UNIVERSITÉ DU QUÉBEC À MONTRÉAL

VÉRIFICATION DES POLITIQUES XACML

AVEC LE LANGAGE EVENT-B

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

MOHAMMED ERRACHID

MARS 2011

REMERCIEMENTS

Avant tout, je tiens à remercier sincèrement mon directeur de recherche, le professeur Aziz Salah, qui m'a proposé un sujet intéressant et qui m'a appuyé tout au long de la réalisation de ce mémoire.

J'offre également ma gratitude à mes professeurs du département informatique de l'UQAM, qui m'ont communiqués leur savoir et leur savoir-faire durant ma scolarité à la maîtrise.

Je n'oublie pas mes parents, ma femme et mes enfants qui m'ont soutenus par leurs prières, leurs encouragements et qui m'ont accompagné avec patience pendant plusieurs mois consacrés à ce sujet.

Enfin je remercie les autres personnes qui ont contribué de près ou de loin à la réussite de mes études.

TABLE DES MATIÈRES

Liste des tableaux

Liste des figures

ACRONYMES ET ABRÉVIATIONS

XACML	*eXtensible Access Control Markup Language*
PDP	*Policy Decision Point*
PEP	*Policy enforcement Point*
PIP	*Policy Information Point*
MTDD	*Multi-Terminal Diagrams Decision*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*
XSD	*XML Schema Definition*
CSP	*Communicating Sequential Processes*
MIT	*Massachusetts Institute of Technology*
OP	*Obligation de Preuve*

RÉSUMÉ

Les politiques permettent de définir les règles de la sécurité et de la gestion des différents composants du système. Cela implique l'emploi d'un langage pour exprimer les règles d'affaires et les règles non fonctionnelles, et de donner aux utilisateurs la possibilité de tester et de corriger les politiques. Plusieurs langages tels que **XACML, Rei ou PONDER,** sont utilisés pour exprimer les politiques par rapport aux objectifs du système d'information. Ces langages peuvent définir plusieurs règles et politiques, mais la plupart de ces langages ne donnent pas de mécanisme pour tester et vérifier la présence des conflits et de l'incohérence entre les politiques du système.

Ce mémoire vise la vérification des politiques de contrôle d'accès. Notre approche consiste à traduire les politiques XACML sous forme d'un ensemble de machines abstraites de la méthode B. Nous exprimons aussi les propriétés à vérifier par des formules logiques. L'approche offre aux utilisateurs des moyens pour vérifier les politiques afin de s'assurer que les règles expriment bien les objectifs régissant le comportement et les interactions des systèmes gérés.

Dans la première phase, les composantes des politiques XACML ont été exprimées avec des expressions formelles basées sur la logique du premier ordre. Par la suite, les outils développés pour la méthode B, comme le langage Event-B sous la plate forme Rodin, ont été utilisés pour vérifier les règles des politiques par rapport à un ensemble de propriétés que nous avons définies.

Notre approche est plus flexible et permet aux utilisateurs de tester et de vérifier les règles avant l'implémentation de ces politiques. Une telle vérification est fondée sur les preuves avec logique du premier ordre, où des propriétés importantes de la politique peuvent être énoncées et prouvées.

Mots clés : Politique, XACML, Méthode formelle, Event-B, Vérification.

INTRODUCTION

Une politique est un ensemble de règles permettant aux utilisateurs d'obtenir une décision selon un ensemble des paramètres. Le terme « politique » en informatique est utilisé de façon interchangeable avec le mot « régulation ». Ce concept a été utilisé pour décrire nettement la règlementation, les objectifs généraux de la gestion d'un système ou pour la prescription d'un plan d'action.

Les politiques permettent de définir les objectifs sous forme de règles exprimées avec un langage informatique. Ces règles, représentées par des expressions booléennes énonçant les conditions à évaluer, expriment les contraintes fonctionnelles des systèmes, les droits d'accès et les contraintes non fonctionnelles. Les politiques, sont complexes et difficiles à utiliser surtout dans les systèmes distribués. Cette complexité favorise l'existence de conflits et d'incohérences qui risquent de menacer la qualité et la sécurité de tout le système informatique, rendant ainsi difficile la gestion et la mise à jour des politiques.

Différents langages sont utilisés en vue d'exprimer les politiques, ayant chacun sa propre spécification [1] [2]. Dans ce mémoire nous visons l'analyse et la vérification des politiques exprimées dans le langage XACML. Notre objectif est de proposer une approche permettant la découverte des conflits dans les politiques par la vérification de certaines propriétés (comme la non redondance).

XACML est un langage de politique de contrôle d'accès basé sur les attributs. Parmi ses caractéristiques, nous en citons:

- Les règles XACML sont écrites en XML selon une norme ouverte bien documentée sous la responsabilité d'un organisme reconnu (OASIS) ;
- L'élaboration et le traitement des politiques XACML se fait de manière indépendante de la plateforme technologique des applications clientes ;

- Les changements et l'évolution des applications n'affectent en rien les politiques XACML puisqu'elles sont gérées et traitées en dehors de celles-ci ;
- La spécification XACML est extensible par la définition de nouveaux types de données et de fonctions et par l'ajout d'algorithmes de combinaison des règles et des politiques.

Notre travail de recherche s'inscrit dans le cadre des méthodes formelles utilisant la méthode B et plus précisément Event-B[21] . Celui-ci offre un langage basé sur la théorie des ensembles et la logique des prédicats. Event-B est muni de techniques de raffinement et permet la vérification des propriétés des systèmes par l'intermédiaire de preuves mathématiques. Event-B est implémenté par la plateforme Rodin qui offre un support pour les raffinements et les preuves.

Notre objectif étant la vérification des politiques XACML en utilisant la plateforme Rodin, notre approche est divisée en deux étapes. La première étape consiste à développer un outil pour traduire les politiques XACML sous vérification en un modèle Event-B équivalent. La deuxième étape c'est la vérification des propriétés. Pour cela nous avons conçu un modèle générique en Event-B qui décrit les propriétés à vérifier communes à toutes les politiques XACML pour la découverte de certains interactions et conflits. D'autres propriétés spécifiques aux politiques sous vérification peuvent aussi être vérifiées par instanciation de requêtes spécifiques. Toutes les vérifications des propriétés des politiques XACML se font la plateforme Rodin sous forme de preuves d'obligations.

Ce mémoire est composé de cinq chapitres. Le chapitre I présente le principe, l'architecture de l'environnement et le diagramme de flux du langage XACML. Nous décrirons par la suite la syntaxe du langage à travers un exemple.

Au chapitre II nous présenterons l'état de l'art relatif aux méthodes de vérification des politiques et aux techniques d'analyse. Nous présentons aussi les travaux sur la transformation des politiques XACML.

Au chapitre III nous décrirons les principes du langage Event-B et les techniques formelles de la méthode B. Nous détaillons aussi la technique de preuve d'obligation qui permet de vérifier le modèle formel.

Le chapitre IV présente le modèle formel des politiques XACML avec le langage Event-B, et décrit la structure des contextes et des machines pour l'évaluation d'une requête XACML.

Enfin, dans le chapitre V nous décrirons les étapes de la transformation des politiques de contrôle d'accès XACML vers un modèle formel décrit avec le langage Event-B. Nous présenterons ensuite l'ensemble des propriétés que nous avons définies en vue de valider les politiques de contrôle d'accès XACML définit dans un exemple d'application.

CHAPITRE I : LE LANGAGE XACML

XACML (*eXtensible Access Control Markup Language*) [5] est un langage de politique permettant d'exprimer les règles utilisées pour le contrôle d'accès. La spécification du langage XACML est définie par un ensemble de schémas XML, qui décrit la syntaxe des règles et des politiques d'accès. Le langage XACML vise à atteindre plusieurs objectifs comme :

- Assurer une protection efficace pour les ressources du système ;
- Permettre de concevoir un système indépendant de la plate-forme utilisée ;
- Permettre d'intégrer les politiques XACML dans des applications déjà existantes.

Dans ce chapitre, nous décrivons le langage XACML et nous présentons les étapes de l'évaluation d'une requête par rapport aux politiques XACML, ainsi que l'architecture du langage XACML.

1.1 Description du langage XACML.

Une politique en XACML (figure 1.1) est composée d'une cible, d'une ou plusieurs règles et d'un algorithme de combinaison des règles.

```
<Policy PolicyId="deny-test"
RuleCombiningAlgId= " rule-combining-algorithm:first-applicable"
<Description> structure de la politique </Description>
<Target>
<Subjects> ... </Subjects>
<Resources>... </Resources>
<Actions> ... </Actions>
</Target>
<Rule/> ... </Rule> # régle 1
<Rule/> ... </Rule> # régle 2
                                        </Policy>
```

Figure 1.1 Exemple de la structure d'une politique XACML

La décision de la politique est calculée en utilisant des algorithmes de combinaison sur les décisions des règles.

La figure 1.2 présente un diagramme de classe qui décrit la structure de la politique XACML. Un ensemble de politiques « *PolicySet* » doit spécifier une cible et un algorithme de combinaison qui permet de combiner les décisions issues des politiques. Une politique « *Policy* » doit aussi avoir une cible ainsi qu'un algorithme de combinaison, qui permet de choisir une décision parmi les règles de la politique. Enfin, une règle spécifie sa cible, sa condition et son effet.

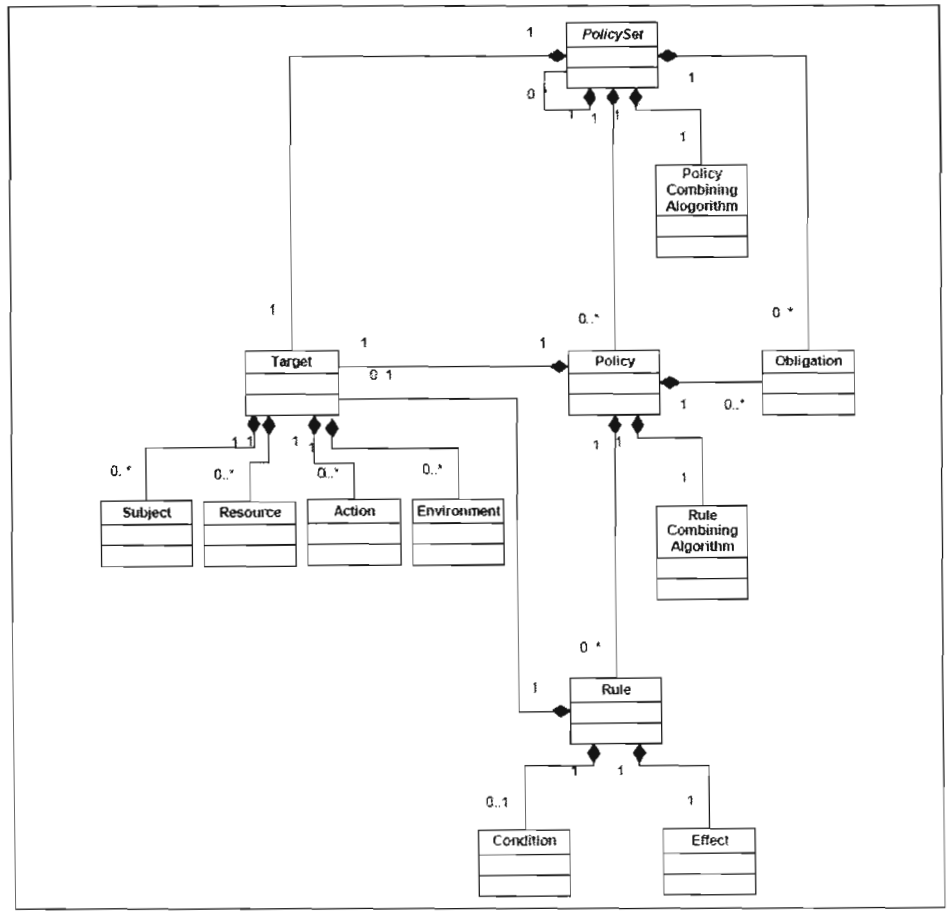

Figure 1.2 Modèle du langage XACML [5]

1.1.1 La cible

XACML a introduit la notion cible « *Target* » afin d'identifier les règles et les politiques qui concernent une requête. La cible est composée **d'attributs** qui décrivent le sujet, les ressources, les actions et l' l'environnement :

- **Le sujet** décrit les attributs de l'utilisateur qui a fait une demande d'accès ;
- **La ressource** décrit les attributs de l'objet auquel l'accès est demandé ;
- **L'action** représente les attributs qui décrivent les mesures que le sujet veut prendre sur la ressource demandée ;
- **L'environnement** concerne les attributs détenant des informations sur le contexte.

Chacun de ces composants est déterminé par des propriétés. Un sujet peut être défini par un identificateur, un groupe auquel il appartient, un rôle etc. Une ressource peut être caractérisée par un identificateur, des propriétés et un type. La même chose pour l'action qui peut être définie par un identificateur, le nom d'action à effectuer. Par exemple, nous considérons la requête suivante :

Un étudiant identifie par « *user_id* » qui appartient au groupe « A » veut accéder à un document public en mode écriture. Pour cette requête nous pouvons distinguer :

- Le sujet : user_id ;
- La ressource : document public ;
- L'action : écriture.

Les propriétés des sujets, ressources et actions sont appelées attributs. Chaque attribut possède une valeur.

1.1.2 La règle

Une règle de contrôle d'accès est définie avec le langage XACML comme étant un ensemble de prédicats qui répondent aux questions suivantes :

- Quels sont les sujets concernés ?

- Quelles sont les ressources demandées ?
- Quelles sont les actions demandées ?
- Quelle est la décision à renvoyer ?

Une règle XACML est composée d'un effet, d'une cible et d'une condition :

- **effet** : détermine la décision de la règle. C'est soit « *Permit* » soit « *Deny* » ;
- **cible** : permet de déterminer si la règle correspond à la requête ou non ;
- **condition** : décrite par un prédicat sur les attributs de la règle.

Par exemple, soit « ***R*** » une requête que nous désirons évaluer par rapport à une règle.

La première étape de validation consiste à évaluer la cible de cette requête afin de savoir si la règle « R » peut être appliquée ou non :

- Si la cible de la règle satisfait celle de la requête, les conditions de la règle seront évaluées ;
- Dans le cas où ces conditions seraient satisfaites, la réponse de la règle sera l'effet spécifié ;

Dans le cas contraire ou si la cible de la règle ne correspond pas à la requête, la règle « R» ne sera pas considérée.

Prenons l'exemple suivant :

- Cible :
  - L'identificateur du sujet est l'étudiant ;
  - La ressource est un document public ;
  - L'action est écriture.
- Condition :
  - L'étudiant doit appartenir au groupe des étudiants de maitrise (groupe A).
- Effet :

- Autorisation d'écriture.

Si un étudiant de groupe « A » demande de modifier un document public, cette règle sera appliquée et retournera la décision « *permit* ». Si un étudiant de groupe « A » demande de modifier un document privé, cette règle ne sera pas appliquée car elle traite seulement les documents publics. Si l'étudiant n'appartient pas au groupe « A », cette règle ne sera pas appliquée car sa condition n'est pas satisfaite.

1.1.3 La politique

Une **politique** est exprimée par une cible, un ensemble de règles et un algorithme de combinaison.

Pour identifier les politiques appropriées à l'évaluation de la requête, il faut d'abord comparer la cible de la requête avec la cible de la politique, et par la suite, vérifier les conditions des règles de la politique afin de déterminer la décision « permit » ou « deny ».

Il est possible que les règles d'une politique retournent des décisions différentes par rapport à une requête donnée. L'algorithme de combinaison des règles permet de spécifier comment déterminer la décision de la politique.

Les décisions possibles sont :

- ***Permit*** : l'accès est autorisé ;
- ***Deny*** : l'accès à la ressource est refusé ;
- ***Indeterminate*** : il n'est pas possible d'appliquer la politique à la requête parce qu'un élément (sujet, ressource, etc) est inconnu ou parce que la construction de la politique ne permet pas d'aboutir à une décision (erreur) ;
- ***NotApplicable*** **:** il n'est pas possible d'appliquer la politique à la requête car elle ne contient aucune règle qui s'applique à la requête.

Un ensemble de Politiques (*Policyset*) est une agrégation de plusieurs politiques ou des ensembles de politiques. *Policyset* contient aussi un algorithme de combinaison pour combiner la décision de politiques.

1.1.4 Algorithme de combinaison

Nous rappelons qu'un algorithme de combinaison permet de calculer la décision d'une politique et d'un ensemble de politiques à partir des décisions de leurs agrégats. XACML 2.0 offre quatre **algorithmes de combinaison** prédéfinis [5] :

- ***Permit-overrides*** : s'il y a une règle évaluée avec effet « *Permit* » alors la décision de combinaison donne également un effet « *permit* » ;
- ***Deny-overrides*** : s'il y a une règle évaluée avec effet « *Deny* » alors la décision de combinaison donne un effet « *Deny* » ;
- ***First-applicable*** : avec l'algorithme First-applicable, l'ordre d'évaluation des règles est important. La politique prend l'effet de la première règle qui s'applique (on ignore la règle non applicable) ;
- ***Only-one-applicable*** : si plusieurs règles sont applicables, la décision «*Indeterminate*» est retournée, sinon la décision de la politique est celle de la règle applicable.

Une extension de ces algorithmes est ajoutée dans la version XACML 3.0, par exemple :

- *Ordered-deny-overrides* utilise le même principe que l'algorithme *« deny-overrides »*, sauf que l'ordre dont les règles sont évaluées est le même que l'ordre dans lequel elles sont décrites dans la politique. La même chose pour *l'algorithme « Ordered-permit-overrides » ;*
- *Deny-unless-permit* est destiné pour les cas où une décision « *Permit* » doit avoir la priorité par rapport à la décision « Deny » et qu'aucune règle ne retourne la décision « Indeterminate ou *NotApplicable* ». Cet algorithme est particulièrement utile dans une structure politique et qu'un PDP retournera toujours une décision « *permit* ou *deny* » ;
- *Legacy Deny-overrides* est conçu pour les cas où une décision « *Deny* » devrait avoir priorité sur une décision « *Permit* ».

1.1.5 Évaluation de la politique

En plus des informations fournies dans la requête, une politique XACML pourrait exiger des informations supplémentaires pour prendre une décision. Ces informations sont récupérées à partir d'une base de données externe.

Lors de l'évaluation d'une requête par rapport à une règle ou une politique, plusieurs types d'erreurs peuvent engendrer la décision « *Indeterminat* » :

- Des erreurs de **réseau** : une politique XACML peut contenir une règle qui réside sur une machine distante temporairement inaccessible ;
- Des erreurs de **syntaxe** : la requête et la politique XACML peuvent comporter des erreurs de syntaxe ;
- Les **requêtes incomplètes** : si la requête ne contient pas les valeurs de certains attributs utilisés dans la cible d'une politique.

Si aucune erreur ne se produit au moment de l'évaluation de la cible d'un contenant, sa décision dépend de l'évaluation des décisions des contenus en utilisant l'algorithme de combinaison.

1.2 Exemple de la politique XACML

Soit « *Pol_document* » une politique qui permet le contrôle d'accès aux documents. L'accès à un document dépend du rôle de l'utilisateur, du type de document et de l'action demandée, voir le tableau 1.1 :

- La politique est décrite de la façon suivante : Tous les utilisateurs ont le droit de consulter les documents « public » ;
- Les gestionnaires ont le droit d'accès aux documents « privé » en lecture seulement ;
- Par contre l'administrateur a le droit de consulter et de modifier les documents « privé ».

	Attribut	Valeur
Sujets	Rôle	administrateur
		manager
		employé
Ressources	Document	privé
		public
Actions	Action	consulter
		modifier

Tableau 1.1 Exemple des valeurs de la cible XACML

Pour mettre en place la politique « ***Pol_document*** », nous décrivons d'abord les règles suivantes :

- R1 : L'administrateur a le droit de consulter et de modifier tous les documents.
- R2 : Les gestionnaires ont le droit de consulter les documents privés.
- R3 : Tous les utilisateurs ont le droit de consulter les documents publics.

La politique « ***Pol_document*** » est représentée schématiquement par la figure 1.3. Son format XML est donnée à l'annexe C.

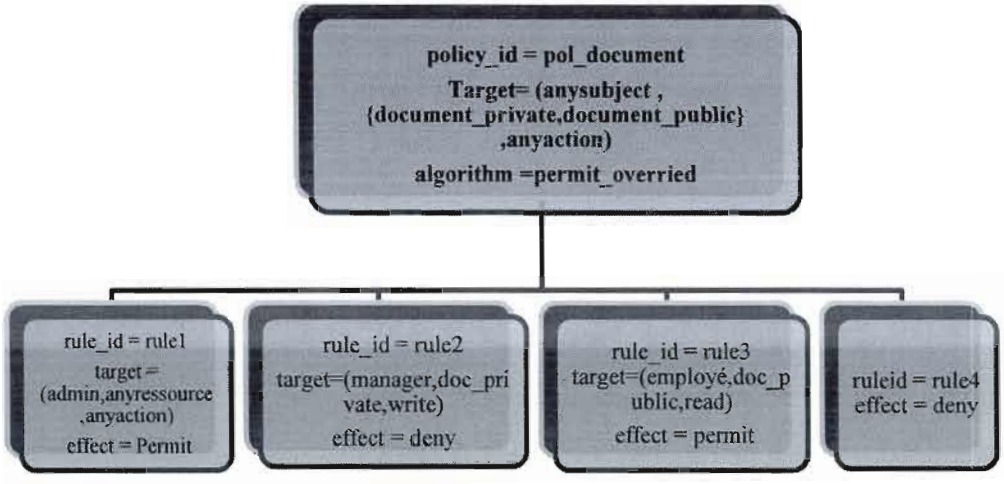

Figure 1.3 Un exemple de la politique XACML

1.3 L'architecture de XACML

L'architecture du système utilisant XACML se compose d'entités. La figure 1.4 synthétise l'ensemble des entités de l'architecture de XACML :

- **Seuil d'application de politique** (*Policy Enforcement Point* (*PEP*)) est l'entité du système qui contrôle la protection des ressources. Le *PEP* fonctionne en collaboration avec le gestionnaire de contexte qui permet d'obten*ir les valeurs des attri*buts des entités du système.
- **Centre de décision de politique** *(PDP : Policy Decision Point)* est l'entité qui prend en charge de déterminer les règles et les politiques applicables à une demande. Après l'évaluation des cibles et des conditions de l'ensemble des règles, le PDP renvoie la réponse au PEP.
- **La source d'information de politique** (*PIP : Policy Information Point*) a le rôle d'extraire des informations supplémentaires non présentes dans la demande d'accès. Le PIP permet de chercher les informations au sein des sources externes dans différentes plates-formes.

Le centre de décision de politique « PDP : *Policy Decision Point* » est considéré comme une boîte noire, la requête XACML serait à l'entrée de PDP, puis la sortie serait la réponse du XACML. Sur la base des informations fournies par la requête, une politique XACML est vérifiée pour déterminer si la demande d'accès à une ressource est autorisée ou non.

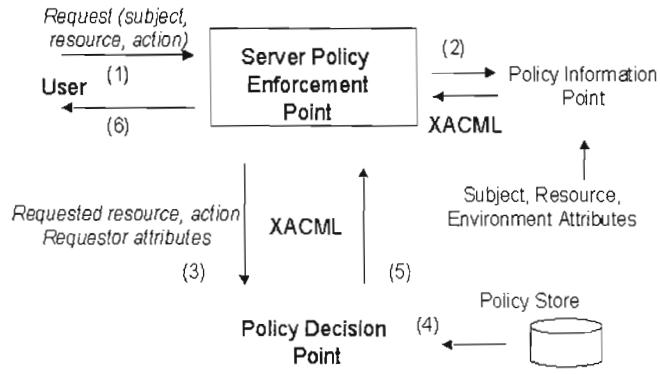

Figure 1.4 Schéma d'une requête de permission avec le langage XACML[6].

CHAPITRE II : MÉTHODES ET OUTILS DE VÉRIFICATION DES POLITIQUES

L'intérêt de l'utilisation des politiques de contrôle d'accès a suscité beaucoup de recherches. Une panoplie d'outils et de méthodes a été développée à des fins de vérification et de validation des politiques.

Notre objectif dans ce chapitre est de présenter d'autre langage des politiques *Ponder et Rei*, et de faire un état de l'art sur les méthodes de test ainsi les travaux réalisés pour la vérification et l'analyse des politiques XACML. Les deux sections du chapitre sont :

- Une description de deux langages de politiques *Rei* et *Ponder*, et par la suite nous comparons ces deux langages avec XACML. Nous mettons en évidence quelques critères généraux et des propriétés pour déterminer le langage le plus adéquates pour exprimer les politiques de contrôle d'accès ;
- Une présentation de littérature de certaines méthodes de test et les travaux réalisés pour la vérification et l'analyse des politiques *XACML*.

2.1 Les langages des politiques

2.1.1 Langage de politiques ***Rei***

Le langage de politiques *Rei*[1][7] permet aux utilisateurs de spécifier des politiques basées sur la logique déontique qui est une formalisation mathématique qui permet de modéliser les obligations, les interdictions et les permissions dans une organisation.

Chaque politique décrit avec le langage *Rei* est associée à une entité. Elle est définie par des règles qui gèrent l'ensemble des droits de l'entité, ses interdictions et ses obligations.

[1] Rei est un mot japonais qui signifie "universel"

La spécification du langage *Rei* offre des moyens d'analyse des politiques et une résolution des conflits avec l'utilisation de la classe « *RulePrioriy* » qui spécifie la priorité des règles.

Par exemple, pour fixer les priorités entre deux règles potentiellement contradictoires RuleA (a la permission d'imprimer) et RuleB (une interdiction de l'impression), la classe rulePriority (la figure 2.1) peut être utilisée pour indiquer que l'interdiction détient la priorité sur l'autorisation.

```
<policy:rulePriority rdf:resource="#PriorityBA"/>
   <metapolicy:RulePriority rdf:ID="PriorityBA">
         <metapolicy:ruleOfGreaterPriority rdf:resource="#RuleB"/>
         <metapolicy:ruleOfLesserPriority rdf:resource="#RuleA"/>
   <metapolicy:RulePriority>
```

Figure 2.1 Exemple de la politique « rulePriority »

- **La spécification du langage *Rei***

La spécification du langage ***Rei*** est composée de plusieurs ontologies :

- *ReiPolicy* ;
- *ReiMetaPolicy* ;
- *ReiEntity* ;
- *ReiDeontic* ;
- *ReiConstraint* ;
- *ReiAnalysis* ;
- *ReiAction* .

Chacune des ces ontologies décrit des classes et des propriétés. L'ontologie de base « ***ReiAction*** » comprend la description des actions.

Chaque action est décrite par :

- Son identificateur unique ;
- Les objets de la cible, sur laquelle l'action peut être effectuée ;
- Un ensemble de conditions préalables qui doivent être remplies avant que l'action puisse être effectuée ;
- Les effets.

- **Analyse des politiques *Rei***

La classe « *What-if* » permet d'analyser les politiques. Plus précisément, elle permet de spécifier des modifications temporaires portées à la politique ou à des entités afin de tester leurs effets.

Il existe deux sous-classes d'analyse de la classe *«what-if» :*

- ***WhatIfProperty*** cette classe permet d'ajouter ou de retirer temporairement une propriété d'une certaine valeur à une entité pour vérifier des modifications portées aux entités ;
- ***WhatIfPolicyRule*** cette classe permet d'ajouter ou de retirer une règle pour vérifier les changements portés aux politiques.

2.1.2 Langage de politiques ***Ponder***

Ponder est un langage orienté objet permettant de spécifier les politiques de sécurité et de gestion des systèmes d'objets distribués [8]. Il fournit des techniques de structuration des politiques visant à répondre à la complexité de gestion des politiques dans les grands systèmes d'information de l'entreprise.

Le langage ***Ponder*** permet de décrire quatre types de politiques : les autorisations, les retenues, les délégations, et les obligations :

- **Politique d'autorisation** : Ce type de politique définit, pour un ensemble de domaines, les actions qui peuvent être effectuées ou non. Une politique

d'autorisation **positive** définit les actions autorisées pour les sujets. Une politique d'autorisation **négative** précise les actions interdites pour les sujets.

- **Politique de retenue** *(Restrain Policies)* : Consiste à définir les actions que les sujets ne doivent pas effectuer sur les objets. Ce type de politiques agit comme des restrictions sur les actions mises en œuvre par les sujets.
- **Politiques de délégation :** ce type de politique décrit les autorisations et les droits d'accès à transférer d'un groupe d'utilisateur à un autre.
- **Politiques d'obligation** : ce type de politique permet de préciser les actions qui doivent être effectuées au sein du système quand certains événements se produisent, fournissant ainsi la capacité de réagir aux circonstances changeantes.

Les deux exemples de la figure 2.2 décrivent l'autorisation d'accès au document privé :

1. La première règle décrit une autorisation positive : l'utilisateur avec le profil administrateur « *admin* » a le droit d'accès au document de type « *document_privat*» en mode lecture, écriture, et modification ;
2. La deuxième règle décrit une autorisation négative : les sujets ayant le profil gestionnaire « manager » n'ont pas le droit d'écrire ou de modifier les documents de type « *document_privat* ».

```
inst auth+ documentPolicyOps {
subject /Admin;
target <document_privat> /document;
action read(), write(), update;
}
inst auth- documentPolicyOps {
subject /manager;
target <document_privat> /document;
action write(), update;
}
```

Figure 2.2 Exemple de politique d'autorisation

Les composantes clés du langage ***Ponder*** (groupes, rôles, relations et le rôle hiérarchie) permettent de structurer les règles, et de réutiliser les spécifications[8] :

- **Les groupes** : Il s'agit d'un regroupement des politiques. C'est un concept commun à la plupart des langages de politique (comme *Policy Set* pour le langage XACML) ;
- **Le rôle** : cette composante fournit un regroupement sémantique des politiques avec un thème commun. Le rôle est représenté par un ensemble de propriétés précises comme : profile, classe, type ;
- **Les relations** : une relation est une composante qui inclut des politiques ayant des rôles communs ;
- **Les rôles hiérarchie** : *Ponder* permet la spécialisation des types de politiques, à travers le mécanisme d'héritage semblable aux langages orienté objets.

2.1.3 Comparaison des langages de politiques

L'article [9], décrit un ensemble des critères qui permettent de juger la qualité d'un langage de politique :

- **Formalisme** : la syntaxe et la structure du langage de la politique doit être claire et sans ambiguïté. Le sens d'une politique écrite avec le langage devrait être indépendant de sa mise en œuvre particulière ;
- **La flexibilité** est la définition des abstractions pour gérer une grande variété de types. L'architecture du système devrait être suffisamment souple pour autoriser l'ajout de nouveaux types ;
- **L'interopérabilité** : le langage doit disposer une architecture lui permettant d'inter opérer avec d'autres architectures qui peuvent exister dans d'autres plates-formes ;
- **Détection des conflits** : le langage doit être en mesure de vérifier qu'une politique n'entre pas en conflit avec une autre ;
- **Évolutivité** : il convient de maintenir des performances de qualité sous une charge accrue du système.

Le tableau suivant présente une comparaison des langages XACML, Ponder et Rei selon un ensemble des critères préétablis considérés comme importantes :

Langage	XACML	Ponder	Rei
Flexibilité	Oui	Oui	Non
Formalisme	Oui	Oui	Non
L'interopérabilité	Oui	Non	Oui
Détection des conflits	Oui	Oui	Oui
Évolutivité	Moyen	Moyen	Élevé

Tableau 2.1 Comparaison des langages de politiques **[9]**

2.2 Les travaux de test et de vérification des politiques XACML.

Nous allons introduire dans cette section une revue de la littérature sur les techniques de test et les méthodes d'analyse et de vérification des politiques *XACML*. Dans la première sous section, nous décrivons les méthodes de test utilisés pour tester les politiques *XACML*, cette sous section décrit :

1. La technique de mutation;
2. Les métriques de couverture pour analyser les politiques *XACML* ;
3. L'outil *Margave* pour analyser les politiques *XACML*.

Dans la deuxième sous section nous décrirons la vérification des politiques XACML avec des méthodes et des langages formels : VDM++, et Alloy

2.2.1 Technique de mutation

L'approche de mutation des opérateurs permet de déterminer les résultats non acceptables par l'utilisateur. L'article [10] décrit une liste d'opérations de mutation pour les éléments de politiques XACML. Cette technique permet de décrire un ensemble des opérations de mutation sur les attributs de la cible de la requête ou sur l'effet des règles de

politiques, par la suite, on évalue les règles de politiques par rapport aux requêtes, ce qui permet d'identifier la différence entre les spécifications des politiques et le résultat de l'évaluation.

Soit la règle *R1* : un gestionnaire a le droit d'accès au document de type privé en mode lecture. La figure 2.3 présente l'opération de mutation de changement d'effet de la règle CRE (*Change Rule Effect*) pour la règle R1.

La règle avant l'application de l'opération CRE	**La règle après l'application de l'opération CRE**
<Rule Effect="Permit" RuleId="rule2"> <Target> ..... </Target> </Rule>	<Rule Effect="Deny" RuleId="rule2"> <Target> <Subject> ..... </Target> </Rule>

Figure 2.3 Operation de mutation CRE[10]

Dans la première colonne de la figure 2.3, on décrit la règle avant l'application de l'opération de mutation CRE, dans la deuxième colonne, on décrit l'opération de mutation CRE, qui consiste de changer l'effet de la règle.

L'utilisation des opérateurs de mutation permet d'analyser et d'imiter les erreurs logiques produites lors de la construction de politiques [10].

En conséquence, pour que cette méthode soit fiable, il est nécessaire de définir un ensemble de critères permettant une caractérisation adéquate de tests qui soit dépendent de la nature de la requête en cours d'évaluation et qui permettent d'améliorer la vérification de politiques.

2.2.2 La métrique de couverture

L'utilisation de la méthode de couverture des politiques XACML consiste à définir des critères qui permettent de maximiser le nombre des éléments de la politique qu'une requête peut couvrir. En d'autres termes, la métrique de couverture consiste à sélectionner seulement les requêtes qui couvrent le maximum d'éléments de la politique.

Plus précisément, Martin et Xie [11] ont défini une métrique de couverture de la façon suivante :

- Couverture de la politique : une politique est couverte par une requête si la politique est applicable à la requête. la métrique de la couverture de la politique est le nombre de politiques couvertes divisé par le nombre total des politiques.
- Couverture des règles : la couverture d'une règle exige qu'une requête soit applicable à la politique et à la règle. la métrique de la couverture des règles est le nombre des règles applicables à une requête par rapport au nombre total de règles.
- La condition d'une requête : Une condition est couverte par une requête signifie que la requête est applicable à la politique et à la condition. La métrique de couverture des conditions est définie par la somme des conditions qui sont évaluées sur le nombre total des conditions.

L'approche considère qu'une requête est applicable à un élément de la politique, si elle remplit les conditions de l'élément à évaluer. Les métriques utilisées pour la couverture des politiques sont définies avec un ratio. Ce ratio détermine le pourcentage de l'implication du composant dans l'évaluation de la requête.

Soit q une requête et r une règle de la politique P. Nous disons que q couvre r si la règle est applicable à la requête q. Soit Q un ensemble de requêtes. La couverture de la politique P par Q est le ratio R définit par :

$$\frac{\text{le nombre de règles couvert par au moins une requête de } Q}{\text{le nombre total des règles de } P}$$

Martin et Xie [11] ont proposé une technique qui permet de générer d'une manière aléatoire un ensemble de requêtes afin de vérifier les décisions des politiques. Les requêtes générées sont présentées avec une valeur binaire.

Pour une politique ***P*** dont les attributs sont composés par :

- **Sujets** : *Student, Faculty*
- ***Ressources*** *: ExtGrades , IntGrades*
- ***Actions*** *: Assign , Receive*

Le tableau 2.2 représente un exemple de la génération aléatoire des valeurs des attributs de la politique ***P***.

Valeur d'attribut	Requête1	Requête2	Requête3
Student	0	1	0
Faculty	1	0	1
Assign	1	1	0
Receive	0	0	1
ExtGrades	0	1	1
IntGrades	1	1	0

Tableau 2.1 Génération aléatoire des requêtes

Une valeur est incluse dans une requête si sa valeur binaire est 1, sinon, la valeur n'est pas présente.

2.2.3 Margrave

Margrave [12] est un outil développé pour la validation des politiques XACML. La validation se base sur une transformation des règles vers des diagrammes de décision *MTDD* (*Multi-Terminal Decision Diagrams)*

MTDD est un arbre qui détermine les décisions et qui utilise un ensemble de variables pour définir l'ensemble des conditions. Chaque combinaison de valeurs booléennes sur ces variables engendre l'un des trois résultats (permis, refusé ou non applicable) signalé par les feuilles de l'arbre MTDD.

La figure 2.4 est extraite de l'article [13] représente l'exemple de la politique qui se compose de deux règles :

- Un sujet dont le profil est « *student* » **(*s*)** peut effectuer l'action « *receive* » **(*r*)** sur la ressource grades.
- Un sujet dont le profil est « *faculty* » **(*f*)** peut effectuer l'action « *assign* » **(*a*)** sur la ressource « *grades* » **(*g*)**.

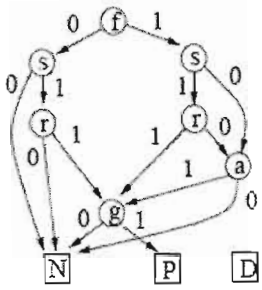

Figure 2.4 Diagramme de décision de MTDD [13]

La politique formée par les deux règles précédentes est représentée par un arbre (figure 2.4). Parcourir l'arbre permet d'obtenir une décision (N : *NotApplicable*, P : *Permit* et D : *Deny*). Les nœuds de cet arbre représentent les valeurs des attributs. Il y a deux profils pour les sujets {f : ***faculty***, s :***student***}. Les actions possibles sont {a : ***assign***, r : ***receive***}, et la ressource de cette politique est **{g : *grades*}**.

- Chaque nœud est étiqueté par une variable ;
- les arcs sortants de chaque nœud représentent les valeurs possibles la variable ;
- le nœud ***f*** représente la valeur « *faculty* » qui désigne un profil du sujet ;
- l'arc ***(f, s) = 1,*** correspond au cas où le profil du sujet est « *faculty* » ;
- l'arc ***(f, s) =0,*** correspond le cas contraire, c'est-à-dire le profil du sujet ne correspond pas à « *faculty* ».

Il faut noter que nous pouvons atteint le nœud D « Deny » si les variables de la politique correspondent sont : « role=student, action=assign, et resource= grade »

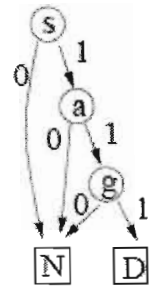

Figure 2.5 Diagramme MTDD de décision *Deny*

Margrave prévoit la vérification des politiques et permet d'analyser l'impact de changement des politiques en utilisant les diagrammes MTDD qui représentent ces politiques.

L'inconvénient de Margrave est que la validation des règles devient complexe dans le cas où la politique est représentée avec des règles compliquées, parce que la transformation des règles en diagrammes de décision génère un grand nombre de nœuds.

2.2.4 Les langages formels de vérification.

- **VDM++**

VDM++ est une extension du langage VDM (*Vienna Development Method*). L'article[14] présente un modèle qui consiste à décrire les politiques de contrôle d'accès XACML avec le langage VDM++. Un ensemble de requêtes de test peut être évalué par l'exécution du modèle décrit avec VDM++ en utilisant l'interpréteur « *VDMTools* » [15] (figure 2.5).

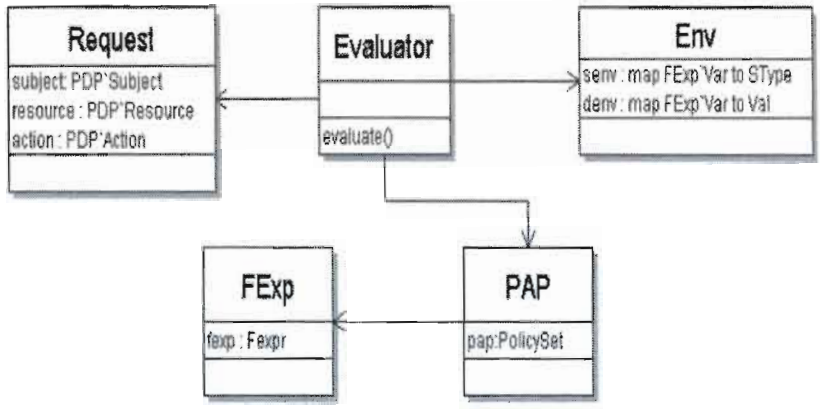

Figure 2.6 Modèle VDM++ [14]

Le modèle évalue une demande par rapport aux politiques de la classe *PAP* selon les paramètres de l'environnement (une instance de la classe *Env*). Le diagramme de classes (figure 2.5) représente la structure du modèle VDM ++, qui représente étroitement celle du XACML[14] :

- La classe de requête ***Request*** décrit les demandes que l'utilisateur peut faire ;
- Les paramètres d'environnement sont représentés par la classe ***Env*** ;
- les politiques sont représentées par des objets de la classe *PAP* ;
- La classe ***FEXP*** représente les règles et les conditions ;
- La classe ***Evaluator*** reçoit la demande et recueille les informations de la classe environnement et du centre de décision de politiques *PAP*.

Le modèle permet aux développeurs d'évaluer les conséquences des décisions des politiques dans des environnements dynamiques. Dans l'article [14] on présente un exemple de comparaison des environnements de test, on suppose qu'une ressource « *haz_an* » existe dans l'environnement « 1 » et n'existe pas dans l'environnement « 2 », par la suite on évalue un ensemble de requêtes par rapport aux politiques du système dans les deux enivrements

(figure 2.6).

Request	Environment 1	Environment 2
Request(Anne,haz_an,{write})	<NotApplicable>	<NotApplicable>
Request(Anne,haz_an,{review})	<NotApplicable>	<NotApplicable>
Request(Bob,haz_an,{write})	<NotApplicable>	<NotApplicable>
Request(Bob,haz_an,{review})	<Deny>	<Deny>
Request(Anne,pp,{write})	<NotApplicable>	<Deny>
Request(Anne,pp,{review})	<Deny>	<Deny>
Request(Bob,pp,{write})	<NotApplicable>	<Deny>
Request(Bob,pp,{review})	<NotApplicable>	<NotApplicable>

Figure 2.7 Résultats de l'évaluation des requêtes dans les deux environnements**[14]**

Avec l'application du même ensemble de tests à deux environnements différents, le résultat de la comparaison des décisions de ces deux environnements permet de détecter les incohérences entre les règles des politiques XACML.

Par exemple, la requête (*request(Anne,pp,{write})* ne génère pas la même décision dans les deux environnements ,dans l'environnement 1 on obtient comme décision « *NotApplicable* », alors que dans l'environnement 2 on obtient la décision « *deny* ».

- **Alloy**

Alloy est un langage et un outil de vérification et de validation de modèles formels. Il a été développé par le groupe de recherche *Software Design Group* du MIT [16]. Les démarches de la vérification et de la validation avec Alloy consistent à modéliser les systèmes afin de simuler et vérifier les propriétés.

Alloy offre des moyens pour représenter des systèmes d'une façon simple en faisant l'abstraction des détails d'implémentation et en mettant l'accent sur les contraintes et les propriétés du système. *Alloy* est un langage structurel, déclaratif, et analysable parce que les propriétés d'un modèle *Alloy* peuvent être vérifiées, et simulé avec l'analyseur *Alloy(Alloy analyser)*[17].

[17] présente une autre approche basée sur le langage Alloy. L'auteur a utilisé le langage XSLT afin d'extraire les attributs de politiques XACML pour les intégrer dans le modèle décrit avec le langage Alloy.

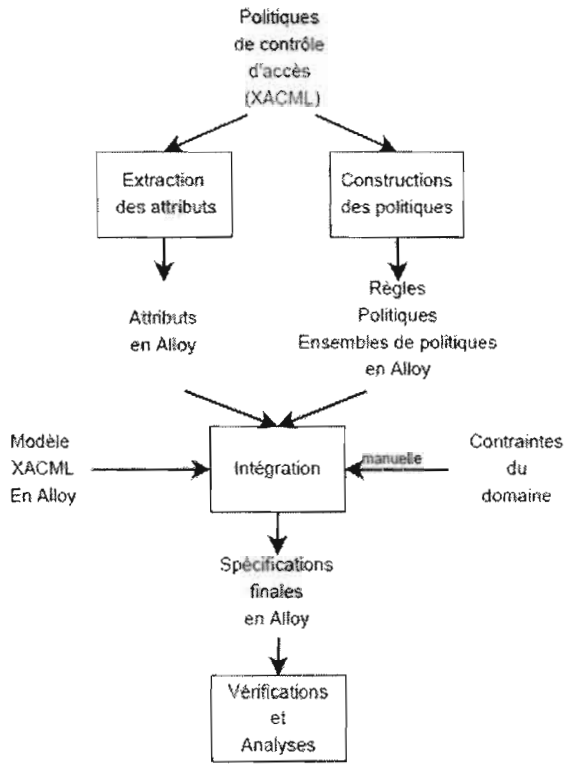

Figure 2.8 Approche de vérification des politiques de contrôle d'accès [17]

Dans cette approche, on définit des propriétés qui caractérisent les cibles dans XACML. Par la suite, et avec l'utilisation de l'analyseur de modèle d'Alloy, on analyse l'impact de ces relations sur les interactions entre les règles, les politiques et l'ensemble de politiques. Ces propriétés sont : Inclusion, Intersection, Disjonction, Égalité.

2.3 Discussion

En général, la vérification des politiques consiste à évaluer les politiques par rapport à toutes les requêtes et à tous les paramètres du contexte d'environnement, puis d'en comparer les résultats. La plupart des travaux cités dans ce chapitre ont le même objectif, celui de soutenir la conception et l'évolution du contexte des politiques.

Plusieurs travaux ont abordé l'utilisation des méthodes formelles qui servent pour analyser et vérifier les politiques XACML, et d'obtenir une présentation des spécifications XACML sous forme d'un ensemble de formes logiques. Nous avons présenté deux approches. La première approche est basée sur le modèle checking [19] comme VDM++ tandis que la seconde est fondée sur le raisonnement logique[20] comme Alloy .

D'autres travaux ont présenté une méthode pour générer un ensemble de requêtes de test, dans le but d'évaluer l'ensemble de politique. Pourtant le test ne peut pas assurer la décision de la politique car généralement, les tests ne garantissent qu'un certain niveau de la qualité.

En conclusion, la littérature nous a encouragés à diriger notre intérêt sur les méthodes d'analyse et de vérification de politiques de contrôle d'accès, ainsi d'approprier des techniques pour identifier les propriétés et les méthodes de vérification basé sur des langages formelles.

Dans le prochain chapitre nous présentons un aperçu sur le langage formel Event-B[4] que nous avons choisi pour exprimer les politiques XACML. Avec l'utilisation du langage Event-B nous construisons un modèle et nous décrivons des propriétés pour vérifier les interactions et les incohérences entre les règles et des politiques XACML.

CHAPITRE III : LE LANGAGE EVENT B

Event-B est un langage formel pour la modélisation de système. C'est une évolution du langage B. Sa principale caractéristique est la combinaison des techniques mathématiques, la théorie des ensembles comme une notation de modélisation et la logique de premier ordre.

Event-B se base sur l'utilisation de raffinement pour représenter des systèmes à différents niveaux d'abstraction. Les preuves mathématiques permettent de la cohérence entre les niveaux de raffinement. Event-B est soutenu par des outils comme ProB[21] et Rodin pour tester et valider le processus décrit par cette méthode.

Le plan de ce chapitre se présente en trois sections :

- Nous présentons dans la première section un aperçu sur la notation utilisée par le langage Event-B ;
- Dans la deuxième section nous présentons les caractéristiques du langage Event-B ;
- Nous décrivons dans la troisième section la plate-forme de développement RODIN pour le langage Event-B, et l'outil de vérification ProB.

3.1 Le langage Event-B

Event-B est un langage formel qui a été développé pour spécifier correctement et modéliser itérativement des systèmes complexes par un mécanisme de raffinement. Le modèle formel d'Event-B repose sur la représentation des événements qui marquent l'évolution du système. Dans cette section, nous présentons les notations d'Event-B, et les éléments principaux qui constituent un modèle décrit avec Event-B.

3.1.1 Notation du langage Event-B

Les machines abstraites et le raffinement sont au cœur du langage Event-B. Une machine abstraite se compose d'un ensemble de constantes et de variables globales. L'état du

système est défini par des variables. Chaque événement dans le modèle abstrait est composé par un ensemble de gardes et d'actions.

Les notations d'Event-B sont généralement sous-forme de notations relationnelles ou fonctionnelles. On peut résumer les notations relationnelles dans le tableau suivant :

Notation	Description
$\mapsto$	Tuple
dom(R)	Domaine de la relation R
ran(R)	Codomaine de la relation R
$\lhd$	Domaine restrictive
$\rhd$	Codomaine restrictive
$\hookleftarrow$	Domaine anti-restrictive
R[A]	Image de la relation R par rapport à l'ensemble A

Tableau 3.1 Notations du langage Event-B

- Soit A et B deux ensembles, la notation « $\leftrightarrow$ » définit la relation entre les deux ensembles. $A \leftrightarrow B$ est l'ensemble des relations entre A et B. $\mathbb{P}$ (A x B) est l'ensemble des sous ensemble. Une relation entre A et B est aussi vue comme une partie du produit cartésien A x B. On peut ainsi écrire :
  - $A \leftrightarrow B \quad = \mathbb{P}(A \times B)$
  - $dom(R) = \{a | a \in A \wedge \exists b.(b \in B \wedge a \mapsto b \in R)\}$
  - $ran(R) = \{b | b \in B \wedge \exists a.(a \in A \wedge a \mapsto b \in R)\}$
  - $X \lhd R = \{a \mapsto b \mid a \mapsto b \in R \wedge a \in X\}$
  - $X \hookleftarrow R = \{a \mapsto b \mid a \mapsto b \in R \wedge a \notin X\}$
  - $Y \rhd R = \{a \mapsto b \mid a \mapsto b \in R \wedge b \in Y\}$

3.1.2 Les substitutions généralisées

Les substitutions généralisées permettent d'exprimer l'aspect dynamique des spécifications d'évent-B. Ces substitutions généralisées sont basées sur le calcul de la plus faible pré-condition de Dijkstra [22].

La sémantique des substitutions est définie sous la forme de transformateurs de prédicats. Soit la formule : **P[S]R** où, **P** exprime la post condition de la substitution, et **R** la post condition de l'instruction **S**. Cette formule signifie que si le prédicat P est vrai et que l'instruction **S** se déclenche, alors l'assertion **R** est vraie après l'exécution de **S**.

3.2 Modèle du système avec le langage Event-B

Dans le langage Event-B, l'état d'un système est représenté par un ensemble des variables typés. Les événements marquent les changements d'état du système. Les événements se composent des gardes sur des actions qui peuvent se produire spontanément au lieu d'être invoqués. Les modèles d'Event-B sont décrits par deux concepts de base : les contextes et les machines.

3.2.1 Contexte

Un contexte représente la partie statique du modèle, il est composé de constantes et d'axiomes décrivant les propriétés de ces constantes. Un contexte peut être étendu par un autre contexte, et il peut être visible pour une machine en utilisant la clause « *sees* » (voir figure 3.1).

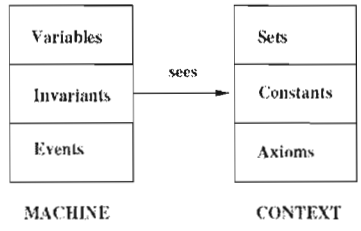

Figure 3.1 Relation entre un contexte et une machine

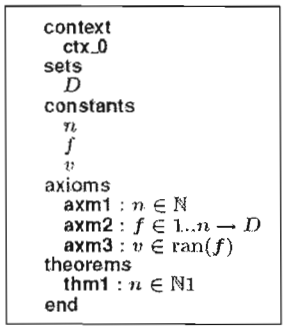

Figure 3.2 Exemple du context d'Event-B

On décrit dans le contexte « *ctx_0* » (la figure 3.2) les éléments suivants:

- L'ensemble **D** ;
- Trois constantes sont définies, **n, f** et **v ;**
- Dans la clause axiomes on définit trois axiomes qui décrivent :
  - **n** est un entier positif (axm1) ;
  - **f** est une fonction totale de l'intervalle **1... n** à l'ensemble D (axm2) ;
  - **v** est supposé appartenir à l'image de la fonction f (axm3) ;
- Un théorème est proposé à la fin du contexte : **n** est un entier positif (thm1) .

3.2.2 La machine Event-B

La machine contient des éléments dynamiques qui décrivent l'état du système. Une machine est constituée de trois sections principales : les invariants, les variables et les événements (figure 3.3). Les événements permettent de changer l'état des variables tout en respectant les conditions décrites dans la clause des invariants.

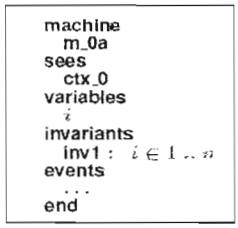

Figure 3.3 Machine event-B

3.2.3 L'événement

Les événements jouent le rôle des opérations de la machine. Chaque événement est composé d'une condition et une action. Une fois les conditions satisfaites, l'événement peut être déclenché à tout moment. Cependant, dès que la condition ne tient plus, l'événement ne peut plus être déclenché.

L'action, comme son nom l'indique, détermine la façon d'évoluer les variables au moment où l'événement se produit. L'action permet de spécifier et de déduire les propriétés du système. L'événement d'une machine abstraite est correcte lorsque celle-ci préserve l'invariant de la machine. La figure ci-dessus présente un exemple d'événement décrit avec l'outil Rodin.

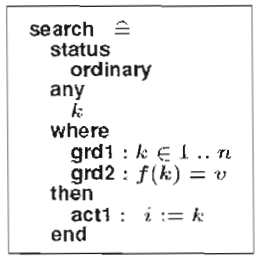

Figure 3.4 Exemple d'événement décrit avec Rodin

3.3 Les outils

3.3.1 L'outil de vérification ProB

ProB s'avère un outil qui automatise la vérification de la cohérence des machines via un modèle de vérification [21]. Nous pouvons utiliser l'interpréteur de ProB pour exécuter directement les scénarios de tests possibles pour vérifier la couverture des événements de la machine, et pour détecter les interblocages *(deadloack)*.

L'outil ProB peut être également utilisé de façon non exhaustive pour analyser et explorer l'espace d'état et chercher des problèmes potentiels. L'outil ProB génère aussi un graphe (figure 3.7) pour montrer les contre-exemples si des violations de l'invariant sont découvertes.

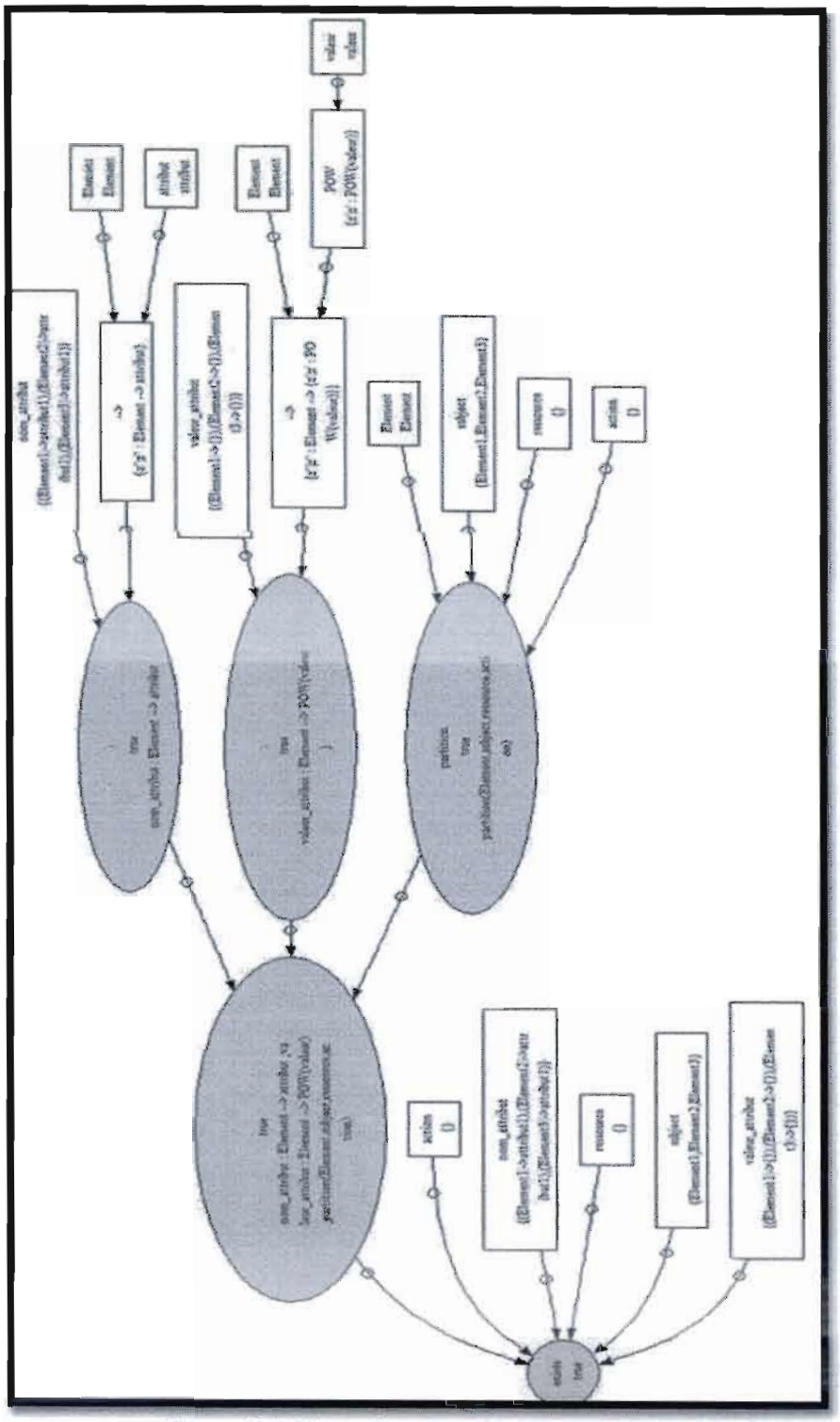

Figure 3.5 Trace d'analyse des événements avec ProB

L'outil ProB permet de détecter les expressions indéfinies comme l'application partielle d'une fonction à des arguments en-dehors de son domaine et peut également être utilisé comme un animateur des spécifications.

ProB offre aussi une méthode de contrôle de remplacement. Dans ce mode de fonctionnement, les états accessibles à partir de l'état initial ne sont pas probablement explorés. ProB vérifie également si le déclanchement d'une opération individuelle peut provoquer une violation d'une contrainte indépendamment de l'initialisation de la machine B.

L'outil proB permet de vérifier si la spécification est conforme aux exigences du système, et de produire un contre-exemple, ce qui permet de renforcer les gardes et les invariants[21]. L'outil ProB utilise deux moyens de contrôle de cohérence :

1. L'utilisation d'un modèle de contrôle qui consiste à trouver, à partir d'un état initial, une séquence d'opérations dont l'état ne vérifie pas les conditions de l'invariant.
2. À l'égard d'une vérification des contraintes de base, si un état « *E1* » vérifie les conditions d'un 'invariant « INV » , il faut vérifier s'il existe un événement « EVE » qui utilise l'état « E1 » pour générer un état « *E2* », et l'état « E2 » ne vérifie pas les conditions de l'invariant « INV ».

ProB permet de visualiser le comportement dynamique d'une machine B et d'analyser systématiquement les états accessibles d'une machine B. L'analyse des événements permet d'obtenir des informations suivantes :

- Le nombre d'interblocage (deadlock) ;
- Le nombre d'invariants ;
- Le nombre d'état en erreur;
- Le nombre des opérations effectuées;
- La liste des événements couverts ;
- La liste des événements non couverts.

3.3.2 Rodin

Rodin est une Plate-forme open source [21] basée sur Éclipse qui offre un soutien efficace pour le raffinement et la preuve mathématique. La plate-forme Rodin est l'acronyme de (*Rigorous Open Development Environment for Complex Systems*). Il s'agit d'un outil de développement, permettant d'intégrer les outils de support à la méthode B.

La plate-forme Rodin a certaines caractéristiques qui en font de lui un outil unique (efficace, fiable et réutilisable) pour le développement de modèles. Ainsi, il aide à la compréhension du système dans son ensemble. Ces principales caractéristiques sont :

- Il permet la **vérification statique** qui valide si les propriétés du système sont en cours de validation. En cas de problème il faut signaler l'erreur puisque la plate-forme n'a pas une syntaxe fixe
- Il offre un **générateur** qui permet de produire les preuves, afin d'examiner si le modèle est valide ou non. Avec l'utilisation d'un démonstrateur qui permet de vérifier automatiquement autant de preuves sans avoir besoin d'ajouter des prédicats (sous forme des hypothèses).
- Il est muni d'une **interface graphique** (figure 3.9) facile a utiliser pour créer / modifier le modèle et raisonner sur le système (par preuves interactives).
- Il offre une plate–forme extensible permettant l'intégration de nouvelles fonctionnalités à l'outil (par exemple, les vérificateurs modèle, démonstrateurs, animateurs, UML-B [25], latex, etc.), par le développement de plug-ins.

CHAPITRE IV : XACML VERS EVENT-B

La vérification des politiques XACML consiste de vérifier une forme normale des prédicats, chaque règle de la politique est représentée par une formule logique qui retourne une décision booléenne. En effet, pour vérifier et analyser les politiques XACML, nous inspirons de la stratégie de la logique premier ordre pour transformer les politiques XACML sous forme des formules basée sur la logique classique.

L'objectif principal de ce chapitre vise à définir une traduction des politiques de contrôle d'accès XACML vers Event-B. Le modèle permet de représenter les politiques de contrôle d'accès XACML afin de s'assurer la cohérence et l'exactitude des règles.

Nous allons présenter dans ce chapitre le modèle et la sémantique des politiques XACML. Nous tentons d'avoir un modèle avec une sémantique fidèle à celle du langage XACML afin d'avoir la possibilité d'analyser les politiques selon un contexte donné.

Les démarches suivies pour réaliser un modèle de politiques de contrôle d'accès XACML sont les suivantes :

1. Définir un modèle avec l'utilisation du diagramme d'UML-B de Rodin, afin de décrire la relation entre les machines et les contextes des politiques de contrôle d'accès XACML ;
2. Définir les axiomes et les événements qui décrivent la structure des politiques de contrôle d'accès XACML ;
3. Ajouter les propriétés à vérifier sous forme de prédicats et d'événements en vue d'évaluer les règles avec la plate-forme Rodin et l'outil ProB.

4.4 Modèle de contrôle d'Accès XACML avec UML-B

4.4.8 Approche et démarches de vérification.

Pour concevoir un modèle des politiques de contrôle d'accès XACML, nous avons constaté qu'il est impératif d'effectuer en amont un travail de modélisation de processus afin de décrire formellement les événements, les prédicats, les règles de transition et les données du contexte.

Nous commençons par la définition d'un ensemble de propriétés, en termes de logique de premier ordre, qui servent à définir les différentes relations caractérisant les cibles dans XACML. Ensuite, nous ajoutons les contraintes et les prédicats à l'étape de raffinement en utilisant des variables concrètes.

4.4.9 Modèle XACML avec UML-B

Le modèle de la figure 4.1 décrit les éléments qui constituent la politique de contrôle d'accès XACML, sous forme d'un ensemble de contextes et de machines avec UML-B sous la plate-forme de Rodin.

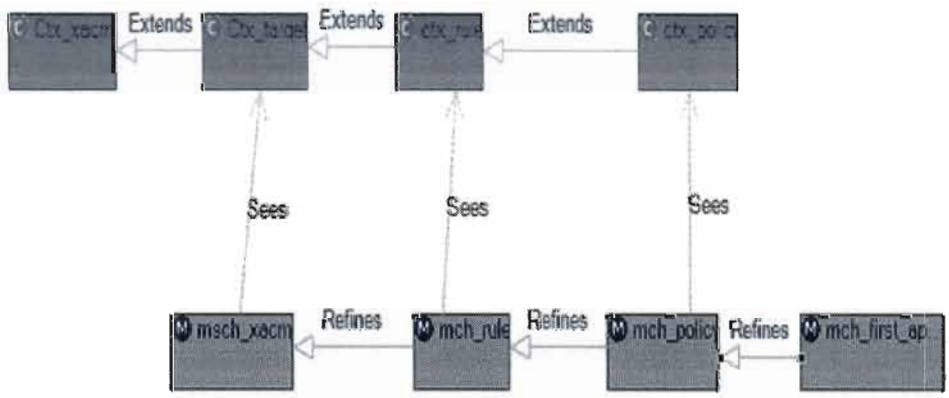

Figure 4.6 Modèle des politiques XACML avec Event-B (sous Rodin)

Le modèle se compose de contextes et de machines suivants :

- **Contextes**
  - ***Ctx_xacml*** : définit les types abstraits et l'ensemble des constantes utilisés dans les politiques XACML comme les valeurs de l'effet des règles et les noms d'algorithme de combinaison.
  - ***Ctx_target*** : ce contexte est une extension du contexte *ctx_xacml*, dans ce contexte, on ajoute la définition de la cible et les axiomes pour contrôler la définition des attributs de la cible.
  - ***Ctx_rule*** : c'est une extension du contexte *ctx_target* où nous avons ajouté la définition des éléments de la règle sous forme de relations et d'axiomes.
  - ***Ctx_policy*** : comme son nom indique, il s'agit d'un contexte pour les politiques qui est représenté comme extension de *ctx_rule*.
- **Machines**
  - **Mch_xacml** : cette machine contient des prédicats et des événements pour évaluer les attributs de la cible.
  - **Mch_rule:** la machine *mch_rule* est un raffinement de la machine mch_xacml, nous ajoutons dans cette machine d'autres événements pour évaluer l'effet de chaque règle selon l'évaluation de la cible.
  - **Mch_policy**: dans cette machine, nous définissons autres événements pour évaluer la cible de la politique.
  - **Mch_evaluate**: cette machine permet de déterminer la décision de la politique selon l'algorithme de combinaison et selon l'évaluation de la cible par rapport à celle de la politique et de ses règles.
  - **Mch_conflit**: c'est une machine qui contient un ensemble d'événements pour évaluer certaines propriétés afin d' analyser le conflit et la relation entre les règles et les politiques XACML.

Dans la prochaine section, nous détaillons le modèle UML-B avec l'utilisation de la syntaxe du langage Event-B.

4.2 Structure de la politique XACML avec Event-B

Cette étape consiste à analyser la structure de la politique XACML afin de construire un ensemble de relations et de prédicats. Ces prédicats permettent de déterminer des relations et des contraintes appliquées sur les attributs.

Dans tous le processus de modélisation de politiques XACML avec Event-B, nous utilisons des types abstraits et des constants, afin de définir les éléments de base. Nous considérons la liste des types abstraits suivants :

```
sets attribut
     valeur
     effect
     Element
     Rules
     Policy
     Algorithmes
```

- ***Element*** : un type abstrait pour exprimer les attributs des sujets, ressources et actions ;
- ***Attribut***: un type abstrait regroupant les noms des attributs ;
- ***Valeur*** : un type abstrait qui contient l'ensemble des valeurs permises pour les attributs ;
- ***Effect*** : un type abstrait pour définir les décisions possibles des règles XACML ;
- **Rules** : identification des règles ;
- **Policy** : identification des politiques ;
- **Algorithmes** : un type abstrait représentant les algorithmes de combinaison.

4.2.1 Les attributs

Les sujets, ressources et actions sont des éléments du langage XACML qui ont des propriétés identiques. Dans le modèle décrit avec le langage Event-B, nous considérons que tous les attributs de la cible des politiques XACML ont le même type abstrait « *Element* ».

Avec le langage Event-B, nous définissons dans la clause « *constants* » du contexte « *ctx_xacml* » trois ensembles « *subject, resource et action* » renseignant respectivement sur les sujet, ressource et action de la cible.

```
constants subject
          ressource
          action
          nom_attribut
          valeur_attribut
```

« *nom_attrribut* » et « *valeur_attribut* » sont deux relations qui définissent respectivement le nom de l'attribut et les valeurs permises pour chaque élément.

```
@axm7 nom_attribut ∈ Element → attribut
@axm8 valeur_attribut ∈ Element → ℙ(valeur)
```

Nous ajoutons d'autres contraintes, pour spécifier que la liste des sujets, ressources, et actions sont trois ensembles distincts de type « *Element* ».

```
@axm10 partition(Element,subject,ressource,action)
```

4.2.2 La cible

Basée sur la définition des attributs, une cible est une structure définie par trois fonctions « *tag_subject* », « *tag_ressource* » et « *tag_action* » :

- « *tag_subject* » : définit l'ensemble des sujets permis pour chaque cible ;
- « *tag_ resource* »: définit l'ensemble des ressources permises pour chaque cible ;
- « *tag_ action* »: définit l'ensemble des actions permises pour chaque cible.

En effet, pour représenter la cible avec le langage Event-B, nous définissons trois fonctions :

```
@axm1 tag_subject ∈ target → subject
@axm2 tag_ressource ∈ target → ressource
@axm4 tag_action ∈ target → action
```

4.2.3 La requête

En général, dans XACML la requête d'accès est définie par un sujet, une ressource et une action. Dans notre modèle nous représentons la requête de la manière suivante :

```
@axm5 req_subject ∈ requete → subject
@axm6 req_ressource ∈ requete → ressource
@axm7 req_action ∈ requete → action
```

Puisque la cible détermine l'application de la politique face à des demandes d'accès ou d'autorisation, nous représentons la cible de la politique de la même façon que celle de la règle. Par conséquent, nous définissons la cible de la manière suivante :

```
@axm8 tag_req ∈ requete → target
@axm9 tag_pol ∈ policy → target
@axm10 tag_rule ∈ rules → target
```

4.2.4 Effet

Les effets envisageables dans le langage XACML sont : « *Permit* », « *Deny* », « *NotApplicable* » et « *Indeterminate* ». Donc, avec le langage Event-B nous définissons le type « *effect* » et la liste des valeurs possibles de la manière suivante :

```
@axm11 partition(effect,{permit},{deny},{Notapplicable},
{Indeterminate})
```

4.2.5 L'algorithme de combinaison

Une politique décrite avec XACML peut inclure plusieurs règles, et avec l'utilisation des algorithmes de combinaison des règles nous pouvons sélectionner la réponse. Soit un ensemble de noms des algorithmes utilisés lors de l'évaluation des requêtes, cet ensemble sera constitué comme suit :

```
    @axm21 partition(algorithms,{Permit_overried},{Deny_overried},
{First_one_applicable})
```

4.3 Processus d'évaluation d'une requête

4.3.1 Évaluation de la cible

Pour vérifier si un élément (sujet, ressource ou action) d'une requête est couvert par une politique, nous définissons une relation entre les attributs définis par la cible de la requête et ceux définis par de la cible de la politique.

Nous définissons aussi une relation qui lie un élément de la cible de la requête à son homologue dans la cible de la politique.

```
@axm20 ∀ ele,ens_ele·(ele ∈ Element ∧ ens_ele ∈ ℙ(Element) ∧
(∃ ele1·(ele1 ∈    ens_ele ∧ nom_attribut(ele) = nom_attribut(ele1) ∧
valeur_attribut(ele) ⊆ valeur_attribut(ele1)))
   ⇒ (ele ↦ ens_ele) ∈ match_element)
```

Par conséquent, pour déterminer si une politique ou une règle est appropriée pour une requête donnée, il faut vérifier leurs cibles. Le prédicat « *match_target* » permet de déterminer si le sujet (respectivement la ressource et l'action) défini par la requête, correspond à l'un des sujets (respectivement ressources et actions) de la cible de la règle ou de la politique selon le contexte d'évaluation.

```
@axm11 ∀  req,tag·(req ∈ requete ∧ tag ∈ target ∧
      req_subject(req) ↦ tag_subject(tag) ∈ match_element
      req_ressource(req) ↦ tag_ressource(tag) ∈ match_element ∧
      req_action(req) ↦ tag_action(tag) ∈ match_element
 ⇒  req ↦ tag ∈ match_target)
```

Le principe d'évaluation d'une requête par rapport à une politique dépend en premier lieu de sa cible. La machine « *mch_xacml* » comporte deux variables « *v_req* », et

« *match_pol* », pour recevoir les résultats de l'évaluation des cibles de la politique par rapport à celle de la requête.

```
variables v_req match_pol

invariants
       @inv2 v_req ∈ requete
       @inv3 match_pol ∈ policy →  {match_yes,match_no,undef}
```

Nous utilisons un exemple afin d'illustrer le processus de l'évaluation. Soit « ReqA » une requête et la liste des politiques P = {P1, P2,...Pn}. Nous utilisons l'événement d'Initialisation de la machine pour initialiser « *match_pol* » avec la valeur « *undef* ». Par la suite, si la cible de la requête « ReqA » correspond à la cible de la politique « Pi » un événement se déclenche pour changer le statut de l'évaluation de la requête : le statut reçoit la valeur « *match_yes* », sinon la valeur du statut devient « *match_No* ». La figure 4.2 ci-dessous schématise une simulation de l'évaluation:

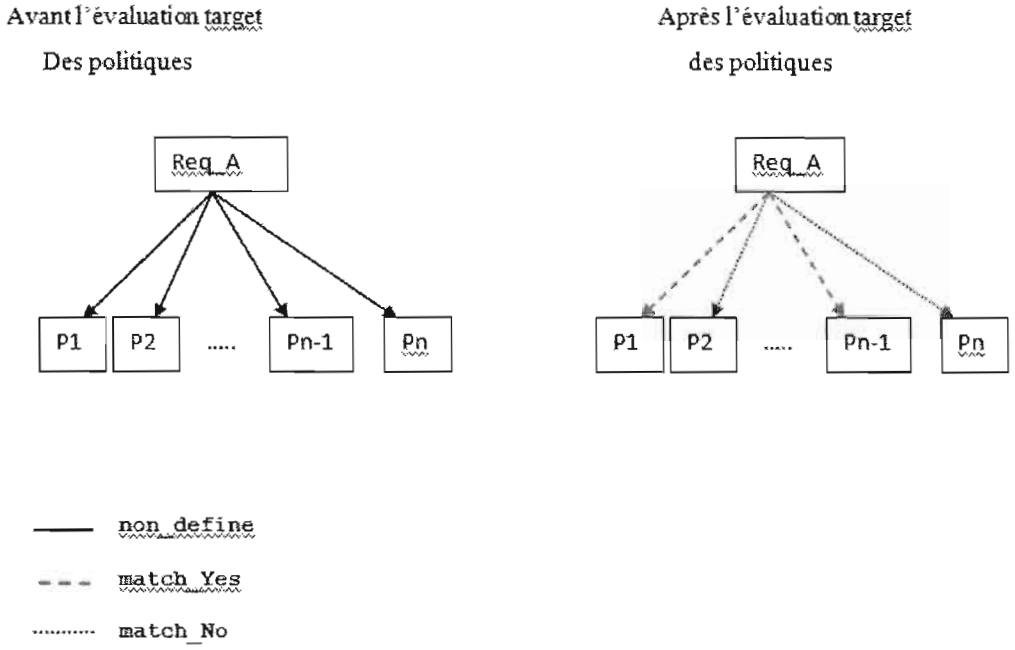

Figure 4.2 Avant et après l'évaluation de la cible des politiques

4.3.2 Décisions des règles

Nous définissons la fonction « *match_rule* » qui détermine le résultat de l'applicabilité d'une règle pour une requête. Si la fonction retourne « *match_Yes* », alors la décision sera l'effet de la règle. Sinon la réponse sera « *NotApplicable* ».

En effet, après la vérification de la cible de chaque politique par rapport à la cible de la requête « *req_A* », nous initialisons le statut de « *match_rule* » de chaque couple (policy, rule). Par la suite, nous traitons seulement les règles qui correspondent à des politiques dont la cible couvre la requête.

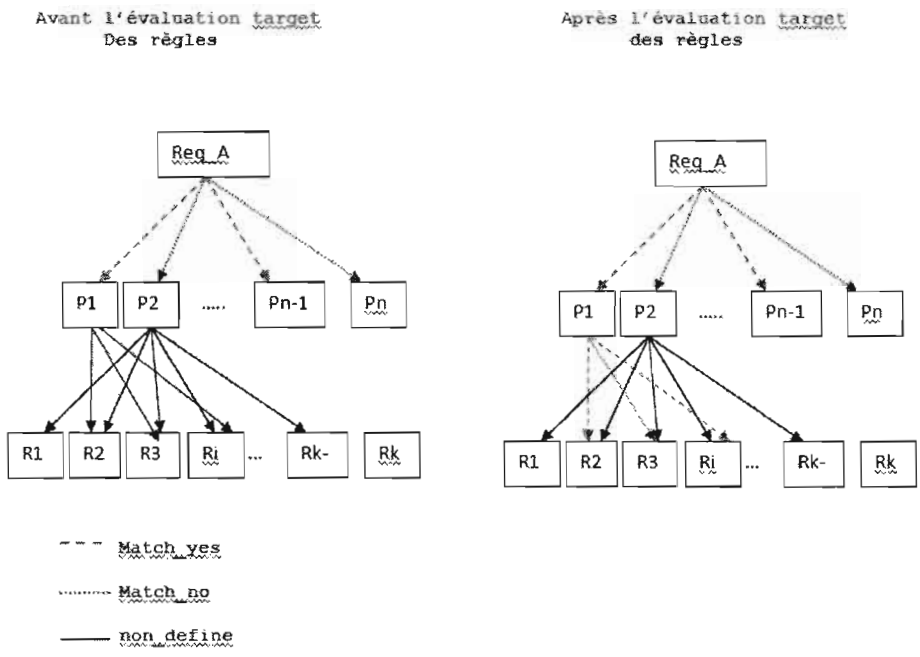

Figure 4.3 Avant et après l'évaluation la cible des règles

Toutefois, la fonction « *match_rule* » détermine la décision de l'évaluation de chaque règle de la politique par rapport à la cible de la requête :

$$match\_rule \in rules \times policy \rightarrow \{match\_yes, match\_no, undef\}$$

Dans la figure 4.3, toutes les règles vont être initialisées avec la valeur « *undef* » (qui signifie non traité). Par la suite, l'événement « *match_rule_true* » utilise les valeurs des paramètres « *rul et pol* » pour déterminer le statut selon les conditions définies dans la clause « *Where* ». L'événement déclenche une action qui permet de changer le statut (la couleur rouge ou verte), les conditions de l'événement sont :

- La cible de la requête correspond à la cible de la politique « *pol* » ;
- La règle « *rul* » appartient à l'ensemble des règles de la politique « *pol* » ;
- La valeur de **Match_rule(rul,pol)** = *undef.*

Avec le langage Event-B, ces conditions sont exprimées de la manière suivante :

```
event match_pol_true
    any pol req
    where
      @type1 pol ∈ policy
      @grd1 req ∈ requete
      @Guard1 match_pol(pol ↦ req ) = undef
      @grd3 {(req ↦ tag_pol(pol))} ⊆ match_target
    then
      @Action1 match_pol(pol ↦ req ) ≔ match_yes
  end
```

Figure 4.4 Événement d'évaluation une requête XACML (*match_pol_true*)

```
event match_pol_false
    any pol req
    where
      @type1 pol ∈ policy
      @grd1 req ∈ requete
      @Guard1 match_pol(pol ↦ req) = undef
      @grd3 req ↦ tag_pol(pol) ∉ match_target
    then
      @Action1 match_pol(pol ↦ req)≔ match_no
  end
```

Figure 4.5 Événement d'évaluation une requête XACML (*match_false*)

4.3.3 Réponse des politiques.

Après la définition des éléments de base qui bâtissent les règles décrites avec le langage XACML, il devient possible de construire des machines pour évaluer une requête donnée, par rapport à un ensemble de politiques et selon l'algorithme de combinaisons utilisées pour ces politiques.

La réponse de la politique dépend de trois éléments : la cible, les règles et l'algorithme de combinaison des politiques. 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.

En effet, nous ajoutons des événements pour contrôler et valider la décision de l'évaluation des requêtes :

```
    @grd1 pol ∈ policy
    @grd2 algo_pol(pol) = Permit_overried
    @grd3 ∀ rul·(rul ∈ rules ∧ rul ∈ ens_rul_policy(pol) ⇒
                  match_rule(rul ↦ pol) ≠ undef )
    @grd4 ∃rul·(rul ∈ rules ∧ rul ∈ ens_rul_policy(pol) ∧
                  effect_rule(rul ) = permit)
    @grd5 decision_pol(pol) = Notapplicable
then
    @act1 eval_pol(pol) ≔ match_yes
    @act2 decision_pol(pol) ≔   permit
```

Figure 4.6 Événement pour déterminer la réponse d'évaluation d'une politique

L'événement (figure 4.6) permet de générer toujours une décision pour une politique donnée « *pol* » par rapport à la requête « *req* ». Si la politique « *pol* » utilise l'algorithme de combinaison « *permit_overried* », et que dans la liste des règles de la politiques « *pol* » il existe au moins une règle qui retourne « permit » comme effet, alors la réponse de la politique sera « *permit* ». Cela s'applique aussi pour l'algorithme de combinaison des politiques « *Deny_Override* ».

Dans ce chapitre, nous avons construit un modèle de XACML basé entièrement sur la logique relationnelle de premier ordre en utilisant le langage Event-B. Event-B offre la possibilité de vérifier et d'analyser le modèle obtenu. Dans le chapitre suivant, nous allons définir encore une fois, en termes de logique de premier ordre, un ensemble de propriétés et d'interactions que nous proposons d'analyser et de vérifier.

CHAPITRE V: ANALYSE ET VÉRIFICATION DES POLITIQUES DE CONTRÔLE D'ACCÈS XACML

Une fois que le modèle des politiques de contrôle d'accès XACML est crée, il est important d'ajouter des propriétés pour analyser les politiques et vérifier la cohérence entre les règles. Le modèle décrit dans le chapitre IV permet d'évaluer les requêtes avec les changements des états des variables. Ces états sont aussi utilisés pour déclencher d'autres événements liés à la vérification de propriétés.

Nous présentons dans ce chapitre certaines propriétés à vérifier dans les politiques de contrôle d'accès. L'analyse des politiques avec des outils permet l'exploitation des avantages d'Évent-B sous la plate-forme Rodin.

5.1 Démarches d'analyse et de vérification des propriétés

Nous définissons un ensemble de relations pour analyser les interactions entre les règles et les politiques. Ce chapitre est composé de trois sections :

1. Nous proposons une méthode de génération d'un contexte Event-B. à partir du fichier des politiques XACML. Nous utilisons ce contexte dans la machine de test.
2. Nous décrivons le contexte de test qui représente le vocabulaire des politiques XACML.
3. Nous proposons les prédicats et les événements que nous avons développés pour analyser certains propriétés comme :
    - Détecter les conflits entre les règles;
    - Détecter les règles redondantes;
    - Vérifier la couverture des règles.

5.2 Extraction des attributs de politique XACML

Cette étape consiste à extraire l'ensemble des attributs qui constituent le contexte décrivant le vocabulaire des politiques du système. Avec l'utilisation du langage *XSLT*, chaque attribut dans la

politique XACML est représenté d'une manière systématique et sans ambiguïté, de sorte que la transformation soit plus facile à écrire.

À partir du schéma de la spécification XACML, nous pouvons extraire des objets structurés représentant les attributs des sujets, ressources et actions, voir annexe B (le code XSLT). En effet, pour l'application de transformation, nous utilisons en entrée le fichier contenant les politiques de contrôle d'accès en XACML.

Par la suite, l'application de transformation génère un fichier XML contenant l'ensemble des attributs utilisés dans les politiques. La structure du fichier XML est définie dans la figure 5.1.

Chaque attribut de la politique XACML est défini par les attributs suivants :

- *name* : Indique si l'attribut est un attribut de sujet, ressource ou action ;
- *attribut_name* : indique le nom de l'attribut ;
- *Attribut_valeur* : contient une liste des valeurs possibles.

La figure ci-dessous, indique comment une politique XACML doit être transformée en un autre fichier XML. Avec l'utilisation des feuilles de style XSLT, nous définissons des opérations à réaliser sur certains éléments de la politique XACML.

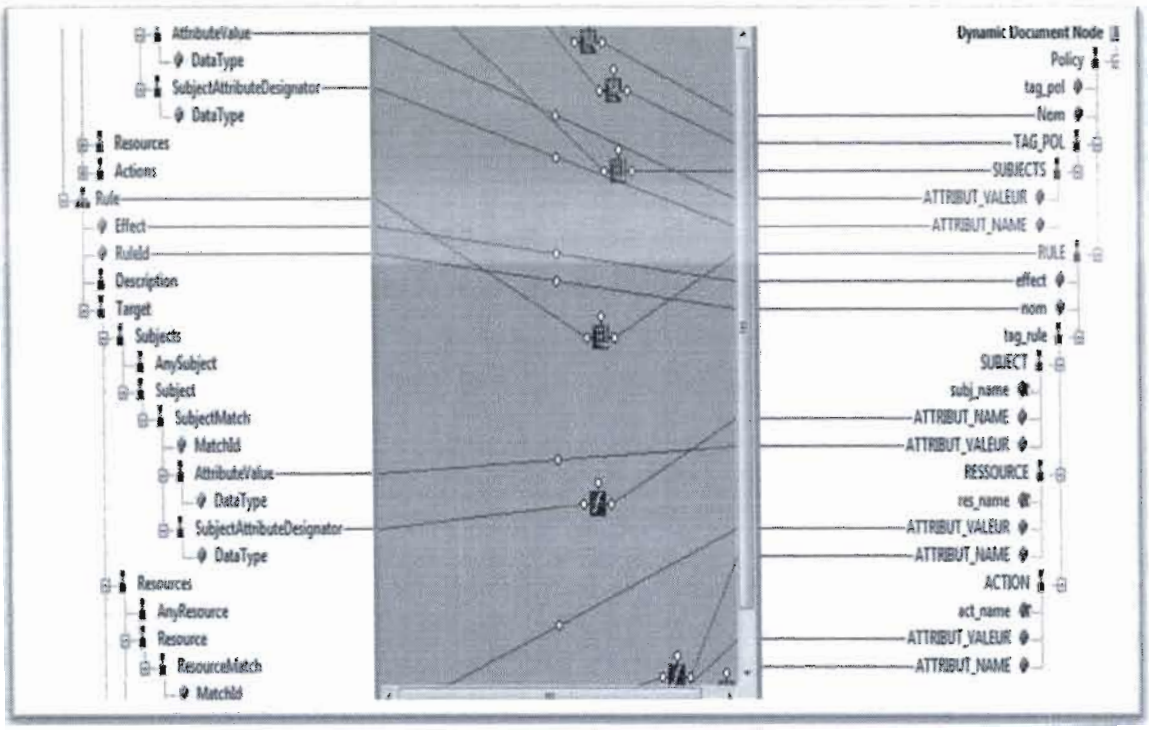

Figure 5.1 Extraction des attributs XACML

Au début, nous définissons un nœud « *policy* ». À partir de ce nœud on récupère les noms des attributs avec leurs valeurs. Par exemple, l'instruction ci-dessous indique au processeur XSLT qu'il doit extraire la valeur textuelle de l'élément **target/Subject**. Puis l'insérer entre la balise d'ouverture et de fermeture du nouvel élément **nom_attribut** qui doit figurer dans le document résultant :

<nom_attribut> **xsl:value-of select**= '**target/Subject**'/></nom_attribut>

Nous utilisons aussi des boucles en XLST, pour obtenir les différents noms et valeurs des attributs utilisés dans la politique XACML.

Le résultat de la transformation permet de générer un fichier XML qui contient l'ensemble des noms et des valeurs des attributs de la politique XACML (la figure 5.2).

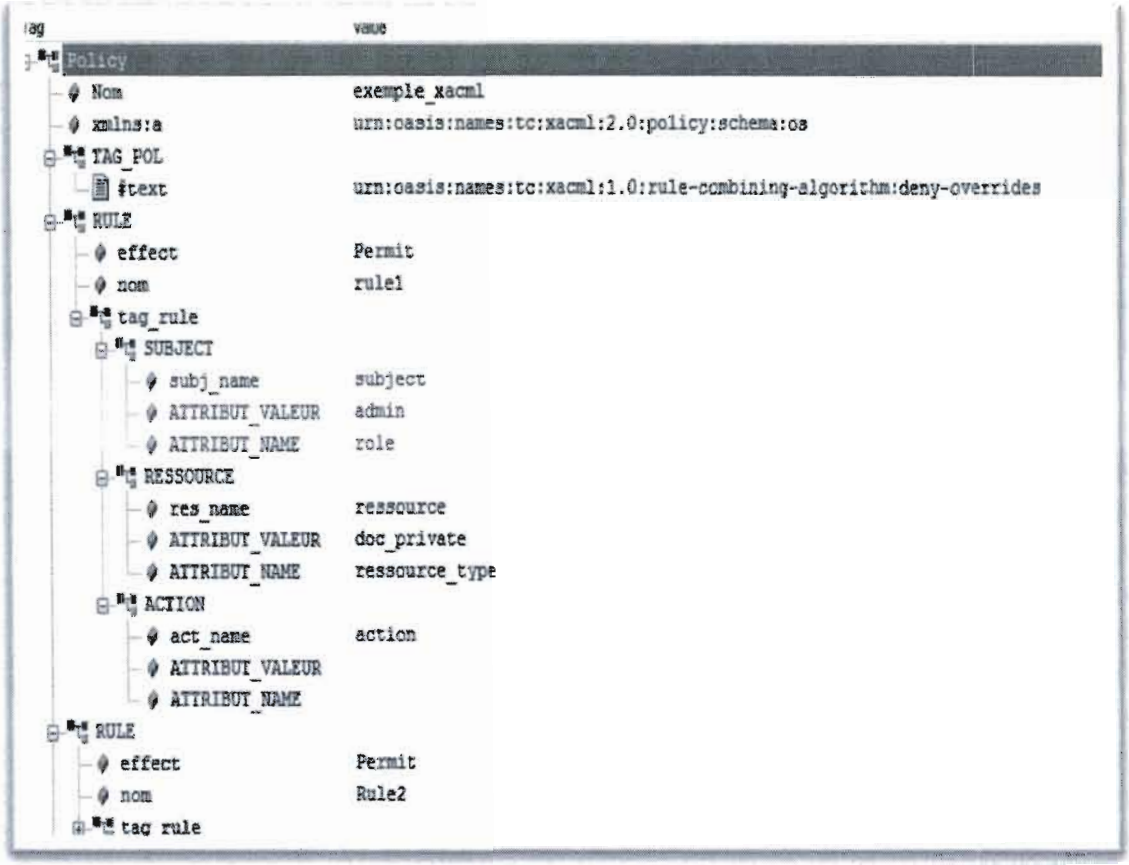

Figure 5.2 Fichier XML généré des attributs de la politique

Par la suite, nous avons développé un programme Java pour lire le document XML et construire les prédicats du contexte du modèle Event-B (voir le code Java dans l'annexe B). Dans le programme Java, nous utilisons des API DOM (Document Object Model) afin de modéliser, de parcourir et de manipuler le document XML généré. À partir du fichier XML nous générons un document texte qui contient les prédicats constituant le contexte d'Event-B.

La figure 5.3 illustre ce contexte d'Event-B :

```
constants subject1 subject2   ressource1 ressource2
  action1 action2
  role type_document  action_name // liste des attributs
  doc_private doc_public admin visiteur read write control
  target1 target2 target3
  rule1 rule2 rule3 policy1 requete1
axioms @axm2  role ∈  attribut ∧
```

```
          type_document ∈  attribut ∧
          action_name ∈  attribut
  @axm3   admin ∈  valeur ∧
          visiteur ∈  valeur ∧
          doc_public ∈  valeur  ∧
          doc_private  ∈  valeur
@axm1  {subject1,subject2 } ∈ subjects
@axm5  {ressource1 , ressource2  } ∈ ressources
@axm9  {action1 , action2} ∈ actions
@axm4   nom_attribut(subject1) = role ∧
        valeur_attribut(subject1) = { admin }
@axm42 nom_attribut(subject2) = role ∧
        valeur_attribut(subject2) = { visiteur   }
@axm7   nom_attribut(ressource1) = type_document ∧
        nom_attribut(ressource2) = type_document
@axm8   valeur_attribut(ressource1) = { doc_private ,doc_public}
@axm9   valeur_attribut(ressource2) = { doc_public}
@axm10 nom_attribut(action1) = read  ∧
        nom_attribut(action2) = write
@axm11 target1 ∈ target ∧ target2 ∈ target  ∧
        target3 ∈ target
@axm12 tag_subject(target1) = {subject1 } ∧
        tag_ressource(target1) = {ressource1} ∧
        tag_action(target1) = {action1}
@axm13 tag_subject(target2) = {subject1} ∧
        tag_ressource(target2) = {ressource2} ∧
        tag_action(target2) = {action2}
  @axm15  rule1 ∈ Rule ∧ rule2 ∈ Rule ∧ rule3 ∈ Rule
  @axm16  policy1  ∈ Policy
  @inv17 requete1 ∈ requete

end
```

Figure 5.3 Contexte de test

5.3 Analyse et vérification des politiques XACML.

Après la création des contextes et des machines qui décrivent les politiques XACML avec Event-B, et après la génération les axiomes qui représentent le vocabulaire des politiques XACML. Nous présentons dans cette section les propriétés à analyser et à vérifier avec l'outil Rodin.

La vérification consiste à démontrer la formule H $\Longrightarrow$ P. Cette formule signifie qu'il faut démontrer le but *P* sous l'hypothèse *H*. Généralement l'hypothèse *H* est une conjonction de prédicats, sachant que la vérification des opérations consiste à vérifier les invariants de la machine ainsi que sa substitution de l'initialisation.

Nous décrivons une machine de test qui contient les propriétés et les prédicats des invariants pour analyser et vérifier les politiques XACML. Cependant, l'utilisation de l'outil ProB [21] offre plusieurs fonctionnalités permettant de produire des simulations de tests, et nous permet de vérifier et d'analyser les politiques décrites avec le langage XACML.

Nous commençons d'abord avec la définition d'un ensemble de propriétés, en termes de logique de premier ordre, pour définir les différentes relations qui caractérisent les cibles dans XACML. Ensuite, nous allons analyser l'impact de ces relations sur les interactions entre les règles et les politiques, et nous ajoutons d'autres propriétés supplémentaires que nous pouvons analyser et qui sont liées au contexte des requêtes.

5.3.1 Conflit des règles.

Pour la gestion de conflit entre les règles, XACML a mis en place les algorithmes de combinaison. Un algorithme de combinaison permet de retourner toujours une décision à partir d'un ensemble de décision souvent contradictoire. Mais est-ce-que c'est la bonne ? Il n'y a aucun moyen pour en être sûr sauf si l'on fasse une validation qui pose les « bonnes » questions.

Soit R1 et R2 deux règles définies comme suit :

- **R1** : tous les clients « S1 » ne sont pas autorisés à consulter les dossiers privés « *File1* » (*effect* = *Deny*)
- **R2** : tous les clients du groupe A « S2 » sont autorisés à consulter les dossiers publics et les dossiers privés « *File2* » (*effect* = *permit*)

Les deux règles ont des effets différents, mais si nous utilisons les deux règles R1 et R2 dans une politique définie avec l'algorithme « *only-one-applicable* », ceci peut provoquer un conflit, parce que S2 ⊆ S1 et File1 ⊆ File2.

Par conséquent, pour détecter ce genre de situation, nous définissons le prédicat « *conflitrule* » (voir la figure 5.4) qui vérifie s'il existe des politiques contenant des règles dont les cibles se chevauchent et les effets sont contradictoires.

```
event conflitrule
  any pol rule1 rule2
  where
  @grd1 pol ∈ policy
  @grd2 rule1 ∈ rules
  @grd3 rule2 ∈ rules
  @grd4 {rule1 ,rule2}⊆ ens_rul_policy (pol)
  @grd6 effect_rule(rule1) ≠ effect_rule(rule2)
  @grd7 tag_rule(rule1) ↦ tag_rule(rule2) ∈ relation_target
  then
     @act1 v_rule_conflit(pol ↦ rule1 ↦ rule2) ≔ match_yes
```

Figure 5.4 Événement pour détecter les conflits entre les règles.

5.3.2 Règle redondante

Une règle R est appelée superflue lorsqu'elle est retirée de la politique définie, et le comportement du système par rapport à cette politique ne change pas.

Par exemple, pour une politique *P* qui contient trois règles *R1, R2,* et *R3*, si la décision de cette politique est toujours la même que la politique Q qui contient la règle R1 et R2, alors la règles R3 est une règle redondante :

- *Règle1* : les visiteurs ont la permission de consulter seulement les produits de type « A ».
- *Règle 2* : les clients ont la permission de consulter tous les produits.

Pour une politique contenant les deux règles R1 et R2, nous remarquons que l'intersection entre la cible de la règle R1 et R2 est non vide, ce qui présente des possibilités de redondance.

Pour détecter la redondance, nous ajoutons l'événement ***rule_redondant*** (voir extrait ci-dessous) pour vérifier si une politique contient des règles qui se chevauchent et dont les effets sont identiques.

```
event rule_redondant
  any pol rule1 rule2
  where
  @grd1 pol ∈ policy
  @grd2 rule1 ∈ rules
  @grd3 rule2 ∈ rules
  @grd4 {rule1 ,rule2}⊆ ens_rul_policy (pol)
  @grd6 effect_rule(rule1) = effect_rule(rule2)
  @grd7 tag_rule(rule1) ↦ tag_rule(rule2) ∈ relation_target
  then
     @act1 v_rule_redondant(pol ↦ rule1 ↦ rule2) := 0
end
```

Figure 5.5 Événement pour vérifier la redondance entre les règles

5.3.3 Autres propriétés

Nous considérons deux politiques **P** et **Q** incompatibles si la décision de la politique P est différente de celle de Q pour l'ensemble des requêtes. De même, le prédicat « *IncompatibleRules* » permet de vérifier si une politique contient deux règles contradictoires pour une requête particulière.

En particulier, on vérifie si une politique « *P* » contient deux règles R1 et R2 pour lesquelles il existe une requête dont les réponses de ces règles sont contradictoires.

```
event incompatiblerule
  any pol rule1 rule2
  where
  @grd1 pol ∈ policy
  @grd2 rule1 ∈ rules
  @grd3 rule2 ∈ rules
  @grd4 {rule1 ,rule2}⊆ ens_rul_policy (pol)
  @grd6 effect_rule(rule1) ≠ effect_rule(rule2)
  @grd7 tag_rule(rule1) ↦ tag_rule(rule2) ∈ relation_target
  then
     @act1 v_rule_incompatible(pol ↦ rule1 ↦ rule2) :=  match_yes
  end
```

Figure 5.6 Événement pour détecter les règles incompatibles

Il faut noter que d'autres propriétés permettent d'analyser les politiques. L'événement (figure 5.7 et 5.8) permet de vérifier si une politique **P** ne retourne jamais la décision « *Permit* » ou

« *Deny* ». Avec ces deux événements, nous pouvons déterminer l'ensemble des règles nécessaires et applicables par rapport à une requête.

```
event pol_permit_for_all_req
  any pol
  where
  @grd1 pol ∈ policy
  @grd2 ∀ req·(req ∈ requete ∧ decision_pol(pol ↦ req) = permit)
  @grd3 pol ∈ list_pol_permit
  then
     @act1 list_pol_permit  :=  list_pol_permit  ∪ {pol}
   End
```

Figure 5.7 Événement pour détecter les politiques positives

```
event pol_deny_for_all_req
  any pol
  where
  @grd1 pol ∈ policy
  @grd2 ∀ req·(req ∈ requete ∧ decision_pol(pol ↦ req) = deny)
  @grd3 pol ∈ list_pol_deny
  then
     @act1 list_pol_deny :=  list_pol_deny  ∪ {pol}
   end
```

Figure 5.8 Événement pour détecter les politiques négatives

5.4 Exemple de vérification des politiques XACML

Nous proposons dans cette section un exemple pour montrer certains résultats des analyses des preuves d'obligations proposées dans ce chapitre. Nous allons considérer l'exemple de l'annexe B :

- **Régle 1** : les visiteurs ont le droit de consulter les documents public ;
- **Règle 2** : les utilisateurs du group B ont le droit de consulter les documents public ;
- **Règle 3** : les utilisateurs du groupe A n'ont pas le droit de consulter les documents public ;
- **Règle 4** : l'administrateur a le droit de modifier les documents public et privé.

Afin de vérifier les obligations de preuve, nous avons besoin d'ajouter une série de nouveaux invariants au modèle. Ces invariants sont décrits dans la machine « *mch_analyse_policy* ».

Nous allons interroger les spécifications en ProB sur la possibilité de détecter des conflits entre deux règles de la politique. L'événement « conflitrule » figure 5.4 permet de détecter si deux règles sont conflictuelles.

En effet, la règle1 autorise aux visiteurs de consulter les documents public, alors que la règle3 défend aux personnes identifiée par le groupe A d'accéder aux locaux. Les deux cibles se chevauchent puisqu'un visiteur peut appartenir au groupe A. Le résultat de la vérification ci-dessous est générée automatiquement par ProB et il montre que la règle rulex1 et rulex3 sont en conflit.

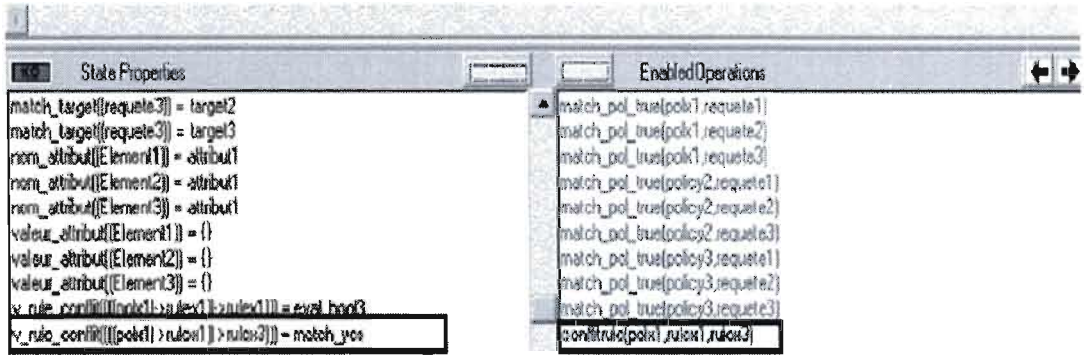

Figure 5.9 la règle *rulex1* et rulex2 sont en conflit

- **Détection des règles redondantes**

L'analyse de l'événement « rule_redondant » permet de détecter, comme le montre la figure 5.10, que les règles 1 et 2 causent une redondance. La règle 1 permet à tous les visiteurs de consulter les documents publics, et la règle 2 permet aux clients du groupe B de consulter les documents publics. La cible de la règle 2 chevauche avec la cible de la règle 1 et les deux règles ont le même effet.

Le résultat de la vérification ci-dessous montre que la règle rulex1 et rulex2 sont en redondance.

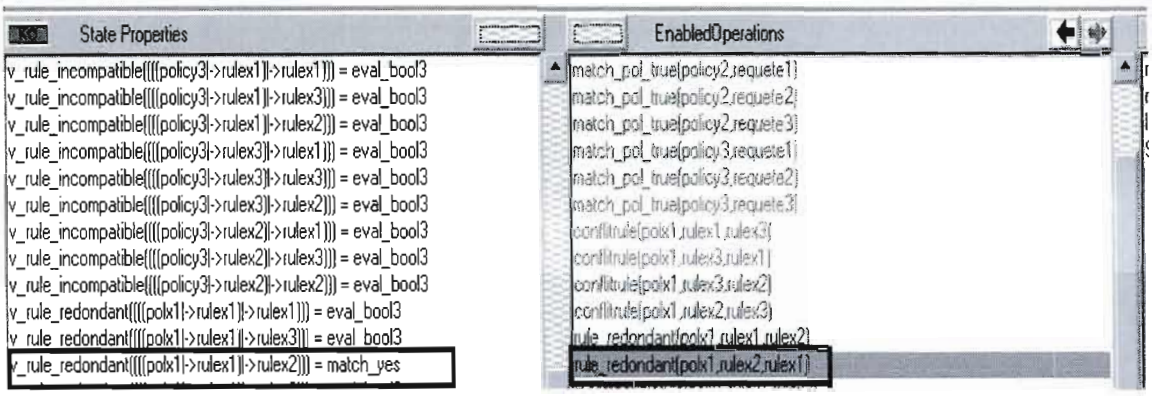

Figure 5.10 la règle *rulex1* et rulex2 sont en redondance

Dans notre mémoire nous avons traité certaines propriétés de vérification des politiques XACML, les questions suivantes ont toujours restée ouvertes :

- Quelle est la complexité de vérification d'une requête d'accès avec le langage XACML ?
- Si la vérification d'une requête d'accès est intraitable par des politiques XACML, qui sont les sous-ensembles qui font une formule logique pour traiter la requête? Comment déterminer les caractéristiques qui conduisent à intraitable?
- Comment classifier les politiques et les règles dans des clusters de politiques et la politique selon des techniques de réorganisation ?

CONCLUSION

Le sujet de ce mémoire ne manque pas d'intérêt car, l'évolution rapide des systèmes d'information rend le défi de la sécurité plus complexe. La sécurité ne se limite pas au cryptage de données ou au contrôle d'accès d'une seule plate-forme, mais à la protection de l'ensemble du système.

Ce mémoire propose un cadre formel pour décrire et analyser les politiques du langage XACML. Ce cadre est essentiellement composé d'un noyau d'expression avec le langage Event-B sur la base des spécifications des politiques XACML.

Dans la première partie de ce travail, nous avons proposé une vue générale des politiques, les méthodes utilisées pour exprimer et vérifier les politiques en matière de sécurité telles que le contexte du contrôle d'accès, le contrôle de l'utilisation et les obligations. En outre, nous avons décrit les différents concepts de vérification des politiques comme la technique de couverture et la méthode de faille.

Dans la deuxième partie, nous avons proposé une approche pour analyser les politiques XACML basée sur la modélisation formelle et la logique des prédicats de premier ordre.

Le travail a démontré les possibilités de valider et vérifier les politiques XACML en utilisant la modélisation formelle. Grâce à ce travail, des failles de sécurité et des interruptions de service peuvent être identifiées et évitées. Ce travail a permis d'apporter les **contributions** suivantes :

- Identifier les concepts fondamentaux pour les politiques XACML, et représenter les éléments de base, qui modélisent les politiques sous forme d'expressions logiques; et proposer un modèle logique des politiques XACML en utilisant le langage Event-B ;
- Transformer des politiques XACML vers les machines abstraites de la méthode B ;

- Développer un processus inclut une méthode pour construire une chaîne de raffinement d'Event-B à partir de modèles XACML ;
- Ajouter des prédicats qui permettent de vérifier des propriétés pour détecter le conflit et la redondance entre les règles de la politique ;

- **Les limites de notre travail et les perspectives de recherche** concernent les aspects suivants : Nous n'avons traité que les fonctions simples d'égalité alors que XACML propose un ensemble de fonctions traitant plusieurs types de données. Dans ce mémoire, nous avons restreint le domaine d'analyse pour se concentrer uniquement sur les fonctions d'égalité de ce langage.
- Nous n'avons pas traité les attributs de l'environnement comme les dates et l'heure.
- Enfin, les obligations qui sont des actions que le système fait suite à une requête XACML (comme une modification des données du système suite à une requête), ne sont pas considérées.

La suite de ce travail pourrait être **l'extension du modèle** présenté, afin de prendre en considération les obligations de politiques et les autres fonctions utilisées par le langage XACML.

ANNEXE A

```
1.   context ctx_xacml
       //  dans ce context nous  definissons l'ensemble des types et
  les
           constants utilisées
       // dans les politiques XACML. la clause sets contient les types
          abstraits , dans cette liste
       // on considere le type Element comme un type abstrait , ou tous
          les attributs de subject , ressource , et action
       // appartient au type Element.et la clause constants représente
         l'ensemble des valeurs et les relations qui
       // permettent de déterminer les valeurs des attributs


       constants subject ressource action subjects ressources actions
                 permit deny Notapplicable
                 Permit_overried Deny_overried First_one_applicable
                 nom_attribut valeur_attribut
                 requete1
                 match_target match_element

       sets Element attribut  valeur effect algorithms
            requete target rules policy

       axioms
         @axm7 nom_attribut  ∈  Element → attribut
         @axm8 valeur_attribut ∈ Element → ℙ(valeur)
         @axm1 subject ∈ ℙ(Element)
         @axm2 subjects ⊆  ℙ(Element)
         @axm3 ressource ∈ ℙ(Element)
         @axm4 ressources ⊆ ℙ(Element)
         @axm5 action ∈ ℙ(Element)
         @axm6 actions ⊆  ℙ(Element)
         @axm10 partition(Element,subject,ressource,action)
         // la fonction partition permet de préciser  que l'ensemble
  Element est un union de subject , ressource
         // et action et que l'intersection entre subject et ressource
  est un ensemble vide , la même chose
         // pour subject et action , ainsi ressource et action
         @axm11 partition(effect,{permit},{deny},{Notapplicable})
         @axm21                  partition(algorithms,{Permit_overried},
  {Deny_overried}, {First_one_applicable})
         @axm18 requete1 ∈ requete
```

```
      // la relation match_element permet de determiner l'ensemble
des couples qui verifie certain conditions;
      // on dit que (ele ,ens_ele) : match_lement s'il existe un
element ele1 dans l'ensemble ens_ele et que
      //  le nom de l'attribut de ele a le mème nom d'attributt de
ele1 et les valeurs des attributs de
      // ele sont inclu dans la liste des  valeurs de ele1
      @axm17 match_element = Element  × ℙ(Element)
      @axm19 match_target =   requete × target
      @acm20 ∀ ele,ens_ele·(ele ∈ Element ∧ ens_ele ∈ ℙ(Element) ∧
            (∃ ele1·(ele1 ∈       ens_ele ∧ nom_attribut(ele) =
nom_attribut(ele1) ∧ valeur_attribut(ele) ⊆ valeur_attribut(ele1)))
            ⇒ (ele ↦ ens_ele) ∈ match_element)
    End
```

2. context ctx_target

```
    // le contexte ctx_target est une extension du context
ctx_xacml. ctx_target est utilisé pour la machine mch_policy
    // dance ce contexte on définit l'ensemble des relations qui
definit la cible de la politique et des régles

    extends ctx_xacml
    constants tag_subject tag_ressource tag_action
              req_subject req_ressource req_action
              tag_req tag_pol tag_rule
              ens_rul_policy effect_rule algo_pol

    axioms
      @axm1 tag_subject ∈ target → subjects
      @axm2 tag_ressource ∈ target → ressources
      @axm4 tag_action ∈ target → actions
      @axm5 req_subject ∈ requete → subject
      @axm6 req_ressource ∈ requete → ressource
      @axm7 req_action ∈ requete → action
      @axm8 tag_req ∈ requete → target
      @axm9 tag_pol ∈ policy → target
      @axm10 tag_rule ∈ rules → target
      @axm13 ens_rul_policy ∈ policy →  ℙ(rules)
      // de la meme façon on definit les conditions qui determinent
les couples qui appartient à
      // l'ensemble match_target
      @axm11  ∀     req,tag·(req  ∈  requete  ∧  tag  ∈  target  ∧
req_subject(req) ↦ tag_subject(tag) ∈ match_element ∧
            req_ressource(req)      ↦      tag_ressource(tag)      ∈
match_element ∧
            req_action(req) ↦ tag_action(tag) ∈ match_element
            ⇒  req ↦ tag ∈ match_target)
```

```
    @axm12 effect_rule ∈ rules → effect
    @axm15 algo_pol ∈ policy → algorithms
End
```

3. context ctx_rule

```
    // dans ce context on définit l'ensemble des relations et des
critères qui permt de déterminer la liste des règles et l'ordre
sur lequel sont représentés dans la politique XACML
    // pour cela nous définissons deux relations nombre_rule pour
déterminer le nombre des règles qui constitue la politique ,
seq_rule qui permet de déterminer d'identifier le nom de la regle
dans chaque position
    // par exemple si la politiquer P est constitue de
(ruleA,ruleB,ruleC,ruleD) alors nombre_rule(P) = 4 et
    // seq_rule(P,1) = ruleA ,seq_rule(P,2) = ruleB,seq_rule(P,3)
=ruleC et seq_rule(P,4)= ruleD

    extends ctx_target

    constants seq_rule nombre_rule

    axioms
      @axm24 nombre_rule ∈ policy ⇸ ℕ
      @axm2 ∀ pol·(pol ∈ policy ⇒ pol ∈ dom(nombre_rule))
      @axm1 seq_rule ∈ policy × ℕ → rules
      @axm11 finite(seq_rule)
      @axm22 finite(nombre_rule)
      // pour specifier que pour chaque regle definit dans sequence
est inclu dans la liste des rgles de la politique

      @axm3 ∀ pol,i·(pol ∈ policy ∧ pol ∈ dom(nombre_rule) ∧ i ∈ ℕ ∧
pol↦ i ∈ dom(seq_rule) ⇒ seq_rule(pol↦ i) ∈ ens_rul_policy(pol) )
      // pour specifier que dans chaque position i , avec i ≤
nombre des règle de P il existe une regle R et seq_(p,i) = R
      @axm23 ∀ pol,i·(pol ∈ policy ∧ pol ∈ dom(nombre_rule)
              ∧ i ∈ ℕ ∧ i ≤ nombre_rule(pol)
          ⇒ pol↦ i ∈ dom(seq_rule)
              ∧ (∃ rul·(rul ∈ rules ∧ seq_rule(pol↦ i) = rul)))

      @axm25 ∀ pol,i·(pol ∈ policy
                    ∧ i ∈ ℕ ∧ pol ∈ dom(nombre_rule)
                    ∧ i > nombre_rule(pol)
              ⇒ pol↦ i ∉ dom(seq_rule))
    // pour specifier que chaque position dans la politique XACML P
contient une seule r'egle, nous ajoutons axiome 26
      @axm26 ∀ pol,i,j·(pol ∈ policy ∧ i ∈ ℕ
```

```
                    ∧ pol ∈ dom(nombre_rule)
                    ∧ i ≤   nombre_rule(pol)
                    ∧  j ≤   nombre_rule(pol)
        ⇒  (seq_rule(pol↦ i) =seq_rule(pol↦ j)  ⇔  i = j) )
End
```

4. machine mch_xacml

```
sees ctx_target

variables v_req match_pol

invariants
  @inv2 v_req ∈ requete
  @inv3 match_pol ∈ policy →  {match_yes,match_no,undef}

events
  event INITIALISATION
    then
      @init2 v_req ≔ requete1
      @init3 match_pol  ≔  policy × {undef}
  end

  event match_pol_true
    any pol
    where
      @type1 pol ∈  policy
      @Guard1 match_pol(pol) = undef
      @grd3 {(v_req ↦ tag_pol(pol))} ⊆ match_target
    then
      @Action1 match_pol(pol) ≔ match_yes
  end

  event match_pol_false
    any pol
    where
      @type1 pol ∈  policy
      @Guard1 match_pol(pol) = undef
      @grd3 v_req ↦ tag_pol(pol) ∉ match_target
    then
      @Action1 match_pol(pol)≔  match_no
  end
end
```

5. machine mch_rule

```
refines mch_xacml
```

```
sees ctx_target

variables v_req match_pol match_rule

invariants
 @inv1 match_rule ∈ rules × policy → {match_yes,match_no, undef }

events
  event INITIALISATION extends INITIALISATION
    then
      @init4 match_rule ≔ rules × policy × { undef }
  end

  event match_rul_true
    any rul pol
    where
      @grd1 rul ∈ rules
      @grd2 pol ∈ policy
      @grd3 rul ∈ ens_rul_policy(pol)
      @grd4 match_pol(pol) = match_yes
      @grd5 {(v_req ↦ tag_rule(rul))} ⊆ match_target
      @Grd6 match_rule(rul ↦ pol) = undef
    then
      @Action1 match_rule(rul ↦ pol) ≔ match_yes
  end

  event match_rul_false
    any rul pol
    where
      @grd1 rul ∈ rules
      @grd2 pol ∈ policy
      @grd3 rul ∈ ens_rul_policy(pol)
      @grd4 match_pol(pol) = match_yes
      @grd5 {(v_req ↦ tag_rule(rul))} ⊆ match_target
      @Grd6 match_rule(rul ↦ pol) = undef
    then
      @Action1 match_rule(rul ↦ pol) ≔ match_no
  end

  event match_pol_true extends match_pol_true
  end

  event match_pol_false extends match_pol_false
  end
end
```

6. machine mch_policy

```
refines mch_rule
sees ctx_target

variables v_req match_pol match_rule eval_pol decision_pol

invariants
  @inv5 eval_pol ∈ policy → { undef,match_no,match_yes}
  @inv6 decision_pol ∈ policy  → effect
  @inv7 finite(decision_pol)

events
  event INITIALISATION extends INITIALISATION
    then
      @init5 eval_pol ≔  policy × { undef }
      @init6 decision_pol ≔  policy × {Notapplicable}
  end

  event match_rul_true extends match_rul_true
  end

  event match_rul_false extends match_rul_false
  end

  event match_pol_true extends match_pol_true
  end

  event match_pol_false extends match_pol_false
  end

  event eval_policy_permit_true
    any pol
    where
      @grd1 pol ∈ policy
      @grd2 algo_pol(pol) = Permit_overried
      @grd3 ∀ rul·(rul ∈ rules ∧ rul ∈ ens_rul_policy(pol) ⇒
match_rule(rul) ≠ -1 )
      @grd4 ∃rul·(rul ∈ rules ∧ rul ∈ ens_rul_policy(pol) ∧
effect_rule(rul) = permit)
      @grd5 decision_pol(pol) = Notapplicable
    then
      @act1 eval_pol(pol) ≔ 0
      @act2 decision_pol(pol) ≔  permit
  end

  event eval_policy_deny_true
    any pol
    where
      @grd1 pol ∈ policy
      @grd2 algo_pol(pol) = Deny_overried
      @grd3 ∀ rul·(rul ∈ rules ∧ rul ∈ ens_rul_policy(pol) ⇒
match_rule(rul) ≠ -1 )
```

```
          @grd4 ∃rul·(rul ∈ rules ∧ rul ∈ ens_rul_policy(pol) ∧
effect_rule(rul) = deny)
        then
          @act1 eval_pol(pol) ≔ 0
          @act2 decision_pol(pol) ≔   deny
      end
    end
```

```
7.  machine match_first_applicable
      refines mch_policy  sees ctx_rule

    variables v_req match_pol match_rule eval_pol decision_pol

    events
      event INITIALISATION extends INITIALISATION
      end

      event match_rul_true extends match_rul_true
      end

      event match_rul_false extends match_rul_false
      end

      event match_pol_true extends match_pol_true
      end

      event match_pol_false extends match_pol_false
      end

      event eval_policy_permit_true extends eval_policy_permit_true
      end

      event eval_policy_deny_true extends eval_policy_deny_true
      end

      event eval_first_app
        any pol rul
        where
          @grd1 pol ∈ policy
          @grd4 rul ∈ rules
          @grd2 algo_pol(pol) = First_one_applicable
          @grd3 ∃ i·(i ∈ ℕ ∧  i ≤ nombre_rule(pol) ∧
                     seq_rule(pol ↦ i) = rul ∧
                     (∀ j·(j ∈ ℕ ∧ j < i ∧ match_rule(seq_rule(pol ↦ j))=
1)))
        then
          @act1 eval_pol(pol) ≔ 0
          @act2 decision_pol(pol) ≔  effect_rule(rul)
      end
    end
```

8 mch_analyse_policy

```
machine mch_analyse_policy refines mch_policy  sees ctx_analyse

variables v_req match_pol match_rule eval_pol decision_pol
          v_rule_conflit v_rule_redondant list_pol_deny list_pol_permit
v_rule_incompatible
invariants
    @inv1    v_rule_conflit ∈  policy  × rules ×  rules → eval_bool
    @inv10   v_rule_incompatible  ∈  policy  × rules ×  rules → eval_bool
    @inv2    v_rule_redondant  ∈  policy  × rules ×  rules → eval_bool
    @inv4    list_pol_deny  ⊆  policy
    @inv5    list_pol_permit ⊆  policy
    @inv6    list_pol_permit ∩ list_pol_deny = ∅
    @inv7    ∃ pol1,req1 ·(pol1 ∈ policy  ∧  req1 ∈ requete ∧ pol1 ∈
list_pol_deny ⇒ effect_pol(pol1↦ req1) = permit)
    @inv8    ∃ pol1,req1 ·(pol1 ∈ policy  ∧  req1 ∈ requete ∧ pol1 ∈
list_pol_permit ⇒ effect_pol(pol1↦ req1) = deny)
    @inv9    ∃ pol1,req1 ·(pol1 ∈ policy  ∧  req1 ∈ requete ∧ pol1 ∈
list_pol_permit ⇒ effect_pol(pol1↦ req1) = deny)
events
  event INITIALISATION extends INITIALISATION
  then
  @inita1  v_rule_conflit       :=   policy  × rules ×  rules × {undef}
  @inita2  v_rule_redondant :=   policy  × rules ×  rules × {undef}
  @inita3  list_pol_permit :=  ∅
  @inita4  list_pol_deny :=  ∅
  end

  event match_rul_true extends match_rul_true
  end
  event match_rul_false extends match_rul_false
  end
  event match_pol_true extends match_pol_true
  end
  event match_pol_false extends match_pol_false
  end
  event eval_policy_permit_true extends eval_policy_permit_true
  end
  event eval_policy_deny_true extends eval_policy_deny_true
  end
  event conflitrule
  any pol rule1 rule2
  where
  @grd1 pol ∈ policy
  @grd2 rule1 ∈ rules
  @grd3 rule2 ∈ rules
  @grd4 {rule1 ,rule2}⊆ ens_rul_policy (pol)
  @grd6 effect_rule(rule1) ≠ effect_rule(rule2)
```

```
@grd7 tag_rule(rule1) ↦ tag_rule(rule2) ∈ relation_target
then
   @act1 v_rule_conflit(pol ↦ rule1 ↦ rule2) ≔  match_yes
 end

event incompatiblerule
any pol rule1 rule2
where
@grd1 pol ∈ policy
@grd2 rule1 ∈ rules
@grd3 rule2 ∈ rules
@grd4 {rule1 ,rule2}⊆ ens_rul_policy (pol)
@grd6 effect_rule(rule1) ≠ effect_rule(rule2)
@grd7 tag_rule(rule1) ↦ tag_rule(rule2) ∈ relation_target
then
   @act1 v_rule_incompatible(pol ↦ rule1 ↦ rule2) ≔  match_yes
 end

event rule_redondant
any pol rule1 rule2
where
@grd1 pol ∈ policy
@grd2 rule1 ∈ rules
@grd3 rule2 ∈ rules
@grd4 {rule1 ,rule2}⊆ ens_rul_policy (pol)
@grd6 effect_rule(rule1) = effect_rule(rule2)
@grd7 tag_rule(rule1) ↦ tag_rule(rule2) ∈ relation_target
then
   @act1 v_rule_redondant(pol ↦ rule1 ↦ rule2) ≔ match_yes
 end

event pol_permit_for_all_req
any pol
where
@grd1 pol ∈ policy
@grd2 ∀ req·(req ∈ requete ∧ decision_pol(pol ↦ req) = permit)
@grd3 pol ∈ list_pol_permit
then
   @act1 list_pol_permit  ≔  list_pol_permit  ∪ {pol}
 end

event pol_deny_for_all_req
any pol
where
@grd1 pol ∈ policy
@grd2 ∀ req·(req ∈ requete ∧ decision_pol(pol ↦ req) = deny)
@grd3 pol ∈ list_pol_deny
then
   @act1 list_pol_deny ≔  list_pol_deny  ∪ {pol}
 end
```

ANNEXE B

- **Code XSLT**

```
<?xml version='1.0' ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:a="urn:oasis:names:tc:xacml:2.0:policy:schema:os">
  <xsl:output method="xml"/>
  <xsl:template match="/">
    <Policy>
      <xsl:attribute name="Nom">
        <xsl:value-of select="Policy/@PolicyId"/>
      </xsl:attribute>
      <TAG_POL>
        <xsl:value-of select="Policy/@RuleCombiningAlgId"/>
        <xsl:for-each select="a:Policy/a:Target/a:Subjects/a:Subject">
          <SUBJECTS>
            <xsl:attribute name="ATTRIBUT_VALEUR">
              <xsl:value-of select="a:SubjectMatch/a:AttributeValue"/>
            </xsl:attribute>
            <xsl:attribute name="ATTRIBUT_NAME">
              <xsl:value-of select="a:SubjectMatch/a:SubjectAttributeDesignator"/>
            </xsl:attribute>
          </SUBJECTS>
        </xsl:for-each>
      </TAG_POL>
      <xsl:for-each select="a:Policy/a:Rule">
        <RULE>
          <xsl:attribute name="effect">
            <xsl:value-of select="@Effect"/>
          </xsl:attribute>
          <xsl:attribute name="nom">
            <xsl:value-of select="@RuleId"/>
          </xsl:attribute>
          <tag_rule>
            <SUBJECT subj_name="subject">
              <xsl:attribute name="ATTRIBUT_VALEUR">
                <xsl:value-of
select="a:Target/a:Subjects/a:Subject/a:SubjectMatch/a:AttributeValue"/>
              </xsl:attribute>
              <xsl:attribute name="ATTRIBUT_NAME">
```

```
                    <xsl:value-of
select="string(a:Target/a:Subjects/a:Subject/a:SubjectMatch/a:SubjectAttributeDesignator)"/>
                  </xsl:attribute>
                </SUBJECT>
                <RESSOURCE res_name="ressource">
                  <xsl:attribute name="ATTRIBUT_VALEUR">
                    <xsl:value-of
select="a:Target/a:Resources/a:Resource/a:ResourceMatch/a:AttributeValue"/>
                  </xsl:attribute>
                  <xsl:attribute name="ATTRIBUT_NAME">
                    <xsl:value-of
select="string(a:Target/a:Resources/a:Resource/a:ResourceMatch/a:ResourceAttributeDesignator)"/>
                  </xsl:attribute>
                </RESSOURCE>
                <ACTION act_name="action">
                  <xsl:attribute name="ATTRIBUT_VALEUR">
                    <xsl:value-of select="a:Target/a:Actions/a:ActionMatch/a:AttributeValue"/>
                  </xsl:attribute>
                  <xsl:attribute name="ATTRIBUT_NAME">
                    <xsl:value-of
select="a:Target/a:Actions/a:ActionMatch/a:ResourceAttributeDesignator/@AttributeId"/>
                  </xsl:attribute>
                </ACTION>
              </tag_rule>
            </RULE>
          </xsl:for-each>
        </Policy>
      </xsl:template>
</xsl:stylesheet>
```

- **Programme Java**

```
    static void listNodes(Node node, String indent) {
        String nodeName = node.getNodeName();
        //System.out.println(indent + nodeName + " Node, type is " +
node.getClass().getName() + ":");
        // System.out.println(indent + " " + node);

        if (node instanceof Element && node.hasAttributes()) {
            // System.out.println(indent + "Element Attributes are:");
          NamedNodeMap attrs = node.getAttributes();
          for (int i = 0; i < attrs.getLength(); i++) {
            Attr attribute = (Attr) attrs.item(i);

            if(attribute.getName()=="ATTRIBUT_NAME"&    attribute.getValue()
!= "")
            {
            System.out.print(indent + attribute.getName() + "("
                + attribute.getValue() + ")=");
            }
```

```
            if                          (attribute.getName()=="ATTRIBUT_VALEUR"&
attribute.getValue() != "")
            {
            System.out.println(    attribute.getValue());
            }
          }
```

- **Fichier XML d'extraction des attributs**

```
<Policy Nom="" xmlns:a="urn:oasis:names:tc:xacml:2.0:policy:schema:os">
<TAG_POL/>
<RULE effect="Permit" nom="rule1">
<tag_rule>

<SUBJECT subj_name="subject" ATTRIBUT_VALEUR="admin" ATTRIBUT_NAME="role"/>
<RESSOURCE res_name="ressource" ATTRIBUT_VALEUR="doc_private" ATTRIBUT_NAME="ressource_type "/>
<ACTION act_name="action" ATTRIBUT_VALEUR="" ATTRIBUT_NAME=""/>
</tag_rule>
</RULE>
<RULE effect="Permit" nom="Rule2">
<tag_rule>
<SUBJECT subj_name="subject" ATTRIBUT_VALEUR="" ATTRIBUT_NAME=""/>
<RESSOURCE res_name="ressource" ATTRIBUT_VALEUR="doc_public" ATTRIBUT_NAME="ressource_type "/>
<ACTION act_name="action" ATTRIBUT_VALEUR="CONSULTER" ATTRIBUT_NAME="action2"/>
</tag_rule>
</RULE>
<RULE effect="Deny" nom="Rule3"><tag_rule>
<SUBJECT subj_name="subject" ATTRIBUT_VALEUR="" ATTRIBUT_NAME=""/>
<RESSOURCE res_name="ressource" ATTRIBUT_VALEUR="doc_public" ATTRIBUT_NAME="ressource_type "/>
<ACTION act_name="action" ATTRIBUT_VALEUR="WRITE" ATTRIBUT_NAME="action3"/>
</tag_rule>
</RULE>
<RULE effect="DENY" nom="Rule4">
<tag_rule>
<SUBJECT subj_name="subject" ATTRIBUT_VALEUR="" ATTRIBUT_NAME=""/>
<RESSOURCE res_name="ressource" ATTRIBUT_VALEUR="" ATTRIBUT_NAME=""/>
<ACTION act_name="action" ATTRIBUT_VALEUR="" ATTRIBUT_NAME=""/>
</tag_rule>
</RULE>
</Policy>
```

```
Document root element "Policy", must match DOCTYPE root "null".
   ATTRIBUT_NAME(role)=admin
   ATTRIBUT_NAME(ressource_type )=doc_private
   ATTRIBUT_NAME(ressource_type )=doc_public
   ATTRIBUT_NAME(action2)=CONSULTER
   ATTRIBUT_NAME(ressource_type )=doc_public
   ATTRIBUT_NAME(action3)=WRITE
```

ANNEXE C

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
PolicyId="Policy Test"
RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
algorithm:permit-overrides">
  <Description>Politique de controle d'accÃ©es au document</Description>
  <Target>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-
equal">
          <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">admin</AttributeValue>
          <SubjectAttributeDesignator AttributeId="Role"
DataType="http://www.w3.org/2001/XMLSchema#string" />
        </SubjectMatch>
      </Subject>
      <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-
equal">
          <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">manager</AttributeValue>
          <SubjectAttributeDesignator AttributeId="role"
DataType="http://www.w3.org/2001/XMLSchema#string" />
        </SubjectMatch>
      </Subject>
    </Subjects>
    <Resources>
      <Resource>
        <ResourceMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">doc_private</AttributeVal
ue>
          <ResourceAttributeDesignator AttributeId="document" />
        </ResourceMatch>
      </Resource>
    </Resources>
    <Actions>
      <AnyAction />
    </Actions>
  </Target>
  <Rule Effect="Permit" RuleId="rule1">
    <Description>l'utilisateur a le droit de modifier les fichier des
promotions</Description>
    <Target>
```

```
      <Subjects>
        <Subject>
          <SubjectMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">admin</AttributeValue>
            <SubjectAttributeDesignator AttributeId="role"
DataType="http://www.w3.org/2001/XMLSchema#string" />
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <ResourceMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">docu_private</AttributeVa
lue>
            <ResourceAttributeDesignator AttributeId="document" />
          </ResourceMatch>
        </Resource>
      </Resources>
      <Actions>
        <AnyAction />
      </Actions>
    </Target>
  </Rule>
  <Rule Effect="Permit" RuleId="rule2">
    <Target>
      <Subjects>
        <Subject>
          <SubjectMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">manager</AttributeValue>
            <SubjectAttributeDesignator AttributeId="role" />
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <ResourceMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">doc_private</AttributeVal
ue>
            <ResourceAttributeDesignator AttributeId="document" />
          </ResourceMatch>
        </Resource>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">read</AttributeValue>
            <ActionAttributeDesignator AttributeId="action_id"
DataType="http://www.w3.org/2001/XMLSchema#string" />
```

```
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
  </Rule>
  <Rule Effect="Deny" RuleId="rule3">
    <Target>
      <Subjects>
        <Subject>  <SubjectMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">manager</AttributeValue>
            <SubjectAttributeDesignator AttributeId="role" />
          </SubjectMatch>
        </Subject>
      </Subjects>
      <Resources>
        <Resource>
          <ResourceMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">doc_private</AttributeVal
ue>
            <ResourceAttributeDesignator AttributeId="document" />
          </ResourceMatch>
        </Resource>
      </Resources>
      <Actions>
        <Action>
          <ActionMatch
MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">write</AttributeValue>
            <ActionAttributeDesignator AttributeId="action_id" />
          </ActionMatch>
        </Action>
      </Actions>
    </Target>
  </Rule>
  <Rule Effect="Deny" RuleId="rule4" />
</Policy>
```

ANNEXE D

Dans le tableau 3.2, on trouve la description des clauses utilisées dans un contexte d'event-B

Clauses	Description
Extends	Signifie que le contexte peut être généré à partir d'un autre contexte qui existe déjà.
Sets	décrit les types de données utilisés pour définir les relations et les variables.
Constants	Comme le nom l'indique, les constantes sont utilisées pour initialiser les variables qui sont des éléments des ensembles.
Axioms	définissent des règles pour la partie statique en fonction des constantes et des ensembles déjà définis. Les axiomes utilisent des prédicats qui font appel à la logique du premier ordre.
Theorems	Les théorèmes sont utilisés pour prouver les propriétés du système. C'est comme les axiomes mais les théorèmes doivent être prouvés pour être valides.

Tableau 5.1 Les clauses du contexte

Le tableau ci-dessous décrit les clauses utilisées dans les machines :

Clauses	**Description.**
Refines	Permet de créer une machine basée sur une autre déjà créée.
Sees	Permet de sélectionner des contextes qui sont considérés par cette machine. Par contre, Il n'y a pas de limitation sur le nombre de contextes vus par la même machine.
Variables	Les variables décrivent les propriétés du système sur la base des invariants. Le changement de la valeur du variable dépend de l'exécution des événements.
Invariant	Les invariants sont des règles globales pour le système, ces règles Exprime les propriétés du système. Ces invariants doivent être conservés pendant tout le modèle de traitement.
Theorems	Similaire au théorème du contexte.
Variant	Dans cette section on définit une variable « variant » de type numérique ou sous-forme d'un ensemble fini d'expressions. Certains événements peuvent être sélectionnés pour diminuer la valeur du variable « variant », afin d'éviter une boucle infinie.

Tableau 5.2 Les clauses de la machine d'Event-B

Clauses	Description.
Status	Détermine le statut de l'événement. les valeurs possibles sont : ▪ **ordinary** : indique que l'événement est «normal». ▪ **convergent**: l'événement doit être un nouvel événement dans une machine raffinée (qui n'apparaît pas dans l'abstraction). ▪ **anticipated** : l'événement doit subir un raffinement ultérieur par un événement avec le statut «convergent»
Refines	Spécifier les événements à raffiner.
Any	Cette clause permet de définir les paramètres utilisés par l'événement.
Where	Permet d'exprimer des gardes sous formes d'une conjonction de prédicats.
witnesses	Quand un événement concret raffine un événement abstrait, tous les paramètres de cet événement doit recevoir une valeur dans l'événement concret. Ces valeurs sont appelées les témoins.
Actions	Une action consiste à affecter des valeurs aux variables qui permettent de changer l'état du système.

Tableau 5.3 Les clauses de l'événement en Event-B

RÉFÉRENCES BIBLIOGRAPHIQUES

1. Sloman, M., *Policy Driven Management for Distributed Systems.* J. Network Syst. Manage., 1994. **2**(4).
2. Uszok, A., et al. *KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement.* in *POLICY.* 2003.
3. Yun-qing, F. and Y. Chun-xiao. *Using XACML to define access control policy in information system.* in *Wireless, Mobile and Sensor Networks, 2007. (CCWMSN07). IET Conference on.* 2007.
4. Abrial, J.R., *The B-book: assigning programs to meanings.* 1996, New York, NY, USA: Cambridge University Press.
5. Holzmann, G.J., *The Model Checker SPIN.* IEEE Trans. Softw. Eng., 1997. **23**(5): p. 279-295.
6. Al-Amayreh, A. and A.M. Zin. *Probe: a formal specification-based testing system.* in *ICIS '99: Proceedings of the 20th international conference on Information Systems.* 1999. Atlanta, GA, USA: Association for Information Systems.
7. *<http://www.cs.umbc.edu/~lkagal1/rei/.*
8. Damianou, N., et al., *The Ponder Policy Specification Language.* POLICY, 2001. **1995**: p. 18-38.
9. Kumaraguru, P.C., Lorrie & Lobo, Jorge & Calo, Seraphin. , *A survey of privacy policy languages* Workshop on Usable IT Security Management (USM 07). In SOUPS '07: Proceedings of the 3rd symposium on Usable privacy and security (New York, NY, USA, March 2007), ACM., , 2007.
10. Evan, M. and X. Tao, *A fault model and mutation testing of access control policies.* 2007.
11. Martin, E., T. Xie, and T. Yu. *Defining and Measuring Policy Coverage in Testing Access Control Policies.* in *ICICS.* 2006.
12. .
13. Kathi, F., et al., *Verification and change-impact analysis of access-control policies.* 2005.
14. Bryans, J.W. and J.S. Fitzgerald, *Formal engineering of XACML access control policies in VDM++*, in *Proceedings of the formal engineering methods 9th international conference on Formal methods and software engineering.* 2007, Springer-Verlag: Boca Raton, FL, USA.
15. *VDMTools: advances in support for formal modeling in VDM.* SIGPLAN Not., 2008. **43**(2): p. 3-11.
16. Jackson, D., *Alloy: a lightweight object modelling notation.* ACM Trans. Softw. Eng. Methodol., 2002. **11**(2): p. 256-290.

17. Mankai, M., *Verification et analyse des politiques de controle d'acces: Application au langage XACML ,by Mahdi Mankai, Msc Thesis, Universite du Quebec en Outaouais, janvier 2005*. 2005.
18. Jery, B., *Reasoning about XACML policies using CSP*. 2005.
19. Hu, V., R. Kuhn, and T. Xie, *Property Verification for Access Control Models via Model Checking*. 2008, North Carolina State University Department of Computer Science.
20. Joseph, Y.H. and W. Vicky, *Using First-Order Logic to Reason about Policies*. ACM Trans. Inf. Syst. Secur., 2008. **11**(4): p. 1-41.
21. Bendisposto, J., et al., *La validation de modeles Event-B avec le plug-in ProB pour RODIN*. TSI, 2008: p. 1065--1084.
22. Dijkstra, E.W., *Guarded commands, nondeterminacy and formal derivation of programs*. Commun. ACM, 1975. **18**(8): p. 453-457.
23. Hallerstede, S., *On the Purpose of Event-B Proof Obligations*, in *Proceedings of the 1st international conference on Abstract State Machines, B and Z*. 2008, Springer-Verlag: London, UK.
24. *Abrial, J.-R., Summary of Event-B Proof Obligations. http://deploy-eprints.ecs.soton.ac.uk/53/3/sld_po.pdf, March 2008.*
25. Butler., C.S.a.M., *UML-B: Formal modelling and design aided by UML*. ACM Transactions on Software Engineering and Methodology,To appear.

eprints.ecs.soton.ac.uk/10169/.
, 2006.