = subjects = \{Subject2\}$

$resources = \{Resource1\}$

$actions = \{Action1\}$

Avec deux règles nous avons pu définir deux sujets, une ressource, une action et deux cibles. Il est clair que si un sujet qui est directeur en vacance demande de visiter les locaux, alors il aura deux réponses possibles puisque la première règle lui interdit l'accès et la seconde le lui accorde. Une telle requête qui génère un conflit, définit une nouvelle instance de sujet :

$Subject3 = \{(profile, Manager), (state, vacation)\}$

Nous désirons que *ALLOY Analyzer* trouve ces nouvelles instances. Nous supposons qu'une seule requête à la fois peut parvenir au système. Donc, au plus, nous avons besoin

de trois instances supplémentaires par rapport à celles déjà extraites : une instance pour chaque élément (sujet, ressource et action).

C'est pourquoi nous avons adopté comme limite de vérification le nombre d'objets total dans les spécifications ajouté de trois. Ce nombre d'instances est suffisant pour trouver des exemples et des contres exemples aux propriétés définies dans le chapitre 5.

7.5.2 Contraintes du domaine

Les politiques de contrôle d'accès contiennent des informations partielles sur le domaine des systèmes analysés. Tenir compte seulement des informations contenues dans les règles XACML générera peut être un modèle incomplet, mais surtout, avec des contraintes incomplètes. Ceci a pour conséquence d'avoir une simulation et une vérification sur un domaine plus large que celui du système réel. Afin d'affiner le modèle obtenu, nous avons besoin d'ajouter les contraintes du domaine qui sont des contraintes spécifiques au contexte général du système. L'intégration de ces contraintes sera manuelle dans notre cas puisqu'elles ne sont pas fournies dans les politiques de contrôle d'accès qui constituent le point de départ de notre travail.

Comme exemple de contraintes de domaine nous pouvons spécifier que dans une requête :

- un sujet est défini par un seul identifiant
- un sujet est défini par un seul profil
- une ressource est définie par un seul identifiant
- une ressource est soit privée, soit publique

```
fact {
  all q : Request  | one SubjectId.(q.subject.attributes)
  all q : Request  | one Profile.(q.subject.attributes)
  all q : Request | one ResourceId.(q.resource.attributes)
  all q : Request | one Privacy.(q.resource.attributes)
}
```

Prise en compte des conditions

Dans ce mémoire nous n'avons pas traité les conditions dans la transformation automatique de XACML vers ALLOY. Un moyen possible de les prendre en compte est de considérer les contraintes imposées par ces dernières et les intégrer manuellement dans

les spécifications des politiques de contrôle d'accès en ALLOY. Ainsi la fonction qui retourne la réponse d'une règle devient :

```
fun ruleResponse (r : Rule, req : Request) : Effect {
  if targetMatch(r.ruleTarget, req) && evalCondition(r, req)
    then r.ruleEffect
  else NotApplicable
}
```

Le prédicat *evalCondition* évalue la condition de chaque règle. L'exemple ci-dessus suppose que la ressource possède un attribut supplémentaire appelé *Owner* qui indique le propriétaire de la ressource. Une condition de la règle *Rule1* a été ajouté et elle vérifie si le sujet demandant l'accès est le propriétaire de la ressource demandée.

```
pred evalCondition (r : Rule, q : Request){
  r = Rule1 =>
    SubjectId.(q.subject.attributes) = Privacy.(q.resource.attributes)
}
```

Nous rappelons encore que nous n'avons pas traité les fonctions complexes de XACML et que nous nous sommes limités aux fonctions simples telles que l'opérateur d'égalité.

7.6 Analyses et vérifications

Dans cette section nous présenterons l'utilisation et les fonctionnalités de l'outil intégré développé pour transformer les spécifications XACML du contrôle d'accès vers les spécifications ALLOY. Cet outil présente une interface graphique montrée par la figure 7.9.

Cet outil permet de saisir un fichier au format XACML et assure les fonctionnalités suivantes :

- Visualisation des politiques de contrôle d'accès
- Transformation des politiques de contrôle d'accès vers ALLOY
- Analyse des interactions et des conflits

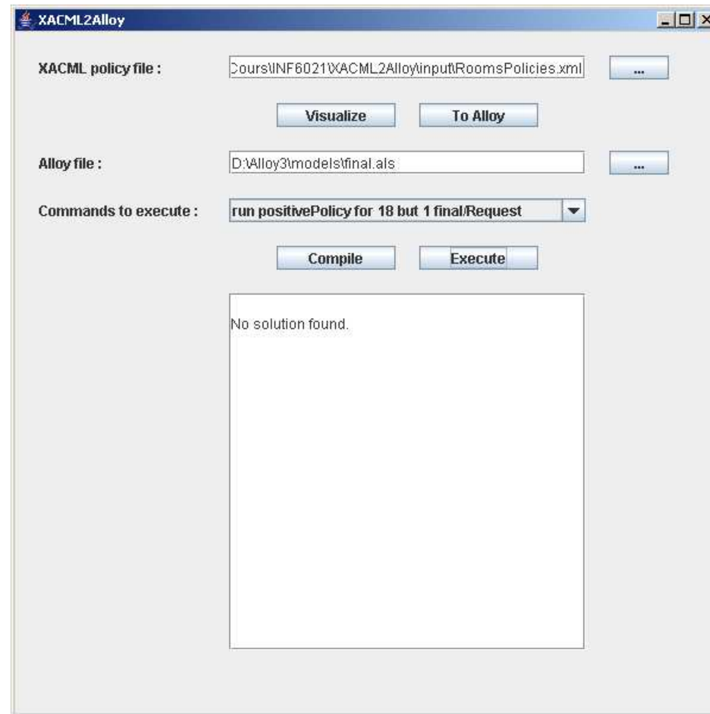

FIG. 7.9 – Outil intégré d'analyse

7.6.1 Visualisation

Nous proposons un format plus ergonomique et plus compréhensible que le format XML pour les politiques de contrôle d'accès. Nous avons utilisé pour réaliser cette interface les transformations *XSL* pour afficher les politiques XACML sous le format *HTML* et en langage naturel comme montré par la figure 7.10. Pour obtenir ce nouveau format des politiques il suffit de cliquer sur le bouton *Visualize* de la figure 7.9.

7.6.2 Transformation vers ALLOY

Cet outil permet également de transformer les politiques XACML vers des spécifications ALLOY en suivant les démarches montrées tout au long de ce chapitre. Pour ce faire, il faut d'abord saisir un fichier XACML dans le premier champ de la figure 7.9. Ensuite, il faut saisir dans le deuxième champ le fichier ALLOY destination. Enfin, cliquer sur le bouton *ToAlloy* permet de créer le fichier ALLOY des politiques de contrôle d'accès.

Identificateur de la politique : Default
Algorithme de combination des règles :Permit-overrides
Description :
Policy to control the rooms visists
Cible :

- **Sujet :** Tous les sujets
- **Ressource :**
  - Si attribut **ResourceId** correspond à la valeur **Room**
- **Action :** Toutes les actions

Règles de contrôle d'accès :

Règle 1

Identificateur de la règle: Rule6
Effet: Permit
Description :
A visiotr can visit any public room
Cible :

- **Sujet :**
  - Si attribut **Profile** correspond à la valeur **Visitor**
- **Ressource :**
  - Si attribut **Privacy** correspond à la valeur **Public**
- **Action :**
  - Si attribut **ActionId** correspond à la valeur **Visit**

Règle 2

Identificateur de la règle: Rule2
Effet: Permit
Description :
Anyone can visit any public room
Cible :

Fig. 7.10 – Visualisation de XACML

7.6.3 Analyses et vérifications

Nous proposons dans cette sous-section de montrer certains résultats des analyses des interactions proposées dans le chapitre 6. Nous allons considérer l'exemple de la section 7.2. L'outil permet de compiler les spécifications d'ALLOY en utilisant la bibliothèque de classes fournie par *ALLOY Analyzer*. Il est également possible d'exécuter des commandes (prédicats ou assertion). Le résultat obtenu est au format textuel et il fournit les informations complètes sur les instances trouvées.

Règles redondantes

Dans la politique de contrôle d'accès *Default* définie dans la section 7.2, l'analyse du prédicat *reduntantRules* permet de détecter, comme le montre la figure 7.11, que les règles 2 et 4 causent une situation de redondance. En effet, la cible de la règle 2 permet à tous les sujets de visiter les locaux publics et elle chevauche avec la cible de la règle 4 qui permet aux employeurs de visiter les locaux. Donc, l'intersection des deux cibles est une intersection non vide.

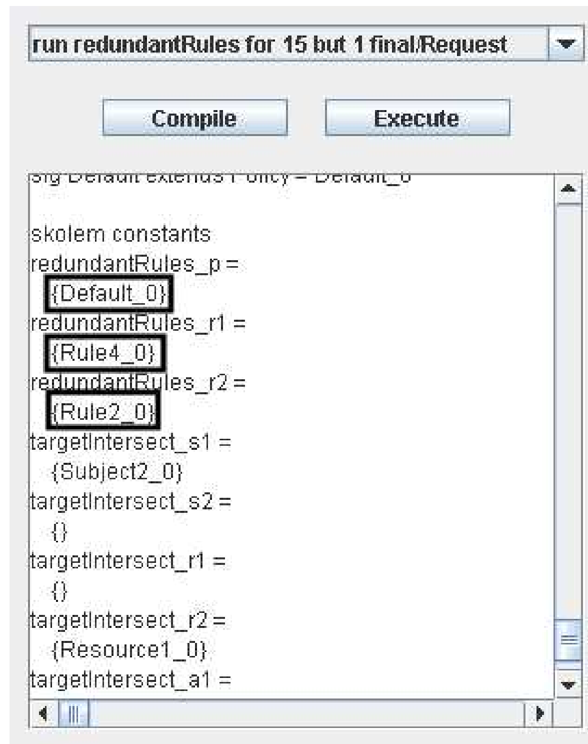

FIG. 7.11 – Règles redondantes

Règles conflictuelles

Dans la politique de contrôle d'accès *Default*, l'analyse du prédicat *conflictingRules* permet de détecter que les règles 6 et 11 sont deux règles conflictuelles (voir figure 7.12. En effet la règle 6 autorise aux visiteurs de visiter les locaux publics alors la règle 11 défend à la personne identifiée par *EMP01* d'accéder aux locaux. Les deux cibles chevauchent puisqu'un visiteur peut être identifié par EMP01.

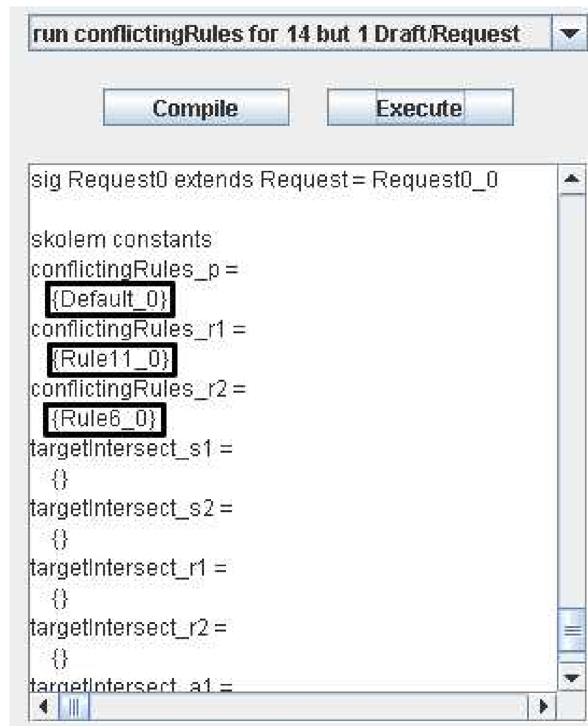

Fig. 7.12 – Règles conflictuelles

Cibles incohérentes

Dans la politique de contrôle d'accès *VacationPolicy*, comme le montre la figure 7.13, la règle 14 est une règle inutile puisque sa cible est incohérente avec le domaine d'application de la politique qui la contient. En effet la politique s'applique aux employés, gardiens et directeurs alors que la règle 14 s'adresse aux visiteurs.

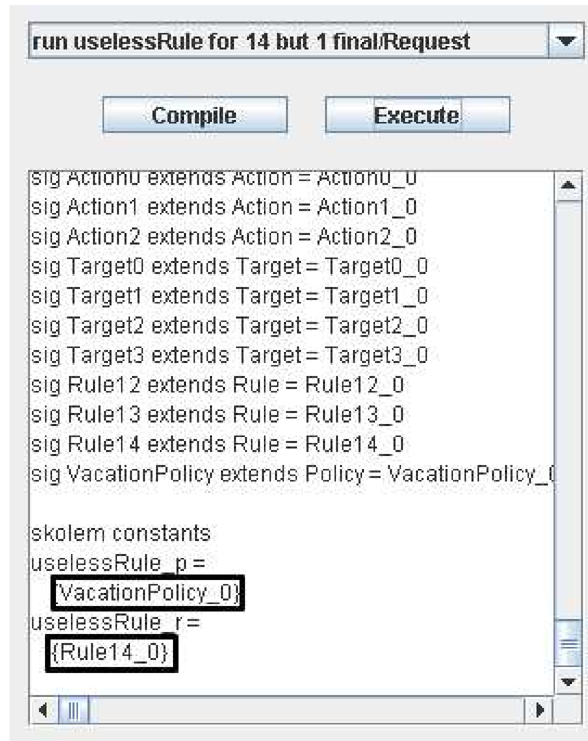

FIG. 7.13 – Règle inutile

Requêtes conflictuelles

Finalement, nous allons interroger les spécifications en ALLOY sur la possibilité d'avoir des requêtes pouvant être appliquées par deux règles de la politique *Default*. La figure 7.14 montre que les règles 2 et 11 sont appliquées pour la requête *Request_ 0*. Cette dernière demande pour un sujet identifié par EMP01 et qui est aussi un directeur, de visiter un local public. La règle 2 autorise cet accès alors que la règle 11 l'interdit.

La sortie ci-dessous (extraite de la fenêtre montrée par la figure 7.14) est générée automatiquement par ALLOY et elle montre que la requête *Request_ 0* est définie par le sujet *Subject_ 1*, la ressource *Resource_ 0* et l'action *Action0_ 0*.

```
sig Request extends univ = {Request_0}
  subject : one final/Subject =
    {Request_0 -> Subject_1}
  resource : one final/Resource =
    {Request_0 -> Resource_0}
  action : one final/Action =
```

```
{Request_0 -> Action0_0}
```

De même ALLOY indique par la sortie ci-dessous (également extraite de la fenêtre montrée par la figure 7.14) que *Subject_1* est défini par un profil de directeur (*Manager*) et un identifiant *EMP01*, que la ressource *Resource_0* est une ressource publique identifiée par *Room* et que l'action *Action0_0* est *Visit*.

```
Subject_1 -> {Profile_0 -> Manager_0, SubjectId_0 -> EMP01_0}
Resource_0 -> {Privacy_0 -> Public_0, ResourceId_0 -> Room_0}
Action0_0 -> ActionId_0 -> Visit_0
```

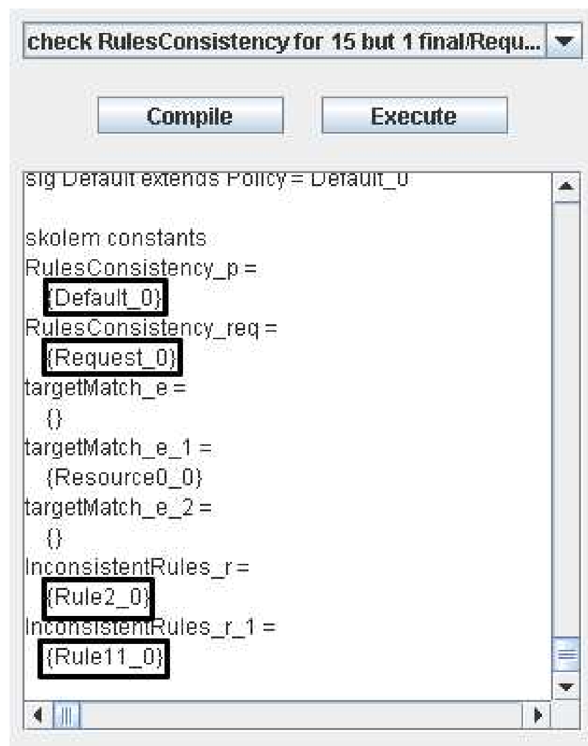

FIG. 7.14 – Requête conflictuelle

Chapitre 8

Conclusion

Dans ce mémoire, nous avons présenté une méthode basée sur la modélisation et la logique relationnelle de premier ordre pour analyser et vérifier différents types d'interactions dans les politiques de contrôle d'accès exprimées en XACML.

Ce travail a montré les possibilités de la modélisation logique de premier ordre dans la validation et la vérification des politiques de contrôle d'accès. En effet, un ensemble de règles peut contenir plusieurs conflits et incohérences potentiels et non évidents. Un outil tel qu'ALLOY nous permet de détecter ces situations et même d'en avoir des exemples.

Grâce à ce travail, des failles de sécurités et des interruptions de service peuvent être identifiées et évitées. De plus, le format textuel que nous proposons rend les politiques de contrôle d'accès plus lisibles et plus compréhensibles.

Ce travail a permis d'apporter les contributions suivantes :

- la proposition d'un modèle logique de l'environnement XACML et ce en utilisant le langage ALLOY
- la spécification formelle des interactions et conflits dans les politiques de contrôle d'accès XACML
- la transformation des politiques de contrôle d'accès XACML vers des spécifications ALLOY vérifiables et analysables
- la réalisation d'un outil de vérification des politiques de contrôle d'accès
- la visualisation des politiques de contrôle d'accès XACML avec un format textuel ergonomique.

Toutefois, dans ce mémoire, nous n'avons considéré qu'un sous-ensemble du langage XACML. En effet, les conditions dans les règles d'accès n'ont pas été transformées automatiquement vers ALLOY, elles ont été considérées comme des contraintes du domaine.

De plus, nous n'avons traité que les fonctions simples d'égalité, alors que XACML propose un ensemble de fonctions traitant plusieurs types de données. Nous n'avons pas traité également les attributs de l'environnement tels que les dates et l'heure. Enfin, les obligations de XACML n'ont pas été considérées.

Nous devons aussi souligner que notre outil n'est pas capable d'identifier toutes les interactions possibles, à cause des caractéristiques fonctionnelles de l'outil ALLOY.

La suite de ce travail pourrait être l'extension de la méthode présentée aux obligations et aux autres fonctions du langage telles que les opérateurs d'inégalité.

Annexe A

Modèle *ALLOY* : Famille

```
module Famille
abstract sig Personne {
  /* Une personne possède zéro ou un seul père */
  pere : lone Personne,
  /* Une personne possède zéro ou une seule mère */
  mere : lone Personne
}
{pere in Homme
mere in Femme}
/* Une personne peut être soit */
/* un homme soit une femme     */
abstract sig  Homme, Femme extends Personne {}
/* Paul et Jean sont deux hommes */
one sig Paul, Jean extends Homme {}
/* Anna et Emma sont deux femmes */
one sig Anna, Emma extends Femme {}
/* Contraintes générales */
fact {
  no p:Personne | p.pere = p
  no p:Personne | p.mere = p
  no p,m:Personne | p.mere = m && m.pere = p
  no p,m:Personne | p.mere = m && m.mere = p
  no p,m:Personne | p.pere = m && m.pere = p
}
/* Définitions des relations entre */
/* les membres de la famille       */
fact {
Jean.pere = Paul
Jean.mere = Anna
Emma.pere = Paul
Emma.mere = Anna }
/* Relation de frère et soeur */
pred FrereSoeur(f:Homme, s:Femme) {
f.pere = s.pere or f.mere = s.mere }
/* Une personne ne peut pas être */
/* un père et une mère à la fois */
assert PereMere { no p : Personne | p.mere = p.pere }
/* Simulation */
run FrereSoeur for 1
/* Vérifiaction de propriété */
check PereMere for 1
```

Annexe B

Politique de contrôle d’accès en XACML

Nous montrons dans cet annexe l’exemple de la politique *VacationPolicy*, définie dans la section 7.2, sous le format original de XACML.

```
<Policy PolicyId="VacationPolicy" RuleCombiningAlgId="Deny-overrides">
  <Description>
      Policy to control the access of persons in vacation
  </Description>
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Doorman</AttributeValue>
          <SubjectAttributeDesignator AttributeId="Profile" DataType="string"/>
        </SubjectMatch>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Vacation</AttributeValue>
          <SubjectAttributeDesignator AttributeId="State" DataType="string"/>
        </SubjectMatch>
      </Subject>
      <Subject>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Employee</AttributeValue>
          <SubjectAttributeDesignator AttributeId="Profile" DataType="string"/>
        </SubjectMatch>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Vacation</AttributeValue>
          <SubjectAttributeDesignator AttributeId="State" DataType="string"/>
        </SubjectMatch>
      </Subject>
      <Subject>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Manager</AttributeValue>
          <SubjectAttributeDesignator AttributeId="Profile" DataType="string"/>
        </SubjectMatch>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Vacation</AttributeValue>
          <SubjectAttributeDesignator AttributeId="State" DataType="string"/>
        </SubjectMatch>
            </Subject>
    </Subjects>
        <Resources>
          <Resource>
        <ResourceMatch MatchId="string-equal">
          <AttributeValue DataType="string">Room</AttributeValue>
          <ResourceAttributeDesignator AttributeId="ResourceId" DataType="string"/>
        </ResourceMatch>
          </Resource>
    </Resources>
    <Actions>
```

```
    <AnyAction/>
  </Actions>
</Target>
<Rule RuleId="Rule12" Effect="Deny">
  <Description>
    Doormen and employees are not allowed to access any private rooms while in vacation
  </Description>
  <Target>
    <Subjects>
       <Subject>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Doorman</AttributeValue>
          <SubjectAttributeDesignator AttributeId="Profile" DataType="string"/>
        </SubjectMatch>
      </Subject>
      <Subject>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Employee</AttributeValue>
          <SubjectAttributeDesignator AttributeId="Profile" DataType="string"/>
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="string-equal">
          <AttributeValue DataType="string">Private</AttributeValue>
          <ResourceAttributeDesignator AttributeId="Privacy" DataType="string"/>
        </ResourceMatch>
      </Resource>
    </Resources>
          <Actions>
      <AnyAction/>
    </Actions>
  </Target>
</Rule>
<Rule RuleId="Rule13" Effect="Deny">
  <Description>
    Doormen and employees are not allowed to lock or unlock any public rooms while in vacation
  </Description>
  <Target>
    <Subjects>
       <Subject>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Doorman</AttributeValue>
          <SubjectAttributeDesignator AttributeId="Profile" DataType="string"/>
        </SubjectMatch>
      </Subject>
      <Subject>
        <SubjectMatch MatchId="string-equal">
          <AttributeValue DataType="string">Employee</AttributeValue>
          <SubjectAttributeDesignator AttributeId="Profile" DataType="string"/>
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch MatchId="string-equal">
          <AttributeValue DataType="string">Public</AttributeValue>
          <ResourceAttributeDesignator AttributeId="Privacy" DataType="string"/>
        </ResourceMatch>
      </Resource>
    </Resources>
          <Actions>
      <Action>
        <ActionMatch MatchId="string-equal">
          <AttributeValue DataType="string">Lock</AttributeValue>
          <ActionAttributeDesignator AttributeId="ActionId" DataType="string"/>
        </ActionMatch>
              </Action>
              <Action>
        <ActionMatch MatchId="string-equal">
          <AttributeValue DataType="string">Unlock</AttributeValue>
          <ActionAttributeDesignator AttributeId="ActionId" DataType="string"/>
        </ActionMatch>
              </Action>
    </Actions>
  </Target>
</Rule>
<Rule RuleId="Rule14" Effect="Deny">
```

```
    <Description>
      Visitor are not allowed to visit private rooms
    </Description>
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch MatchId="string-equal">
            <AttributeValue DataType="string">Visitor</AttributeValue>
            <SubjectAttributeDesignator AttributeId="Profile" DataType="string"/>
          </SubjectMatch>
                </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <ResourceMatch MatchId="string-equal">
            <AttributeValue DataType="string">Private</AttributeValue>
            <ResourceAttributeDesignator AttributeId="Privacy" DataType="string"/>
          </ResourceMatch>
        </Resource>
      </Resources>
            <Actions>
        <Action>
          <ActionMatch MatchId="string-equal">
            <AttributeValue DataType="string">Visit</AttributeValue>
            <ActionAttributeDesignator AttributeId="ActionId" DataType="string"/>
          </ActionMatch>
                </Action>
      </Actions>
    </Target>
  </Rule>
</Policy>
```

Annexe C

Modèle ALLOY complet

Les spécifications ci-dessous sont les spécifications complètes des politiques de contrôle d'accès présentées dans la section 7.2.

```
module final
/* Element signature */
abstract sig Element {
  attributes :Attribute -> Value
}{
  attributes in values
  not (no attributes)
}
/* Subject, resource, and action inherit form Element */
sig Subject, Resource, Action extends Element{}
/* Request signature */
sig Request {
  subject : one Subject,
  resource : one Resource,
  action : one Action
}
/* Target signature */
abstract sig Target {
  subjects : set Subject,
  resources : set Resource,
  actions : set Action
}
/* Value signature */
abstract sig Value {}
/* Attribute signature */
abstract sig Attribute { values : set Value }
/* Rule's effect signature */
abstract sig Effect {}
/* Possible effects */
one sig Permit, Deny, NotApplicable, Indeterminate extends Effect {}
/* Rule signature */
abstract sig Rule {
  ruleTarget : one Target,
  ruleEffect : one Effect
}
/* Policy signature */
abstract sig Policy {
  policyTarget : one Target,
  rules : set Rule,
  ruleCombiningAlgo : one CombiningAlgo
}
/* Policy set signature */
abstract sig PolicySet {
  policySetTarget : one Target,
  policies : set Policy,
  policySets : set PolicySet,
  policyCombiningAlgo : one CombiningAlgo
}
/* Combining algorithm signature */
abstract sig CombiningAlgo {}
```

```
/* Possible combining algorithms */
one sig PermitOverrides, DenyOverrides extends CombiningAlgo {}
/* Predicate that verifies that a target matches a request */
pred targetMatch (t : Target, r : Request) {
  {t.subjects = none || some e: t.subjects | elementMatch(r.subject, e)}
  {t.resources = none || some e: t.resources | elementMatch(r.resource, e)}
  {t.actions = none || some e: t.actions | elementMatch(r.action, e)}
}
/* Predicate that verifies that an element matches another */
pred elementMatch(e1: Element, e2 : Element){
  e2 = none ||
  all a2 : e2.attributes.Value {
    some a1 : e1.attributes.Value
       | a1=a2 and a2.(e2.attributes) in a1.(e1.attributes)
  }
}
/* Function that generates a rule's response against a request */
fun ruleResponse (r : Rule, req : Request) : Effect {
  if targetMatch(r.ruleTarget, req) then r.ruleEffect
  else NotApplicable
}
/* Predicate that verifies that a policy contains a rule returning a deny */
pred existDeny(p : Policy, q : Request) {
  some r : p.rules | ruleResponse(r, q) = Deny
}
/* Predicate that verifies that a policy contains a rule returning a permit */
pred existPermit(p : Policy, q : Request) {
  some r : p.rules | ruleResponse(r, q) = Permit
}
/* Function that generates a policy's response against a request */
fun policyResponse (p : Policy, req : Request) : Effect {
  if targetMatch(p.policyTarget, req)
  then ruleCombinedResponse(p, req)
  else NotApplicable
}
/* Function that generates a policy's response against a request according to the combining algorithm*/
fun ruleCombinedResponse (p : Policy, req : Request) : Effect {
  if p.ruleCombiningAlgo = PermitOverrides then rulePermitOverrides(p, req)
  else if p.ruleCombiningAlgo = DenyOverrides then ruleDenyOverrides(p, req)
  else Indeterminate
}
/* Permit-Overrides */
fun rulePermitOverrides ( p : Policy, req : Request) : Effect {
  if existPermit(p,req) then Permit
  else if existDeny(p,req) then Deny
  else NotApplicable
}
/* Deny-Overrides */
fun ruleDenyOverrides ( p : Policy, req : Request) : Effect {
  if existDeny(p,req) then Deny
  else if existPermit(p,req) then Permit
  else NotApplicable
}
/* Predicates that verifies if a target is included into another */
pred targetIsIncluded(t1, t2:Target){
  all s1 : t1.subjects
    {some s2 : t2.subjects | elementMatch(s1,s2)}
  all r1 : t1.resources
    {some r2 : t2.resources | elementMatch(r1,r2)}
  all a1 : t1.actions
    {some a2 : t2.actions | elementMatch(a1,a2)}
}
/* Predicate that verifies whether two elements intersect */
pred inter(e1, e2 : Element){
  elementMatch(e1, e2) or elementMatch(e2, e1)
}
/* Predicate that verifies whether two elements are equal */
pred eq(e1, e2 : Element){
  elementMatch(e1, e2) and elementMatch(e2, e1)
}
/* Predicates that verifies whether two targets intersect */
pred targetIntersect(t1, t2 : Target){
  some s1 : t1.subjects, s2 : t2.subjects | inter(s1,s2)
  some r1 : t1.resources, r2 : t2.resources | inter(r1,r2)
  some a1 : t1.actions, a2 : t2.actions | inter(a1,a2)
```

```
}
/* Predicates that verifies whether two targets are disjunctive */
pred targetDisj(t1, t2 : Target){
  no s1 : t1.subjects, s2 : t2.subjects | inter(s1,s2)
  no r1 : t1.resources, r2 : t2.resources | inter(r1,r2)
  no a1 : t1.actions, a2 : t2.actions | inter(a1,a2)
}
/* Predicates that verifies whether two targets are equal */
pred tagetEqual(t1, t2 : Target){
  all s1 : t1.subjects, s2 : t2.subjects | eq(s1,s2)
  all r1 : t1.resources, r2 : t2.resources | eq(r1,r2)
  all a1 : t1.actions, a2 : t2.actions | eq(a1,a2)
}
/* Predicate that verifies whether a policy contains redundancy between rules */
pred redundantRules(p : Policy){
  some disj r1, r2 : p.rules {
    (targetIsIncluded(r1.ruleTarget, r2.ruleTarget)
    or
    targetIntersect(r1.ruleTarget, r2.ruleTarget)
    or
    tagetEqual(r1.ruleTarget, r2.ruleTarget))
    r1.ruleEffect = r2.ruleEffect
  }
}
/* Predicate that verifies whether a policy contains conflict between rules */
pred conflictingRules(p : Policy){
  some disj r1, r2 : p.rules {
    (targetIsIncluded(r1.ruleTarget, r2.ruleTarget)
    or
    targetIntersect(r1.ruleTarget, r2.ruleTarget)
    or
    tagetEqual(r1.ruleTarget, r2.ruleTarget))
    r1.ruleEffect != r2.ruleEffect
  }
}
/* Predicate that verifies whether a policy contains a useless rule */
pred uselessRule(p : Policy){
  some r : p.rules | targetDisj(p.policyTarget, r.ruleTarget)
}
/* Predicate that verifies whether a policy set contains a useless policy */
pred uselessPolicy(s : PolicySet){
  some p : s.policies | targetDisj(s.policySetTarget, p.policyTarget)
}
/* Predicate that verifies whether a policy set contains a useless policy set */
pred uselessPolicySet(s : PolicySet){
  some p : s.policySets | targetDisj(s.policySetTarget, p.policySetTarget)
}
/* Predicate that verifies if a request generates two conflicting rules within a policy */
pred InconsistentRules (p : Policy, req : Request) {
  targetMatch(p.policyTarget, req)
  some r : p.rules | ruleResponse(r, req) = Permit
  some r : p.rules | ruleResponse(r, req) = Deny
}
/* Predicate that verifies if a request generates two conflicting policies within a policy set*/
pred InconsistentPolicies (s : PolicySet, req : Request) {
  targetMatch(s.policySetTarget, req)
  some p : s.policies | policyResponse(p, req) = Permit
  some p : s.policies | policyResponse(p, req) = Deny
}
/* Assertion for non existance of conflicting rules */
assert RulesConsistency {
no p : Policy, req : Request | InconsistentRules(p,req)
}
/* Assertion for non existance of conflicting policies */
assert PoliciesConsistency {
no s : PolicySet, req : Request | InconsistentPolicies(s,req)
}
/* Predicates that verifies whether a policy returns always a permit */
pred positivePolicy(p : Policy){
  all q : Request | policyResponse(p, q) = Permit
}
/* Predicates that verifies whether a policy returns always a deny */
pred negativePolicy(p : Policy){
  all q : Request | policyResponse(p, q) = Deny
}
/* Only one requested action */
```

```
fact {
  one Request.action.attributes
}
run positivePolicy for 18 but 1 Request
check RulesConsistency for 18 but 1 Request
run negativePolicy for 18 but 1 Request
run uselessRule for 18 but 1 Request
run redundantRules for 18 but 1 Request
/* Domain contraints (Manual) */
fact {
  all q : Request  | one SubjectId.(q.subject.attributes)
  all q : Request  | one Profile.(q.subject.attributes)
  all q : Request | one Privacy.(q.resource.attributes)
  all q : Request | one ResourceId.(q.resource.attributes)
}
one sig ResourceId extends Attribute {}{
  values = Room
}
one sig Profile extends Attribute {}{
  values = Doorman + Manager + Employee + Visitor
}
one sig ActionId extends Attribute {}{
  values = Visit + Unlock + Lock
}
one sig Privacy extends Attribute {}{
  values = Public + Private
}
one sig SubjectId extends Attribute {}{
  values = EMP01
}
one sig State extends Attribute {}{
  values = Vacation
}
one sig Room extends Value{}
one sig Doorman, Manager, Employee, Visitor extends Value{}
one sig Visit, Unlock, Lock extends Value{}
one sig Public, Private extends Value{}
one sig EMP01 extends Value{}
one sig Vacation extends Value{}
fact {
Subject.attributes.Value =Profile + SubjectId + State
Resource.attributes.Value =ResourceId + Privacy
Action.attributes.Value =ActionId
}
one sig Subject0 extends Subject{}{
  attributes = Profile->Doorman
}
one sig Subject1 extends Subject{}{
  attributes = Profile->Manager
}
one sig Subject2 extends Subject{}{
  attributes = Profile->Employee
}
one sig Subject3 extends Subject{}{
  attributes = Profile->Visitor
}
one sig Subject4 extends Subject{}{
  attributes = SubjectId->EMP01
}
one sig Subject5 extends Subject{}{
  attributes = Profile->Doorman + State->Vacation
}
one sig Subject6 extends Subject{}{
  attributes = Profile->Employee + State->Vacation
}
one sig Subject7 extends Subject{}{
  attributes = Profile->Manager + State->Vacation
}
one sig Resource0 extends Resource{}{
  attributes = ResourceId->Room
}
one sig Resource1 extends Resource{}{
  attributes = Privacy->Public
}
one sig Resource2 extends Resource{}{
  attributes = Privacy->Private
}
```

```
one sig Action0 extends Action{}{
  attributes =  ActionId->Visit
}
one sig Action1 extends Action{}{
  attributes =  ActionId->Unlock
}
one sig Action2 extends Action{}{
  attributes =  ActionId->Lock
}
one sig Target0 extends Target {}{
  subjects =   none
  resources =  none
  actions =  none
}
one sig Target1 extends Target {}{
  subjects =   none
  resources = Resource0
  actions =  none
}
one sig Target2 extends Target {}{
  subjects =  Subject0
  resources =  none
  actions =  none
}
one sig Target3 extends Target {}{
  subjects =  Subject1
  resources =  none
  actions = Action0 + Action1
}
one sig Target4 extends Target {}{
  subjects =  Subject2
  resources =  none
  actions = Action0
}
one sig Target5 extends Target {}{
  subjects =  Subject2
  resources = Resource1
  actions = Action1
}
one sig Target6 extends Target {}{
  subjects =  Subject3
  resources = Resource1
  actions = Action0
}
one sig Target7 extends Target {}{
  subjects =  Subject3 + Subject2 + Subject1
  resources =  none
  actions = Action2
}
one sig Target8 extends Target {}{
  subjects =  Subject2
  resources = Resource2
  actions = Action1
}
one sig Target9 extends Target {}{
  subjects =  Subject3
  resources = Resource2
  actions = Action0
}
one sig Target10 extends Target {}{
  subjects =  Subject3
  resources =  none
  actions = Action2 + Action1
}
one sig Target11 extends Target {}{
  subjects =  Subject4
  resources =  none
  actions =  none
}
one sig Target12 extends Target {}{
  subjects =  Subject5 + Subject6 + Subject7
  resources = Resource0
  actions =  none
}
one sig Target13 extends Target {}{
  subjects =  Subject0 + Subject2
  resources = Resource2
  actions =  none
}
one sig Target14 extends Target {}{
  subjects =  Subject0 + Subject2
  resources = Resource1
  actions = Action2 + Action1
}
```

```
one sig Rule1 extends Rule {}{
  ruleTarget = Target2
  ruleEffect = Permit
}
one sig Rule2 extends Rule {}{
  ruleTarget = Target3
  ruleEffect = Permit
}
one sig Rule3 extends Rule {}{
  ruleTarget = Target4
  ruleEffect = Permit
}
one sig Rule4 extends Rule {}{
  ruleTarget = Target5
  ruleEffect = Permit
}
one sig Rule5 extends Rule {}{
  ruleTarget = Target6
  ruleEffect = Permit
}
one sig Rule6 extends Rule {}{
  ruleTarget = Target7
  ruleEffect = Deny
}
one sig Rule7 extends Rule {}{
  ruleTarget = Target8
  ruleEffect = Deny
}
one sig Rule8 extends Rule {}{
  ruleTarget = Target9
  ruleEffect = Deny
}
one sig Rule9 extends Rule {}{
  ruleTarget = Target10
  ruleEffect = Deny
}
one sig Rule10 extends Rule {}{
  ruleTarget = Target11
  ruleEffect = Deny
}
one sig Rule11 extends Rule {}{
  ruleTarget = Target13
  ruleEffect = Deny
}
one sig Rule12 extends Rule {}{
  ruleTarget = Target14
  ruleEffect = Deny
}
one sig Rule13 extends Rule {}{
  ruleTarget = Target9
  ruleEffect = Deny
}
one sig Default extends Policy {}{
  policyTarget = Target1
  ruleCombiningAlgo = PermitOverrides
  rules = Rule1 + Rule2 + Rule3 + Rule4 + Rule5 + Rule6 + Rule7 + Rule8 + Rule9 + Rule10
}
one sig VacationPolicy extends Policy {}{
  policyTarget = Target12
  ruleCombiningAlgo = DenyOverrides
  rules = Rule11 + Rule12 + Rule13
}
one sig AccessRooms extends PolicySet {}{
  policySetTarget = Target0
  policyCombiningAlgo = DenyOverrides
  policies = Default + VacationPolicy
  policySets = none
}
```

Bibliographie

[1] Ahmed, T., and Tripathi, A. R. Static verification of security requirements in role based CSCW systems. In *SACMAT '03 : Proceedings of the eighth ACM symposium on Access control models and technologies* (New York, NY, USA, 2003), ACM Press, pp. 196–203.

[2] Barth, A., Mitchell, J. C., and Rosenstein, J. Conflict and combination in privacy policy languages. In *WPES '04 : Proceedings of the 2004 ACM workshop on Privacy in the electronic society* (New York, NY, USA, 2004), ACM Press, pp. 45–46.

[3] Benferhat, S., Baida, R. E., and Cuppens, F. A stratification-based approach for handling conflicts in access control. In *SACMAT '03 : Proceedings of the eighth ACM symposium on Access control models and technologies* (New York, NY, USA, 2003), ACM Press, pp. 189–195.

[4] Benferhat, S., Dubois, D., and Prade, H. Representing Default Rules in Possibilistic Logic. In *KR'92. Principles of Knowledge Representation and Reasoning : Proceedings of the Third International Conference*, B. Nebel, C. Rich, and W. Swartout, Eds. Morgan Kaufmann, San Mateo, California, 1992, pp. 673–684.

[5] Bertino, E., Catania, B., Ferrari, E., and Perlasca, P. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur. 6*, 1 (2003), 71–127.

[6] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. Ponder : A Language for Specifying Security and Management Policies for Distributed Systems. The Language Specification - Version 2.3. Imperial College Research Report DoC 2000/1, Oct. 2000.
http ://www.doc.ic.ac.uk/ ncd/policies/files/ accédé en Avril 2005.

[7] Damianou, N., Dulay, N., Lupu, E., and Sloman, M. The Ponder Policy Specification Language. *Lecture Notes in Computer Science 1995* (2001), 18.

[8] Damianou, N., Dulay, N., Lupu, E., Sloman, M., and Tonouchi, T. Tools for Domain-based Policy Management of Distributed Systems. In *IEEE/IFIP Network Operations and Management Symposium (NOMS2002)* (Apr. 2002), pp. 213–218.

[9] Damianou, N. C. *A Policy Framework for Management of Distributed Systems.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, Department of Computing, 2002.

[10] Downs, D. D., Rub, J. R., Kung, K. C., and Jordan, C. S. Issues in Discretionary Access Control. In *1985 IEEE Symposium on Security and Privacy* (1985), pp. 96–109.

[11] Edwards, J., Jackson, D., and Torlak, E. A type system for object models. In *SIGSOFT '04/FSE-12 : Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* (2004), ACM Press, pp. 189–199.

[12] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., and Chandramouli, R. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur. 4*, 3 (2001), 224–274.

[13] Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tschantz, M. C. Margrave : An api for XACML policy verification and change analysis, 2005. http ://www.cs.brown.edu/research/plt/software/margrave/, accédé en mai 2005.

[14] Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tschantz, M. C. Verification and Change-Impact Analysis of Access-Control Policies. In *International Conference on Software Engineering* (2005), ACM.

[15] Fitting, M. *First-order logic and automated theorem proving*, second ed. Springer-Verlag, 1996.

[16] Godik, S., and Moses, T. extensible access control markup language (XACML) version 1.1, Aug. 2003. http ://www.oasis-open.org/committees/xacml/repository/cs-xacml-specification-1.1.pdf, accédé en février 2005.

[17] Group, O. M. UML 2.0 OCL Specification, Oct. 2003. http ://www.omg.org/docs/ptc/03-10-14.pdf, accédé en mai 2005.

[18] Guelev, D. P. Prolog code supporting "Model-checking access control policies", Nov. 2003.
http ://www.cs.bham.ac.uk/ dpg/mcacp/aclsimp.pl, accédé en avril 2005.

[19] Guelev, D. P., Ryan, M., and Schobbens, P. Y. Model-Checking Access Control Policies. In *Seventh Information Security Conference (ISC 2004). Lecture Notes in Computer Science* (Sept. 2004), Springer-Verlag, pp. 219–230.

[20] Holzner, S. *XSLT par la pratique.* Éditions Eyrolles, 2002.

[21] Huth, M., and Ryan, M. *Logic in Computer Science : Modelling and reasonning about systems.* Cambridge University Press, 2001, ch. 2.

[22] Jackson, D. ALLOY Home Page.
http ://alloy.mit.edu/, accédé en mai 2005.

[23] Jackson, D. Tutorial for ALLOY 3.0.
http ://web.mit.edu/ rseater/www/tutorial3/alloy-tutorial.html, accédé en Janvier 2005.

[24] Jackson, D. Automating first-order relational logic. In *SIGSOFT '00/FSE-8 : Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering* (2000), ACM Press, pp. 130–139.

[25] Jackson, D. ALLOY : a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11*, 2 (2002), 256–290.

[26] Jackson, D. *Micromodels of Software : Lightweight Modelling and Analysis with ALLOY*, Feb. 2002.

[27] Jackson, D. *ALLOY 3.0 Reference Manual*, May 2004.

[28] Jackson, D., Schechter, I., and Shlyahter, H. Alcoa : the alloy constraint analyzer. In *ICSE '00 : Proceedings of the 22nd international conference on Software engineering* (2000), ACM Press, pp. 730–733.

[29] Jackson, D., Shlyakhter, I., and Sridharan, M. A micromodularity mechanism. In *ESEC/FSE-9 : Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering* (2001), ACM Press, pp. 62–73.

[30] Jajodia, S., Samarati, P., Sapino, M. L., and Subrahmanian, V. S. Flexible support for multiple access control policies. *ACM Trans. Database Syst. 26*, 2 (2001), 214–260.

[31] Kamoda, H., Yamaoka, M., Matsuda, S., Broda, K., and Sloman, M. Policy conflict analysis using free variable tableaux for access control in web services evironments. In *Proceedings of the 14th International World Wide Web Conference on Policy Management for the Web* (2005), pp. 5–12.

[32] Koch, M., Mancini, L., and Parisi-Presicce, F. Graph Transformations for the Specification of Access Control Policies. *Electronic Notes in Theoretical Computer Science 15* (May 2002), 1–11.
http ://www.sciencedirect.com/science/article/B6WJ0-4F8TW25-1/2/6e420fedd70b1f06489a5dddedb3f030 accédé en avril 2005.

[33] Koch, M., Mancini, L., and Parisi-Presicce, F. Graph-based specification of access control policies. *Journal of Computer and System Sciences* (2004).
http ://www.sciencedirect.com/science/article/B6WJ0-4F8TW25-1/2/6e420fedd70b1f06489a5dddedb3f030 accédé en avril 2005.

[34] Koch, M., Mancini, L. V., and Parisi-Presicce, F. A graph-based formalism for RBAC. *ACM Trans. Inf. Syst. Secur. 5*, 3 (2002), 332–365.

[35] Koch, M., Mancini, L. V., and Parisi-Presicce, F. Conflict Detection and Resolution in Access Control Policy Specifications. *Lecture Notes in Computer Science 2303* (Jan. 2002), 223.

[36] Labs, B. SPIN Home page.
http ://spinroot.com/spin/whatispin.html, accédé en avril 2005.

[37] Li, N., and Tripunitara, M. V. On safety in discretionary access control. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)* (2005), pp. 96–109.

[38] Lorch, M., Proctor, S., Lepro, R., Kafura, D., and Shah, S. First experiences using XACML for access control in distributed systems. In *XMLSEC '03 : Proceedings of the 2003 ACM workshop on XML security* (2003), ACM Press, pp. 25–37.

[39] Lupu, E., and Sloman, M. Conflict Analysis for Management Policies. In *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network management IM'97, San Diego, CA* (1997).

[40] Lupu, E. C., and Sloman, M. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering 25*, 6 (Nov. 1999), 852–869.

[41] Microsystems, S. Java Technology.
http ://java.sun.com/, accédé en décembre 2005.

[42] Moses, T. extensible access control markup language (XACML) version 2.0. Tech. rep., OASIS Committee Draft, Dec. 2004.
http ://docs.oasis-open.org/xacml/access_control-xacml-2_0-core-spec-cd-04.pdf, accédé en février 2005.

[43] OASIS. Context Schema.
http ://www.oasis-open.org/committees/download.php/919/cs-xacml-schema-context-01.xsd, accédé en mai 2005.

[44] OASIS. OASIS eXtensible Access Control Markup Language (XACML) TC.
http ://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=xacml, accédé en mai 2005.

[45] OASIS. Policy Schema.
http ://www.oasis-open.org/committees/download.php/915/cs-xacml-schema-policy-01.xsd, accédé en mai 2005.

[46] Osborn, S. Mandatory access control and role-based access control revisited. In *RBAC '97 : Proceedings of the second ACM workshop on Role-based access control* (New York, NY, USA, 1997), ACM Press, pp. 31–40.

[47] Rensink, A. The GROOVE Simulator : AăTool for State Space Generation, Lecture Notes in Computer Science. *Lecture Notes in Computer Science 3062* (July 2004), 479–485.

[48] Sandhu, R. S., and Samarati, P. Access Control : Principles and Practice. *IEEE Communications Magazine 32*, 9 (1994), 40–48.

[49] Schaad, A., and Moffett, J. A Lightweight Approach to Specification and Analysis of Role-based Access Control Extensions. In *SACMAT '02 : Proceedings of the 7th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2002), ACM Press.

[50] Schunter, M., and Powers, C. The Enterprise Privacy Authorization Language (EPAL 1.1), 2003.
http ://www.zurich.ibm.com/security/enterprise-privacy/epal/, accédé en avril 2005.

[51] Spivey, J. M. *The Z Notation : A Reference Manual*, second ed. Prentice Hall International, 1992.

[52] Taentzer, G. AGG : A Tool Environment for Algebraic Graph Transformation. *Lecture Notes in Computer Science 1779* (Jan. 2000), 481.

[53] The European Telecommunications Standards Institute (ETSI). Testing and Test Control Notation 3 (TTCN-3).
http ://www.ttcn-3.org/, accédé en décembre 2005.

[54] W3C. Document Object Model (DOM).
http ://www.w3.org/DOM/, accédé en décembre 2005.

[55] W3C. XSL Transformations (XSLT).
http ://www.w3.org/TR/xslt, accédé en mai 2005.

[56] Zhang, N. Evaluating Access Control Polices Through Model Checking, Apr. 2005.
http ://www.cs.bham.ac.uk/ nxz/ acc, accédé en avril 2005.

[57] Zhang, N., Ryan, M., and Guelev, D. P. Synthesising verified access control systems in XACML. In *FMSE '04 : Proceedings of the 2004 ACM workshop on Formal methods in security engineering* (2004), ACM Press, pp. 56–65.

[58] Zhao, Y., and Parisi-Presicce, F. Policy Analysis and Verification by Graph Transformation Tools. In *Proceedings of Second International Conference on Graph Transformation (ICGT 2004)* (Oct. 2004).