Modélisation de politiques de sécurité à l'aide de méthode de spécifications formelles

Pierre Konopacki

▶ **To cite this version:**

Pierre Konopacki. Modélisation de politiques de sécurité à l'aide de méthode de spécifications formelles. Autre [cs.OH]. Université Paris-Est, 2012. Français. NNT : 2012PEST1089 . tel-00786926

HAL Id: tel-00786926

https://theses.hal.science/tel-00786926

Submitted on 11 Feb 2013

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

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

UNIVERSITÉ PARIS-EST

ÉCOLE DOCTORALE

MSTIC: Mathématiques et Sciences et Technologies de l'Information et de la Communication

Thèse de doctorat

Informatique

présentée par:

Pierre KONOPACKI

Une approche événementielle pour la description de politiques de contrôle d'accès

soutenue le 4 mai 2012 devant le jury suivant :

Président du jury :		
	Yves Ledru	Professeur à l'Université Joseph Fourier
Rapporteurs:		
	Frédéric Cuppens	Professeur à l'ENST-Bretagne
	Franck Pommereau	Professeur à l'Université d'Évry
Examinateurs:		
	Catalin Dima	Professeur à l'Université Paris-Est Créteil
	Jean Goulet	Professeur à l'Université de Sherbrooke
Directeurs de thèse:		
	Régine Laleau	Professeur à l'Université Paris-Est Créteil
	Marc Frappier	Professeur à l'Université de Sherbrooke

Sommaire

Le contrôle d'accès permet de spécifier une partie de la politique de sécurité d'un système d'informations (SI). Une politique de contrôle d'accès (CA) permet de définir qui a accès à quoi et sous quelles conditions. Les concepts fondamentaux utilisés en CA sont : les permissions, les interdictions (ou prohibitions), les obligations et la séparation des devoirs (SoD). Les permissions permettent d'autoriser une personne à accéder à des ressources. Au contraire les prohibitions interdisent à une personne d'accéder à certaines ressources. Les obligations lient plusieurs actions. Elles permettent d'exprimer le fait qu'une action doit être réalisée en réponse à une première action. La SoD permet de sécuriser une procédure en confiant la réalisation des actions composant cette procédure à des agents différents. Différentes méthodes de modélisation de politiques de contrôle d'accès existent. L'originalité de la méthode EB^{3}SEC issue de nos travaux repose sur deux points :

- permettre d'exprimer tous les types de contraintes utilisées en CA dans un même modèle,
- proposer une approche de modélisation basée sur les événements.

En effet, aucune des méthodes actuelles ne présente ces deux caractéristiques, au contraire de la méthode EB^{3}SEC. Nous avons défini un ensemble de patrons, chacun des patrons correspond à un type de contraintes de CA.

Un modèle réalisé à l'aide de la méthode EB^{3}SEC peut avoir différentes utilisations :

- vérification et simulation,
- implémentation.

La vérification consiste à s'assurer que le modèle satisfait bien certaines propriétés, dont nous avons défini différents types. Principalement, les blocages doivent être détectés. Ils correspondent à des situations où une action n'est plus exécutable ou à des situations où plus aucune action n'est exécutable. Les méthodes actuelles des techniques de preuves par vérification de modèles ne

permettent pas de vérifier les règles dynamiques de CA. Elles sont alors combinées à des méthodes de simulation. Une fois qu'un modèle a été vérifié, il peut être utilisé pour implémenter un filtre ou noyau de sécurité. Deux manières différentes ont été proposées pour réaliser cette implémentation : transformer le modèle EB^{3}SEC vers un autre langage, tel XACML, possédant une implémentation ayant déjà atteint la maturité ou réaliser un noyau de sécurité utilisant le langage EB^{3}SEC comme langage d'entrée.

Remerciements

Je remercie les membres du jury pour leur disponibilité et leur présence. Je remercie Frédéric Cuppens et Franck Pommereau d'avoir rapporté cette thèse. Mes remerciements vont aussi à Yves Ledru, Catalin Dima, Jean Goulet pour leur présence à mon jury de soutenance.

Je remercie le professeur Marc Frappier pour m'avoir offert l'opportunité de réaliser une thèse sous sa direction et la professeure Régine Laleau de m'avoir accepté en cotutelle. Je les remercie tous deux pour leurs conseils, leur aide et leur soutien ainsi que pour leur patience.

Je remercie Flore, Nathalie, Lynn et Lise pour leur aide dans toutes les tâches administratives. Mes remerciements vont aussi aux équipes du LACL, du GRIL et de l'IUT de Fontainebleau pour leur accueil, leur aide au niveau de la recherche et aussi dans l'art de l'enseignement.

Je tenais à remercier Frédéric Gervais, Benoît Fraikin, Richard Saint-Denis, Abderrahman Matoussi, Muath Alrammal ... pour leur aide et leur présence durant ces années de thèse.

Mes remerciements s'adressent aussi à Lise, Francis, Vincent, Mélanie, Olivier et Jérémy pour leur soutien, leur accueil, leur écoute et pour m'avoir aidé à surmonter les difficultés.

Mes remerciements vont aussi à tous mes amis que je n'ai pas eu la place de citer (Béné, Anneclaire, Chloé, Bertrand, Rémi, Pierrot, Juju, Jérémie ...) mais qui m'ont fait partager des moments de joie et de grands moments de bonheur.

Mes remerciements vont aussi à mes parents qui ont dû me supporter malgré mes pointes de mauvaise humeur durant ces dernières années.

Table des matières

Table des figures **xiv**

Liste des codes **xv**

Liste des abréviations **xvii**

Introduction **1**

Contexte 1

Objectifs et motivations 2

EB^{3}SEC 4

Contributions et structure du mémoire 4

1 État de l'art **7**

1.1 Les types de CA 7

1.2 Concepts de base du CA 9

1.3 Définition détaillée des quatre concepts 10

1.3.1 Contraintes statiques 10

1.3.1.1 Les permissions sans contraintes 11

1.3.1.2 Les interdictions sans contraintes 11

1.3.1.3 La résolution statique des contraintes de SoD 11

1.3.2 Contraintes dynamiques 12

1.3.2.1 Les permissions avec contraintes 12
1.3.2.2 Les interdictions avec contraintes 12
1.3.2.3 Résolution dynamique des contraintes de type SoD 12
1.3.2.4 Contraintes d'obligation 13
1.4 Les modèles de CA 13
1.4.1 Le modèle RBAC 14
1.4.2 Extensions du modèle RBAC 14
1.4.3 OrBAC 15
1.4.4 ABAC 18
1.4.5 SoDA 19
1.4.6 Expression de contrôle de transaction pour la séparation des devoirs 19
1.5 Les utilisations des différents modèles 21
1.5.1 Vérification 22
1.5.2 Implémentation 23
1.5.2.1 SecureUML 23
1.5.2.2 Implémentation en CSP de modèles exprimés en SoDA 24
1.5.2.3 XACML et quelques travaux connexes 25
1.6 Comparaison 25
1.7 Conclusion 30

2 Exemple **33**
2.1 Aspect fonctionnel 33
2.2 Politique de CA 34

3 Les méthodes EB^3 et EB^{3}SEC **37**
3.1 Modélisation de la partie fonctionnelle à l'aide de la méthode EB^3 37
3.1.1 La méthode EB^3 37
3.1.2 Modélisation de l'exemple 39

3.2 Modélisation d'une politique de CA à l'aide de la méthode EB^{3}SEC 42
3.2.1 Le diagramme de classes de sécurité 43
3.2.1.1 Les permissions sans contraintes 45
3.2.1.2 Les permissions avec contraintes 46
3.2.1.3 Les interdictions 46
3.2.1.4 Les interdictions avec contraintes 47
3.2.1.5 La séparation des devoirs statique (séparation des devoirs statique (SSD)) 47
3.2.2 Les expressions de processus 48
3.2.2.1 Définition d'une action sécurisée 49
3.2.2.2 Première approche : sécuriser les expressions de processus fonctionnelles 50
3.2.2.3 Deuxième approche : créer une expression de processus pour chaque règle de CA 52
3.2.2.4 Troisième approche : créer une expression de processus par action . 53
3.2.3 Discussion 54
3.2.3.1 Adaptabilité de la méthode EB^{3}SEC 54
3.2.3.2 Comparaison des modélisations précédentes 55

4 Patrons de conception de règles en EB^{3}SEC **57**
4.1 Quelques définitions 57
4.2 Permissions et interdictions 59
4.2.1 Permissions 59
4.2.1.1 Permissions sans contraintes 59
4.2.1.2 Équivalence entre les permissions du diagramme de classes et les expressions obtenues à l'aide des patrons de permission 61
4.2.1.3 Permissions avec contraintes 61

4.2.2 Interdictions 63
4.2.2.1 Interdictions sans contraintes 63
4.2.2.2 Interdictions avec contraintes 64
4.3 séparation des devoirs dynamique (DSD) et obligations 64
4.3.1 SoD dynamique (DSD) 65
4.3.1.1 DSD séquentielles sans contraintes 65
4.3.1.2 DSD séquentielle avec contraintes sans sanctions 66
4.3.1.3 DSD séquentielle avec contraintes avec sanctions 67
4.3.1.4 DSD parallèle sans contraintes sans sanctions 68
4.3.1.5 DSD parallèle avec contraintes sans sanctions 69
4.3.1.6 DSD parallèle avec contraintes avec sanctions 70
4.3.2 Obligations 71
4.3.2.1 Obligations séquentielles sans sanctions sans contraintes 72
4.3.2.2 Obligations séquentielles sans sanctions avec contraintes 73
4.3.2.3 Obligations séquentielles avec sanctions avec contraintes 74
4.3.2.4 Obligations parallèles sans sanctions sans contraintes 75
4.3.2.5 Obligations parallèles sans sanctions avec contraintes 75
4.3.2.6 Obligations parallèles avec sanctions avec contraintes 77
4.3.2.7 Remarque 78
4.4 Mise en commun 78
4.4.1 Permissions et interdictions 78
4.4.1.1 Mise en commun des permissions 79
4.4.1.2 Mise en commun des interdictions 79
4.4.1.3 Mise en commun des permissions et des interdictions 80
4.4.1.4 Mise en application 80
4.4.2 Obligation et SoD 82
4.4.2.1 Choix 82

4.4.2.2 Parallèle 83
4.4.2.3 Entrelacement 83
4.4.2.4 Mise en commun 84
4.4.3 Le main 84
4.5 Discussion sur l'utilité 84

5 Outils pour la vérification et la validation **87**
5.1 Types de propriétés 87
5.1.1 Propriétés de vivacité 88
5.1.2 Propriétés de sûreté 89
5.1.3 Quelques exemples de propriétés 89
5.1.3.1 Définitions préliminaires 90
5.1.3.2 Typage de la politique de CA (type **INV**) 91
5.1.3.3 Atteignabilité de tous les événements du système (type **SCEF**) . . . 92
5.1.3.4 Atteignabilité de tous les événements sécurisés (type **SCEF**) 92
5.1.3.5 Faisabilité des permissions (type **SCEF**) 92
5.1.3.6 Faisabilité des instances de l'association "**joue**" (type **SCEF**) 93
5.1.3.7 Utilisabilité des instances de la classe "**Organisation**" (type **SCEF**) 93
5.1.3.8 Utilisabilité des instances de la classe "**Personne**" (type **SCEF**) . . . 94
5.1.3.9 Utilisabilité des instances de la classe "**Role**" (type **SCEF**) 94
5.1.3.10 Application des propriétés précédentes 94
5.1.3.11 Non-faisabilité des interdictions 94
5.2 Vérification des propriétés 95
5.2.1 Vérification de la partie statique 95
5.2.1.1 Choix des patrons implémentés 96
5.2.1.2 Le langage d'entrée 96
La clause *play* 96

La clause *permission* . . . 97
La clause *interdiction* . . . 97
La clause *sod* . . . 98
La clause *obligation* . . . 99
La clause *complex* . . . 99
5.2.1.3 Algorithme de vérification de la partie statique . . . 100
5.2.2 Simulation de la partie dynamique . . . 103
5.2.2.1 La plateforme APIS . . . 104
5.2.2.2 Utilisation de la plateforme APIS pour simuler des modèles EB^{3}SEC . 105
5.3 Un profil XACML pour la méthode EB^{3}SEC . . . 105
5.3.1 XACML . . . 106
5.3.1.1 Un aperçu du langage XACML . . . 106
5.3.1.2 L'architecture d'une solution XACML . . . 107
5.3.1.3 Politique de CA en XACML . . . 108
5.3.1.4 Policyset . . . 109
Target : . . . 111
Subjects : . . . 112
Resources, *actions*, *environments* : . . . 113
Policy : . . . 113
Rule : . . . 114
Algorithmes de combinaison (des règles et des politiques) : . . . 115
Obligation : . . . 116
5.3.1.5 Les requêtes . . . 117
5.3.1.6 Les décisions . . . 119
5.3.2 Implémentation . . . 122
5.3.2.1 L'implémentation de SUN . . . 122
5.3.2.2 L'implémentation de Google . . . 122

5.3.3 Les outils . . . 122
5.3.3.1 L'éditeur XACML UMU . . . 123
5.3.3.2 Un outil de modélisation de politiques de CA de type RBAC . . . 123
5.4 Quelque extensions XACML . . . 123
5.4.1 Un profil du modèle RBAC . . . 123
5.4.1.1 Élément de type *Policy* ou *Policyset* utilisé pour l'affectation des rôles 124
5.4.1.2 L'élément *Policyset* correspondant aux rôles . . . 127
5.4.1.3 Les éléments *Policyset* correspondant aux permissions données à un rôle . . . 127
5.4.1.4 L'élément *Policyset* utilisé pour la SoD . . . 128
5.4.2 Un profil XACML pour la SoD . . . 130
5.4.2.1 La SoD statique . . . 130
5.4.3 La SoD dynamique . . . 130
5.4.4 Le profil OrBAC . . . 131
5.5 Un profil XACML pour la méthode EB^{3}SEC . . . 131
5.5.1 Étude de faisabilité du profil . . . 131
5.5.1.1 Permissions . . . 131
En EB^{3}SEC : . . . 131
En XACML : . . . 133
Comparaison : . . . 135
5.5.1.2 Diagramme de classes . . . 135
Première version . . . 135
Deuxième version . . . 135
Troisième version . . . 136
5.5.1.3 Expression de processus . . . 136
La séquence . . . 136
Entrelacement quantifié . . . 137

Le choix 138
5.5.2 Résultats 138

Conclusion **141**

Synthèse 141
Perspectives 143

Table des figures

1.1 La relation "*employee*" du modèle OrBAC 16
1.2 La relation "*use*" du modèle OrBAC 16
1.3 La relation "*considere*" du modèle OrBAC 17
1.4 La relation "*permission*" du modèle OrBAC 17
1.5 Exemple d'utilisation des opérateurs ; et • pour l'expression de la séparation des devoirs 20
1.6 Exemple de pondérations des actions pour l'expression de la séparation des devoirs 20
1.7 Exemple d'assignation de valeurs et de choix pour l'expression de la séparation des devoirs 20

3.1 Diagramme de classes de la procédure de dépôt de chèques 40
3.2 Diagramme de classes utilisé pour sécuriser la procédure de dépôt de chèque 43
3.3 Instanciation du diagramme de classes de l'exemple de l'institution bancaire 45
3.4 Exemple de permissions avec contraintes 46
3.5 Exemple d'interdictions sans contraintes 47
3.6 Exemple d'interdictions avec contraintes 47
3.7 Exemple de SSD 48

5.1 Exemple de table *joue* 97
5.2 Exemple pour la partie permission 97

5.3 Exemple pour la partie interdiction 97
5.4 Exemple pour la partie SoD 98
5.5 Exemple pour la partie obligation 99
5.6 Exemple pour la partie complexe 100
5.7 Résultat de l'algorithme $PerEff$ 102
5.8 Result of $ProhEff$ 102
5.9 Result of $PermReal$ 103
5.10 Architecture du prototype de noyau de sécurité 104
5.11 Architecture standard de l'implémentation d'une solution XACML 107
5.12 Modèle utilisé pour décrire les politiques de CA en XACML 109
5.13 Diagramme de séquence utilisé pour l'orchestration des composantes du profil OrBAC 132
5.14 Diagramme de classes utilisé en EB^{3}SEC 133

Liste des codes

5.1 Exemple de Policyset utilisé en XACML 110
5.2 Exemple de Target utilisé en XACML 111
5.3 Exemple d'élément Subjects utilisé en XACML 112
5.4 Exemple de Policy utlilisé en XACML 113
5.5 Exemple de Rule utilisé en XACML 114
5.6 Example d'Obligation utilisée en XACML 117
5.7 Exemple de requête 117
5.8 Exemple de décision 120
5.9 Exemple de Policyset utilisé pour l'assignation des rôles 124
5.10 Exemple de Policyset utilisé pour les SoD 128
5.11 Un exemple de permission en XACML 133

Liste des abréviations

CA contrôle d'accès

DAC contrôle d'accès discrétionnaire

DSD séparation des devoirs dynamique

MAC contrôle d'accès obligatoire

PDP point de prise de décision de la politique

PEP point de mise en application de la politique

RBAC contrôle d'accès basé sur les rôles

SI système d'informations

SoD séparation des devoirs

SSD séparation des devoirs statique

Introduction

Contexte

Un système d'informations (SI) correspond aux moyens mis en œuvre par une organisation pour regrouper, stocker, traiter et gérer les informations qu'elle utilise. Grâce à l'avènement de l'informatique, les SI se sont automatisés et informatisés. Bien qu'informatisés, les SI n'étaient consultables que par peu d'utilisateurs. Leurs accès n'étaient possible par une présence physique. Seuls les employés, présents sur un site de l'organisation, pouvaient accéder aux données de l'organisation. L'essor que connaissent les technologies de l'information et principalement le développement des réseaux de télécommunication rendent les SI accessibles par une multitude d'utilisateurs à travers le monde. Dans l'exemple du domaine bancaire, il y a peu, seuls les employés de la compagnie avaient accès aux comptes de leurs clients. Aujourd'hui, le SI d'une organisation bancaire permet à la fois, aux employés de gérer les comptes de leurs clients, aux courtiers de la banque de gérer les produits financiers, aux clients d'accéder à leurs comptes en banque aussi bien via leur ordinateur familial que leur téléphone cellulaire intelligent. Les SI doivent aujourd'hui gérer un grand nombre d'utilisateurs, réalisant des tâches différentes.

Avec le développement des technologies de la communication, la sécurité des SI a pris beaucoup d'importance. Les données regroupées et stockées au sein des SI peuvent avoir une grande valeur et attirer la convoitise. Les organisations veulent s'assurer que leurs données ne peuvent être piratées. Les moyens mis en place sont devenus considérables. Ces moyens revêtent différents aspects : chiffrement des données, cryptage des communications, moyens sophistiqués d'iden-

tification ... Cependant l'ensemble des outils ne permet pas de s'assurer que les données sont protégées [73]. Face à ces problèmes, des nouvelles lois ont été votées [53, 69], ces lois obligent les organisations à sécuriser les données présentes dans leurs SI.

En plus des composantes précédemment citées faisant partie intégrante de la sécurité, il ne faut pas oublier le contrôle d'accès (CA). Le CA permet d'exprimer les contraintes relatives aux permissions accordées aux utilisateurs. En d'autres termes, le CA définit qui a accès à quoi au sein des SI. L'ensemble de ces contraintes forme ce qui est communément appelé la politique de CA. Plusieurs méthodes permettent d'exprimer des politiques de CA. Elles se distinguent principalement par leur expressivité, leur formalisme ou leur approche. Le domaine du CA ne correspond pas seulement aux permissions accordées au sein d'un SI, mais il regroupe différentes catégories de contraintes : les permissions, les interdictions, les obligations et la séparation des devoirs.

Lors de la modélisation d'un système, plusieurs méthodes peuvent être utilisées. Un point de comparaison pour les méthodes de modélisation est leur formalisme. Une méthode est dite formelle lorsqu'elle permet de décrire rigoureusement à l'aide d'objets mathématiques un modèle. L'utilité des méthodes formelles est apparue pour le développement des systèmes critiques. Leur formalisme permet l'utilisation de raisonnements mathématiques afin de prouver que le résultat obtenu est exempt de bogues et qu'il réalise effectivement la tâche qui lui est impartie.

La thèse présentée dans ce document se trouve à l'intersection de ces trois domaines : l'ingénierie des SI, les méthodes formelles de modélisation et le CA.

Objectifs et motivations

Dans cette thèse, nous nous intéressons à l'expression de politiques de CA. Notre objectif est de définir les concepts nécessaires pour spécifier un modèle de politiques de CA. Cependant, beaucoup de méthodes permettent déjà ce résultat. Nous ajoutons des contraintes supplémentaires. Ces contraintes sont les suivantes :

- le modèle créé doit être souple et doit pouvoir s'adapter à toute structure. Notre modèle doit pouvoir s'adapter à toute structure organisationnelle ainsi bien dans le domaine bancaire ou hospitalier. Les méthodes de fonctionnement des entités de ces différents domaines varient d'une structure à l'autre. Notre modèle doit pouvoir s'adapter aux particularités des organisations de chacun des domaines précédemment cités,

- le modèle créé doit être le plus expressif possible. Le langage que nous créons doit permettre aux utilisateurs de pouvoir exprimer, dans un même modèle, l'ensemble des contraintes de CA existant. Cette particularité permet de ne pas avoir recours à différents modèles pour exprimer l'intégralité d'une politique de CA. Le fait que la politique de CA est exprimée au sein d'un même modèle aide à la vérification de cette politique de CA,

- le modèle créé doit permettre la séparation des contraintes de CA et des contraintes fonctionnelles du SI. Cette séparation permet une fois le modèle réalisé d'utiliser des méthodes d'implémentation adaptées pour chacun des aspects, voire de permettre la réalisation d'un noyau de sécurité à partir du modèle de la politique de CA pour un système déjà existant. De plus, elle permet aussi de simplifier la vérification de chacun des modèles et aussi de l'ensemble de la spécification. En effet, différentes propriétés, comme des propriétés de vivacité, pourront être vérifiées sur chacune des parties du modèle puis sur la totalité. Cette séparation aide à la conception d'un modèle global de meilleur qualité et aussi à une meilleure traçabilité entre les exigences de sécurité et le modèle de la politique de CA,

- le modèle créé doit être formel. En effet, l'utilisation de méthodes formelles aide à la vérification du modèle. Si le modèle est réalisé à l'aide d'un langage formel, des raisonnements mathématiques tels que des techniques de vérification de modèles ou des techniques de preuves peuvent utilisées pour vérifier certaines propriétés sur le modèle.

Pour réaliser cette méthode de modélisation de politiques de CA, nous avons essayé de combiner les atouts proposés par les méthodes de modélisation de SI et ceux de différents modèles

de CA. L'étude des méthodes de modélisation de SI nous a mené vers différents langages comme EB^3, B ou UML. Parmi toutes ces méthodes, la méthode EB^3 paraît la plus adaptée : elle dispose d'un langage formel de modélisation, elle correspond à une approche basée sur les événements. Cette approche simplifie l'expression du comportement d'un système dynamique. La méthode EB^3 présente la particularité d'obtenir des modèles regroupant plusieurs composantes. L'une d'elle est un diagramme de classes permettant de décrire les entités du système et leurs attributs, une autre composante correspond à des expressions de processus et permet de décrire la composante dynamique.

EB^{3}SEC

La méthode EB^3 a été étendue de manière à créer EB^{3}SEC. Une spécification EB^{3}SEC est composée d'un diagramme de classes et d'une expression de processus. Le diagramme de classes permet de décrire les entités et leurs attributs nécessaires à l'expression d'une politique de CA. L'expression de processus permet de définir les contraintes de CA. L'étude de différents modèles de CA, nous a permis de déterminer les concepts couramment utilisés en CA, afin d'étendre l'algèbre de processus en y incorporant la notion d'attributs de sécurité. Les attributs de sécurité sont les données correspondant à un événement et nécessaires à la prise de décision pour l'exécution de l'action.

Contributions et structure du mémoire

Les objectifs fixés au début de nos travaux nous ont permis d'arriver aux contributions suivantes :

- réalisation d'une étude approfondie des différents modèles de CA, de manière à bien comprendre les concepts sous-jacents de ce domaine et de les exprimer le mieux possible au sein de la méthode EB^3. Cette étude a été réalisée en gardant pour objectif que le résultat devait s'appliquer aux SI et non à tout type de systèmes informatiques,

- définition du langage EB^{3}SEC. Ce langage, permettant la description formelle de politiques de CA, peut être utilisé dans tout type d'organisations, et s'adapter à n'importe quel SI,

- définition d'un processus d'utilisation du langage EB^{3}SEC. En plus de fournir un langage de modélisation de politiques de CA, une méthodologie permet de guider l'utilisateur dans la modélisation d'une politique de CA. Cette méthodologie se base principalement sur l'utilisation de patrons adaptés : pour chaque type de contraintes de CA, un patron a été défini,

- définition des méthodes de vérification des modèles EB^{3}SEC. Une fois le modèle réalisé, les méthodes de vérification permettent à l'utilisateur de vérifier quelques propriétés sur le modèle. Des exemples typiques de propriétés ont été établis. Ces propriétés permettent de vérifier l'absence de blocage dans le modèle mais aussi que les éléments modélisés dans le diagramme de classes sont pertinents. La vérification d'une partie de ces propriétés peuvent être réalisée sur le modèle : par exemple l'absence de blocage entre les permissions et les interdictions.

Une définition plus précise du contrôle d'accès est donnée au chapitre 1. Ce chapitre définit aussi les différents types de contraintes rencontrées en CA avant de donner un aperçu de différentes méthodes de spécification de politiques de CA. Le chapitre 2 donne un exemple de politique de CA utilisé au long de cette thèse dans un but d'illustration. La méthode EB^3 est présentée au chapitre 3. La méthode EB^{3}SEC étant basée sur EB^3, cette présentation permet de définir les différentes composantes d'un modèle EB^{3}SEC. Le chapitre 3 décrit ensuite comment sont intégrés les attributs de sécurité au sein de la méthode EB^{3}SEC. Le chapitre 4 décrit comment modéliser les différents types de contraintes de CA en EB^{3}SEC. Finalement, le chapitre 5 illustre les utilisations possibles d'un modèle réalisé en EB^{3}SEC.

Chapitre 1

État de l'art

Les politiques de contrôle d'accès (CA) correspondent à la description de qui a accès à quoi. Par *qui*, nous comprenons tous les acteurs du système. Par *quoi*, nous entendons toutes les ressources du système. Cet état de l'art présente un ensemble de méthodes de modélisation de politiques de CA. D'autres revues de littérature existent : la thèse [12] propose un inventaire exhaustif des différentes méthodes de CA, la thèse [54] s'intéresse à des composantes, telles que l'administration des politiques de CA, non traitées dans ce document, de même la thèse [42] s'intéresse à des concepts tels que le contrôle d'interface qui ne sont pas traités dans ce document et la thèse [62] présente une étude des méthodes de validation des modèles de politiques de CA.

1.1 Les types de CA

Cette section donne une première définition des politiques de CA. En fonction du système considéré, les politiques de CA peuvent être réparties en trois catégories.

Le **contrôle d'accès discrétionnaire (DAC)** est défini comme un moyen de limiter les accès aux objets basés sur l'identité des sujets ou des groupes auxquels ils appartiennent. Les droits sont dits discrétionnaires car un sujet avec une certaine autorisation d'accès a le droit de transmettre cette permission (peut-être indirectement) à n'importe quel autre sujet. Cette

définition est donnée par le Département de la Défense Américaine, au travers de critères définis dans le TCSEC [60] (*Trusted Computer System Evaluation Criteria*). Ces critères permettent d'évaluer la fiabilité des systèmes informatiques.

Le **contrôle d'accès obligatoire (MAC)** correspond aux politiques de CA dans lesquelles les permissions d'accès aux données ne sont pas définies par l'utilisateur créant les données mais par le système, selon la politique créée par l'administrateur. Les critères définis dans le TCSEC [60] permettent aussi l'évaluation de la fiabilité des systèmes informatiques utilisant une politique de CA de type MAC.

Le **contrôle d'accès basé sur les rôles (RBAC)** correspond aux politiques de CA dans lesquelles les permissions d'accès accordées à un utilisateur sont basées sur les rôles qui lui sont rattachés. Ce type de politiques de CA est principalement utilisé dans les entreprises où les rôles proviennent de la structure de l'entreprise.

Dans le cadre des politiques de type DAC, plusieurs modèles existants peuvent être utilisés : les matrices HRU [34], le modèle de Bell et LaPadula [11], le modèle de Clark et Wilson [18], les listes de contrôle d'accès ... Notre travail s'intéresse à l'expression de politiques de CA appliquées à un système d'informations (SI). Un SI est un ensemble organisé de ressources (matériel, logiciel, personnel, données et procédures) permettant d'acquérir, traiter, stocker, communiquer, etc., des informations nécessaires à la réalisation des processus de l'organisation [65]. Dans la suite de cette thèse, nous nous intéressons aux modèles de politiques de CA de type RBAC. Cependant, dans une partie de la littérature comme dans [61], les modèles de politiques de CA de type RBAC ne sont qu'une catégorie de modèles de politique de CA de type MAC.

Dans la suite de cette thèse, nous nous intéressons aux modèles de politiques de CA de type RBAC, définis pour des SI.

1.2 Concepts de base du CA

La logique déontique formalise les rapports existants entre les quatre alternatives d'une loi : la permission, l'interdiction, l'obligation et le facultatif. Elle fût décrite comme une logique modale à partir des années 1670 par Gottfried Wilheim Leibniz, cette approche ayant été formalisée dans les années 1950, par Georg Henrik von Wright dans [77]. Les concepts de permission, d'interdiction, d'obligation et de facultatif sont formalisés de la manière suivante. Pour une action nommée A, on définit :

- la permission de réaliser l'action A par PA,
- l'interdiction de réaliser l'action A est décrite par $\sim (PA)$ (l'opérateur $\sim$ désigne l'opérateur de négation),
- l'obligation de réaliser une action correspond à la non permission de ne pas la réaliser, l'obligation de réaliser l'action A peut s'écrire $\sim (P \sim A)$. Dans un souci de lisibilité on introduit l'opérateur OA,
- le caractère facultatif d'une action peut se traduire par le fait qu'elle peut être réalisée ou non. Le caractère facultatif d'une action A peut se noter $(PA)\&(P \sim A)$ (l'opérateur $\&$ correspond à l'opérateur de conjonction).

Les trois premiers concepts, permission, interdiction et obligation, sont des concepts utiles en CA. En plus de ces concepts, un autre concept appelé séparation des devoirs (SoD) est introduit dans [44]. Cette contrainte de CA divise une tâche en un ensemble de sous-tâches, chacune de ces sous-tâches doit alors être exécutée par une personne différente. La SoD permet, entre autres, d'éviter les cas de corruption.

Dans la littérature actuelle, il est communément admis que le CA regroupe les quatre types de contraintes précédentes : permission, interdiction (encore appelée prohibition), obligation et SoD. Cependant, certains, par exemple dans [61], considèrent que le CA ne regroupe que les contraintes

de type permission, interdiction et SoD. Ils définissent alors le contrôle d'usage regroupant le CA et les obligations. Dans la suite de cette thèse, nous considérons que le CA regroupe l'ensemble des quatre types de contraintes précédemment cités.

1.3 Définition détaillée des quatre concepts

Pour modéliser un système, deux approches peuvent être utilisées.

Approche basée sur les événements : l'état du système est représenté par un ensemble de variables d'état (classes et attributs dans un SI) et l'évolution est codée par le biais des valeurs, en ajoutant éventuellement de nouvelles variables, appelées variables de contrôle.

Approche basée sur les événements : l'évolution du système est représentée explicitement par un modèle qui permet de spécifier des contraintes d'ordonnancement comme des algèbres de processus ou des automates. L'état se déduit alors de cette représentation. Cependant, dans le domaine des SI, il est plus usuel de représenter l'état par un diagramme de classes, l'évolution du SI étant représentée par des automates ou des expressions de processus (par exemple, comme avec la méthode EB^3).

Pour les contraintes de CA nous introduisons la distinction statique/dynamique. Les contraintes statiques sont les contraintes qui ne dépendent pas de l'état du système ou de son évolution. Au contraire, les contraintes dynamiques dépendent de l'état du système et/ou de son évolution.

1.3.1 Contraintes statiques

Ces contraintes ne dépendent pas de l'état du système. Cette catégorie de contraintes est divisée en trois sous-catégories : les permissions, les interdictions et la résolution statique des contraintes de SoD.

1.3.1.1 Les permissions sans contraintes

La définition que nous utilisons pour ce type de permissions est la définition utilisée en logique déontique et aussi communément admise dans les modèles de politiques de CA. Les permissions expriment le fait qu'un sujet puisse réaliser une action. Elles sont toujours vraies au sein du système. Elles permettent à un sujet de réaliser une action pour un objet donné. Les permissions sans contraintes, utilisées dans nos travaux, sont les mêmes que celles utilisées dans la plupart des modèles.

1.3.1.2 Les interdictions sans contraintes

Ces interdictions, encore appelées prohibitions, sont définies, en logique déontique, comme l'opposée des permissions. Elles correspondent au fait qu'un sujet n'a pas le droit de réaliser une action pour un objet donné. Tous les modèles de CA permettent d'exprimer des permissions, mais seuls quelques uns [14, 17, 40] permettent d'utiliser le concept d'interdiction. Les interdictions sans contraintes sont vraies tout au long de l'exécution du système. Elles permettent d'interdire une action sur un objet pour un sujet.

1.3.1.3 La résolution statique des contraintes de SoD

Les contraintes de SoD (*Separation of Duty*), ou séparation des devoirs, ont été introduites pour éviter les corruptions lors de l'exécution de procédures complexes. La séparation des devoirs se définit comme la division d'une procédure en tâches élémentaires, chacune devant être réalisée par une entité différente [23]. Le fait de faire intervenir plusieurs acteurs pour chacune des tâches élémentaires d'une procédure permet de limiter les risques de corruptions de la procédure. La séparation des devoirs statique (SSD) correspond à la résolution statique des contraintes de SoD, c'est-à-dire que les permissions d'exécution des tâches élémentaires provenant de la décomposition d'une procédure sont attribuées à des sujets différents. Une même personne ne peut se voir attribuer la permission d'exécuter toutes les tâches provenant d'une même procédure. Les permissions

et interdictions induites par une contrainte de type SSD sont vraie durant toute l'exécution du système.

1.3.2 Contraintes dynamiques

Les contraintes dynamiques dépendent de l'état du système et/ou de son évolution. Elles sont divisées en quatre sous-catégories : les permissions avec contraintes, les interdictions avec contraintes, la résolution dynamique des contraintes de type SoD et les contraintes de type obligation.

1.3.2.1 Les permissions avec contraintes

Une contrainte associée à une permission restreint son domaine d'application. La contrainte correspond à un prédicat logique défini sur les valeurs des attributs des entités du système. Elle représente la condition sous laquelle la permission est valide. Par exemple, on peut restreindre une permission à une certaine plage horaire de la journée. Les permissions avec contraintes (aussi appelées permissions sous contraintes) ne sont pas utilisées dans tous les modèles de CA, seuls quelques modèles permettent leur expression [13, 57].

1.3.2.2 Les interdictions avec contraintes

Elles permettent de définir des interdictions qui ne sont valables que sous certaines conditions. Les conditions sont, comme pour les permissions avec contraintes, des prédicats logiques définis sur les valeurs des variables du système.

1.3.2.3 Résolution dynamique des contraintes de type SoD

Les contraintes de type SoD peuvent être résolues de manière dynamique. Dans ce cas, on définit la notion de séparation des devoirs dynamique (DSD), contrainte qui dépend de l'état d'exécution du système. Dans le cadre des DSD, les permissions et interdictions induites par la contraintes de SoD ne sont pas vraies durant toute l'exécution du système, mais dépendent de l'état d'exécution

du système. Dans la plupart des modèles [78] elles sont vraies pour une instance de la procédure donnée. Par exemple, si une contrainte de DSD est valable pour deux actions a et b toutes deux s'appliquant pour les objets $o_1, \ldots, o_n$, alors la personne ayant réalisé l'action a pour l'objet o_1 ne peut pas exécuter l'action b pour l'objet o_1, mais elle peut réaliser l'action b pour l'objet o_2 si elle n'a pas réalisé l'action a pour l'objet o_2.

1.3.2.4 Contraintes d'obligation

Les contraintes d'obligation, utilisées en CA, diffèrent de la définition donnée en logique déontique, la raison principale étant qu'un système informatique ne peut obliger un utilisateur à réaliser une action. Les contraintes de type obligation correspondent en CA au fait que deux actions sont liées. Pour deux actions liées par une contrainte d'obligation, lorsqu'une des deux est réalisée l'autre aussi doit être exécutée. Trois cas sont envisageables. L'obligation doit être remplie après l'action, par exemple, dans le domaine hospitalier, lorsqu'un médecin examine un patient, il doit, à la fin de l'examen, remplir le dossier médical du patient. L'obligation doit être remplie avant l'action, dans le domaine bancaire pour qu'un chèque soit crédité sur un compte, il faut qu'il soit préalablement validé par un employé de l'agence. Dans certains cas, l'ordre des actions n'a pas d'importance. Dans [59], trois catégories d'obligations sont répertoriées : les pré-obligations, correspondant au cas où l'obligation doit être réalisée avant l'action, les post-obligations, correspondant au cas où l'obligation doit être réalisée après l'action, et les obligations conditionnelles. Les obligations conditionnelles, correspondent à des obligations devant être réalisée lorsqu'une contrainte est vraie. Le type de contraintes principalement rencontré correspond à des contraintes temporelles exprimant une durée.

1.4 Les modèles de CA

Après avoir défini les concepts de base du CA, nous nous intéressons maintenant à différents modèles existants.

1.4.1 Le modèle RBAC

Le modèle RBAC [68, 38] répond à trois besoins.

Le besoin de "*least privilege*" : l'utilisateur n'a que les droits minimums pour réaliser une action.

La séparation des responsabilités : plusieurs personnes doivent intervenir dans la réalisation d'une action.

Abstraction des données : on peut définir des actions plus abstraites que lecture/écriture.

Le modèle RBAC modélise le système informatique considéré en séparant les entités actives des entités passives. Les entités actives correspondent aux utilisateurs du système et les entités passives correspondent aux objets du système. Au contraire du modèle de LaPadula, qui associe des niveaux de sécurité aux différents objets et aux différents utilisateurs, le modèle RBAC regroupe les utilisateurs en groupe d'utilisateurs. Ces groupes sont appelés des rôles. Les privilèges sont accordés à un rôle et sont valables pour exécuter des actions sur des objets donnés. Pour pouvoir réaliser une action, l'utilisateur doit donc activer le rôle adéquat. Les permissions ne sont plus attribuées à une personne particulière mais à des rôles. Il revient alors aux différents acteurs du système d'activer le rôle nécessaire à la réalisation d'une action. Plusieurs variantes ont été créées à partir de ce modèle de sécurité. De plus, d'autres modèles se sont inspirés du modèle RBAC comme point de départ. Différents exemples sont mentionnés dans la suite de cette section.

1.4.2 Extensions du modèle RBAC

Le modèle RBAC s'est vu ajouter de nombreuses extensions. La première à prendre place fut la notion de hiérarchie des rôles. Cette hiérarchie permet de décrire une notion d'héritage. Dans un modèle RBAC hiérarchique, un rôle peut hériter des permissions d'un autre rôle. Une autre extension du modèle RBAC correspond à la notion de contraintes. Ces contraintes peuvent par exemple permettre d'exprimer des contraintes de SSD au sein d'un modèle RBAC. Elles peuvent

aussi correspondre à des contraintes temporelles sur l'activation d'un rôle. Ces extensions sont définies dans [23].

En plus des notions de hiérarchie et de contrainte, on peut citer les extensions suivantes.

TRBAC (*Temporal Role-Based Acces Control*) introduit la notion de temps. Le modèle permet de définir des contraintes temporelles sur l'activation des rôles. TRBAC est décrit dans [13].

GTRBAC (*Generalized Temporal Role Based Access Control*) ajoute des contraintes temporelles au modèle RBAC hiérarchique. Il est décrit dans [41].

W-RBAC adapte l'utilisation du modèle RBAC à la notion de *workflow*. Ce modèle permet alors de décrire une politique RBAC adaptée à un processus. Ce modèle est décrit dans [78].

WS-RBAC introduit la notion de services *web*. Ce modèle est basé sur le langage *WS-Policy* [37, 71, 7] et permet d'exprimer une politique de CA de type RBAC adaptée aux services *web*.

1.4.3 OrBAC

Avant de voir apparaître la notion d'organisation au sein des politiques de CA, la notion d'équipe a permis de simplifier l'expression de politiques de CA. Le modèle TBAC (*Team Based Access Control*) [76] a permis de formaliser l'utilisation de la notion d'équipe dans le modèle RBAC. Le modèle OrBAC [43, 21] se base sur le modèle RBAC mais inclut une certaine hiérarchie dans les rôles ainsi que le concept d'organisation. Pour des raisons de compréhension et de présentation, le modèle OrBAC est présenté de manière séparée. Cependant, les différentes parties du modèle présentées ci-après sont toutes reliées entre elles au sein du même modèle. Dans le modèle OrBAC, les organisations permettent de représenter les structures organisationnelles du système que l'on modélise. Généralement, elle correspond à l'entreprise ou à la succursale à laquelle on s'intéresse.

Ensuite apparaissent les rôles ; ils correspondent, comme leur nom l'indique, au rôle que peut jouer une personne au sein du système. Ce rôle est activé par la personne lorsqu'elle se connecte au SI. Les personnes sont aussi représentées par une entité dans le modèle. Ces trois concepts

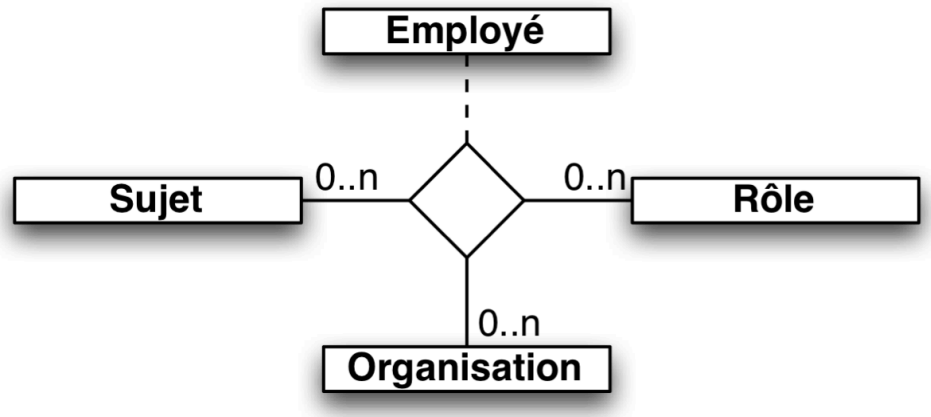

FIGURE 1.1 – La relation "*employee*" du modèle OrBAC

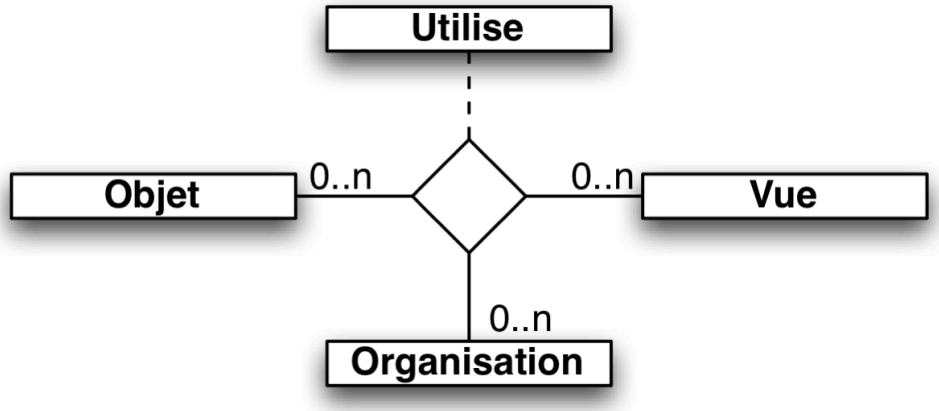

FIGURE 1.2 – La relation "*use*" du modèle OrBAC

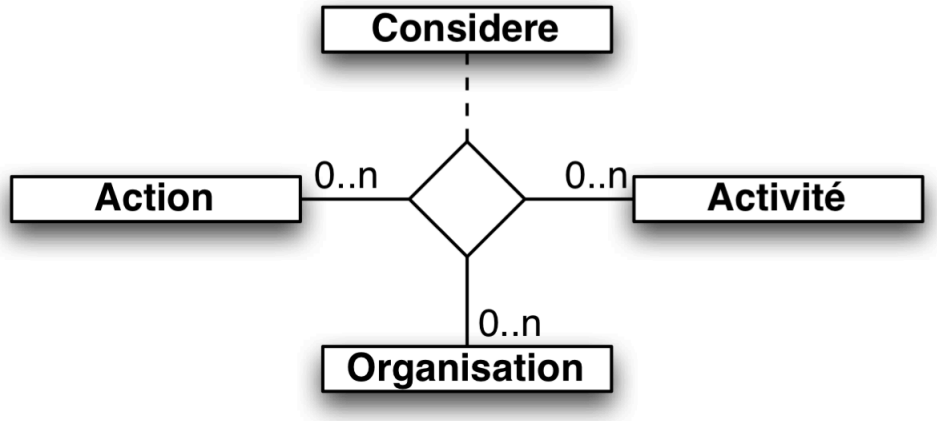

FIGURE 1.3 – La relation "*considere*" du modèle OrBAC

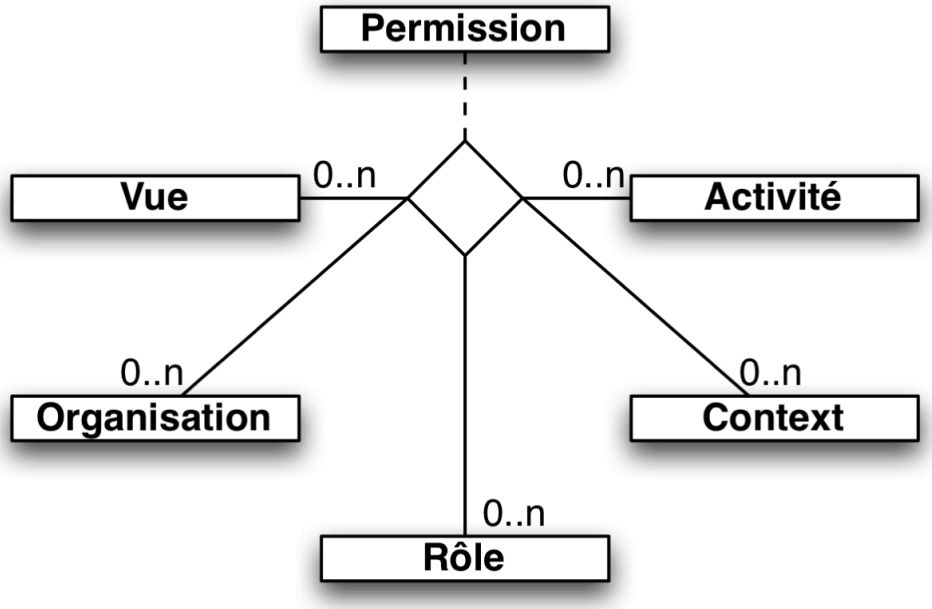

FIGURE 1.4 – La relation "*permission*" du modèle OrBAC

sont représentés par des entités au sein du modèle. Une première relation existe entre ces trois entités et indique qu'une personne est employée pour un certain rôle dans une organisation donnée. Cette relation est présentée à la figure 1.1. L'entité rôle peut être perçue comme une abstraction des personnes au sein du système. Cette abstraction sera aussi réalisée pour les objets. Dans le modèle arrivent deux nouvelles entités : les objets et les vues. Les objets correspondent à l'objet informatique du SI, ils peuvent être des fichiers, des tables de bases de données ... Les vues correspondent à leur abstraction au sein du système. Par exemple on peut considérer au sein du SI d'une banque la vue correspondant à un compte bancaire. Une relation permet de lier les entités correspondant aux organisations aux vues et aux objets. Cette relation permet de signaler que tel objet est utilisé de telle manière au sein d'une certaine organisation. Cette relation est présentée à la figure 1.2.

Pour pouvoir réaliser des actions au sein du système il faut encore ajouter deux nouvelles entités : action et activité. Les actions correspondent aux traitement sur les objets et les activités au traitement sur les vues. Les activités correspondent à une abstraction des actions au sein de la politique de sécurité. Une relation permet de préciser qu'au sein d'une organisation une certaine action sur un objet correspond à telle activité sur une vue. Cette relation est présentée à la figure 1.3. La dernière entité présente dans le modèle permet de rendre compte du contexte dans lequel un utilisateur réalise une action sur un objet. La dernière association utilisée permet de modéliser les permissions au sein du système. En effet, cette relation permet la réalisation d'une certaine activité par un certain rôle sur une vue donnée au sein d'une certaine organisation et dans un contexte précis. Cette relation est présentée à la figure 1.4.

1.4.4 ABAC

La méthode ABAC[1], définie dans [81], permet de décrire des politiques de CA en se basant sur les attributs des entités nécessaires à la description de la politique. Au contraire des méthodes RBAC et OrBAC, qui réalisent une abstraction des sujets au travers du concept de rôle, la méthode ABAC

[1] http ://www.axiomatics.com/attribute-based-access-control.html

attribue des caractéristiques particulières aux différents éléments nécessaires à l'expression d'une requête de demande d'accès. Les différentes entités retenues dans la méthode ABAC sont les sujets, les objets (appelés aussi ressources), les actions et l'environnement. Pour chacune de ces entités, on peut définir les attributs nécessaires à l'expression d'une politique de CA. La méthode ABAC est étroitement liée au langage XACML, dont l'utilisation est détaillée au chapitre 5.3.1.1.

1.4.5 SoDA

Certains modèles se focalisent sur l'expression d'un sous-ensemble des contraintes de CA. Par exemple, SoDA (*Separation of Duty Algebra*), ou algèbre de séparation des devoirs, permet d'exprimer uniquement des contraintes de SoD. Pour exprimer ces contraintes, cette méthode utilise une algèbre dont les constantes correspondent aux utilisateurs ou au rôles que les utilisateurs jouent. SoDA est décrit dans [50].

Pour un processus donné, on peut spécifier en SoDA que les utilisateurs doivent jouer un certain rôle. Par exemple $((Manager \bigodot Comptable) \bigotimes Tresorier)$, spécifie que pour exécuter le processus il faut un manager, un comptable et un trésorier. Le comptable et le manager peuvent correspondre à la même personne.

1.4.6 Expression de contrôle de transaction pour la séparation des devoirs

La notion de séparation des devoirs est récurrente dans le domaine du CA. En effet, cette notion permet d'ajouter une sécurité supplémentaire, qui consiste à découper une tâche en actions élémentaires et à attribuer chacune de ces actions élémentaires à des personnes différentes. De cette manière, pour que la réalisation d'une tâche soit corrompue, il faut que la réalisation de chacune de ses actions élémentaires soit corrompue, ce qui implique plus de personnes et est donc moins envisageable. Cette méthode permet d'exprimer le fait que les actions doivent être réalisées par différentes personnes. Le résultat obtenu [67] peut être comparé à une algèbre de processus, bien qu'il ne s'agisse ici que d'une esquisse de solution complète.

prepare•clerk ;
approve•supervisor ;
issue•clerk ;

FIGURE 1.5 – Exemple d'utilisation des opérateurs ; et • pour l'expression de la séparation des devoirs

L'expression :
prepare•clerk ;
approve•supervisor ;
approve•supervisor ;
approve•supervisor ;
issue•clerk ;
est équivalente à :
1 :prepare•clerk ;
3 :approve•supervisor ;
1 :issue•clerk ;

FIGURE 1.6 – Exemple de pondérations des actions pour l'expression de la séparation des devoirs

requisition•project-leader$\downarrow x$;
prepare•clerk ;
approve•purchasing-manager$\downarrow y_1$;
agree•project-leader$\downarrow x$;
reapprove•purchasing-manager$\downarrow y_2$;
issue•clerk ;

FIGURE 1.7 – Exemple d'assignation de valeurs et de choix pour l'expression de la séparation des devoirs

Les opérateurs et notations utilisés dans cette approche sont définis ci-après.

- Le symbole "•" permet d'associer un rôle à une action. Ici, les notions de rôle et d'action peuvent être comprises comme au sein du modèle RBAC.

- Le symbole ";" permet de réaliser une suite d'actions. Cet opérateur permet d'exprimer le fait que pour réaliser une tâche plusieurs actions élémentaires doivent être réalisées.

La figure 1.5 représente un exemple d'utilisation de ces opérateurs. En plus de ces opérateurs, on peut aussi ajouter l'opérateur ":". Cet opérateur permet d'exprimer une répétition de l'opération, le nombre de fois étant inscrit de manière préfixée à l'opérateur. Cette notation est illustrée à la figure 1.6.

De plus, on ajoute les opérateurs suivants :

- Le symbole "↓" permet d'associer une valeur à un rôle. Cette assignation est à mettre en relation avec la notion de "user" de la méthode RBAC. Cet opérateur permet d'assigner des tâches à deux personnes différentes qui peuvent avoir le même rôle.

- Le symbole "$\{\ldots\}$" permet de réaliser une fermeture. Cet opérateur permet d'exprimer le fait qu'une opération peut être réalisée un nombre arbitraire de fois.

- Le symbole "+" permet de réaliser un choix dans les actions. Il permet d'exprimer le fait que des actions ne sont pas obligatoires ou possèdent des alternatives.

L'utilisation de ces opérateurs est illustrée à la figure 1.7. Cette notation peut s'approcher d'une algèbre de processus. Du moins, elle donne des notions à rajouter dans une algèbre de processus de manière à la rendre utilisable pour exprimer une politique de sécurité.

1.5 Les utilisations des différents modèles

Les modèles de politiques de CA permettent d'exprimer des politiques de CA. Selon le formalisme utilisé pour exprimer le modèle, la modélisation peut être utilisée dans différents buts. Dans le

cadre de méthodes formelles, la politique de CA peut être vérifiée, c'est-à-dire vérifier si elle ne crée pas de *deadlocks*. Une spécification une fois vérifiée peut être implémentée [72], c'est-à-dire transformée en un composant logiciel permettant de vérifier que les requêtes reçues sont conformes à la politique de CA. D'autres approches [22, 64] intègrent les notions de sécurité et de CA tout au long du cycle de développement du logiciel.

1.5.1 Vérification

La principale prérogative, pour réaliser de la vérification sur un modèle, est que le langage utilisé pour réaliser le modèle soit formel. De nombreux langages formels de modélisation existe, cependant, les langages formels préférés en CA sont : Alloy [39], B [2], Event-B [3], Z [75],CSP [66] et Prolog. En effet, ces langages présentent pour la plupart la particularité d'être outillés pour réaliser de la vérification de modèles (*model checking*). Les travaux, qui se focalisent sur la vérification de modèles de politiques de CA, permettent de vérifier principalement la cohérence du modèle (c'est-à-dire qu'il ne présente pas de *deadlocks*) ou des propriétés quelconques (principalement des propriétés de sécurité exprimées en logique temporelle). Les problèmes de conflits (*i.e. deadlock*) apparaissent, par exemple, lorsqu'un modèle ne permet plus l'exécution d'aucune action lors de son exécution. Cette situation apparaît, par exemple, lorsque les interdiction ont été mal modélisées et ne permettent plus de réaliser aucune action.

Dans [82, 70], les auteurs présentent comment transformer un modèle de type RBAC en spécification Alloy. Le résultat obtenu est utilisé pour vérifier la cohérence du modèle ainsi que des propriétés de sécurité. Dans [70], le modèle de politiques de CA correspond au modèle RBAC auquel ont été ajoutées des contraintes d'administration et de SSD. Une fois le modèle réalisé les outils de vérification de Alloy sont utilisés afin de résoudre les conflits apparaissant dans la spécification. Dans [82], les politiques de CA correspondent au modèle RBAC, auquel ont été ajoutées des contraintes de hiérarchie de rôles. Les outils de vérifications de Alloy sont utilisés afin de vérifier des propriétés algébriques sur le modèle. Par exemple, les auteurs montrent l'équivalence entre le modèle de Bell et LaPadula et son implémentation réalisée en utilisant le

modèle RBAC. Dans [49], les auteurs expriment en langage Z une politique de CA. Le modèle de politique de CA est proche du modèle RBAC, il correspond au modèle RBAC utilisé en SecureUML. Les auteurs expriment aussi le modèle fonctionnel en Z. Les modèles de l'aspect fonctionnel et de la politique de CA sont mis en parallèle. Les outils d'animation permettent la validation des modèles en vérifiant que les scénarios normaux s'exécutent alors que les scénarios malicieux sont refusés. Dans [80], des modèles de type RBAC contenant des contraintes de SoD sont transformés en Z. Le résultat obtenu permet de vérifier la cohérence de la politique. Dans [48, 56], Les auteurs utilisent un modèle de l'aspect fonctionnel du système exprimé en UML. Les contraintes de CA sont exprimées à l'aide de la méthode UMLSec [51]. Les contraintes de CA ne pouvant être exprimées à l'aide la méthode UMLSec sont exprimées à l'aide de la méthode ASTD [27]. Les trois modèles sont traduits en B. Les outils de vérification de modèles utilisables en B peuvent alors être mis en œuvre pour vérifier des propriétés sur l'ensemble de la modélisation. Dans [35], les auteurs combinent en un modèle B des contraintes fonctionnelles et des contraintes de sécurité exprimées en CSP-OZ [9]. Ils utilisent ensuite la notion de raffinement pour prouver que le résultat vérifie les conditions de sécurité imposées. Dans [83], la politique de CA est exprimée en RW [31]. Le langage RW permet de décrire des permissions générales et ne respecte pas un modèle de CA particulier. La cohérence du modèle peut être prouvée à l'aide des outils associés au langage Prolog. Une fois la vérification réalisée le modèle est transformé en politique XACML.

1.5.2 Implémentation

1.5.2.1 SecureUML

SecureUML [8, 51] présente une solution efficace de l'implémentation d'une politique de CA lors du développement de logiciels distribués. Le modèle de politique de CA correspond au modèle RBAC. Nous allons voir dans quelles conditions cette méthode peut être mise en place ainsi que les résultats que l'on peut en attendre. Cette méthode s'adresse aux logiciels développés selon une approche MDA (*Model Driven Architecture*). Le principe est la création d'un logiciel en deux ou plu-

sieurs étapes. La première étape consiste en l'élaboration d'un PIM (*Platform Independent Model*). Ce modèle a la particularité d'être indépendant des technologies utilisées pour l'implémentation du logiciel final. Il correspond à une abstraction du résultat escompté. La deuxième étape consiste à réaliser un PSM (*Platform Specific Model*). Ce modèle est lié à la solution finale, donc aux technologies utilisées. Dans une approche MDA, le PSM correspond à l'implémentation du PIM. L'intérêt d'utiliser une approche MDA réside dans le fait que plusieurs outils permettent de réaliser une implémentation quasi automatique du PSM en fonction du PIM. L'exemple le plus courant est d'utiliser le langage UML (*Unified Modeling Langage*) de manière à réaliser le PIM, et d'utiliser différents outils pour transformer cette modélisation abstraite en un logiciel fonctionnant sur une architecture donnée. La plupart du temps, cette implémentation n'est pas complète, seul un squelette d'application est fourni. Les développeurs doivent alors le compléter en fonction des besoins spécifiés précédemment. De nos jours, le langage UML est l'un des plus populaire, c'est pourquoi la méthode Secure-UML s'en sert comme langage de description de systèmes. Cette méthode apporte une solution quant à la gestion d'accès aux données. Elle permet de considérer la politique de sécurité dès la phase de conception du SI considéré.

1.5.2.2 Implémentation en CSP de modèles exprimés en SoDA

Le langage SoDA permet d'exprimer des contraintes de SoD. L'article [10] présente une méthode permettant l'implémentation d'une solution de *monitoring* de contraintes exprimées en SoDA. Le modèle résultant, exprimé en CSP, comporte la simulation de trois processus. Le premier processus correspond aux contraintes de CA basées sur un modèle RBAC. Le second processus correspond aux contraintes de SoD exprimées en SoDA. Le troisième processus correspond aux contraintes fonctionnelles. L'ensemble de la spécification peut alors être vérifiée et le modèle de la politique de CA peut être utilisé pour l'élaboration de contrôleurs de sécurité.

1.5.2.3 XACML et quelques travaux connexes

Le langage XACML [57] est basé sur le langage XML et permet d'exprimer des politiques de CA correspondant au modèle ABAC. Une description détaillée du langage XACML est donnée à la section 5.3.1.1. Plusieurs travaux permettent de formaliser le langage XACML. Dans [45],une manière systématique de transformer une politique de CA exprimée en XACML en logique descriptive est décrite. Le résultat peut être utilisé pour comparer différentes politiques de CA ou faire de la vérification de propriétés sur une politique de CA donnée. Dans [16], les auteurs montrent comment transformer un modèle XACML en modèle CSP. La spécification résultante peut être utilisée pour faire du vérification de modèles à l'aide des outils disponibles en CSP. Dans [52], un modèle de la méthode XACML en logique du premier ordre est décrit. Des règles de traduction des instances de ce modèle vers une spécification Alloy sont décrites. Les modèles obtenus en Alloy sont utilisés pour faire de la vérification (vérification de la cohérence de la politique de CA et détection des conflits). Une algèbre permettant d'exprimer une politique de CA est décrite dans [63]. Cette algèbre permet de prendre en compte des permissions et des interdictions. Une méthode de transformation systématique de modèles exprimés à l'aide de cette algèbre vers un modèle XACML est aussi détaillée.

1.6 Comparaison

D'autres projets [32, 4, 74, 33, 58] s'intéressent à l'intégration de politiques de sécurité au sein des SI. Ces différents projets établissent des comparaisons des modèles de politiques de sécurité. Le projet ORKA définit un *framework* permettant de comparer les différents modèles de CA. Ce *framework* est décrit dans [5]. Il établit trente-quatre points de comparaison classés en huit catégories. Dans la suite, nous décrivons les différentes catégories avant de proposer un ensemble de critères inspirés de ce *framework*.

La première catégorie concerne la spécification des modèles. Cette catégorie permet de définir comment la politique est modélisée. Les critères de cette catégorie permettent de préciser si la po-

litique cible des actions de haut niveau (présentes dans la spécification) ou de bas niveau (présentes dans l'implémentation). Elle s'intéresse aussi au formalisme utilisé, en permettant de préciser si le langage utilisé est formel, semi-formel, informel ou structuré. Les différents critères permettent de définir le domaine applicatif ciblé par la méthode et aussi l'utilisation du modèle. L'utilisation du modèle correspond soit à la vérification de la politique, l'implémentation ou la vérification en temps réel.

La seconde catégorie concerne l'expressivité de la politique de CA. Cette catégorie concerne l'ensemble des concepts exprimables dans la politique. Par exemple, elle rend compte du nombre de niveaux d'abstraction utilisés (modélisation, implémentation, ...), du nombre de concepts de CA (permission, interdiction, obligation et SoD) qui peuvent être exprimés dans la politique. Pour pouvoir prendre en compte certains concepts de CA dynamiques, la politique de CA doit connaître l'état des *workflows* du SI. Cette catégorie rend aussi compte de la manière dont les instances des *workflows* du système sont représentées en mémoire.

La troisième catégorie permet de déterminer si un langage utilise des notations graphiques ou textuelles. Elle permet aussi de déterminer si un modèle représenté est compréhensible par un humain. Cette catégorie juge aussi de la modularité, de la capacité d'extension et de la définition de la sémantique du langage.

Une autre catégorie juge de la manière dont la modélisation est implémentée. Elle permet de savoir si la méthode a pour but d'être utilisée avec un noyau de sécurité ou d'être intégrée dans le système existant. Dans le cas où la méthode est utilisée avec un noyau de sécurité, les critères de cette catégorie permettent de savoir si l'implémentation du noyau est compatible avec les notions de point de prise de décision de la politique (PDP) et point de mise en application de la politique (PEP).

Une catégorie concerne la validation et permet de savoir si les méthodes de modélisation de politiques de CA permettent de réaliser la validation de la politique. Le cas échéant, elle permet de répertorier les différents outils permettant de réaliser du vérification de modèles.

D'autres catégories permettent de comparer les modèles selon l'administration qu'ils permettent de réaliser sur le modèle mis en place, la façon dont l'implémentation s'adapte aux systèmes existant ou encore de la maturité de la méthode.

Comme nous voulons établir une comparaison des modèles précédemment présentés, nous considérons les méthodes suivantes :

RBAC : Cette méthode n'inclut pas seulement le standard. Nous incluons sous la dénomination RBAC l'ensemble des extensions présentées : hiérarchie des rôles, contraintes d'activation des rôle, contraintes de SSD et DSD. Nous considérons secureUML comme une implémentation de cette méthode inclus dans ce modèle.

SoDA : Dans ce modèle nous incluons l'algèbre SoDA permettant d'exprimer des contraintes de SoD. De plus nous incluons aussi un modèle RBAC permettant d'exprimer les permissions du système. L'inclusion du modèle RBAC est nécessaire pour pouvoir considérer l'implémentation en CSP de politiques SoDA.

XACML : Nous incluons dans ce modèle le standard XACML, le modèle ABAC et l'ensemble des travaux de formalisation présentés dans la section 1.5.2.3.

OrBAC Nous appelons OrBAC la méthode OrBAC ainsi que son implémentation [20].

Les critères présents dans [5] sont trop nombreux. Tous ces critères ne sont pas forcément probant pour la comparaison de modèles de politiques de sécurité. Nous détaillons ci-après les critères que nous utilisons pour comparer les différentes méthodes.

représentation des workflows : Ce critère permet d'exprimer le fait que la méthode permet de représenter les instances de processus de l'organisation. Ce critère permet de savoir si le modèle peut prendre en compte les contraintes dynamiques de CA.

concepts de CA : Ce critère permet de comparer l'expressivité des différentes méthodes. En plus de préciser le nombre de catégories de critères pris en considération par la méthode, ce critère définit quels types de contraintes sont pris en compte par la méthode.

formalisme : Ce critère permet d'indiquer le formalisme des méthodes comparées. Le formalisme utilisé peut être formel ou structuré.

présentation : Ce critère permet de comparer la lisibilité des spécifications réalisées dans le modèle. Par lisibilité, nous entendons la représentation des spécifications réalisées : graphique, textuelle ...

vérification Ce critère permet de préciser si les spécifications réalisées dans la méthode peuvent être formellement vérifiées ou s'ils peuvent servir à réaliser des preuves.

critères \ modèles		RBAC	SoDA	XACML	OrBAC
représentation des workflows		*state-based*	*event-based*	non	*state-based (1)*
concepts de CA	permissions	oui	oui	oui	oui
	permissions avec contraintes	oui *(2)*	non	oui	oui
	interdictions	non	non	oui	oui
	interdictions avec contraintes	non pertinent	non pertinent	oui	oui
	obligations	non	non	non pertinent *(3)*	oui
	SSD	oui	oui	oui	oui
	DSD	oui	oui	non pertinent	oui
formalisme		formel	formel	structuré	formel
présentation		*(4)*	textuelle *(5)*	textuelle *(6)*	textuelle *(7)*
vérification		*(8)*	vérification de modèles possible *(9)*	*(10)*	non *(11)*

(1) pour exprimer les SoD, les *workflows* sont modélisés à l'aide de réseaux de Petri, avant d'être transformés en contextes. Dans le modèle, les *workflows* ne sont pas représentés mais des contextes permettent de savoir si les événements nécessaires à la prise de décision ont été exécutés.

(2) seules les contraintes temporelles d'activation des rôles peuvent être exprimées.

(3) un mécanisme d'exception existe en XACML. Il est décrit dans la section 5.3.1.1.

(4) de nombreux outils et méthodes se basent sur la méthode RBAC. La description d'une politique peut se faire par des moyens allant d'un diagramme entité relation à l'écriture de fichiers XML.

(5) la sémantique et la syntaxe d'une politique en SoDA sont formellement décrits. Dans le cadre de cette comparaison, le modèle est présenté sous la forme d'un modèle CSP.

(6) une politique XACML est décrite à l'aide de fichiers XML présentés dans la section 5.3.1.1.

(7) une politique OrBAC se présente sous la forme d'un ensemble de formules logiques.

(8) selon la méthode utilisée, des techniques de vérification de modèles, de simulation ou de preuve peuvent être envisagées.

(9) en vue des outils utilisés, des méthodes de vérification de modèles peuvent être envisagées ; leur utilisation n'a pas été présentée par les auteurs de la méthode.

(10) le langage XACML n'est pas formel. Cependant des travaux permettent d'en formaliser un sous-ensemble et d'effectuer des vérifications sur ce sous-ensemble.

(11) les problèmes de conflits sont gérés à l'aide de priorités définies dans la politique.

1.7 Conclusion

Cet état de l'art a permis d'éclaircir plusieurs points.

1. Il existe des modèles de CA permettant d'exprimer des contraintes de tous types dans un même modèle.

2. Il existe des modèles de CA basé sur les événements.

3. Il existe des modèles de CA permettant de faire de la vérification, des preuves ou de la simulation.

4. Il existe des modèles de CA permettant de vérifier simultanément l'aspect fonctionnel d'un modèle et la politique de CA associée.

Le tableau comparatif précédent nous apprend qu'aucune méthode précédemment présentée ne satisfait aux quatre critères précédents. Le but de cette thèse est de décrire la méthode EB^{3}SEC et son utilisation. Cette méthode a pour but de :

1. permettre aux utilisateurs d'exprimer tout type de contraintes de CA,

2. présenter une approche basée sur les événements,

3. présenter un formalisme pouvant permettre la vérification de propriétés sur les modèles exprimés,

4. permettre de vérifier la cohérence du système entier : aspect fonctionnel et CA,

5. s'adapter simplement à n'importe quelle organisation.

Chapitre 2

Exemple

Ce chapitre décrit l'exemple utilisé dans le reste de la thèse. Cet exemple est emprunté au domaine bancaire. L'aspect fonctionnel et la politique de CA du SI sont décrits.

2.1 Aspect fonctionnel

Cette partie décrit le principe de fonctionnement du SI d'une banque. L'ensemble de l'aspect fonctionnel n'est pas considéré, seuls sont décrits les éléments utiles pour la suite du document.

La partie ciblée du SI concerne les clients, les chèques et la procédure de dépôt les unissant. Au sein du système, les clients possèdent un nom et un compte. Seul est utilisé dans la suite le compte de dépôt, dans un souci de concision. Les clients peuvent être créés (creer_client), leur nom peut changer (modifier_nom) ainsi que la valeur de leur compte de dépôt (modifier_montant) et finalement le client peut aussi quitter la banque (supprimer_client). Les chèques sont représentés par un numéro unique, ils possèdent chacun leur propre montant. Les chèques sont créés (creer_cheque), leur valeur peut être modifiée (modifier_valeur) et ils sont finalement détruits (supprimer_cheque). La procédure de dépôt reliant les chèques et les clients correspond à plusieurs actions : le dépôt proprement dit (deposer), puis l'annulation ou la validation (annuler ou valider ou valider_directeur), dans certains cas il peut y avoir la vérification et l'enregistrement (verifier et enregistrer).

2.2 Politique de CA

La politique de CA permet de sécuriser les actions d'un SI. Elle décrit les droits accordés aux utilisateurs du SI. Ces droits assurent l'intégrité des données. L'exemple utilisé est le suivant :

- **règle 1 :** Seuls les guichetiers et les conseillers ont le droit d'effectuer un dépôt de chèque.
- **règle 2 :** Seuls les guichetiers et les directeurs sont autorisés à valider ou annuler un dépôt de chèque.
- **règle 3 :** Seuls les guichetiers et les conseillers sont autorisés à modifier le montant d'un compte bancaire.
- **règle 4 :** La validation ou l'annulation d'un dépôt de chèque ne peut être réalisée par la personne qui a effectué le dépôt de chèque.
- **règle 5 :** Si le montant d'un chèque excède une certaine somme limite, le dépôt de ce chèque doit être validé par deux personnes différentes. L'une de ces deux personnes doit être le directeur de la succursale. Dans la succursale de Québec ce seuil vaut $10\ 000\ \$$, tandis que dans la succursale d'Ottawa ce montant vaut $8\ 000\ \$$.
- **règle 6 :** La modification du montant du compte bancaire du client ayant déposé un chèque doit être fait par l'employé qui a effectué ce dépôt de chèque, avec le même rôle, dans la même succursale.
- **règle 7 :** Pour les chèques déposés dont le montant est supérieur à la somme limite il faut vérifier le chèque auprès de l'organisme émetteur, par une personne ayant le rôle de conseiller.
- **règle 8 :** Pour un chèque dont le montant est supérieur à la somme limite, la personne ayant effectué son dépôt doit l'enregistrer dans un registre spécial.

L'analyse de ces règles permet de mettre en avant certains points.

- L'ensemble des règles, sauf **règle 4** et **règle 8**, font appel à la notion de *rôle*. Les rôles suivants doivent apparaître dans la modélisation : guichetier, conseiller et directeur.
- Les actions du SI concernées par cette politique de CA sont les suivantes : deposer (**règle 1, 4, 5, 6, 7, 8**), annuler (**règle 2, 4**), valider (**règle 2, 4, 5**), valider_directeur (**règle 2, 4, 5**), crediter (**règle 3, 6**), verifier (**règle 7**), enregistrer (**règle 8**).
- Les règles **règle 1, 2, 3** correspondent à des règles de CA de type *permission*.
- La règle **règle 4**, correspond à un regroupement de deux règles de CA correspondant chacune à une SoD. La première SoD (respectivement la seconde) intervient entre deposer et valider (respectivement entre deposer et annuler). Cette règle introduit aussi une SoD entre les actions deposer et valider_directeur.
- La règle **règle 5** correspond à une contrainte de CA de type SoD.
- La règle **règle 6** correspond à une contrainte de CA de type *obligation*, cette obligation porte sur les actions deposer et crediter. Les règles **règle 7** et **règle 8** correspondent aussi à des contraintes de CA de type *obligation*.

Pour illustrer, certains concepts décrits dans le chapitre 4, nous avons défini des alternatives aux règles précédentes (les différences sont indiquées par l'utilisation d'une police en *italique*).

- **règle 4 bis :** La validation ou l'annulation d'un dépôt de chèque ne peut être réalisée par la personne qui a effectué le dépôt de chèque. *L'annulation ou la validation du dépôt doit être réalisée dans l'heure suivant l'exécution du dépôt.*
- **règle 4 ter :** La validation ou l'annulation d'un dépôt de chèque ne peut être réalisée par la personne qui a effectué le dépôt de chèque. *L'annulation ou la validation du dépôt doit être réalisée dans l'heure suivant l'exécution du dépôt. Si la validation ou l'annulation n'a pas eu lieu alors, un courriel doit être envoyé à la personne ayant exécuté le dépôt de chèque.*

- **règle 4 quater :** La validation ou l'annulation d'un dépôt de chèque ne peut être réalisée par la personne qui a effectué le dépôt de chèque. *Cependant, la directrice ou le directeur de la succursale peut déposer, valider et annuler un chèque.*

- **règle 5 bis :** *Un dépôt de chèque doit être validé par deux personnes différentes, quelque soit son montant.*

- **règle 5 ter :** *Un dépôt de chèque doit être validé par deux personnes différentes dans l'heure suivant le dépôt, sinon un courriel est envoyé à la personne ayant réalisé le dépôt du chèque.*

- **règle 5 quater :** *Un dépôt de chèque doit être validé par deux personnes différentes, dont un directeur, quelque soit son montant.*

- **règle 6 bis :** La modification du montant du compte bancaire du client ayant déposé un chèque doit être faite par l'employé qui a effectué ce dépôt de chèque, avec le même rôle, dans la même succursale. *La modification du montant doit être faite dans l'heure suivant le dépôt du chèque.*

- **règle 8 bis :** *Pour tous les chèques déposés*, la personne ayant effectué son dépôt doit l'enregistrer dans un registre spécial.

- **règle 8 ter :** Pour un chèque dont le montant est supérieur à la somme limite, la personne ayant effectué son dépôt doit l'enregistrer dans un registre spécial. *Si l'enregistrement n'est pas fait un courriel doit être envoyé à la personne ayant réalisé le dépôt.*

Chapitre 3

Les méthodes EB^3 et EB^{3}SEC

Cette section présente les méthodes EB^3 [26, 28] et EB^{3}SEC [47], la première permettant la modélisation fonctionnelle d'un SI, la seconde permettant la modélisation de la politique de CA d'un SI. Pour illustrer l'utilisation de ces méthodes, l'exemple du chapitre 2 est utilisé.

3.1 Modélisation de la partie fonctionnelle à l'aide de la méthode EB^3

La méthode EB^3 est une méthode formelle de spécification orientée événement. Cette méthode a été spécialement conçue pour la spécification des SI.

3.1.1 La méthode EB^3

En EB^3, les types d'entités et les associations sont décrites comme des boîtes noires à l'aide d'expressions de processus. L'algèbre de processus, utilisée en EB^3, est inspirée de CSP [36] et de LOTOS [15], mais est adaptée aux SI. Les expressions de processus permettent de préciser les traces valides du système, c'est-à-dire les séquences valides d'événements reçus en entrée. Les sorties du système sont obtenues à l'aide de fonctions définies sur les traces valides du système.

Une spécification EB^3 se compose de quatre parties. Le diagramme de classes est utilisé de manière à définir les différentes classes et les associations évoluant au sein du SI. Le comportement du système est défini à l'aide d'expressions de processus. Celles-ci permettent d'expliciter les traces valides du système. Les autres composantes d'une spécification EB^3 ne sont pas détaillées, car elles ne sont pas réutilisées en EB^{3}SEC.

En EB^3, les actions du système sont représentées avec leurs arguments dans la spécification ; les événements reçus par le système de la part des utilisateurs correspondent à des instances des actions du système. Une action constitue une expression de processus élémentaire. Une expression de processus non élémentaire est constituée à l'aide des opérateurs suivants. Soit $\mathbf{e}_1$, $\mathbf{e}_2$ et $\mathbf{e}(\ldots)$ trois expressions de processus.

- La *séquence* est utilisée pour exécuter deux expressions de processus à la suite. Elle est représentée par un point : $\mathbf{e}_1 \,.\, \mathbf{e}_2$.

- Le *choix* est utilisé pour des expressions de processus qui n'ont pas à être toutes réalisées. Mais pour continuer l'une au moins doit être achevée. Le choix est représenté par une barre verticale : $\mathbf{e}_1 \mid \mathbf{e}_2$.

- Le choix quantifié est une variante de l'opérateur précédent. Lorsqu'une expression de processus dépend de la valeur d'une variable libre, le choix quantifié exprime le fait que cette expression de processus peut être exécutée pour une valeur particulière de cette variable libre. Le choix quantifié s'utilise de la manière suivante : $\mid x \in ens \;:\; \mathbf{e}(x)$. Cette notation signifie que $\mathbf{e}(x)$ est exécuté avec une valeur spécifique de x prise dans l'ensemble ens.

- Le caractère générique _ est un raccourci syntaxique utilisé à la place d'un choix quantifié. Par exemple, $\mathbf{e}(\_)$ signifie $\mid x \in ens \;:\; \mathbf{e}(x)$.

- La *composition parallèle* permet d'exprimer que deux expressions de processus peuvent être exécutées dans n'importe quel ordre. S'il s'agit d'expressions de processus possédant des

actions en commun, ces actions doivent être synchronisées. La composition parallèle est représentée par deux barres parallèles : $\mathbf{e}_1 \parallel \mathbf{e}_2$.

- L'*entrelacement* correspond à une composition parallèle sans synchronisation. L'entrelacement est représenté par trois barres verticales : $\mathbf{e}_1 \lll \mathbf{e}_2$.

- L'entrelacement quantifié noté $\lll x \in ens \ : \ \mathbf{e}\,(\,x\,)$ correspond à l'entrelacement des expressions de processus $\mathbf{e}\,(\,x\,)$ pour toutes les valeurs de x présentes dans l'ensemble ens.

- La *garde* permet de conditionner l'exécution d'une expression de processus à l'état du système. Les gardes s'expriment de la manière suivante : $condition \implies \mathbf{e}_1$. L'opérande gauche d'une garde, ici appelée $condition$, correspond à une formule logique quantifiée du premier ordre pouvant utiliser les attributs des différentes classes décrites dans le diagramme de classes.

- La *fermeture de Kleene* permet de répéter une opération un nombre arbitraire de fois, elle se note de la manière suivante : $\mathbf{e}_1{}^*$.

3.1.2 Modélisation de l'exemple

L'exemple présenté dans le chapitre 2 est modélisé ici. Toutefois seule la partie fonctionnelle est modélisée en EB^3. Deux classes sont utilisées : Client et Cheque. Leurs attributs clés sont respectivement clId et cId. Les attributs valeur, nom et montant permettent de spécifier la valeur d'un chèque déposé, le nom d'une personne et le montant de son compte de dépôt. Le chèque et son bénéficiaire sont reliés par l'association depot. Les actions autorisées au sein du système sont la création et la suppression des objets provenant des différents classes. La modification des valeurs des attributs des différents objets est aussi permise. Le diagramme de classes est présenté à la figure 3.1. Dans la suite du document, la convention suivante est adoptée : chaque classe NomClasse a un attribut clé nId ou noId dont le type est nomClasseId.

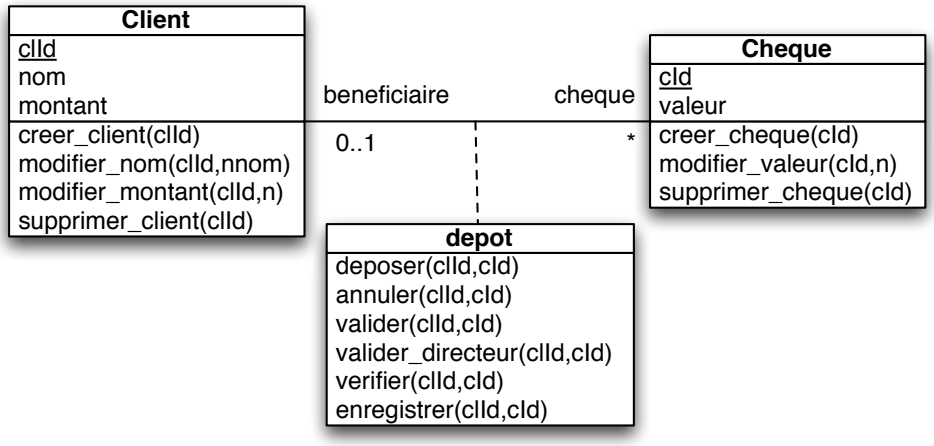

FIGURE 3.1 – Diagramme de classes de la procédure de dépôt de chèques

Les expressions de processus décrivant le comportement des classes et des relations sont obtenues à l'aide des patrons de conception présentés dans [28]. L'expression de processus obtenue pour la classe Client est :

1. **Client** $(\, clId : clientId \,) \triangleq$
2. creer_client($clId$)
3. **.** ((modifier_nom($clId, \_$)
4. | modifier_montant($clId, \_$))*
5. ||| ||| $cId \in chequeId$: **depot** $(\, clId, cId \,)$ *)
6. **.** supprimer_client($clId$)

Cette expression permet dans l'ordre de créer un client, de le modifier et de le supprimer. Les modifications d'un client permettent le changement des valeurs de ses attributs mais aussi le fait de l'associer à aucun ou plusieurs chèques. L'expression de processus décrivant le comportement de la classe Cheque est donnée par :

1. **Cheque** ($cId : chequeId$)$\triangleq$
2. creer_cheque(cId)
3. **.** (modifier_valeur($cId, \_$)*
4. ||| **depot** ($\_, cId$)*)
5. **.** supprimer_cheque(cId)

Cette expression de processus permet dans l'ordre de créer, de modifier et de supprimer les objets de la classe Cheque. Les modifications possibles sont le changement de valeurs de ses attributs et le fait de pouvoir l'associer à un seul objet de type Client. L'expression de processus décrivant le fonctionnement de l'association depot est :

1. **depot** ($clId : clientId, cId : chequeId$)$\triangleq$
2. deposer($clId, cId$)
3. **.** (verifier($clId, cId$)|| enregistrer($clId, cId$))
4. **.** (annuler($clId, cId$)
5. | (valider($clId, cId$)
6. | (valider($clId, cId$)|| valider_directeur($clId, cId$))))

Cette expression de processus permet d'associer un chèque à son bénéficiaire. Le compte du bénéficiaire est crédité après la ou les validations. Si le dépôt de chèque n'est pas validé, le compte du bénéficiaire n'est pas crédité.

1. **main** ( )$\triangleq$
2. (||| $clId \in clientId$: **Client** ($clId$))
3. ||
4. (||| $cId \in chequeId$: **Cheque** (cId))

Cette expression de processus $main$ est l'expression de processus principale. Dans notre cas, elle permet d'appeler les expressions de processus concernant les chèques et les clients.

3.2 Modélisation d'une politique de CA à l'aide de la méthode EB^{3}SEC

Cette partie décrit l'algèbre EB^{3}SEC. La méthode EB^3 permet de modéliser la procédure de dépôts. Elle permet aussi d'exprimer les contraintes de CA au sein de la modélisation. Cependant, la méthode EB^3 ne permet la séparation de la modélisation de la partie fonctionnelle et de la modélisation de la politique de CA. La méthode EB^{3}SEC, basée sur la méthode EB^3 et présentée dans la suite, permet cette séparation. La suite de cette section présente les différents éléments composants un modèle EB^{3}SEC.

L'exemple utilisé n'est pas celui donné dans le chapitre 2 mais un exemple simplifié.

- **Règle a** : Les guichetiers peuvent réaliser toutes les actions, sauf **annuler** et **valider**, pour les clients et les chèques de l'agence dans laquelle ils travaillent.
- **Règle b** : Les directeurs peuvent réaliser toutes les actions pour les clients et les chèques de l'agence dans laquelle ils travaillent.
- **Règle c** : Seul l'utilisateur ayant créé un chèque peut le supprimer.

De manière à ce que l'argent des dépôts de chèque ne soit pas détournée, la réalisation des actions aboutissant au dépôt d'un chèque est déléguée à différentes personnes. Ainsi, selon leur rôle, les employés de la banque pourront ou non réaliser certaines actions. Toutefois, le chef d'agence peut tout de même réaliser toutes les actions, car il doit pouvoir réaliser le dépôt de chèque d'un client en toutes circonstances. De manière à réaliser cette séparation des devoirs, les contraintes de CA associées à la procédure de dépôt de chèque au sein d'une institution bancaire sont les suivantes. Dans un souci d'illustration, nous considérons aussi les variantes suivantes des règles précédentes :

- **Règle a bis** : Les guichetiers peuvent réaliser toutes les actions, sauf **annuler** et **valider**, pour les clients et les chèques de l'agence dans laquelle ils travaillent. *Toutefois, les guichetiers*

peuvent réaliser l'action valider *pour les chèques dont le montant est inférieur à une certaine limite.*

Les contraintes concernant la limitation de la réalisation des actions à un certain rôle (*i.e.* **Règle a, b**) ne dépendent pas de l'état des processus d'affaire de l'institution bancaire, elles sont toujours valides, peu importe l'état du système. Au contraire, l'obligation de réaliser toutes les opérations concernant un chèque (*i.e.* **Règle c**) dépend de l'état des processus d'affaire de l'institution. En effet, la validation des modifications et de l'annulation ou de la validation du dépôt de chèque dépend de l'endroit où le chèque a été déposé.

3.2.1 Le diagramme de classes de sécurité

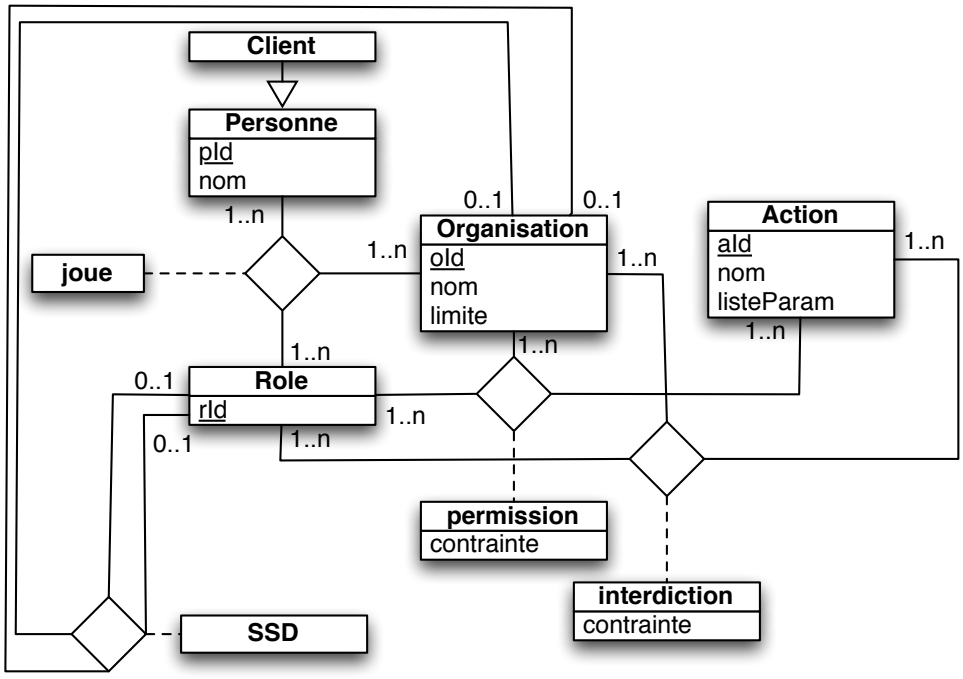

FIGURE 3.2 – Diagramme de classes utilisé pour sécuriser la procédure de dépôt de chèque

En EB^{3}SEC, le diagramme de classes est utilisé pour exprimer les règles statiques de la politique de CA. Il s'inspire de la méthode OrBAC [43]. Cependant, au contraire du modèle utilisé par la méthode OrBAC, le modèle utilisé par la méthode EB^{3}SEC est ajustable : il s'adapte aux besoins de sécurité de l'application que l'on veut sécuriser. Pour exprimer la possibilité d'exécuter un événement, il faut en connaître l'acteur. En effet, la politique de sécurité peut vouloir filtrer les événements en fonction de l'utilisateur désirant les réaliser. De manière plus générale, la politique peut aussi filtrer les événements en fonction du rôle joué par l'utilisateur voulant réaliser une action. Dans l'exemple de la banque, les permissions accordées aux guichetiers sont différentes de celles accordées aux directeurs. Le filtrage des événements peut aussi se réaliser en fonction de l'organisme dans lequel l'action se réalise. Dans l'exemple de la banque, les guichetiers et directeurs n'ont accès qu'aux clients et aux chèques de la succursale dans laquelle ils travaillent. La politique de CA dépend aussi de l'action dont l'événement est instancié. Les notions, précédemment citées, sont communes aux différentes politiques de sécurité. Ces notions sont appelées *paramètres de sécurité*. Le diagramme de classes utilisé pour l'exemple de l'institution bancaire est présenté à la figure 3.2.

La classe Organisation permet de pouvoir spécifier les différents organismes utiles à la conception de la politique de CA, cette classe permet par exemple de pouvoir créer les différentes succursales d'une institution bancaire. La classe Personne permet d'instancier les différents utilisateurs du système. De plus, la classe Client permet de référencer les clients du système déjà présents dans le SI. La classe Role permet de créer les différents rôles présents dans une institution bancaire, comme par exemple, les clients, les guichetiers et les directeurs. La classe Action contient l'ensemble des actions visées par la politique de sécurité modélisée. Pour l'exemple de l'institution bancaire, cette classe contient les actions propres à chacune des entités présentes dans la modélisation EB^3 (*i.e.* : creer_client( )...). L'attribut nom de la classe action est à valeur unique, il peut servir de clé pour les différentes instances de cette classe. L'association joue permet de mettre en relation les personnes, les rôles, et les organisations. Elle permet d'exprimer le rôle joué par chaque utilisateur au sein d'une organisation. Par exemple elle permet d'indiquer nominativement

personne

pId	nom
1	alphonse
2	boris
3	catherine
4	damien
5	elise
6	franck

role

rId	nom
1	client
2	guichetier
3	directeur

organisation

oId	nom	limite
1	Montreal	8000
2	Toronto	10000

joue

pId	rId	oId
1	1	1
2	2	1
3	3	1
4	1	2
5	2	2
6	3	2

action

aId	nom	listeParam
1	creer_client	clId
2	modifier_nom	pId,nnom
3	modifier_montant	pId,n
4	supprimer_client	clId
5	deposer	pId,cId
6	annuler	pId,cId
7	valider	pId,cId
8	creer_cheque	cId
9	modifier_valeur	cId,n
10	supprimer_cheque	cId

permission

rId	oId	aId	contrainte
3	1	1	$TRUE$
	...		
3	1	10	$TRUE$
2	1	1	$TRUE$
	...		
2	1	5	$TRUE$
2	1	8	$TRUE$
	...		
2	1	10	$TRUE$
	...		

FIGURE 3.3 – Instanciation du diagramme de classes de l'exemple de l'institution bancaire

les directeurs de chaque succursale. Les autres classes et associations sont utilisées pour exprimer les différents types de règles de CA.

3.2.1.1 Les permissions sans contraintes

L'association permission entre les classes role, action et organisation permet d'exprimer les permissions sans contraintes de la politique de CA. Les permissions sans contraintes sont les permissions qui ne dépendent pas de l'état du système. La figure 3.3 est une instanciation de cette classe. Elle indique que dans la première succursale, alphonse est un client, boris est guichetier et catherine est directrice. Tandis que dans la seconde succursale, damien est un client, élise est guichetière et franck est directeur. L'association permission permet d'indiquer que les guichetiers peuvent réaliser toutes les actions sauf annuler et valider. De plus, cette même association permet de préciser que les directeurs ont le droit de réaliser toutes les actions. Dans cet exemple, les permissions ont été indiquées uniquement pour la première succursale, il en va de même pour la seconde.

permission

rId	oId	aId	contrainte
2	1	7	$cheque(cId).valeur < organisation(oId).limite$
2	2	7	$cheque(cId).valeur < organisation(oId).limite$

FIGURE 3.4 – Exemple de permissions avec contraintes

3.2.1.2 Les permissions avec contraintes

Les permissions avec contraintes permettent d'établir une permission qui est valable uniquement si une condition est vraie. Les conditions utilisées peuvent porter sur l'état du système et les paramètres de l'action concernée. La règle **Règle a bis** est un exemple de permission avec contraintes. La limite utilisée pour la condition de cette permission dépend de la succursale et est représentée en ajoutant un attribut *limite* à la classe organisation. La contrainte est définie par l'attribut contrainte de l'association permission. La figure 3.4 donne un exemple de permission avec contraintes. La syntaxe utilisée pour décrire les gardes est définies dans [46]. Elles correspondent à des prédicats logiques du premier ordre. L'exemple $cheque(cId).valeur < organisation(oId).limite$ explicite le fait que l'action valider ne peut être réalisée par les guichetiers que lorsque le montant du chèque est inférieur à la limite de la succursale. Le *oId* utilisé dans le prédicat correspond à la succursale intervenant dans la permission tandis que le *cId* fait référence au paramètre de l'action utilisée dans la permission. Les variables utilisées dans les contraintes proviennent soit de la table action, pour une action donnée, ou du prédicat statique, décrit dans la section 3.2.2.1.

3.2.1.3 Les interdictions

Cette section introduit les interdictions et une manière de les modéliser en EB^{3}SEC. Le problème du conflit enter les permissions et les interdictions est traité à la section 3.2.2.1. L'association interdiction entre les classes role, action et organisation permet d'exprimer les interdictions sans contraintes de la politique de CA. Elle permet d'exprimer le fait que les guichetiers ne peuvent réaliser les actions valider et annuler. La figure 3.5 montre les instances supplémentaires à rajouter pour intégrer ces interdictions à la modélisation.

interdiction			
rId	oId	aId	contrainte
2	1	7	$TRUE$
2	2	7	$TRUE$
2	1	6	$TRUE$
2	2	6	$TRUE$

FIGURE 3.5 – Exemple d'interdictions sans contraintes

interdiction			
rId	oId	aId	contrainte
2	1	7	$cheque(cId).valeur > organisation(oId).limite$
2	2	7	$cheque(cId).valeur > organisation(oId).limite$

FIGURE 3.6 – Exemple d'interdictions avec contraintes

3.2.1.4 Les interdictions avec contraintes

De la même manière qu'il est possible d'exprimer des permissions avec contraintes, on peut définir des interdictions avec contraintes. L'exemple de la figure 3.6 exprime la règle **Règle a bis** à l'aide d'une interdiction avec contraintes : il est interdit pour les guichetiers de réaliser l'action **valider** pour les chèques dont la valeur est supérieure à une certaine limite.

3.2.1.5 La séparation des devoirs statique (SSD)

La SoD permet de s'assurer qu'une procédure, comme un dépôt de chèque, n'est pas corrompue. La transaction est alors découpée en actions dont la réalisation est confiée à des entités différentes. De cette manière, le risque de corruption est moindre du fait que plusieurs acteurs interviennent dans le déroulement de la transaction. La SoD peut être résolue de différentes manières. Nous présentons ici comment la résoudre de manière statique.

Pour des besoins d'illustration, nous modélisons une SSD pour sécuriser la procédure de dépôt de chèques : Les guichetiers ont le droit de réaliser toutes les actions sauf annuler et valider (*i.e.* **Règle a**) et les directeurs peuvent réaliser toutes les actions sauf deposer (contradiction avec la règle **Règle b** et le fait que les directeurs puissent toujours réaliser toutes les actions inhérentes à un dépôt de chèque). La classe-association SSD de la figure 3.2 contient les

SSD			
rId	oId	rId	oId
2	1	3	1
2	2	3	2

FIGURE 3.7 – Exemple de SSD

couples rôles/organisations qui sont en conflit. Elle correspond à la manière habituelle de traiter les problème de SSD en RBAC [23], adaptée au besoin de la modélisation, c'est-à-dire en incluant la notion d'*organisation*. Pour rendre effective cette contrainte de SSD, il faut ajouter à la modélisation les instances de la figure 3.7.

Pour s'assurer que les contraintes de SSD soient respectées, il faut que toute instance du diagramme de classes vérifie alors le prédicat suivant :

$$\forall x \in personId \cdot \forall (y_1, y_2) \in roleId^2 \cdot \forall (z_1, z_2) \in organisationId^2 \cdot$$

$$(x, y_1, z_1) \in joue \wedge (x, y_2, z_2) \in joue \wedge (y_1, z_1) \neq (y_2, z_2) \Rightarrow \langle y_1, z_1, y_2, z_2 \rangle \notin \text{SSD}$$

3.2.2 Les expressions de processus

Dans l'exemple de l'institution bancaire réalisé en EB^3, les expressions de processus permettent d'ordonnancer les actions et d'exprimer le comportement des instances en fonction des événements acceptés. À l'aide de la méthode EB^{3}SEC, l'exécution des événements est aussi filtrée pour être conforme à la politique de CA. Pour réaliser cette vérification, les événements sont transformés en *événements sécurisés*. Les événements sécurisés doivent être acceptés par l'expression de processus du modèle EB^{3}SEC de la politique de CA. Plusieurs approches sont alors possibles pour définir les expressions de processus sécurisées. La première consiste à prendre les expressions de processus définies en EB^3 et à en sécuriser les actions. La deuxième approche consiste à définir une expression de processus pour chaque règle de CA. La dernière approche considère chaque

action du SI et donne l'expression de processus qui la sécurise. Dans la suite du paragraphe, nous donnons la définition d'une action sécurisée et détaillons chacune des approches.

3.2.2.1 Définition d'une action sécurisée

Les événements sont filtrés en fonction de paramètres de sécurité. Chaque paramètre de sécurité est typiquement représenté par une classe dans le diagramme de classes de sécurité. Si on considère le diagramme de classes de la figure 3.2, ils sont au nombre de trois : l'utilisateur (instance de la classe Personne), le rôle et l'organisation. Une action sécurisée est alors définie par le quadruplet $\langle p, r, o, evt\rangle$ avec $p \in personId$, et $r \in roleId$, $o \in organisationId$ et $label(evt) \in Action().nom$. L'expression $\langle boris, guichetier, montreal, \mathsf{valider}(1,1)\rangle$ est un exemple d'événement sécurisé stipulant que *boris* dans le rôle de *guichetier* réalise dans la succursale *Montréal* l'action valider pour le chèque numéro *1* et le client numéro *1*.

Une spécification de politique de CA est composée de deux parties : un diagramme de classes et une expression de processus. La sémantique est donnée en deux parties. La sémantique du diagramme de classes, de la figure 3.2, est donnée par un prédicat. Ce prédicat indique si une action est acceptée ou refusée. Ce prédicat est appelé prédicat statique, car il donne la sémantique des contraintes statiques. Il dépend de l'état du système mais pas de l'état des expressions de processus utilisées pour modéliser le système.

1. $sp(\langle p, r, o, evt\rangle) \triangleq$
2. $\quad \langle p, r, o\rangle \in joue$
3. $\quad \wedge \langle r, o, idEvt(evt)\rangle \in permission \wedge permission(r, o, idEvt(evt)).contrainte$
4. $\quad \wedge (interdiction(r, o, idEvt(evt)).contrainte \Rightarrow \langle r, o, idEvt(evt)\rangle \notin interdiction)$

La fonction $idEvt$ est définie par $idEvt(evt) = Action(label(evt)).aId$.

La ligne **2.** du prédicat statique permet de vérifier que la relation joue est bien respectée, c'est-à-dire que la personne a bien le droit de jouer le rôle précisé dans l'organisation donnée. La ligne **3.** permet de s'assurer que l'action est permise pour cet environnement de sécurité. Pour

les permissions sans contraintes, l'attribut contrainte leur correspondant vaut $TRUE$, seule la partie vérifiant que l'action est autorisée pour le rôle dans l'organisation est considérée. Pour les permissions avec contraintes, il faut aussi que la contrainte soit vraie sinon le prédicat statique n'est pas vrai. Pour les interdictions, si elles sont sans contraintes, alors le rôle l'organisation et l'action ne doivent pas se trouver dans la relation interdiction pour que l'événement soit autorisé. Pour les interdictions avec contraintes, si la contrainte est fausse alors la ligne **3.** est vraie, mais si le prédicat est vrai alors l'action ne doit pas être interdite pour le rôle dans l'organisation. Lorsque des contraintes sont utilisées, les variables apparaissant dans leur définition obtiennent une valeur, en effectuant une substitution à partir de l'événement sécurisés. Pour l'exemple d'action sécurisée précédent (le montant du chèque numéro *1* est de $11\ 000\$$) le prédicat statique vaut : $\langle boris, guichetier, montreal, \mathsf{valider}(\,1,1\,)\rangle$

1. $sp(\langle boris, guichetier, montreal, \mathsf{valider}(\,1,1\,)\rangle) \triangleq$
2. $\langle 2,2,1\rangle \in joue$
3. $\wedge\langle 2,1,7\rangle \in permission \wedge 11000 < 8000$
4. $\wedge(11000 > 8000 \Rightarrow \langle 2,1,7\rangle \notin interdiction)$

Le prédicat statique ne mentionne pas les contraintes de SSD, car on suppose qu'une instance du diagramme de classes est valide, et donc que la relation joue est cohérente avec la SSD.

La seconde partie de la politique de CA correspond aux expressions de processus et au fait que l'action est permise par ces expressions de processus.

3.2.2.2 Première approche : sécuriser les expressions de processus fonctionnelles

Les expressions de processus EB^{3}SEC, qui prennent en compte les règles de CA, peuvent être obtenues à l'aide des expressions de processus EB^3. Les actions doivent alors être sécurisées à l'aide des quadruplets pour définir les actions sécurisées du modèle EB^{3}SEC. Dans l'exemple simplifié de la banque on obtient les expressions de processus suivantes.

1. **client** ($clId : clientId$)$\triangleq$
2. $|\, o \in organisationId$:
3. $(|\, r \in \{guichetier, directeur\}$: $\langle \_, r, o,$ creer_client($clId$)$\rangle$) .
4. (
5. ($(|\, r \in \{guichetier, directeur\}$:
6. $\langle \_, r, o,$ modifier_nom($clId, \_$)$\rangle$).
7. | $(|\, r \in \{guichetier, directeur\}$:
8. $\langle \_, r, o,$ modifier_montant($clId, \_$)$\rangle)^*$
9. $|||$
10. $||| \, cId \in chequeId$: **depot** ($clId, cId$) *
11.).
12. $|\, r \in \{guichetier, directeur\}$:
13. $\langle \_, r, o,$ supprimer_client($clId, \_$)$\rangle$

Cette expression ordonne toujours la création, les modifications et la suppression d'une personne. Toutefois, la quantification portant sur la variable o (ligne **2.**), oblige le dossier d'un client à être suivi dans la même agence. Le suivi du dossier est indépendant de l'employé du moment que le rôle est respecté. Le symbole _ permet de ne pas préciser la personne s'occupant du dossier. N'importe quelle personne de l'agence, du moment qu'elle ait le rôle de guichetier ou de directeur, peut réaliser l'action utilisée dans cette expression. La quantification sur la variable r (ligne **12.**) permet uniquement aux guichetiers et aux directeurs de réaliser ces actions. De cette manière, un client ne pourra pas créer d'autres clients ou modifier son propre dossier.

1. **cheque** ($cId : chequeId$)$\triangleq$
2. $|\, o \in organisationId$: $(|\, p \in personId$:
3. $(|\, r \in \{guichetier, directeur\}$:
4. $\langle p, r, o,$ creer_cheque(cId)$\rangle$
5. . $((|\, r' \in \{guichetier, directeur\}$:
6. $\langle \_, r', o,$ modifier_valeur($cId, \_$)$\rangle)^*$
7. $|||$ **depot** ($\_, cId$)*)
8. . $\langle p, r, o,$ supprimer_cheque(cId)$\rangle$))

Cette expression autorise toujours la création, la modification, puis la suppression d'un chèque. En revanche, elle oblige pour un chèque donné à réaliser toutes ces opérations dans la même agence. De plus, ces actions ne peuvent être exécutées que par les guichetiers et les chefs de cette agence. Les choix quantifiés sur les variables r et p permettent de respecter la contrainte de sécurité fonctionnelle obligeant l'utilisateur à être la même personne avec le même rôle pour la création et la suppression d'un chèque.

1. **depot** ($clId : clientId, cId : chequeId$)$\triangleq$
2. $\quad | \, o \in organisationId$:
3. $\quad\quad (| \, r \in \{guichetier, directeur\}$:
4. $\quad\quad\quad \langle \_, r, o,$ deposer($clId, cId$)$\rangle$
5. $\quad\quad\quad .($ $\langle \_, r, o,$ verifier($clId, cId$)$\rangle \| \langle \_, r, o,$ enregistrer($clId, cId$)$\rangle$
6. $\quad\quad\quad .($ $\langle \_, directeur, o,$ annuler($clId, cId$)$\rangle$
7. $\quad\quad\quad\quad | ($ $cheque(cId).valeur < oraganisation(o).limite \Longrightarrow$
8. $\quad\quad\quad\quad\quad \langle \_, r, o,$ valider($clId, cId$)$\rangle$
9. $\quad\quad\quad\quad | (cheque(cId).valeur \geq oraganisation(o).limite \Longrightarrow$
10. $\quad\quad\quad\quad\quad (\langle \_, directeur, o,$ valider($clId, cId$)$\rangle$
11. $\quad\quad\quad\quad\quad \| \quad \langle \_, directeur, o,$ valider_directeur($clId, cId$)$\rangle)))))$
12. $\quad\quad)$

Grâce à cette expression, les actions concernant un dépôt de chèque donné ne peuvent être effectuées que dans une même agence.

3.2.2.3 Deuxième approche : créer une expression de processus pour chaque règle de CA

Pour l'exemple simplifié de la banque, les règles **Règle a** (ou **Règle a bis** le cas échéant) et **Règle b** sont modélisées à l'aide du diagramme de classes. En effet, ces règles de CA correspondent à des contraintes statiques. La règle **Règle c** correspond, quant à elle, à une contrainte de CA de type dynamique. Elle est modélisée à l'aide d'une expression de processus :

1. reglec ( ) $\triangleq$
2. $|||\, cId \in chequeId\ :\ |\, p \in personId\ :$
3. $\langle p, \_, \_,$ creer_cheque(cId)$\rangle$
4. . $\langle p, \_, \_,$ supprimer_cheque(cId)$\rangle$

Cette expression de processus met en séquence deux actions sécurisées. La première concerne l'action **creer_cheque** et la seconde l'action **supprimer_cheque**. L'entrelacement sur le paramètre fonctionnel cId permet de spécifier que la règle s'applique pour tous les chèques. Pour chacun des chèques traités par la banque, le choix quantifié sur le paramètre de sécurité p oblige les deux actions de la séquence à être réalisées par la même personne.

3.2.2.4 Troisième approche : créer une expression de processus par action

Cette approche crée une expression de processus pour chaque action concernée par les règles de la politique de CA. Dans l'exemple simplifié de la banque, seule l'action **supprimer_cheque** concernée par la règle **Règle c** nous intéresse. On obtient l'expression de processus suivante :

1. regle_supprimer_cheque ( ) $\triangleq$
2. $|||\, cId \in chequeId\ :\ |\, p \in personId\ :$
3. $cheque(cId).created\_by\_person = p \Longrightarrow$
4. $\langle p, \_, \_,$ supprimer_cheque(cId)$\rangle$

Cette expression de processus utilise une garde pour s'assurer que les actions **creer_cheque** et **supprimer_cheque** sont réalisées par la même personne. Pour pouvoir utiliser cette méthode, il faut rajouter des attributs à différentes classes du diagrammes de classes de sécurité. Ces attributs servent à conserver les valeurs des paramètres de sécurité nécessaires pour exprimer les contraintes sur les actions (*i.e.* pour cet exemple l'attribut ajouté permet de conserver la personne ayant créé le chèque pour s'assurer qu'elle seule puisse le détruire).

3.2.3 Discussion

3.2.3.1 Adaptabilité de la méthode EB^{3}SEC

La méthode EB^{3}SEC permet l'adaptation d'une politique de sécurité à de nouvelles contraintes. Par exemple, si la politique de sécurité de l'institution bancaire vient à changer et n'autorise la suppression d'un chèque par un guichetier que durant l'heure suivant sa création, la description de la politique de sécurité en EB^{3}SEC est modifiée de la manière suivante. Les quadruplets, présents dans les expressions de processus, sont alors transformés en quintuplets. Ces quintuplets contiennent les mêmes éléments que les quadruplets précédemment définis ainsi qu'un élément supplémentaire correspondant au temps et de type temps. Les expressions de processus sont alors modifiées de façon à utiliser des quintuplets en lieu et place des quadruplets. Pour l'exemple précédent, uniquement l'expression de processus correspondant à la classe cheque est modifiée de manière significative et devient :

1. **cheque** $(\, cId : chequeId \,) \triangleq$
2. $\quad (|\, t \in temps \;:\; (|\, o \in organisationId \;:\; (|\, p \in personneId \;:$
3. $\quad\quad (|\, r \in \{guichetier, directeur\} \;:$
4. $\quad\quad\quad \langle p, r, o, t, \mathsf{creer\_cheque}(\, cId \,)\rangle$
5. $\quad\quad\quad .\, ((|\, r' \in \{guichetier, directeur\} \;:$
6. $\quad\quad\quad\quad\quad \langle \_, r', o, \_, \mathsf{modifier\_valeur}(\, cId, \_ \,)\rangle)^*$
7. $\quad\quad\quad\quad \|\| \;\mathbf{depot}\,(\, \_, cId \,)^*)$
8. $\quad\quad\quad .\, |\, t' \in temps \;:\; t' < t + 3600 \Longrightarrow$
9. $\quad\quad\quad\quad\quad \langle p, r, o, t', \mathsf{supprimer\_cheque}(\, cId \,)\rangle))))$

La garde, utilisée pour l'action supprimer_cheque permet de lier les instants de création et de suppression d'un chèque. De plus, elle stipule que la suppression du chèque doit se réaliser dans l'heure suivant sa création.

3.2.3.2 Comparaison des modélisations précédentes

La première approche permet d'exprimer les règles de CA en s'appuyant sur les expressions de processus fonctionnelles. Si la modélisation de l'aspect fonctionnel du SI n'a pas été réalisée ou réalisée dans un autre langage, cette méthode oblige la modélisation de l'aspect fonctionnel en premier lieu. De plus cette méthode ne permet pas d'établir une séparation nette entre les aspects fonctionnels et de CA du SI. Cette séparation est toutefois permise par les deux autres approches. La troisième approche, quant à elle, peut demander l'ajout de nombreux attributs au diagramme de classes si les contraintes de CA dynamiques sont nombreuses dans la politique à modéliser. La seconde approche est plus orientée événements que états, au sens où elle représente explicitement les contraintes d'ordonnancement entre les événements. Cependant, cette méthode ne présente pas une approche systématique au problème de la modélisation d'une politique de CA en EB^{3}SEC. Cette lacune est en partie comblée grâce aux patrons de conception présentés dans la section 4.

Chapitre 4

Patrons de conception de règles en EB^{3}SEC

Le chapitre 3 illustre trois méthodes permettant d'obtenir les expressions de processus modélisant une politique de CA en EB^{3}SEC. Cependant, ces trois méthodes ne permettent pas de modéliser les contraintes de CA de manière systématique. Ce chapitre illustre un ensemble de patrons de conception, chacun de ces patrons correspond à un type de contraintes de CA. L'utilisation de ces patrons s'inscrit dans une stratégie de séparation entre la politique de CA et les règles fonctionnelles. Ces patrons servent aussi à présenter une approche systématique pouvant être utilisée lors d'une modélisation en EB^{3}SEC.

Afin de guider le concepteur pour l'expression des règles de CA en EB^{3}SEC, des patrons de conception ont été définis pour chaque catégorie de règles décrites dans le chapitre 1. La méthode EB^{3}SEC est utilisable pour n'importe quel domaine, le diagramme de classes utilisé n'est pas figé et peut comporter des entités propres à chaque domaine. Les patrons introduits dans ce chapitre doivent être formalisés de manière à être applicables quelque soit le diagramme de classes utilisé pour la spécification. La spécification de ces patrons requiert quelques définitions.

4.1 Quelques définitions

Dans un soucis de généralisation, les entités définies dans le diagramme de classes sont nommées : $E_1, \ldots, E_n$. Dans le cadre de l'exemple 2.1, le diagramme 3.2 est le diagramme utilisé. Les en-

tités $E_1, \ldots, E_n$ correspondent donc aux entités *personne*, *role*... présentes dans le diagramme de classes. On définit $E_{sec} = E_1 \times \ldots \times E_n$. On définit de même E_{fonc} le produit cartésien des entités utilisées pour la modélisation des aspects fonctionnels. Les actions utilisées dans le système d'informations sont regroupées dans l'ensemble Σ_{IS}, qui contient toutes les actions du système, ciblée ou non par la politique de CA. Les actions ciblées par la politique de CA sont regroupées au sein de l'ensemble Σ_{sec}. Dans le cadre de l'exemple 2.1, Σ_{sec} est instancié par :

$$\begin{aligned}\Sigma_{sec} = \{&\mathsf{deposer}(\, pId, cId\,), \mathsf{annuler}(\, pId, cId\,),\\ &\mathsf{valider}(\, pId, cId\,), \mathsf{valider\_directeur}(\, pId, cId\,),\\ &\mathsf{crediter}(\, pId, n\,), \mathsf{verifier}(\, clId, cId\,),\\ &\mathsf{enregistrer}(\, clId, cId\,)\}\end{aligned}$$

Pour la modélisation de la politique de CA en EB^{3}SEC, les expressions de processus utilisent les actions sécurisées. Elles sont définies par : $\sigma = \langle \overrightarrow{p_{sec}}, e \rangle$, où $\overrightarrow{p_{sec}} \in E_1 \times \cdots \times E_i$ avec $[\![1..i]\!]$ une sous-suite de $[\![1..n]\!]$ et $e \in \Sigma_{sec}$. L'ensemble des actions sécurisées sont regroupées dans l'ensemble Σ. L'environnement de sécurité noté $\overrightarrow{p_{sec}}$ est un vecteur dont les composantes sont définies sur un sous-ensemble des entités du diagramme de classes. Pour les besoins de la modélisations de l'exemple 2.1, l'environnement de sécurité comporte les informations relatives à la personne ayant réalisé l'action, le rôle qu'elle jouait au moment où l'action a été exécutée, l'organisation dans laquelle l'action a été réalisée et le moment de l'exécution. Dans la suite de ce chapitre, les attributs clés des classes sont remplacés par le nom des instances, dans un soucis de clarté et de compréhension. Pour la modélisation de la politique de CA donnée en exemple la signature des environnements de sécurité est la suivante :

1. $\langle p, r, o, t, evt \rangle$ avec $p \in pId$ et $r \in rId$
2. et $o \in oId$ et $t \in TIMESTAMP$ et $label(evt) \in Action().nom$

Dans les exemples illustrant les patrons de conception, le prédicat statique utilisé est le suivant :

1. $sp(\langle p, r, o, t, evt\rangle) \triangleq$
2. $\quad \langle p, r, o\rangle \in joue$
3. $\quad \wedge \langle r, o, idEvt(evt)\rangle \in permission \wedge permission(r, o, idEvt(evt)).contrainte()$
4. $\quad \wedge (interdiction(r, o, idEvt(evt)).contrainte() \Rightarrow \langle r, o, idEvt(evt)\rangle \notin interdiction)$

Le prédicat statique est utilisé pour réaliser le lien entre le diagramme de classes et l'expression de processus :

$$sp(\sigma_i) \ \wedge \ (label(\sigma_i) \in label(\textbf{main}_i)) \Rightarrow \textbf{main}_i \xrightarrow{\sigma_i} \textbf{main}_{i+1}$$

4.2 Permissions et interdictions

Cette section présente les patrons correspondant aux contraintes de CA de type permission ou interdiction. Bien que ces contraintes de CA ont déjà été modélisées à l'aide du diagramme de classes et du prédicat statique, nous donnons dans la suite leur équivalent en expressions de processus.

4.2.1 Permissions

Au sein d'une politique de CA, les permissions correspondent au fait d'autoriser une entité à réaliser une action. En EB^{3}SEC, les permissions autorisent l'exécution d'une action pour certaines valeurs des paramètres de sécurité. Les permissions sont elles-mêmes divisées en deux catégories, les permissions sans contraintes et avec contraintes.

4.2.1.1 Permissions sans contraintes

Une permission sans contraintes correspond à une permission qui est toujours valide au sein du SI. Une permission sans contrainte s'exprime en EB^{3}SEC à l'aide du patron suivant :

1. **permission** ( ) $\triangleq$
2. $\quad \langle \overrightarrow{p_{sec}}, \mathsf{action}(\_)\rangle$

Ce patron définit pour l'action $\mathsf{action} \in \Sigma_{sec}$ un ensemble de valeurs pour l'environnement de sécurité. Ces valeurs sont exprimées à l'aide de $\overrightarrow{p_{sec}}$. Dans ce patron, $\overrightarrow{p_{sec}}$ ne peut contenir que des valeurs constantes ou le caractère générique "_".

Il est aussi utile de pouvoir exprimer plusieurs valeurs pour l'environnement de sécurité associées à une action dans une même permission. Pour ce faire, le patron suivant introduit des variables quantifiées :

1. **permission**^{q} ( ) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}} \in E_{sec}$:
3. $\quad\quad \langle \overrightarrow{p_{sec}}, \mathsf{action}(\_) \rangle$

La seconde ligne de ce patron permet d'utiliser l'opérateur de choix quantifié afin de définir les différentes valeurs que peuvent prendre l'environnement de sécurité pour que l'action action soit permise. Dans le cas de ce patron, $\overrightarrow{p_{sec}}$ ne possède que des composantes scalaires égales à l'élément _ ou étant une des variables quantifiées.

Dans la politique de CA donnée en exemple, la première règle correspond à des exemples typiques de permission. Ces permissions n'autorisent que les personnes ayant le rôle *guichetier* et *conseiller* à pouvoir réaliser l'action deposer. Ces permissions sont modélisées séparément à l'aide du premier patron :

1. **permission**$_1$ ( ) $\triangleq$
2. $\quad \langle \_, guichetier, \_, \_, \mathsf{deposer}(\_) \rangle$

et

1. **permission**$_2$ ( ) $\triangleq$
2. $\quad \langle \_, conseiller, \_, \_, \mathsf{deposer}(\_) \rangle$

Le second patron peut aussi être utilisé pour modéliser les deux permissions précédentes en une seule expression :

1. **permission**$_1^q$ ( ) $\triangleq$
2. $\quad | r \in \{guichetier, conseiller\}$:
3. $\quad\quad \langle \_, r, \_, \_, \mathsf{deposer}(\_) \rangle$

4.2.1.2 Équivalence entre les permissions du diagramme de classes et les expressions obtenues à l'aide des patrons de permission

Le patron de conception correspondant aux permissions sans contraintes peut être utilisé à la place de l'élément $\langle r, o, idEvt(evt)\rangle \in permission$ du prédicat statique. Dans l'exemple **permission**$^q_1$, les environnements de sécurité ne précisent que le rôle utilisé. Cependant la seconde ligne du prédicat statique peut aussi être remplacée par des expressions de processus. En effet, les différentes valeurs possibles des paramètres de sécurité peuvent être précisées dans l'expression de processus. Pour reprendre l'exemple de la figure 3.3, on peut utiliser l'expression de processus suivante :

1. **exemple** ( ) $\triangleq$
2. $\quad\langle catherine, directeur, Montreal, \_, \mathsf{creer\_client}(\_)\rangle$
3. $\mid\quad\langle catherine, directeur, Montreal, \_, \mathsf{modifier\_nom}(\_, \_)\rangle$
4. $\mid\quad\ldots$

Un modèle de sécurité de style RBAC peut aussi être utilisé. Dans ce cas, les paramètres de sécurité sont au nombre de deux et correspondent à l'utilisateur et le rôle qu'il peut jouer.

4.2.1.3 Permissions avec contraintes

Les permissions avec contraintes sont des permissions qui ne sont pas toujours applicables dans le système. La contrainte associée à la permission permet de restreindre leur champs d'application à la valeur d'un prédicat booléen défini sur les variables du système (les attributs du diagramme de classes de la partie fonctionnelle, les attributs du diagramme de classes de la politique de CA) et les paramètres de l'action. Pour que la permission puisse être effective, la contrainte doit être évaluée à vraie. Les permissions avec contraintes sont modélisées en EB^{3}SEC à l'aide du patron suivant :

1. **permission**^{c} ( ) $\triangleq$
2. $\quad\mid \overrightarrow{p_{fonc}} \in E_{fonc}\ :$
3. $\qquad contr(\overrightarrow{p_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \langle \overrightarrow{p_{sec}}, \mathsf{action}(\overrightarrow{p_{fonc}})\rangle$

Deux nouvelles notations sont introduites dans ce patron. Le prédicat $contr(\overrightarrow{x_1}, \ldots, \overrightarrow{x_n})$ correspond à une contrainte, c'est-à-dire une fonction à valeur booléenne définie sur les éléments utilisés dans $\overrightarrow{p_{sec}}$ et $\overrightarrow{p_{fonc}}$. La garde utilisée à la ligne **3.** permet d'assujettir la validité de la permission à la valeur de la garde. Le vecteur $\overrightarrow{p_{fonc}}$ permet de représenter les variables fonctionnelles du SI nécessaires à l'expression de la permission avec contraintes. Dans ce patron seules les variables fonctionnelles (*i.e.* $\overrightarrow{p_{fonc}}$) sont quantifiées, $\overrightarrow{p_{sec}}$, représentant les paramètres de sécurité, ne contient que des constantes ou des caractères génériques. La syntaxe des contraintes est la même que pour l'entité Contrainte du diagramme de classe de sécurité. Les contraintes correspondent à une formule logique du premier ordre pouvant utiliser toutes les constantes et les variables présentes dans l'expression de processus.

Comme pour les permissions sans contraintes, les paramètres de sécurité peuvent aussi être quantifiés de manière à regrouper plusieurs règles au sein d'une même expression. Le patron suivant est alors utilisé dans ce cas :

1. **permission**^{c,q} ( ) $\triangleq$
2. $\quad | \overrightarrow{p_{fonc}}, \overrightarrow{p_{sec}} \in E_{fonc} \times E_{sec}$:
3. $\quad\quad contr(\overrightarrow{p_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \langle \overrightarrow{p_{sec}}, \mathsf{action}(\overrightarrow{p_{fonc}}) \rangle$

Dans ce patron, les variables fonctionnelles (*i.e.* $\overrightarrow{p_{fonc}}$) et les paramètres de sécurité (*i.e.* $\overrightarrow{p_{sec}}$) peuvent être des variables quantifiées.

La règle **règle 7** correspond à une permission sous contraintes. Elle autorise les personnes ayant le rôle de *conseiller* à réaliser l'action verifier pour les chèques d'un certain montant. Elle se modélise par :

1. **permission**$_1^{c,q}$ ( ) $\triangleq$
2. $\quad | c \in cId$: $| o \in oId$:
3. $\quad\quad Cheque(c).montant > Organisation(o).limite$
4. $\quad\quad \Longrightarrow \quad \langle \_, conseiller, o, \_, \mathsf{verifier}(c) \rangle$

4.2.2 Interdictions

Les interdictions permettent d'interdire l'exécution de certaines actions pour certaines valeurs des paramètres de sécurité. Elles se classent en deux catégories : les interdictions sans contraintes et les interdictions avec contraintes.

4.2.2.1 Interdictions sans contraintes

Comme les permissions sans contraintes, les interdictions sans contraintes ne dépendent pas de l'état du SI. Leur validité n'est pas soumise à la valeur d'une fonction booléenne définie sur les variables du système. Elles interdisent l'exécution d'une action pour certaines valeurs des paramètres de sécurité. Elles se modélisent en EB^{3}SEC à l'aide du patron :

1. **interdiction** ( ) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}} \in (E_{sec})$:
3. $\quad\quad \overrightarrow{p_{sec}} \notin \{\overrightarrow{v}\}$
4. $\quad\quad \Longrightarrow \langle \overrightarrow{p_{sec}}, \mathsf{action}(\_) \rangle$

L'ensemble $\{\overrightarrow{v}\}$ de la troisième ligne correspond aux valeurs interdites pour l'environnement de sécurité associé à l'action action. La garde de la quatrième ligne permet de s'assurer que l'environnement de sécurité de l'événement sécurisé reçu n'est pas dans l'ensemble $\{\overrightarrow{v}\}$. Toutefois, ces valeurs ne seront interdites que pour les événements sécurisés instanciés de l'action action.

Dans la politique de CA donnée en exemple, la règle **règle 1** peut être vue comme une interdiction pour les personnes ayant le rôle de *directeur* de réaliser l'action deposer. On peut ainsi la modéliser à l'aide d'une interdiction :

1. **interdiction**$_1$ ( ) $\triangleq$
2. $\quad | r \in rId$:
3. $\quad\quad r \neq directeur$
4. $\quad\quad \Longrightarrow \langle \_, r, \_, \_, \mathsf{deposer}(\_) \rangle$

Dans cet exemple, seul le rôle a été spécifié. La remarque concernant l'équivalence entre les permissions du diagramme de classes et celles obtenues à l'aide des patrons de permission est aussi

applicable aux interdictions. Pour se passer du diagramme de classes et n'utiliser que les expressions de processus, il faudrait donc spécifier les différentes valeurs possibles en lieu et place des _.

4.2.2.2 Interdictions avec contraintes

Au contraire des interdictions sans contraintes, les interdictions avec contraintes ne sont valables que lorsque la contrainte associée est vraie. Lorsque la contrainte associée est vraie, certaines valeurs d'environnements de sécurité de l'action concernée sont interdites. Les interdictions avec contraintes s'expriment en EB^{3}SEC à l'aide du patron :

1. **interdiction**^{c} ( ) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}}, \overrightarrow{p_{fonc}} \in E_{sec} \times E_{fonc}$:
3. $\quad\quad contr(\overrightarrow{p_{sec}}, \overrightarrow{p_{fonc}}) \rightarrow \overrightarrow{p_{sec}} \notin \{\overrightarrow{v}\}$
4. $\quad\quad \Longrightarrow \langle \overrightarrow{p_{sec}}, \mathsf{action}(\overrightarrow{p_{fonc}}) \rangle$

La garde présente à la quatrième ligne permet de s'assurer que l'interdiction ne s'applique que lorsque la contrainte $contr(\overrightarrow{p_{sec}}, \overrightarrow{p_{fonc}})$ est vraie. L'implication dans la garde permet de s'assurer que l'interdiction sans contraintes ne s'applique pas lorsque la contrainte est fausse.

La règle **règle 7** peut être vue comme une interdiction pour les personne ayant le rôle de *guichetier* ou de *directeur* d'effectuer l'action verifier pour les chèques au dessus d'un certain montant. Cela s'exprime par :

1. **interdiction**$^c_1$ ( ) $\triangleq$
2. $\quad | c \in cId$: $| o \in oId$: $| r \in rId$:
3. $\quad\quad (\quad Cheque(c).montant > Organisation(o).limite$
4. $\quad\quad \rightarrow \; r \notin \{directeur, clerk\})$
5. $\quad\quad\quad \Longrightarrow \langle \_, r, o, \_, \mathsf{verifier}(c) \rangle$

4.3 DSD et obligations

Cette section présente les patrons modélisant les contraintes de CA de type DSD et obligations.

4.3.1 SoD dynamique (DSD)

Cette section s'intéresse à la SoD dynamique (DSD). C'est à dire à la résolution de manière dynamique des contraintes de sécurité de type SoD. Une contrainte de CA de type SoD divise une tâche en sous-tâches afin de confier la réalisation de ces sous tâches à des acteurs différents. Ces tâches sont alors : i) réalisées dans un ordre précis (SoD séquentielles) ii) réalisées sans contraintes d'ordonnancement(SoD parallèles).

4.3.1.1 DSD séquentielles sans contraintes

En plus d'une distinction séquentielle ou parallèle s'appliquant à une SoD, les concepts de contraintes ou de sanctions sont utiles. Les SoD séquentielles sans contraintes s'expriment en EB^{3}SEC à l'aide du patron :

1. **SoD**^{s} ($\overrightarrow{p_{fonc}} : \overrightarrow{E}$) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}$:
3. $\quad\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
4. $\quad\quad .\ (\ \overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}}$
5. $\quad\quad\quad \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle)$

Les expressions de processus résultant de l'utilisation de ce patron ont comme arguments les paramètres fonctionnels du SI. En effet, comme la contrainte de SoD lient deux actions, il faut pouvoir retrouver l'objet concerné par les deux actions. La deuxième ligne de ce patron permet d'exprimer l'existence de deux environnements de sécurité, la quatrième ligne permet d'exprimer le fait que ces deux environnements de sécurité sont distincts. Les troisième et cinquième lignes utilisent les actions a et b sur lesquelles portent la contrainte de SoD. Comme ce patron sert à l'expression des SoD séquentielles, les deux actions sont reliées par une séquence, à la quatrième ligne.

La règle **règle 4** donne deux exemples de SoD : il y a une SoD entre les actions deposer et annuler et entre les actions valider et deposer, ces SoD portent sur l'utilisateur. La SoD entre deposer et annuler est considérée comme séquentielle et elle s'exprime à l'aide du patron par :

1. $\mathbf{SoD}_1^s\,(\,c : cId\,) \triangleq$
2. $\quad |\, p \in pId\ :\ |\, p' \in pId\ :$
3. $\quad\quad \langle p, \_, \_, \mathsf{deposer}(\,c\,)\rangle$
4. $\quad\quad .\ \ (\quad p' \neq p$
5. $\quad\quad\quad \Longrightarrow\!\!\!\!\!\!\diagup \langle p', \_, \_, \_, \mathsf{annuler}(\,c\,)\rangle)$

4.3.1.2 DSD séquentielle avec contraintes sans sanctions

Dans le cadre d'une SoD avec contraintes, les deux actions concernées par la SoD doivent être exécutées lorsque la contrainte est vraie. Une SoD séquentielle avec contraintes mais sans sanctions, est une SoD entre deux actions qui doivent être exécutées dans un certain ordre. De plus, une fois que la première action concernée par la SoD est exécutée, l'exécution de la seconde action est soumise à contrainte. Généralement les contraintes utilisées sont temporelles. Par exemple, on peut imaginer une contrainte qui oblige la seconde action de la SoD à être exécutée durant l'heure suivant l'exécution de la première action. Une SoD séquentielle avec contrainte sans sanction est de la forme :

1. $\mathbf{SoD}^{s,c}\,(\,\overrightarrow{p_{fonc}} : \overrightarrow{E}\,) \triangleq$
2. $\quad |\, \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}\ :$
3. $\quad\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\,\overrightarrow{p_{fonc}}\,)\rangle$
4. $\quad\quad .\ \ (\quad\ (\overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \wedge contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}})$
5. $\quad\quad\quad\ \Longrightarrow\!\!\!\!\!\!\diagup \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\,\overrightarrow{p_{fonc}}\,)\rangle)$
6. $\quad\quad\quad |\quad\ (\neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}})$
7. $\quad\quad\quad\ \Longrightarrow\!\!\!\!\!\!\diagup \lambda))$

Le résultat de ce patron a pour arguments les paramètres fonctionnels. Il utilise aussi deux environnements de sécurité, définis à la deuxième ligne. Ces deux environnements de sécurité sont différents grâce à la garde de la quatrième ligne, et chacun d'eux sert à exécuter une des actions concernées par la SoD. Comme ce patron traite le problème des SoD séquentielles avec contraintes sans sanction, l'opérateur de séquence permet de relier les actions concernées par la SoD. Toutefois la seconde action concernée par la SoD est exécutable si la contrainte sur les paramètres fonctionnels et les deux environnements de sécurité est vraie. De manière à ne pas créer de *deadlock* , les

sixième et septième lignes permettent, une fois que la contrainte n'est plus vraie de réaliser l'action dénotée par λ, correspondant à l'action neutre.

Pour illustrer ce patron, la règle **règle 4 bis** est modélisée. Toutefois seule l'annulation est prise en compte dans l'exemple suivant.

1. $\mathbf{SoD}_1^{s,c}(\, c : cId, cl : clId\,) \triangleq$
2. $\quad |\, p \in pId \;:\; |\, p' \in pId \;:\; |\, t \in Timestamp \;:\; |\, t' \in Timestamp \;:$
3. $\quad\quad \langle p, \_, \_, t, \mathsf{deposer}(\, cl, c\,)\rangle$
4. $\quad\quad . \quad (\quad\quad (p' \neq p \wedge (t' - t) < 3600$
5. $\quad\quad\quad\quad \Longrightarrow \langle p', \_, \_, t', \mathsf{annuler}(\, cl, c\,)\rangle)$
6. $\quad\quad\quad | \quad\quad (currentTime() - t) \geq 3600$
7. $\quad\quad\quad\quad \Longrightarrow \lambda))$

La garde de la sixième utilise la primitive $currentTime()$ de manière à pouvoir déterminer si la condition temporelle est encore valide. Cette primitive permet de connaître le temps actuel et d'exécuter l'action λ une fois la contrainte dépassée.

4.3.1.3 DSD séquentielle avec contraintes avec sanctions

Ce type de contrainte de CA exprime une SoD entre actions devant être réalisée de manière séquentielle ; de plus, l'exécution des actions est soumise à contrainte et si la contrainte n'est pas respectée, alors une sanction doit être exécutée.

1. $\mathbf{SoD}^{s,c,a}(\, \overrightarrow{p_{fonc}} : \overrightarrow{E}\,) \triangleq$
2. $\quad |\, \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E} \;:$
3. $\quad\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\, \overrightarrow{p_{fonc}}\,)\rangle$
4. $\quad\quad . \quad (\quad\quad (\overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \wedge contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}})$
5. $\quad\quad\quad\quad \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\, \overrightarrow{p_{fonc}}\,)\rangle)$
6. $\quad\quad\quad | \quad\quad (\neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}})$
7. $\quad\quad\quad\quad \Longrightarrow \mathbf{sanction}\,(\, \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}\,)))$

Le résultat de ce patron, comme les autres patrons de SoD, a comme argument les paramètres fonctionnels du SI. Les actions a et b sont reliées par l'intermédiaire de l'opérateur de séquence

de la quatrième ligne. La garde de la quatrième ligne exprime que les environnements de sécurité utilisés pour les deux actions doivent être différents et aussi que la seconde action de la SoD doit être réalisée tant que la contrainte est vraie. Les sixième et septième lignes expriment le fait qu'une fois la contrainte rompue la sanction doit être exécutée. La *sanction* est représentée ici comme une expression de processus.

Pour illustrer ce patron, la règle **règle 4 ter** est utilisée. Seulement la partie concernant la validation est modélisée dans l'exemple suivant.

1. $\textbf{SoD}_1^{s,c,a}\,(\,c : cId, cl : clId\,) \triangleq$
2. $\quad |\, p \in pId \,:\, |\, p' \in pId \,:\, |\, t \in Timestamp \,:\, |\, t' \in Timestamp \,:$
3. $\quad\quad \langle p, _, _, t, \mathsf{deposer}(\,c\,)\rangle$
4. $\quad\quad\; .\;\; (\quad (p' \neq p \wedge (t' - t) < 3600$
5. $\quad\quad\quad\quad \Longrightarrow \langle p', _, _, t', \mathsf{valider}(\,cl, c\,)\rangle)$
6. $\quad\quad\quad\; |\quad (t' - t) \geq 3600$
7. $\quad\quad\quad\quad \Longrightarrow \langle _, _, _, t', \mathsf{mail}(\,p, c\,)\rangle))$

Dans cet exemple, la sanction correspond à un courriel envoyé à la personne ayant fait le deposer pour lui signaler que le chèque n'a pas été validé.

4.3.1.4 DSD parallèle sans contraintes sans sanctions

Ce patron concerne les actions liées par une SoD. Mais cette SoD n'est pas soumise à contrainte. Ce patron a la forme :

1. $\textbf{SoD}^{P}\,(\,\overrightarrow{p_{fonc}} : \overrightarrow{E}\,) \triangleq$
2. $\quad |\, \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E} \;:$
4. $\quad\quad \overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}}$
3. $\quad\quad\quad \Longrightarrow (\langle \overrightarrow{p_{sec}}, \mathsf{a}(\,\overrightarrow{p_{fonc}}\,)\rangle$
5. $\quad\quad\quad\quad ||| \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\,\overrightarrow{p_{fonc}}\,)\rangle)$

Comme l'ordre d'exécution des actions a et b n'est pas obligatoire les deux actions sont en entrelacement. Deux cas peuvent être envisagés : soit deux actions différentes sont concernées par ce patron ou ce patron concerne deux fois la même action. Si les actions ont un nom différent elles

sont en parallèle. Si le patron concerne deux fois la même action, elle est réalisée deux fois, un *deadlock* étant ainsi évité. Lorsque la première action est exécutée, les deux environnements de sécurité ne sont pas encore instanciés, de ce fait la garde ne peut être calculée. Pour utiliser ce patron avec les outils actuels il faut le remplacer par :

1. $\mathbf{SoD}^{P}\,(\overrightarrow{p_{fonc}} : \overrightarrow{E}\,) \triangleq$
2. $\quad |\,\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}\ :$
3. $\qquad (\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\,\overrightarrow{p_{fonc}}\,)\rangle$
4. $\qquad\ \ .\quad \overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\,\overrightarrow{p_{fonc}}\,)\rangle)$
5. $\quad\ \ |\quad (\quad \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\,\overrightarrow{p_{fonc}}\,)\rangle$
6. $\qquad\ \ .\quad \overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \Longrightarrow \langle \overrightarrow{p_{sec}}, \mathsf{a}(\,\overrightarrow{p_{fonc}}\,)\rangle)$

De manière à illustrer ce patron nous utilisons la règle **règle 5 bis** :

1. $\mathbf{SoD}_1^{P}\,(\,c : cId, cl : clId\,) \triangleq$
2. $\quad |\,p \in pId\ :\ |\,p' \in pId\ :$
4. $\qquad p' \neq p$
3. $\qquad \Longrightarrow (\langle p, _, _, \mathsf{valider}(\,cl, c\,)\rangle$
5. $\qquad\quad |||\,\langle p', _, _, \mathsf{valider}(\,cl, c\,)\rangle)$

4.3.1.5 DSD parallèle avec contraintes sans sanctions

Ce patron, comme le précédent, concerne les actions soumises à une contrainte de CA de type SoD. Cependant les deux actions concernées n'ont pas d'ordre d'exécution établi et peuvent être interverties. Le patron est le suivant :

1. $\mathbf{SoD}^{p,c}\,(\,\overrightarrow{p_{fonc}} : \overrightarrow{E}\,) \triangleq$
2. $\quad |\,\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}\ :$
3. $\qquad (\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\,\overrightarrow{p_{fonc}}\,)\rangle$
4. $\qquad\ \ .\qquad (contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \wedge \overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\,\overrightarrow{p_{fonc}}\,)\rangle$
5. $\qquad\qquad |\quad \neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \lambda))$
6. $\quad\ \ |\quad (\quad \langle \overrightarrow{p_{sec}}, \mathsf{b}(\,\overrightarrow{p_{fonc}}\,)\rangle$
7. $\qquad\ \ .\qquad (contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \wedge \overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{a}(\,\overrightarrow{p_{fonc}}\,)\rangle$
8. $\qquad\qquad |\quad \neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \lambda))$

De par la présence des gardes, l'entrelacement ne peut être utilisé. Ce patron explicite le choix des deux possibilités d'ordre d'exécution des actions a et b. Les gardes des lignes **4** et **7** permettent de s'assurer que la DSD est exécutée seulement lorsque la contrainte est vraie. De plus, les lignes **5** et **8** permettent de s'affranchir d'un *deadlock* lorsque la seconde action n'est pas exécutée.

Ce patron sert à modéliser la règle **règle 5** de la politique de contrôle d'accès donnée en exemple :

1. $\mathbf{SoD}_1^{p,c}\,(\,c : cId, cl : clId\,) \triangleq$
2. $\quad |\, p \in pId \;:\; |\, p' \in pId \;:\; |\, o \in oId \;:$
3. $\quad\quad (\quad \langle p, \_, \_, \_, \mathsf{valider}(\,c\,)\rangle$
4. $\quad\quad \cdot \quad\quad (p' \neq p \wedge Cheque(c).montant > Organisation(o).limite$
5. $\quad\quad\quad\quad \Longrightarrow \langle p', \_, o, \_, \mathsf{valider}(\,cl, c\,)\rangle$
6. $\quad\quad\quad |\quad \neg(p' \neq p \wedge Cheque(c).montant > Organisation(o).limite) \Longrightarrow \lambda))$
7. $\quad\quad |\quad (\quad \langle p', \_, o, \_, \mathsf{valider}(\,c\,)\rangle$
8. $\quad\quad\quad \cdot \quad\quad (p' \neq p \wedge Cheque(c).montant > Organisation(o).limite$
9. $\quad\quad\quad\quad \Longrightarrow \langle p', \_, o, \_, \mathsf{valider}(\,cl, c\,)\rangle$
10. $\quad\quad\quad |\quad \neg(p' \neq p \wedge Cheque(c).montant > Organisation(o).limite) \Longrightarrow \lambda))$

4.3.1.6 DSD parallèle avec contraintes avec sanctions

Ce patron concerne uniquement les contraintes de CA de type SoD, intervenant entre deux actions dont l'ordre d'exécution n'est pas fixé. Pour ce patron, l'exécution des actions est soumise à contrainte et si la contrainte n'est pas respectée une sanction doit être exécutée. Le patron correspondant aux SoD soumises à contrainte et sanction est de la forme :

1. $\textbf{SoD}^{p,c,a}\,(\,\overrightarrow{p_{fonc}} : \overrightarrow{E}\,) \triangleq$
2. $\quad |\,\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}\ :$
3. $\quad\quad (\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\,\overrightarrow{p_{fonc}}\,)\rangle$
4. $\quad\quad .\quad\quad (contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \wedge \overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\,\overrightarrow{p_{fonc}}\,)\rangle$
5. $\quad\quad\quad |\quad \neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \textbf{sanction}\,(\,\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}\,)))$
6. $\quad\quad |\quad (\quad \langle \overrightarrow{p_{sec}}, \mathsf{b}(\,\overrightarrow{p_{fonc}}\,)\rangle$
7. $\quad\quad .\quad\quad (contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \wedge \overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{a}(\,\overrightarrow{p_{fonc}}\,)\rangle$
8. $\quad\quad\quad |\quad \neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \textbf{sanction}\,(\,\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}\,)))$

Ce patron ne peut lui aussi utiliser l'entrelacement, il explicite les deux possibilités d'ordre d'exécution des actions concernées par la contrainte de DSD. Les lignes **5** et **8** permettent d'exécuter la sanction lorsque la contrainte liée à la DSD n'est plus vraie.

La règle **regle 5 quater** peut être modélisée à l'aide de ce patron :

1. $\textbf{SoD}^{p,c,a}\,(\,c : cId, cl : clId\,) \triangleq$
2. $\quad |\,p \in pId\ :\ |\,p' \in pId\ :\ |\,t \in Timestamp\ :\ |\,t' \in Timestamp\ :$
3. $\quad\quad (\quad \langle p, \_, \_, t, \mathsf{valider}(\,cl, c\,)\rangle$
4. $\quad\quad .\quad\quad (p' \neq p \wedge (t' - t) < 3600 \Longrightarrow \langle p', \_, \_, t', \mathsf{valider\_directeur}(\,cl, c\,)\rangle$
5. $\quad\quad\quad |\quad \neg(p' \neq p \wedge (t' - t) < 3600) \Longrightarrow \langle \_, \_, \_, t', \mathsf{mail}(\,p, c\,)\rangle))$
6. $\quad\quad |\quad (\quad \langle p', \_, \_, t', \mathsf{valider\_directeur}(\,cl, c\,)\rangle$
7. $\quad\quad .\quad\quad (p' \neq p \wedge (t' - t) < 3600 \Longrightarrow \langle p', \_, \_, t', \mathsf{valider}(\,cl, c\,)\rangle$
8. $\quad\quad\quad |\quad \neg(p' \neq p \wedge (t' - t) < 3600) \Longrightarrow \langle \_, \_, \_, t', \mathsf{mail}(\,p, c\,)\rangle))$

La sanction correspond ici à un courriel envoyé à la personne ayant fait le deposer pour lui signaler que son chèque n'a pas été validé.

4.3.2 Obligations

Les contraintes de CA de type obligation, sont des contraintes qui lient au moins deux actions, ces actions devant être exécutées avec des éléments identiques de leur environnement de sécurité. Comme pour les contraintes de type SoD, les obligations sont séparées en deux catégories : les

obligations séquentielles et les obligations parallèles. Les patrons représentant les obligations séquentielles correspondent aux pré-obligations et aux post-obligations, tandis que les patrons utilisés pour les obligations parallèles correspondent aux in-obligations.

4.3.2.1 Obligations séquentielles sans sanctions sans contraintes

Les obligations séquentielles sans contraintes et sans sanctions concernent deux actions devant être réalisées avec certains éléments de leur environnement de sécurité identiques. Les deux actions doivent de plus être exécutées dans un ordre précis. L'exécution de l'obligation n'est pas soumise ni à contrainte, ni à sanction. Elles sont modélisées à l'aide du patron :

1. **obligation**^{s} ($\overrightarrow{p_{fonc}} : \overrightarrow{E}$) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}$:
3. $\quad\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
4. $\quad\quad . \quad \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle)$

Il faut rajouter la contrainte : $\exists i \in \mathbb{N} \,.\, \overrightarrow{p_{sec|i}} = \overrightarrow{p'_{sec|i}}$. Les vecteurs $\overrightarrow{p_{sec}}$ et $\overrightarrow{p'_{sec}}$ représentent les environnements de sécurité nécessaires pour réaliser a et b. Ils peuvent contenir des variables, des constantes et des éléments de type _. L'opérateur de séquence est utilisé pour relier a et b et obliger ces dernières à être exécutées dans cet ordre. Dans le cas d'une pré-obligation (respectivement post-obligation), a correspond à l'obligation (respectivement l'action) et b à l'action (respectivement à l'obligation).

La règle 6 est modélisée à l'aide de ce patron et pour obtenir :

1. **obligation**$^s_1$ ($c : cId, cl : clId$) $\triangleq$
2. $\quad | p \in pId \; : \; | r \in rId \; :$
3. $\quad\quad \langle p, r, \_, \mathsf{deposer}(cl, c) \rangle$
4. $\quad\quad . \quad \langle p, r, \_, \mathsf{modifier\_montant}(cl, \_) \rangle)$

4.3.2.2 Obligations séquentielles sans sanctions avec contraintes

Ce type de contrainte oblige deux actions à être réalisées avec certaines valeurs de paramètres de sécurité égales. L'exécution de la seconde action est soumise à contrainte. Le patron correspondant à ce type de contrainte est donné par :

1. **obligations**^{s,c} ($\overrightarrow{p_{fonc}} : \overrightarrow{E}$) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}$:
3. $\quad\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
4. $\quad\quad .$ ($\quad (contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}})$
5. $\quad\quad\quad \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle)$
6. $\quad\quad | \quad (\neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}})$
7. $\quad\quad\quad \Longrightarrow \lambda))$

La quatrième ligne utilise l'opérateur de séquence pour exprimer le fait que l'obligation doit être réalisée après l'exécution de l'action. Les quatrième et cinquième lignes expriment le fait que l'exécution de l'obligation ne peut être réalisée que lorsque la contrainte est vraie. Les sixième et septième lignes sont utilisées pour éviter d'obtenir un *deadlock* une fois que la contrainte n'est plus vraie. En plus de ce patron il faut ajouter une contrainte sur les environnements de sécurité $\overrightarrow{p_{sec}}$ et $\overrightarrow{p'_{sec}}$. En effet, il doivent comporter tout deux au moins un élément égal pour exprimer une obligation : $\exists i \in \mathbb{N} \,.\, \overrightarrow{p_{sec}}|_i = \overrightarrow{p'_{sec}}|_i$. Les patrons d'obligation et de SoD sont très proches. Cette propriété souligne la différence entre les deux.

La règle **regle 6 bis** est modélisée à l'aide de ce patron :

1. **obligation**$_1^{s,c}$ ($c : cId, cl : clId$) $\triangleq$
2. $\quad | p \in pId : | r \in rId : | t \in Timestamp : | t' \in Timestamp :$
3. $\quad\quad \langle p, r, \_, t, \mathsf{deposer}(cl, c) \rangle$
4. $\quad\quad .$ ($\quad ((t' - t) < 3600$
5. $\quad\quad\quad \Longrightarrow \langle p, r, \_, t', \mathsf{modifier\_montant}(cl, \_) \rangle)$
6. $\quad\quad | \quad (currentTime() - t) \geq 3600$
7. $\quad\quad\quad \Longrightarrow \lambda))$

4.3.2.3 Obligations séquentielles avec sanctions avec contraintes

Les contraintes de CA de ce type concernent deux actions devant être exécutées dans un ordre précis. Ces actions sont aussi liées par une obligation. Si cette obligation n'est pas réalisée tant que la contrainte associée est vraie, alors une sanction est exécutée. Ce type d'obligation est représentée par le patron :

1. **obligations**^{s,c,a} ($\overrightarrow{p_{fonc}} : \overrightarrow{E}$) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}$:
3. $\quad\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
4. $\quad\quad .$ ($\quad (contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}})$
5. $\quad\quad\quad \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle)$
6. $\quad\quad | \quad (\neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}})$
7. $\quad\quad\quad \Longrightarrow$ **sanction** ($\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}$)))

Comme pour le patron précédent il faut ajouter, pour les environnements de sécurité $\overrightarrow{p_{sec}}$ et $\overrightarrow{p'_{sec}}$, la contrainte suivante : $\exists i \in \mathbb{N} \,.\, \overrightarrow{p_{sec}}|_i = \overrightarrow{p'_{sec}}|_i$.

La sixième règle peut se voir ajouter une contrainte temporelle et une sanction, elle est alors modélisée par :

1. **obligation**$_1^{s,c,a}$ ($c : cId, cl : clId$) $\triangleq$
2. $| p \in pId : | r \in rId : | t \in Timestamp : | t' \in Timestamp :$
3. $\quad \langle p, r, \_, t, \mathsf{deposer}(cl, c) \rangle$
4. $\quad .$ ($\quad ((t' - t) < 3600$
5. $\quad\quad \Longrightarrow \langle p, r, \_, t', \mathsf{modifier\_montant}(cl, \_) \rangle)$
6. $\quad | \quad (t' - t) \geq 3600$
7. $\quad\quad \Longrightarrow \langle \_, \_, \_, t', \mathsf{mail}(p, c) \rangle)))$

Dans cet exemple, la section correspond à un courriel envoyé à l'utilisateur ayant réalisé l'action deposer pour lui signaler qu'il n'a pas réalisé l'action modifier_montant une fois la contrainte de temps dépassée.

4.3.2.4 Obligations parallèles sans sanctions sans contraintes

Ces obligations concernent deux actions dont l'ordre d'exécution n'est pas prédéfini, cependant l'exécution de ces actions est liée par une obligation. Les contraintes de ce type sont modélisées à l'aide du patron :

1. **obligation**^{p} ($\overrightarrow{p_{fonc}} : \overrightarrow{E}$) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}$:
3. $\quad\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
4. $\quad\quad ||| \quad \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle)$

Pour modéliser entièrement l'obligation, il faut rajouter la contrainte : $\exists i \in \mathbb{N} \,.\, \overrightarrow{p_{sec|i}} = \overrightarrow{p'_{sec|i}}$. L'entrelacement de la quatrième ligne est utilisée pour relier l'action et l'obligation car leur ordre d'exécution n'est pas prédéfini. Le fait de mettre un entrelacement permet de pouvoir utiliser ce patron avec deux fois la même action.

La règle **regle 8 bis** peut être modélisée à l'aide de ce patron :

1. **obligation**$^{p}_{1}$ ($c : cId, cl : clId$) $\triangleq$
2. $\quad | p \in pId$:
3. $\quad\quad \langle p, \_, \_, \mathsf{deposer}(cl, c) \rangle$
4. $\quad\quad ||| \quad \langle p, \_, \_, \mathsf{enregistrer}(c) \rangle)$

4.3.2.5 Obligations parallèles sans sanctions avec contraintes

Ce patron correspond à une obligation entre deux actions dont l'ordre n'est pas prédéfini. Toutefois, leur exécution est soumise à contrainte :

1. **obligation**^{p,c} ($\overrightarrow{p_{fonc}} : \overrightarrow{E}$) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}$:
3. $\quad\quad (\quad \langle \overrightarrow{p_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
4. $\quad\quad\quad . \quad (contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle$
5. $\quad\quad\quad\quad | \quad \neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \lambda))$
6. $\quad\quad | \quad (\quad \langle \overrightarrow{p_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle$
7. $\quad\quad\quad . \quad (contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
8. $\quad\quad\quad\quad | \quad \neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \lambda))$

Les environnements de sécurité utilisés pour a et b sont définis à la deuxième ligne. Bien que l'ordre n'importe peu entre les deux actions, l'utilisation de l'entrelacement est impossible de part la présence de la contrainte. Le patron utilise l'opérateur de choix entre les deux possibilité d'ordre d'exécution des actions a et b. L'obligation ne sera exécutée que lorsque la contrainte définie par la garde des quatrième ou septième lignes est vraie. Les cinquième et huitième lignes permettent d'éviter d'obtenir un *deadlock* lorsque la contrainte n'est plus vraie.

La huitième règle est modélisée à l'aide de ce patron :

1. **obligation**$_1^{p,c}$ ($c : cId, cl : clId$) $\triangleq$
2. $\quad | p \in pId : | t \in Timestamp : | t' \in Timestamp : | o \in oId :$
3. $\quad\quad (\quad \langle p, \_, o, t, \mathsf{deposer}(cl, c) \rangle$
4. $\quad\quad . \quad (Cheque(c).montant > Organisation(o).limite$
5. $\quad\quad\quad\quad e \Longrightarrow \langle p, \_, \_, t', \mathsf{enregistrer}(c) \rangle$
6. $\quad\quad\quad | \quad \neg(Cheque(c).montant > Organisation(o).limite) \Longrightarrow \lambda))$
7. $\quad | \quad (\quad \langle p, \_, o, t', \mathsf{enregistrer}(c) \rangle$
8. $\quad\quad . \quad (Cheque(c).montant > Organisation(o).limite$
9. $\quad\quad\quad\quad \Longrightarrow \langle p, \_, \_, t, \mathsf{deposer}(cl, c) \rangle$
10. $\quad\quad\quad | \quad \neg(Cheque(c).montant > Organisation(o).limite) \Longrightarrow \lambda))$

4.3.2.6 Obligations parallèles avec sanctions avec contraintes

Une obligation avec contraintes et sanctions concerne une obligation entre deux actions dont l'ordre d'exécution n'est pas définie à l'avance.

1. **obligation**^{p,c,a} ($\overrightarrow{p_{fonc}} : \overrightarrow{E}$) $\triangleq$
2. $\quad | \overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}} \in \overrightarrow{E} \times \overrightarrow{E}$:
3. $\quad\quad$ ($\langle \overrightarrow{p_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
4. $\quad\quad$. $(contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle$
5. $\quad\quad\quad$ | $\neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow$ **sanction** ($\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}$)))
6. $\quad\quad$ | ($\langle \overrightarrow{p_{sec}}, \mathsf{b}(\overrightarrow{p_{fonc}}) \rangle$
7. $\quad\quad$. $(contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow \langle \overrightarrow{p'_{sec}}, \mathsf{a}(\overrightarrow{p_{fonc}}) \rangle$
8. $\quad\quad\quad$ | $\neg contr(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}) \Longrightarrow$ **sanction** ($\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}, \overrightarrow{p_{fonc}}$)))

À la différence du patron précédent, les cinquième et huitième lignes permettent une fois que la contrainte n'est plus respectée de pouvoir exécuter la sanction.

La modélisation de la règle **regle 8 ter** est donnée par :

1. **obligation**$_1^{p,c,a}$ ($c : cId$) $\triangleq$
2. $\quad | p \in pId : | t \in Timestamp : | t' \in Timestamp : | o \in oId :$
3. $\quad\quad$ ($\langle p, \_, o, t, \mathsf{deposer}(c) \rangle$
4. $\quad\quad$. $(Cheque(c).montant > Organisation(o).limite$
5. $\quad\quad\quad \Longrightarrow \langle p, \_, \_, t', \mathsf{enregistrer}(c) \rangle$
6. $\quad\quad\quad$ | $\neg(Cheque(c).montant > Organisation(o).limite)$
7. $\quad\quad\quad \Longrightarrow \langle \_, \_, \_, t', \mathsf{mail}(p, c) \rangle))$
8. $\quad\quad$ | ($\langle p, \_, o, t', \mathsf{enregistrer}(c) \rangle$
9. $\quad\quad$. $(Cheque(c).montant > Organisation(o).limite$
10. $\quad\quad\quad \Longrightarrow \langle p, \_, \_, t, \mathsf{deposer}(c) \rangle$
11. $\quad\quad\quad$ | $\neg(Cheque(c).montant > Organisation(o).limite)$
12. $\quad\quad\quad \Longrightarrow \langle \_, \_, \_, t', \mathsf{mail}(p, c) \rangle))$

4.3.2.7 Remarque

Dans les patrons précédents, la contrainte $\exists i \in \mathbb{N}.\overrightarrow{p_{sec|i}} = \overrightarrow{p'_{sec|i}}$ est utilisée. Dans certains cas, cette contrainte est trop forte : on pourrait imaginer que l'obligation entre deux actions a et b ne porte pas sur la même personne. En effet, la première action pourrait être réalisée par un employé et la seconde doit être réalisée par un de ses collègues. Par exemple, la règle **regle 8 bis** peut être modifier afin de stipuler que les actions deposer et enregistrer doivent être réalisées par deux collègues. Pour la modéliser, on ajouterait au diagramme de classes de sécurité une relation réflexive nommée collegue sur la classe Personne. On modifierait le patron **obligation**^{p} de manière à obtenir :

1. **obligation**$_2^{p'}$ $(\, c : cId, cl : clId \,) \triangleq$
2. $\quad | \, p \in pId \; : \; | \, p' \in pId \; :$
3. $\qquad\qquad (\quad \langle p, \_, \_, \mathsf{deposer}(\, cl, c \,)\rangle$
4. $\qquad\qquad .\quad (p' \in Personne(p).collegue \Longrightarrow \langle p', \_, \_, \mathsf{enregistrer}(\, c \,)\rangle)$
5. $\qquad\quad | \quad (\quad \langle p, \_, \_, \mathsf{enregistrer}(\, c \,)\rangle$
6. $\qquad\qquad .\qquad (p' \in Personne(p).collegue \Longrightarrow \langle p', \_, \_, \mathsf{deposer}(\, cl, c \,)\rangle)$

L'entrelacement a dû aussi être enlevé, en effet l'apparition de la garde permettant d'exprimer l'obligation oblige d'expliciter les deux ordres d'exécution possibles.

4.4 Mise en commun

Cette section précise comment les résultats des patrons précédents peuvent être regroupés dans une même expressions de processus. La section 4.4.1 concerne les résultats des patrons des permissions et interdictions tandis que la section 4.4.2 concerne les résultats des obligations et des SoD. La section 4.4.3 permet de mettre en commun les résultats obtenues à partir de tous les patrons.

4.4.1 Permissions et interdictions

Maintenant que les patrons de permissions et d'interdictions ont été définis, ils peuvent être mis en commun. Dans le cadre de ce chapitre, nous considérons que les permissions et interdictions avec

et sans contraintes ont été modélisées à l'aide des patrons de ce chapitre et ne sont pas représentées à l'aide du diagramme de classes.

Les résultats des patrons sont regroupés au sein d'une même expression de processus. La construction de cette expression de processus est expliquée à la section 4.4.1.1. De même, la section 4.4.1.2 illustre comment regrouper les différentes interdictions en une expression de processus et la section 4.4.1.3 décrit comment les interdictions et les permissions sont réunies dans une même expression de processus.

4.4.1.1 Mise en commun des permissions

Pour qu'un événement soit accepté, il faut qu'au moins une permission autorise son exécution. Les permissions définies à l'aide des précédents patrons vont donc être regroupées entre elle à l'aide d'un choix :

$$
\begin{array}{lll}
\textbf{1.} & \textbf{PermissionGlobale}(\,) \triangleq & \\
\textbf{2.} & & |\, i \in K_1 \,:\, \textbf{permission}_i(\,) \\
\textbf{3.} & | & |\, i \in K_2 \,:\, \textbf{permission}_i^q(\,) \\
\textbf{4.} & | & |\, i \in K_3 \,:\, \textbf{permission}_i^c(\,) \\
\textbf{5.} & | & |\, i \in K_4 \,:\, \textbf{permission}_i^{c,q}(\,)
\end{array}
$$

Les constantes $K_1, \ldots$ et K_4 représentent le nombre de résultats correspondant à chaque type de patron. Toutefois, n'apparaissent dans **PermissionGlobale** seulement les patrons utilisés. Si un patron n'a pas été utilisé durant la phase de modélisation, il n'apparaît pas dans cette expression de processus.

4.4.1.2 Mise en commun des interdictions

Pour qu'un événement soit accepté, il faut qu'aucune des interdictions ne l'interdise. Cependant les patrons utilisés pour définir les interdictions, autorisent les actions en interdisant les valeurs d'environnement de sécurité interdites. Les expressions de processus obtenues à l'aide des patrons d'interdiction sont mises en parallèle :

1. **InterdictionGlobale** ( ) $\triangleq$
2. $\|\, i \in K_1 \mathbb{N}$: **interdiction**$_i$ ( )
3. $\|$ $\|\, i \in K_2$: **interdiction**$_i^c$ ( )

Les constantes K_1 et K_2 représentent le nombre de résultats correspondant à chaque type de patron. En effet, si un événement a été autorisé par cette expression de processus, c'est que toutes les interdictions concernant l'action dont provient l'événement, n'interdisent pas les valeurs des paramètres de sécurité de l'événement.

4.4.1.3 Mise en commun des permissions et des interdictions

Les permissions et les interdictions sont mises en commun de manière à ce qu'elles soient synchronisées sur leurs actions communes :

1. **PermInter** ( ) $\triangleq$
2. (**PermissionGlobale** ( )
3. $\|$ **InterdictionGlobale** ( ))$^+$

Cette expression de processus doit pour chaque événement reçu vérifier qu'il est permis par au moins une des permissions et qu'il est aussi interdit par aucune des interdictions.

4.4.1.4 Mise en application

De manière à illustrer ces patrons, nous exprimons à des fins d'illustration une sous-partie de l'exemple modélisé à l'aide des instances du diagramme de classes présentées dans les figures 3.3, 3.4, 3.5 et 3.6.

1. PermInter ( )$\triangleq$
2. (($\langle catherine, directeur, Montreal, \_, \mathsf{creer\_client}(\_)\rangle$
3. | $\cdots$
4. | $\langle catherine, directeur, Montreal, \_, \mathsf{supprimer\_cheque}(\_)\rangle$
5. | $\langle boris, guichetier, Montreal, \_, \mathsf{creer\_client}(\_)\rangle$
6. | $\cdots$
7. | $\langle boris, guichetier, Montreal, \_, \mathsf{deposer}(\_,\_)\rangle$
8. | $\langle boris, guichetier, Montreal, \_, \mathsf{creer\_cheque}(\_)\rangle$
9. | $\cdots$
10. | $\langle boris, guichetier, Montreal, \_, \mathsf{supprimer\_cheque}(\_)\rangle$
11. | $\cdots$
12. | $|\, c \in cId\ :\ |\, o \in oId\ :$
13. $Cheque(c).valeur < Organisation(o).limite$
14. $\Longrightarrow \langle boris, guichetier, Montreal, \_, \mathsf{valider}(\_, c)\rangle$
15. | $|\, c \in cId\ :\ |\, o \in oId\ :$
16. $Cheque(c).valeur < Organisation(o).limite$
17. $\Longrightarrow \langle elise, guichetier, Toronto, \_, \mathsf{valider}(\_, c)\rangle)$
18. || ($|\, p \in pId\ :\ |\, r \in rId\ :\ |\, o \in oId\ :$
19. $p \neq boris \wedge r \neq guichetier \wedge o \neq Montreal$
20. $\Longrightarrow \langle p, r, o, \_, \mathsf{valider}(\_,\_)\rangle$
21. || $|\, p \in pId\ :\ |\, r \in rId\ :\ |\, o \in oId\ :$
22. $p \neq elise \wedge r \neq guichetier \wedge o \neq Toronto$
23. $\Longrightarrow \langle p, r, o, \_, \mathsf{valider}(\_,\_)\rangle$
24. || $|\, p \in pId\ :\ |\, r \in rId\ :\ |\, o \in oId\ :$
25. $p \neq boris \wedge r \neq guichetier \wedge o \neq Montreal$
26. $\Longrightarrow \langle p, r, o, \_, \mathsf{annuler}(\_,\_)\rangle$
27. || $|\, p \in pId\ :\ |\, r \in rId\ :\ |\, o \in oId\ :$
28. $p \neq elise \wedge r \neq guichetier \wedge o \neq Toronto$
29. $\Longrightarrow \langle p, r, o, \_, \mathsf{annuler}(\_,\_)\rangle$
30. || $|\, p \in pId\ :\ |\, r \in rId\ :\ |\, o \in oId\ :\ |\, c \in cId\ :$
31. $Cheque(c).valeur > Organisation(o).limite$
32. $\rightarrow p \neq boris \wedge r \neq guichetier \wedge o \neq Montreal$
33. $\Longrightarrow \langle p, r, o, \_, \mathsf{valider}(\_,\_)\rangle$
34. || $|\, p \in pId\ :\ |\, r \in rId\ :\ |\, o \in oId\ :\ |\, c \in cId\ :$
35. $Cheque(c).valeur > Organisation(o).limite$
36. $\rightarrow p \neq elise \wedge r \neq guichetier \wedge o \neq Toronto$
37. $\Longrightarrow \langle p, r, o, \_, \mathsf{valider}(\_,\_)\rangle))^+$

Les lignes **2** à **11** sont obtenues à l'aide du patron des patrons de permissions sans contraintes et correspondent aux permissions de la figures 3.3. Les lignes **12** à **17** sont obtenues à l'aide des patrons de permissions sous contraintes et correspondent aux permissions sous contraintes de la figure 3.4. Les lignes **18** à **29** sont obtenues à l'aide des patrons d'interdiction et correspondent aux interdictions de la figure 3.5. Les lignes **30** à **37** sont obtenues à l'aide des patrons d'interdictions sous contraintes et correspondent aux interdictions sous contraintes de la figure 3.6.

4.4.2 Obligation et SoD

Maintenant que les patrons de contraintes de CA de type obligation ou SoD sont exprimés, on peut définir comment les mettre en commun.

L'expression de processus associée aux obligations et SoD, permet d'exprimer la politique de CA applicable à tous les *workflows* en l'exprimant pour un *workflow* donné. Elle contient en racine un entrelacement des paramètres fonctionnels du SI, puis les résultats des différents patrons reliés entre eux à l'aide des opérateurs de choix de parallèle synchronisé et d'entrelacement.

4.4.2.1 Choix

Le choix est utilisé pour rassembler deux patrons dont l'application est disjointe : c'est à dire que durant l'exécution du *workflow* l'un des patrons s'applique mais pas les deux. Comme illustration, on peut citer la règle **regle 4** qui introduit deux obligations entre deposer et valider et entre deposer et annuler. Ces deux obligations peuvent être modélisée à l'aide des patrons d'obligation et reliées à l'aide d'un choix :

1. **regle_4** $(\, c : cId, cl : clId \,) \triangleq$
2. $\quad (|\, p \in pId \; :$
3. $\quad\quad \langle p, \_, \_, \mathsf{deposer}(\, cl, c \,)\rangle$
4. $\quad\quad . \quad \langle p, \_, \_, \mathsf{valider}(\, cl, c \,)\rangle))$
5. $\quad | \quad (|\, p' \in pId \; :$
6. $\quad\quad \langle p', \_, \_, \mathsf{deposer}(\, cl, c \,)\rangle$
7. $\quad\quad . \quad \langle p', \_, \_, \mathsf{annuler}(\, cl, c \,)\rangle))$

4.4.2.2 Parallèle

L'opérateur de parallèle peut être utilisé pour mettre en commun deux patrons qui doivent s'appliquer en même temps et de manière synchronisée : c'est à dire qu'ils s'appliquent simultanément pour les mêmes événements. Comme illustration, on peut utiliser les règles **regle 4** et **regle 8** modélisées respectivement par les expressions de processus **regle_4** et $\mathbf{obligation}_1^{p,c}$. Comme elles doivent se synchroniser sur les actions communes on obtient alors :

1. **regle_4&8** $(\, c : cId, cl : clId \,) \triangleq$
2. $\quad (\quad (|\, p \in pId \; :$
3. $\quad\quad \langle p, \_, \_, \mathsf{deposer}(\, cl, c \,)\rangle$
4. $\quad\quad . \quad \langle p, \_, \_, \mathsf{valider}(\, cl, c \,)\rangle))$
5. $\quad |\quad (|\, p' \in pId \; :$
6. $\quad\quad \langle p', \_, \_, \mathsf{deposer}(\, cl, c \,)\rangle$
7. $\quad\quad . \quad \langle p', \_, \_, \mathsf{annuler}(\, cl, c \,)\rangle))))$
8. $\quad \|\quad (|\, p \in pId \; : \, |\, t \in Timestamp \; : \, |\, t' \in Timestamp \; : \, |\, o \in oId \; :$
9. $\quad\quad (\quad \langle p, \_, o, t, \mathsf{deposer}(\, cl, c \,)\rangle$
10. $\quad\quad . \quad\quad (Cheque(c).montant > Organisation(o).limite$
11. $\quad\quad\quad\quad \Longrightarrow \langle p, \_, \_, t', \mathsf{enregistrer}(\, c \,)\rangle$
12. $\quad\quad\quad |\quad \neg(Cheque(c).montant > Organisation(o).limite) \Longrightarrow \lambda))$
13. $\quad\quad |\quad (\quad \langle p, \_, o, t', \mathsf{enregistrer}(\, c \,)\rangle$
14. $\quad\quad . \quad\quad (Cheque(c).montant > Organisation(o).limite$
15. $\quad\quad\quad\quad \Longrightarrow \langle p, \_, \_, t, \mathsf{deposer}(\, cl, c \,)\rangle$
16. $\quad\quad\quad |\quad \neg(Cheque(c).montant > Organisation(o).limite) \Longrightarrow \lambda)))$

4.4.2.3 Entrelacement

L'opérateur d'entrelacement peut être utilisé pour mettre en commun deux patrons devant s'appliquer en parallèle mais sans synchronisation. Les événements permettant de satisfaire l'une des règles ne permettent pas de satisfaire l'autre, même s'ils proviennent d'une action dont le *label* est commun au deux règles.

4.4.2.4 Mise en commun

L'ensemble des règles dynamiques doit être regroupé dans une expression de processus appelée **OblSod**. Cette expression permet d'instancier la politique de CA pour chaque valeur du workflow. En illustration, on peut reprendre les règle **regle 4** et **regle 8** :

1. **OblSod** ( ) $\triangleq$
2. $|||\, c \in cId$: $|||\, cl \in clId$:
3. **regle_4** (c, cl)
4. $\|$ **obligation**$_1^{p,c}$ (c, cl)

4.4.3 Le main

L'expression de processus **main** permet d'exprimer la politique de contrôle d'accès globale du SI. Elle est obtenue en mettant en parallèle synchronisé l'expression de processus représentant les permissions et interdictions et l'expression de processus représentant les obligations et SoD.

1. **main** ( ) $\triangleq$
2. **PermInter** ( )
3. $\|$ **OblSod** ( )

4.5 Discussion sur l'utilité

Plusieurs remarques peuvent être faites sur les patrons, les plus pertinentes sont traitées dans cette section. Les patrons correspondant aux DSD et au obligations sont relativement semblable. En effet, la différence entre les patrons se trouve dans la contrainte qu'ils induisent sur les environnements de sécurité nécessaires à la réalisation des actions concernées par le patron. Dans le cadre d'un DSD cette contrainte correspond au fait qu'un élément des environnements de sécurité doit être différent pour chaque action concernée par le patron. Pour une obligation, l'élément doit être identique (ou relié grâce au diagramme de classes). Pour les patrons de DSD, cette contrainte est exprimable à l'aide d'une garde tandis que pour les obligation cette contrainte n'est pas directement exprimable dans le patron.

L'action λ utilisée par les patrons avec contraintes mais sans sanctions et les sanctions utilisées dans les patrons avec contraintes et sanctions sont des actions qui ne peuvent être exécutées par un utilisateur du système. En effet, elles correspondent à des actions *systèmes*, c'est-à-dire des actions devant être réalisées par un administrateur voir le système lui-même.

Pour les opérateurs quantifiés, les ensembles de quantification correspondent à l'ensemble des instances des classes du diagramme de classes.

Pour certaines règles de CA, les patrons présentés précédemment ne présentent pas d'utilité. Dans ce cas, la personne en charge de la modélisation peut les modéliser à l'aide des techniques présentées dans le chapitre 3 et les incorporer aux résultats des patrons à l'aide des opérateurs de mise en commun.

Les patrons définis dans ce chapitre peuvent être utilisés de manière à créer de nouveaux opérateurs EB^{3}SEC. En effet, l'utilisation de ces opérateurs peut simplifier la tâche du concepteur utilisant la méthode EB^{3}SEC pour exprimer la politique de CA utilisée dans le SI.

Chapitre 5

Outils pour la vérification et la validation

Les chapitres précédents permettent de modéliser en EB^{3}SEC une politique de CA donnée. Ce chapitre explique comment la spécification obtenue peut être vérifiée ou utilisée dans le cadre d'une implémentation. Nous présentons tout d'abord quels types de propriétés peuvent être vérifiés sur une spécification EB^{3}SEC. Ensuite, nous étudions les façons de pouvoir vérifier ces propriétés sur une spécification. Finalement, nous donnons une implémentation possible.

5.1 Types de propriétés

Cette section présente les différents types de propriétés que l'on désire vérifier sur une spécification EB^{3}SEC. Nous présentons d'abord les trois grandes catégories de propriétés avant de sélectionner les plus intéressantes à notre cadre d'étude et finalement d'en donner des exemples.

Les trois grandes catégories de propriétés sont les suivantes.

Les propriétés de vivacité permettent d'exprimer le fait qu'un événement doit s'exécuter. Ces propriétés sont décrites dans la partie 5.1.1.

Les propriétés de sûreté permettent de spécifier qu'un événement inopportun ne peut s'exécuter. Cette catégorie de propriétés est décrite dans la section 5.1.2.

Les propriétés d'équité permettent d'exprimer qu'une action potentiellement exécutable va être exécutée. Dans le cadre des SI, les événements sont déclenchés par les utilisateurs de l'application et non par le système lui-même. La vérification de propriétés d'équité n'a que peu de sens dans le cadre des politiques de CA.

Les propriétés de vivacité et de sûreté étant très utilisées dans le cadre de SI, elles sont chacune scindées en deux sous-catégories présentées dans [29].

5.1.1 Propriétés de vivacité

Les propriétés de vivacité permettent de spécifier qu'un événement doit s'exécuter. Bien que l'exécution de l'événement ne soit pas nécessairement immédiate, elle doit être obligatoire. Parfois, ce type de propriétés permet aussi de spécifier qu'une action entraîne une réaction de la part du système. Cette deuxième option est très rarement utilisée dans le cas des SI, car les actions sont réalisées par un humain (aujourd'hui la technologie ce permet pas de forcer un humain à réaliser une action). De plus, les situations de *deadlock*, dans le sens où un état ne permet plus de réaliser aucune action, sont exceptionnelles dans un SI.

Condition suffisante pour l'exécution d'un événement (SCE : *Sufficient state condition to enable an event*)

Dans cette catégorie sont regroupées les propriétés permettant d'exprimer le fait qu'un utilisateur du système peut effectuer une action lorsqu'une certaine condition est vérifiée. Dans cette définition, l'expression "effectuer une action" signifie que l'action est immédiatement exécutable dans l'état courant. Dans le cadre des systèmes bancaires, l'exemple peut être : un chèque peut toujours être déposé par un client s'il n'est pas déjà déposé.

Condition suffisante pour l'exécution future d'un événement (SCEF : *Sufficient state condition to enable an event in the future*)

Ce type de propriétés exprime le fait qu'il existe une suite d'événements permettant

d'exécuter un événement particulier. Dans le cadre des systèmes bancaires, un exemple d'une telle propriété est : le compte de dépôt d'un client peut toujours être crédité. Ce type de propriétés correspond au propriétés les plus fréquemment utilisées dans le cadre des SI. Des exemples plus détaillés de ce type de propriété sont donnés dans la partie 5.1.3

5.1.2 Propriétés de sûreté

Les propriétés de sûreté permettent de vérifier qu'un événement non voulu ne s'exécutera jamais lors de l'exécution du système. Adaptées aux SI, ce type de propriété correspond à des invariants, représentant une condition nécessaire à l'exécution d'un événement.

Invariant (INV : *Invariant state property*)

Les propriétés d'invariance concernent des propriétés qui doivent être satisfaites quelque soit l'état du système. Dans le cadre des systèmes bancaires, un exemple de propriété d'invariance peut être : le montant correspondant aux comptes de dépôt d'un client doit toujours être positif...

Condition nécessaire pour l'exécution d'une action (NCE : *Necessary state condition to enable an event*)

Ce type de propriétés exprime le fait que pour que le système accepte un événement, il faut qu'une condition soit vraie dans le système. Par exemple, dans le domaine bancaire : pour qu'un compte soit crédité, il faut que le chèque correspondant ait été validé.

5.1.3 Quelques exemples de propriétés

En plus des types de propriétés précédents, nous avons défini un ensemble de contraintes devant être vérifiées par la politique de CA et la modélisation de la partie fonctionnelle du SI. Cette partie présente quelques exemples de ces contraintes. Tout d'abord, quelques définitions sont introduites dans la première section.

5.1.3.1 Définitions préliminaires

Dans la suite, nous considérons que le SI a été modélisé entièrement : la partie fonctionnelle du système a été modélisée à l'aide de la méthode EB^3 et la partie correspondant à la politique de CA a été spécifiée à l'aide de la méthode EB^{3}SEC. Nous définissons les ensembles suivants (les détails sur la définition de ces ensembles sont donnés dans [28]).

Σ_{fonc} représente l'ensemble des entrées (événements) valides correspondant à la modélisation fonctionnelle.

Σ_{CA} correspond à l'ensemble des entrées (événements sécurisés) valides correspondant à la modélisation de la politique de CA. L'ensemble Σ_{CA} est divisé en deux sous-ensembles : Σ_{stat} et Σ_{dyn}.

Σ_{stat} regroupe l'ensemble des entrées valides de la partie statique.

Σ_{dyn} représente l'ensemble des entrées valides de la partie dynamique.

On définit l'opérateur α permettant de donner le nom de l'action correspondant à un événement. Par extension, cet opérateur s'applique à un ensemble d'événements et retourne l'ensemble des noms d'actions correspondant aux événements présents dans l'ensemble.

$\alpha(\Sigma_{fonc})$ correspond à l'ensemble des noms d'actions présents dans la modélisation fonctionnelle.

$\alpha(\Sigma_{CA})$ correspond à l'ensemble des noms d'actions présents dans la modélisation de la politique de CA.

$\alpha(\Sigma_{stat})$ est l'ensemble des noms d'actions présents dans la partie statique de la politique de CA.

$\alpha(\Sigma_{dyn})$ est l'ensemble des noms d'actions présents dans la partie dynamique de la modélisation de la politique de CA.

La sémantique de la notation EB^3 associe à une expression de processus un système de transitions étiquetées.

$\mathcal{M}_{fonc}$ est le système de transitions obtenu à partir de la modélisation fonctionnelle.

$\mathcal{M}_{CA}$ est le système de transitions obtenu à partir de la modélisation de la politique de CA en EB^{3}SEC. En effet, grâce au prédicat statique, une spécification réalisée en EB^{3}SEC peut être transformée en modélisation EB^3. $\mathcal{M}_{CA}$ peut être séparé en deux sous-systèmes, $\mathcal{M}_{dyn}$ et $\mathcal{M}_{stat}$.

$\mathcal{M}_{dyn}$ correspond au système obtenu à partir de la partie dynamique de la modélisation de la politique de CA.

$\mathcal{M}_{stat}$ est le modèle obtenu à partir de la partie statique de la politique de CA. En effet, les patrons, définis dans le chapitre 4, assurent que la partie statique de la politique de CA est exprimable à l'aide d'une expression de processus.

Dans la suite, e dénote une action utilisée dans la modélisation de la politique de CA et e' l'action correspondante après l'utilisation du prédicat statique. On considère alors les formules de logique CTL bien formées sur ces systèmes et on note $\mathcal{M}_{fonc} \models_0 \mathcal{F}$ le fait que la formule logique CTL bien formée $\mathcal{F}$ est vérifiée par le modèle $\mathcal{M}_{fonc}$ à partir de l'instant initial, pour simplifier la notation on note : $\mathcal{M}_{fonc} \models \mathcal{F}$. En effet, dans la suite nous considérons que toutes les formules doivent être vérifiées à partir de l'état initial.

5.1.3.2 Typage de la politique de CA (type INV)

La première propriété à vérifier sur un modèle EB^{3}SEC correspond à une propriété de typage :

$$\alpha(\Sigma_{dyn}) \subseteq \alpha(\Sigma_{stat})$$

Cette propriétés permet de s'assurer que toutes les actions présentes dans les contraintes dynamiques de CA, font aussi l'objet de contraintes statiques. En effet, toute action apparaissant dans

une contrainte de type *SoD* ou de type *obligation* doit être permise par la partie statique de la politique de CA.

5.1.3.3 Atteignabilité de tous les événements du système (type SCEF)

Toutes les actions exécutables par la partie fonctionnelle doivent pouvoir être exécutées au moins une fois par la politique de CA. Cette propriété peut s'énoncer par :

$$\forall e \in \alpha(\Sigma_{fonc}) : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists \overrightarrow{p_{sec}} \in \overrightarrow{E_{sec}} : \\ (\mathcal{M}_{fonc} \models EF(e(\overrightarrow{p_{fonc}}))) \Rightarrow (\mathcal{M}_{CA} \models EF(e'(\overrightarrow{p_{sec}}, \overrightarrow{p_{fonc}})))$$

Les quantifications permettent d'expliciter le fait que pour toutes les actions présentes dans la modélisation on peut trouver un vecteur de paramètres fonctionnels et un vecteur de paramètres de sécurité tels que si la partie fonctionnelle permet d'exécuter l'action pour le vecteur de paramètres fonctionnels (première partie de l'implication) alors le modèle de la politique de CA permet de l'exécuter pour ces valeurs paramètres fonctionnels et de sécurité (seconde partie de l'implication).

5.1.3.4 Atteignabilité de tous les événements sécurisés (type SCEF)

Pour toutes les actions sécurisées, il faut s'assurer que la partie statique et la partie dynamique de la modélisation de la politique de CA permettent de les exécuter au moins une fois :

$$\forall e \in \alpha(\Sigma_{dyn}) : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists \overrightarrow{p_{sec}} \in \overrightarrow{E_{sec}} : \\ \mathcal{M}_{stat} \models EF(e'(\overrightarrow{p_{sec}}, \overrightarrow{p_{fonc}})) \Leftrightarrow (\mathcal{M}_{dyn} \models EF(e'(\overrightarrow{p_{sec}}, \overrightarrow{p_{fonc}})))$$

En effet, dans le cas contraire, les permissions et interdictions sont trop restrictives et ne permettent pas aux contraintes de SoD ou aux obligations de pouvoir s'exécuter pleinement.

5.1.3.5 Faisabilité des permissions (type SCEF)

Cette propriété a pour but de vérifier que toutes les permissions décrites peuvent être exécutées durant l'exécution du système. Elle s'écrit :

$$\forall p \in Permission : \exists e \in \alpha(\Sigma_{CA}) : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists u \in personne : \exists t \in TIMESTAMP : \\ (p.e = e) \wedge (\mathcal{M}_{CA} \models EF(e'(u, p.r, p.o, t, \overrightarrow{p_{fonc}})))$$

Dans cette formule, l'élément $p.r$ correspond au rôle utilisé dans la permission, $p.o$ à l'organisation et $p.e$ à l'action. Cette propriété n'a pas pour but d'éviter une incohérence dans la spécification. Elle permet à la personne ayant modélisé la politique de CA de s'assurer que toutes les permissions décrites dans la politique sont utiles et peuvent être utilisées par un utilisateur du système.

5.1.3.6 Faisabilité des instances de l'association "joue" (type SCEF)

Cette propriété permet de tester que toutes les instances de l'association **joue** du diagramme de classes de la figure 3.2 peuvent être utilisées au cours de l'exécution du système. Cette propriété s'écrit :

$$\forall j \in joue : \exists e \in \alpha(\Sigma_{CA}) : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists t \in TIMESTAMP : \\ \mathcal{M}_{CA} \models EF(e'(j.p, j.r, j.o, t, \overrightarrow{p_{fonc}}))$$

De même que la propriété précédente, cette propriété ne permet pas de déceler des erreurs de cohérence de la modélisation. Elle permet à la personne ayant réalisé la spécification de s'assurer que les éléments de l'association **joue** qui sont instanciés peuvent être utilisés au moins une fois durant l'exécution du système.

5.1.3.7 Utilisabilité des instances de la classe "Organisation" (type SCEF)

Cette propriété est utilisée pour déterminer si toutes les organisations présentes dans le modèle ont leur utilité. Cette propriété comme les deux précédentes ne permet pas de déterminer si le modèle est cohérent mais aide la personne ayant spécifié le modèle à vérifier que les organisations instanciées sont utiles. Elle s'exprime par :

$$\forall o \in oId : \exists e \in \alpha(\Sigma_{CA}) : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists p \in pId : \exists r \in rId \exists t \in TIMESTAMP : \\ \mathcal{M}_{CA} \models EF(e'(p, r, o, t, \overrightarrow{p_{fonc}}))$$

Cette propriété permet de vérifier que l'on peut trouver une action sécurisée exécutable pour chaque organisation.

5.1.3.8 Utilisabilité des instances de la classe “Personne” (type SCEF)

Cette propriété correspond à la propriété précédente adaptée aux instances de la classe **Personne**. Comme la précédente, cette propriété ne permet pas de vérifier la cohérence ou l'exactitude de la spécification, mais aide la personne ayant réalisé la modélisation à vérifier que la spécification ne comporte pas une instance de la classe **Personne** inutilisable. Cette propriété s'écrit :

$$\forall p \in pId : \exists e \in \alpha(\Sigma_{CA}) : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists o \in oId : \exists r \in rId : \exists t \in TIMESTAMP : \\ \mathcal{M}_{CA} \models EF(e'(p, r, o, t, \overrightarrow{p_{fonc}}))$$

5.1.3.9 Utilisabilité des instances de la classe “Role” (type SCEF)

Les propriétés d'utilisabilité précédentes peuvent être adaptées aux autres classes du diagramme de classes. Par exemple, pour la classe **Role**, on obtient :

$$\forall r \in rId : \exists e \in \alpha(\Sigma_{CA}) : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists o \in oId : \exists p \in pId \exists t \in TIMESTAMP : \\ \mathcal{M}_{CA} \models EF(e'(p, r, o, t, \overrightarrow{p_{fonc}}))$$

5.1.3.10 Application des propriétés précédentes

Les propriétés précédentes peuvent être modifiées de manière à spécifier une certaine action. Par exemple, la propriété de la section 5.1.3.8 peut être modifiée de façon à cibler une action précise.

$$\exists p \in pId : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists o \in oId : \exists r \in rId : \exists t \in TIMESTAMP : \\ \mathcal{M}_{CA} \models EF(deposer'(p, r, o, t, \overrightarrow{p_{fonc}}))$$

Cette propriété permet de vérifier que toute personne cliente de l'institution bancaire peut venir faire un dépôt de chèque au sein de l'institution.

5.1.3.11 Non-faisabilité des interdictions

De même que l'on peut vérifier que toutes les permissions peuvent être utilisées au cours de l'exécution du système, on peut aussi vérifier que les interdictions ne peuvent pas être exécutées au cours de l'exécution du système. Cette propriété peut être vérifiée par :

$$\forall p \in Prohibition : \neg(\exists e \in \alpha(\Sigma_{CA}) : \exists \overrightarrow{p_{fonc}} \in \overrightarrow{E_{fonc}} : \exists u \in personne : \exists t \in TIMESTAMP : (p.e = e) \wedge (\mathcal{M}_{CA} \models EF(e'(u, p.r, p.o, t, \overrightarrow{p_{fonc}}))))$$

5.2 Vérification des propriétés

Pour valider une politique de CA, plusieurs technique peuvent être envisagées. Une façon de procéder est d'exprimer la politique dans un langage formel offrant des outils de simulation, validation ou vérification de modèles. La simulation de politiques de CA à l'aide de la méthode Z a été étudiée dans [49]. L'utilisation de différents outils de vérification de modèles dans le cadre des SI a été étudiée dans [29]. Ce travail indique que le plus gros problème est le nombre important d'instances des différentes entités. Les outils de vérification de modèles ne permettent pas de faire intervenir un nombre important d'instances. Cependant, l'article conclut que la validation des propriétés pourrait être réalisée en utilisant que le nombre minimal d'entités impliquées par la propriété considérée. Nous proposons une méthode permettant la validation d'une politique de CA modélisée en EB^{3}SEC. La section 5.2.1 s'intéresse à la validation de la partie statique tandis que la section 5.2.2 présente une simulation de la partie dynamique d'une politique de CA.

5.2.1 Vérification de la partie statique

Cette section s'intéresse à la partie statique de la politique de CA (*i.e.* les contraintes de CA pouvant être définies en EB^{3}SEC à l'aide du diagramme de classes). Elle décrit l'utilisation d'un outil permettant de décrire une politique de CA à l'aide des patrons décrits dans le chapitre 4. La section 5.2.1.1 décrit les patrons et opérateurs choisis pour cette implémentation, La section 5.2.1.2 définit le langage utilisé en entrée de l'outil et la section 5.2.1.3 explique et illustre les algorithmes utilisés pour la vérification de la partie statique.

5.2.1.1 Choix des patrons implémentés

Un sous-ensemble des patrons, définis dans le chapitre 4, a été sélectionné. Ce sous-ensemble comprend :

- **permission** ( ) : les permissions sans contraintes et non quantifiées,
- **interdiction** ( ) : les interdictions sans contraintes,
- **SoD**^{s} ( ) : les séparations des devoirs séquentielles sans contraintes et sans sanctions,
- **obligation**^{s} ( ) : les obligations séquentielles sans contraintes et sans sanctions.

En plus des opérateurs utilisés pour exprimer les contraintes de CA, il faut ajouter des opérateurs de mise en commun :

- $\|$: le parallèle synchronisé,
- $|||$: l'entrelacement.

De manière à pouvoir représenter la politique de CA, il faut aussi pouvoir expliciter l'association *joue* utilisée en EB^{3}SEC.

5.2.1.2 Le langage d'entrée

Le langage utilisé en entrée permet de détailler les différentes composantes de la politique de CA. Les six différentes parties composant la description d'une politique de CA sont décrites ci-après.

La clause *play* correspond à la table *joue* de la spécification. Elle permet de déclarer quelle personne joue quel rôle dans quelle organisation. La partie *joue* correspond à une liste de triplets ne pouvant contenir que des constantes. Un exemple de cette partie est donnée à la figure 5.1.

```
play () ==
  <Alphonse,guichetier,Montreal>
  & <Boris,conseiller,Montreal>
  & <Catherine,guichetier,Montreal>
  & <Denis,conseiller,Montreal>
  ;
```

FIGURE 5.1 – Exemple de table *joue*

```
permission () ==
   <_,guichetier,Montreal,deposer()>
   & <_,conseiller,Montreal,deposer()>
   & <_,conseiller,Montreal,valider()>
   & <catherine,_,_,valider()>
   ;
```

FIGURE 5.2 – Exemple pour la partie permission

La clause *permission* correspond à la déclaration des permissions sans contraintes. Elle correspond à une liste de *n*-uplets. Les environnements de sécurité de ces *n*-uplets ne contiennent que des constantes ou des éléments de type _. Un exemple de partie *permission* est donné à la figure 5.2.

La clause *interdiction* correspond à la déclaration des interdictions sans contraintes. Elle correspond à une liste de *n*-uplets. Les environnements de sécurité de ces *n*-uplets ne contiennent que des éléments de type _ et des éléments correspondant à des négations de constantes. Les négations de constantes correspondent aux valeurs interdites pour l'action du *n*-uplet. Un exemple de partie *in-*

```
interdiction () ==
   <_,!guichetier,!Montreal,valider()>
   & <!Denis,_,_,valider()>
   ;
```

FIGURE 5.3 – Exemple pour la partie interdiction

```
sod () ==
 (
(
SOD(user,<andre, _, _,_,deposer()>,<!andre, _, _,_ ,valider()>)
  ||| SOD(user,<user, _, _,_,deposer()>,<!user, _, _,_ ,cancel()>)
 )
||
 (
SOD(role,<_,role, _,_,modifier()>,<_, !role, _,_ ,valider()>)
|||
 SOD(user,<user,_,_,_,valider()>,<!user,_,_,_,valider()>)
)
)
;
```

FIGURE 5.4 – Exemple pour la partie SoD

terdiction est donnée à la figure 5.3. Dans cet exemple, la personne nommée **Denis** ne peut réaliser l'action **valider**.

La clause *sod* correspond à l'utilisation des SoD. Elle ne comporte que des opérateurs de synchronisation parallèle, d'entrelacement et de SoD. L'opérateur de SoD est un opérateur ternaire. Le premier argument correspond à la variable de l'environnement de sécurité sur laquelle porte la contrainte de SoD. Les deux autres arguments portent sur les actions sécurisées concernées par la SoD. Un exemple de partie SoD est donné à la figure 5.4. Le premier argument permet de préciser sur quelle paramètre de sécurité porte la SoD : l'utilisateur (le paramètre vaut alors `user`), le rôle (le paramètre vaut alors `role`) ou l'organisation (le paramètre vaut alors `organization`). Les deux autres arguments indiquent les actions sécurisées sur lesquelles porte la SoD. Pour les actions sécurisées y figurant seuls doivent être précisés l'action et le paramètre sur lequel porte la SoD. Si le paramètre de sécurité vaut la même valeur que le premier argument alors la SoD est valable pour toutes les valeurs possibles de ce paramètre. Au contraire, si le paramètre de sécurité prend une valeur constante particulière, alors la SoD n'est valable que pour cette valeur. Par exemple,

```
obligation () ==
( OBL(role,<_,role, _,_ ,deposer()>,<_,role, _,_ ,modifier()>)
||
OBL(role,<_,role, _,_ ,deposer()>,<_,role, _,_ ,modifier()>))
;
```

FIGURE 5.5 – Exemple pour la partie obligation

```
SOD(user,<andre, _, _,_,deposer()>,<!andre, _, _,_ ,valider()>)
```

spécifie que la SoD entre les actions `deposer` et `valider` n'est valable que pour l'utilisateur `andre`. Tandis que

```
SOD(user,<user, _, _,_,deposer()>,<!user, _, _,_ ,valider()>)
```

précise que la SoD entre les actions `deposer` et `valider` est valable pour tous les utilisateurs.

La clause *obligation* concerne les obligations. Elles est composée d'opérateurs de synchronisation parallèle, d'opérateurs d'entrelacement et d'opérateurs d'obligation. L'opérateur d'obligation est aussi un opérateur ternaire. Le premier argument représente l'élément de l'environnement de sécurité sur lequel porte l'obligation et les deux autres arguments représentent les actions sécurisées sur lesquelles porte l'obligation. La figure 5.5 présente un exemple pour la partie obligation. L'opérateur d'obligation est utilisé de la même manière que l'opérateur de SoD. Le premier argument permet de préciser sur quel paramètre de sécurité porte l'obligation. Les deux derniers arguments permettent de préciser sur quelles actions sécurisées porte l'obligation. Si dans les actions sécurisées la variable concernée par l'obligation a pour valeur le premier argument alors l'obligation s'applique à toutes les valeur de ce paramètre de sécurité.

La clause *complex* permet d'utiliser des SoD et des obligations dans la même règle de manière à pouvoir créer des règles de CA plus élaborées. Les SoD et les obligations sont reliées par des opérateurs de synchronisation parallèle et des opérateurs d'entrelacement. La figure 5.6 montre un exemple d'utilisation de la partie complexe.

```
complex () ==
 (
(
OBL(user,<user,_, _,_ ,deposer()>,<user,_, _,_ ,modifier()>)
  ||| SOD(user,<user, _, _,_,deposer()>,<!user, _, _,_ ,cancel()>)
 )
||
 (
SOD(user,<user, _, _,_,modifier()>,<!user, _, _,_ ,valider()>)
|||
 OBL(role,<_,role, _,_ ,deposer()>,<_,role, _,_ ,modifier()>)
)
)
;
```

FIGURE 5.6 – Exemple pour la partie complexe

5.2.1.3 Algorithme de vérification de la partie statique

Les ensembles contenant les permissions et les interdictions peuvent contenir des contradictions voire des *deadlocks*. Cette partie décrit un ensemble d'algorithmes servant à vérifier que la partie statique de la politique de CA (*i.e.* les permissions et les interdictions) ne contiennent pas de telles incohérences. L'idée de ces algorithmes est de calculer pour chaque action l'ensemble des environnements de sécurité qui peuvent être effectivement utilisés pour l'exécuter. Pour chaque action, on calcule l'ensemble des environnements de sécurité autorisés par les permissions auquel on soustrait l'ensemble des environnements de sécurité interdits par les prohibitions. Si pour une action, on obtient un ensemble d'environnements de sécurité effectivement autorisés vide, alors cette action ne peut être exécutée.

Nous définissons deux environnements de sécurité appelés $\overrightarrow{p_{sec}} = (u, r, o)$ et $\overrightarrow{p'_{sec}} = (u', r', o')$, deux relations $isParamComp$ et $isComp$.

$isParamComp(x, x') \triangleq$
1. `if` $(x = \_ \vee x' = \_)$ `return` $TRUE$
2. `else if` $(x = x')$ `return` $TRUE$
3. `else return` $FALSE$

$isComp(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}}) \triangleq$ `return`
1. $isParamComp(u, u')$
2. $\wedge$ $isParamComp(r, r')$
3. $\wedge$ $isParamComp(o, o')$

La relation $isParamComp$ vérifie si deux paramètres de sécurité sont compatibles : si l'un d'entre eux vaut _, ils sont compatibles sinon ils doivent avoir tous deux la même valeur. La relation $isComp$ vérifie si deux environnements de sécurité sont compatibles : c'est-à-dire si tous les paramètres de sécurité sont compatibles deux à deux (*i.e.* si les paramètres correspondant aux utilisateurs sont compatibles entre eux, et de même pour les rôles et les organisations).

La première étape de la vérification consiste à déplier les clauses `play` et `permissions`. Pour chaque action, on obtient l'ensemble des environnements de sécurité autorisés pour cette action (les environnements de sécurité obtenus ne contiennent plus de valeurs _ mais uniquement des constantes). L'algorithme $PerEff$ décrit dans la suite est utilisé pour cette étape ; il calcule la jointure des deux relations `play` et `permissions`. Il est décrit d'une manière fonctionnelle (*i.e.* dans une syntaxe proche du CAML) :

$PerEff(play, permission) \triangleq$ `match` $permission$ `with`
1. $\langle \overrightarrow{p_{sec}}, \mathsf{a}(\ ) \rangle :: queue \rightarrow$ `return` $unfoldPlay(play, \overrightarrow{p_{sec}}, \mathsf{a}) \cup PerEff(play, queue)$
2. $null \rightarrow$ `return` $\emptyset$

$unfoldPlay(play, \overrightarrow{p_{sec}}, \mathsf{a}) \triangleq$ `match` $play$ `with`
1. $\langle \overrightarrow{p'_{sec}} \rangle :: queue \rightarrow$ `if` $(isComp(\overrightarrow{p_{sec}}, \overrightarrow{p'_{sec}})$
2. `return` $\langle \overrightarrow{p'_{sec}}, \mathsf{a}(\ ) \rangle \cup unfoldPlay(queue, \overrightarrow{p_{sec}}, \mathsf{a})$
3. `else return` $unfoldPlay(queue, \overrightarrow{p_{sec}}, \mathsf{a})$
4. $null \rightarrow$ `return` $\emptyset$

La figure 5.7 montre le résultat de cet algorithme sur les exemples de la partie 5.2.1.2 (figure 5.1 et5.2).

$$
\begin{array}{l}
PerEff(play, permission) = \\
\quad [\quad \langle Alphonse, guichetier, Montreal, \textbf{deposer} \rangle \\
\quad , \quad \langle Catherine, guichetier, Montreal, \textbf{deposer} \rangle \\
\quad , \quad \langle Boris, conseiller, Montreal, \textbf{deposer} \rangle \\
\quad , \quad \langle Denis, conseiller, Montreal, \textbf{deposer} \rangle \\
\quad , \quad \langle Boris, conseiller, Montreal, \textbf{valider} \rangle \\
\quad , \quad \langle Denis, conseiller, Montreal, \textbf{valider} \rangle \\
\quad , \quad \langle Catherine, guichetier, Montreal, \textbf{valider} \rangle]
\end{array}
$$

FIGURE 5.7 – Résultat de l'algorithme $PerEff$

La prochaine étape est d'obtenir pour chaque action l'ensemble des valeurs d'environnements de sécurité interdits par la clause `interdiction`. L'algorithme suivant correspond à cette étape :

$ProhEff(play, interdiction) \triangleq$ `match` $interdiction$ `with`

1. $\langle \overrightarrow{p_{sec}}, \mathsf{a}(\ ) \rangle :: queue \rightarrow$ `return` $unfoldPlay(play, \overrightarrow{p_{sec}}, \mathsf{a}) \cup ProhEff(play, reste)$

2. $null \rightarrow$ `return` $\emptyset$

La figure 5.8 présente le résultat de cet algorithme :

$$
\begin{array}{l}
ProhEff(play, interdiction) = \\
\quad [\quad \langle Alphonse, guichetier, Montreal, \textbf{valider} \rangle \\
\quad , \quad \langle Catherine, guichetier, Montreal, \textbf{valider} \rangle \\
\quad , \quad \langle Denis, conseiller, Montreal, \textbf{valider} \rangle]
\end{array}
$$

FIGURE 5.8 – Result of $ProhEff$

La dernière étape consiste à retirer pour chaque action l'ensemble des environnements de sécurité prohibés de l'ensemble des environnements de sécurité autorisés. L'algorithme $PermReal$ est utilisé dans ce but :

$PermReal(pE, iE) \triangleq$ `match` iE `with`

1. $\langle \overrightarrow{p_{sec}}, \mathsf{a}(\ ) \rangle :: queue \rightarrow$ `return` $removeI(pE, \overrightarrow{p_{sec}}, \mathsf{a}) \cup PermReal(pE, queue)$

2. $null \rightarrow$ `return` $\emptyset$

$removeI(pE, \overrightarrow{p_{sec}}, \mathsf{a}) \triangleq$ `match` pE `with`

1. $\langle \overrightarrow{p'_{sec}}, \mathsf{a}' \rangle :: queue \rightarrow$ `if` $(\overrightarrow{p_{sec}} \neq \overrightarrow{p'_{sec}} \wedge \mathsf{a} = \mathsf{a}')$

2. `return` $\langle \overrightarrow{p'_{sec}}, \mathsf{a}(\ ) \rangle \cup removeI(queue, \overrightarrow{p_{sec}}, \mathsf{a})$

3. `else return` $removeI(queue, \overrightarrow{p_{sec}}, \mathsf{a})$

4. $null \rightarrow$ `return` $\emptyset$

La figure 5.9 montre le résultat obtenu pour l'exemple (avec les exemples des figures 5.7 et 5.8).

$$
\begin{aligned}
&PermReal(PerEff(play, genPermission), ProhEff(play, genProhibition)) = \\
&\quad [\ \langle Alphonse, guichetier, Montreal, \textbf{deposer} \rangle \\
&\quad ,\ \langle Catherine, guichetier, Montreal, \textbf{deposer} \rangle \\
&\quad ,\ \langle Boris, conseiller, Montreal, \textbf{deposer} \rangle \\
&\quad ,\ \langle Denis, conseiller, Montreal, \textbf{deposer} \rangle \\
&\quad ,\ \langle Boris, conseiller, Montreal, \textbf{valider} \rangle]
\end{aligned}
$$

FIGURE 5.9 – Result of $PermReal$

Dans l'exemple, pour chaque action, il y a au moins un environnement de sécurité qui permet de l'exécuter. De ce fait la partie statique de la politique de sécurité ne génère pas de *deadlocks*.

5.2.2 Simulation de la partie dynamique

Des outils de vérification de modèles permettraient la vérification de propriétés pour un modèle donné. Nous présentons un outils de simulation de politique EB^{3}SEC. Cette section décrit un outil permettant de simuler les modélisation EB^{3}SEC. Cet outil se base sur la plateforme APIS. La section 5.2.2.1 décrit la plateforme APIS, la section 5.2.2.2 décrit comment est utilisée la plateforme APIS de manière à simuler des modèles EB^{3}SEC.

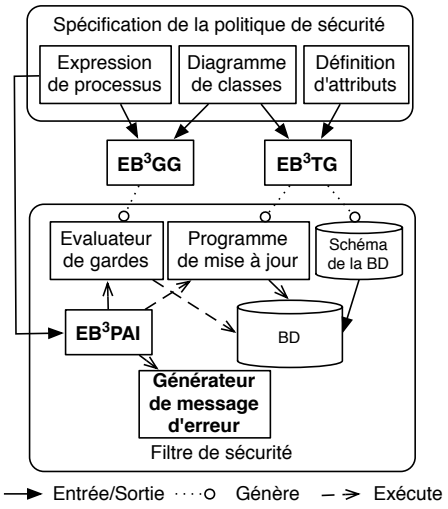

FIGURE 5.10 – Architecture du prototype de noyau de sécurité

5.2.2.1 La plateforme APIS

La plateforme APIS permet de simuler le comportement de modèle EB^3. En plus de comporter un interpréteur efficace de processus, elle permet de mettre à jour une base de données. Cette base de données est durant toute la simulation dans un état cohérent par rapport à la trace et permet l'évaluation des gardes présentes dans la modélisation. Son architecture est présentée à la figure 5.10. EB^{3}PAI [24, 25] est un interpréteur d'expressions de processus EB^3. Les expressions de processus peuvent contenir des prédicats faisant appel à des attributs utilisés par le SI. Pour implémenter le diagramme de classes de la politique de sécurité, nous utilisons EB^{3}TG [30]. Ce module génère un schéma de base de données relationnelle et un programme permettant de mettre à jour les attributs de la base de données en fonction de l'action réalisée. Ce programme est utilisé par EB^{3}PAI pour maintenir la base de données dans un état cohérent pendant l'exécution des actions. Le module EB^{3}GG [46] est utilisé pour générer un programme permettant d'évaluer les gardes contenues dans les expressions de processus. Ce programme est aussi utilisé par EB^{3}PAI pour savoir si une garde est vraie ou fausse. EB^{3}GG génère des requêtes SQL portant sur les bases

de données du SI . Si un événement n'est pas accepté par EB^{3}PAI alors le module présenté dans [55] est utilisé pour générer un message d'erreur adapté.

5.2.2.2 Utilisation de la plateforme APIS pour simuler des modèles EB^{3}SEC

La partie dynamique de la politique de CA ne peut pas être vérifiée à l'aide des algorithmes précédents. Cependant, le comportement de la modélisation EB^{3}SEC peut être simulé. Cette simulation permet, dans une certaine mesure, de découvrir des comportements menant à des *deadlocks* et les corriger le cas échéant. Pour simuler un modèle EB^{3}SEC, on le transforme en modèle EB^3. Cette transformation est possible en utilisant la sémantique des actions sécurisées de manière à les transformer en action EB^3. Cette transformation permet alors à la plateforme APIS de pouvoir simuler le comportement d'un modèle EB^{3}SEC. Cette transformation est possible à l'aide du prédicat statique :

1. $sp(\langle p, r, o, evt\rangle) \triangleq$
2. $\quad \langle p, r, o\rangle \in joue$
3. $\quad \wedge \langle r, o, idEvt(evt)\rangle \in permission \wedge permission(r, o, idEvt(evt)).contrainte$
4. $\quad \wedge (interdiction(r, o, idEvt(evt)).contrainte \Rightarrow \langle r, o, idEvt(evt)\rangle \notin interdiction)$

Le prédicat statique est alors exécuté pour chaque événement sécurisé.

5.3 Un profil XACML pour la méthode EB^{3}SEC

Le langage XACML représente un standard de langage de description de politique de CA. Ce chapitre permet de comparer le langage EB^{3}SEC au langage XACML. Cette comparaison se situe au niveau de l'expressivité des deux langages. Suite à cette comparaison, ce chapitre décrit la possibilité de réaliser un profil EB^{3}SEC en XACML. Une définition du langage XACML est donnée dans la partie 5.3.1.1. La partie 5.3.1.2 présente l'architecture type d'une implémentation du langage XACML. Les éléments du langage sont décrits dans 5.3.1.3. La partie 5.3.2 présente un panorama des différents outils consacrés à l'utilisation du langage XACML et aussi des différentes

implémentations connues. La partie 5.4 présente quelques extensions du langage. Finalement, la partie 5.5 présente les idées utiles à la réalisation d'un profil du langage EB^{3}SEC en XACML.

5.3.1 XACML

5.3.1.1 Un aperçu du langage XACML

XACML est un standard de l'OASIS [6], qui permet d'exprimer une politique de CA. Ce standard se base sur la technologie XML pour la syntaxe du langage. XACML est conçu pour être utilisé dans des SI construits avec une architecture SOA.

Le langage XACML permet de décrire trois types d'objets.

La politique de CA qui définit qui peut accéder à quoi. En XACML, la politique de CA est décomposée en plusieurs règles. Ces règles correspondent à des autorisations ou des refus pour une personne d'obtenir l'accès à un objet. Pour écrire ces règles, les caractéristiques du sujet et/ou des objets du systèmes sont utilisées pour définir des contraintes.

Les requêtes (*Decision Requests*) sont envoyées par le système au PDP (*Policy Decision Point*). Elles contiennent les données nécessaires afin que le PDP puisse prendre une décision en adéquation avec la politique de CA. Les requêtes contiennent les données relatives au sujet (réalisant la requête), l'action qu'il souhaite exécuter, les objets sur lesquels porte la requête et l'environnement (*i.e.* toutes les données ne rentrant pas dans les catégories précédentes : le moment,...).

Les décisions (*Authorization Decision*) correspondent à la réponse du PDP pour une requête donnée.

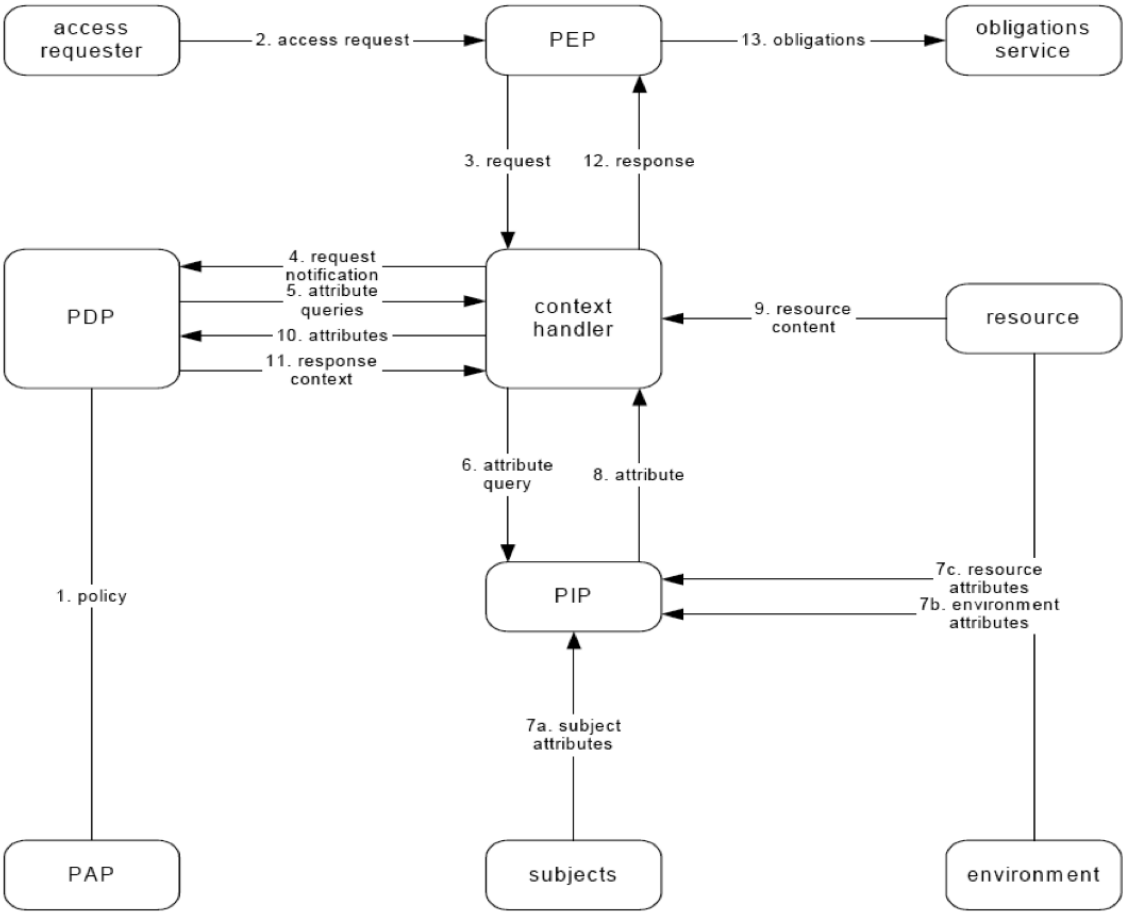

FIGURE 5.11 – Architecture standard de l'implémentation d'une solution XACML

5.3.1.2 L'architecture d'une solution XACML

Le standard XACML fournit une architecture typique pour les implémentations possibles. Cette architecture est présentée à la figure 5.11 tirée de [57]. Cette architecture s'appuie sur différents éléments présents dans une architecture SOA.

La politique de CA est stockée dans le PAP (*Policy Administration Point*). Elle est récupérée par le PDP (flèche 1). Les requêtes sont interceptées par le PEP (*Policy Enforcement Point*) (flèche 2) et envoyées au *context handler* (flèche 3). Ce dernier les convertir de leur forme native vers

la forme canonique utilisée en XACML avant de les envoyer vers le PDP (flèche 4). Le PDP les reçoit, les analyse, rapatrie les règles de CA relatives à la requête et évalue la demande en fonction de la politique avant de renvoyer la décision. Le PIP (*Policy Information Point*) permet de connaître les valeurs des attributs nécessaires à la prise de décision (flèche 5 à 10). La décision est envoyée par le PDP au *context handler* (flèche 11), qui la convertit en langage natif avant de la renvoyer au PEP (flèche 12).

5.3.1.3 Politique de CA en XACML

La figure 5.12, tirée de [57], présente le modèle utilisé par le langage XACML pour décrire une politique de CA. De manière général, une politique de CA est exprimée en XACML à l'aide d'une structure arborescente. La racine de la politique contient toutes les règles, la racine est modélisée à l'aide d'un élément *Policyset* présenté à la section 5.3.1.4. De manière à pouvoir organiser hiérarchiquement la modélisation, la politique utilise des éléments de types *Policyset* ou *policy*. Les éléments de types *Policyset* peuvent contenir des éléments de types *Policyset* ou *policy*. Les éléments terminaux ou feuilles sont des éléments de type *rule* présentés à la section 5.3.1.4.Toutefois les éléments *rule* ne peuvent être contenus que par des éléments de type *policy*. Les éléments de type *rule* correspondent à une décision élémentaire, c'est-à-dire la permission ou l'interdiction exprimée à l'aide d'un élément de type *effect* présenté à la section 5.3.1.4 et une condition permettant de définir des contraintes supplémentaires sur la permission ou l'interdiction. Les conditions sont exprimées à l'aide d'élément de type *condition*. La politique de CA est représentée à l'aide d'une structure arborescente. De manière à préciser pour quelle requête chacun des nœuds ou feuille s'adresse, les éléments de type *Policyset*, *policy* ou *rule* utilisent des éléments de type *target*. Les éléments de type *target*, décrit à la section 5.3.1.4, permettent de décrire les requêtes cibles en décrivant les sujets, les objets, les actions et environnements concernés. Les éléments de type *target* peuvent être laissés vides, dans ce cas ils sont hérités de l'élément père. Pour enlever les ambiguïtés lors de la décision (c'est-à-dire si plusieurs décision différentes peuvent être prises pour une même requête), des éléments de types *Policy Combining Algorithm* sont utilisés par les

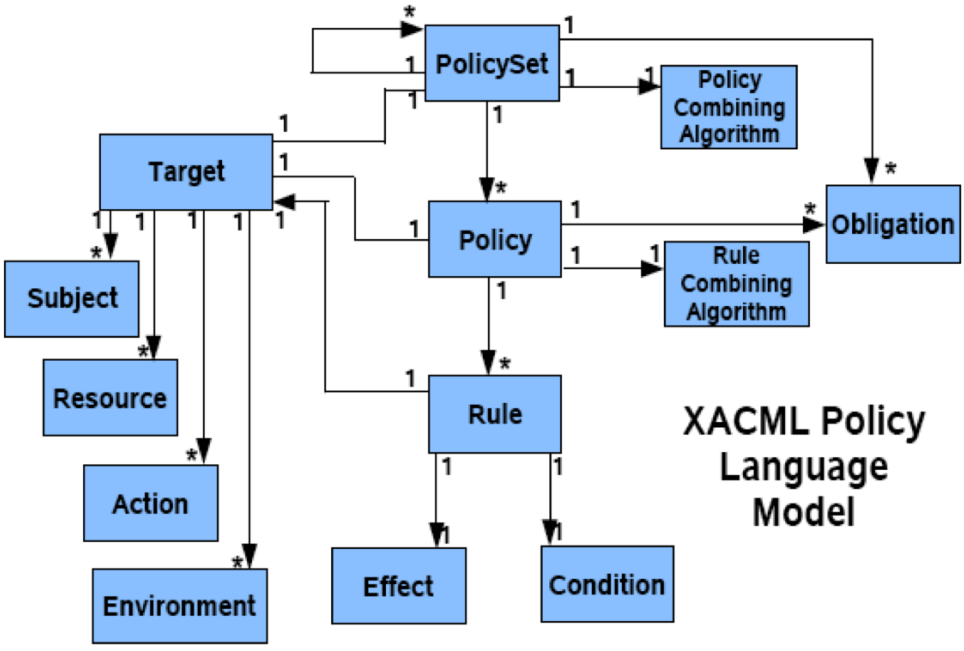

FIGURE 5.12 – Modèle utilisé pour décrire les politiques de CA en XACML

éléments de type *Policyset* et des éléments de type *Rule Combining Algorithm* sont utilisés par les éléments de type *Policy*. Les éléments de type *Policy Combining Algorithm* (*resp. Rule Combining Algorithm*) sont décrits à la section 5.3.1.4 (*resp.* 5.3.1.4). Les éléments de type *Policyset* et *Policy* peuvent contenir des éléments de type *Obligation*. Ces éléments ,décrits à la section 5.3.1.4, permettent l'exécution de procédures par le système lorsque la décision prise pour une requête correspond à une permission. Les différents exemples servant d'illustration sont issus des tests utilisés par l'implémentation fournie par Google [1].

5.3.1.4 Policyset

Le code source 5.1 présente un exemple de Policyset. Les Policyset représentent le constructeur de base utilisé en XACML. Ils peuvent contenir des règles ou des ensembles de règles et peuvent poin-

[1] http ://code.google.com/p/enterprise-java-xacml/

ter vers d'autres Policyset à l'aide d'une URI. Si l'élément Target (décrit dans la partie 5.3.1.4) d'un Policyset correspond au contexte d'une requête, Le PDP pourra utiliser ce Policyset pour résoudre cette requête. Si un Policyset contient des liens vers d'autres politiques à l'aide d'URL, alors ces URLs doivent pouvoir être résolues. Les différentes règles présentes dans un Policyset doivent être combinées à l'aide d'algorithmes précisés par l'élément *PolicyCombiningAlgId*. Les éléments de type *Policyset* sont utilisés de la même manière que les éléments *Policy* par les algorithmes de combinaison.

L'élément *description* est une chaine de caractères contenant des informations textuelles sur l'élément *policy set* (l'élément *description* est aussi disponibles pour les éléments *policy*, *rule* ...).

L'élément *Obligations* contient un ensemble d'obligations devant être réalisées par le PEP une fois la décision reçue. Si l'obligation ou une partie de l'obligation de peut être réalisée par le PEP, alors le PEP doit agir comme si la réponse retournée par le PDP était un refus.

```
<PolicySet
    xmlns="urn:oasis:names:tc:xacml:1.0:policy"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:policy
    cs-xacml-schema-policy-01.xsd"
    PolicySetId="urn:oasis:names:tc:xacml:1.0:conformance-
        test:IIIA028:policyset"
    PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-
        combining-algorithm:only-one-applicable">
  <Description>
    PolicySet for Conformance Test IIIA028.
  </Description>
  <Target>
    ...
  </Target>
  <Policy
    PolicyId="urn:oasis:names:tc:xacml:1.0:conformance-
        test:IIIA028:policy1"
```

```
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
            algorithm:first-applicable">
            ...
    </Policy>
    <Obligations>
        <Obligation
            ObligationId="urn:oasis:names:tc:xacml:1.0:conformance-
                test:IIIA028:policyset:obligation-1"
            FulfillOn="Permit">
                ...
        </Obligation>
    </Obligations>
</PolicySet>
```

Code source 5.1 – Exemple de Policyset utilisé en XACML

***Target* :** Un exemple d'élément *target* est donné au code source 5.2. L'élément *target* permet d'identifier pour quelles requêtes l'élément parent (*i.e.* un élément de type *policy*, *Policyset* ou une requête) est utile. Il contient une définition des sujets, des ressources, des actions et des environnements. L'élément *target* contient une suite d'éléments *subjects*, *resources*, *actions* et *environments*. Pour utiliser un élément de type *Policyset* dans l'évaluation d'une requête, toutes les données de son élément *target* doivent correspondre aux données de la requête. Si un élément n'est pas explicité dans l'élément *target* alors il correspond à toutes les valeurs possibles (*i.e.* c'est le même principe de fonctionnement que le symbole *wildcard* en EB³SEC ou que l'étoile dans une expression régulière). Dans les exemples fournis par SUN aucun élément de type *target* ne contient d'élément de type *environment*.

```
<Target>
    <Subjects>
        ...
    </Subjects>
    <Resources>
```

```
        ...
    </Resources>
    <Actions>
        ...
    </Actions>
</Target>
```

Code source 5.2 – Exemple de Target utilisé en XACML

Subjects : Un élément de type *subjects* contient une suite d'éléments de type *subject*, représentant un ensemble de valeurs possibles. Chacun des élément *subject* a des valeurs utilisées pour tester la correspondant avec le contexte d'une requête. Le code source présenté à la figure 5.3 présente un exemple d'élément de type *subjects*. Chaque élément contient une fonction de correspondance (*matching function*). Les fonctions de correspondance sont des fonctions de comparaison de type ou de valeur. Les arguments de la fonction d'appariement sont précisés à l'aide d'éléments de type *SubjectAttributeDesignator* ou *attributeSelector*. Les deux permettent de retrouver les valeurs des attributs d'une requête, le premier n'est utilisable qu'avec des éléments de type *subject* tandis que le second utilise une description XPATH de manière à retrouver la valeur d'un attribut.

```
<Subjects>
    <Subject>
        <SubjectMatch
            MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal
                ">
            <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">
                    Julius Hibbert</AttributeValue>
            <SubjectAttributeDesignator
                SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-
                    category:access-subject"
                AttributeId="urn:oasis:names:tc:xacml:1.0
                    :subject:subject-id"
```

```
                DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </SubjectMatch>
    </Subject>
</Subjects>
```

Code source 5.3 – Exemple d'élément Subjects utilisé en XACML

***Resources*, *actions*, *environments* :** Les éléments de type *resources*, *actions* et *environments* sont définis par le même mécanisme que les éléments de type *subjects*.

***Policy* :** Un exemple d'utilisation d'un élément de type *policy* est donné dans le code source 5.4. Ce type d'élément comprend des éléments de type *description*, *target*, au moins un élément de type *rule* et aucun ou plusieurs éléments de type *obligation*. Les éléments de type *policy* combinent les différentes règles de CA à l'aide d'un élément de type *rule-combining-algorithm*.

```
<Policy
2       PolicyId="urn:oasis:names:tc:xacml:1.0:conformance-
            test:IIIA028:policy3"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-
            algorithm:first-applicable">
    <Description>
        ...
    </Description>
7   <Target>
        ...
    </Target>
    <Rule
            RuleId="urn:oasis:names:tc:xacml:1.0:conformance-
                test:IIIA028:rule3"
12          Effect="Permit">
        ...
    </Rule>
    <Obligations>
```

```
        ...
17    </Obligations>
</Policy>
```

Code source 5.4 – Exemple de Policy utlilisé en XACML

***Rule* :** Un exemple d'élément de type *rule* est présenté dans la figure 5.5. Les élément de type *rule* contiennent un élément de type *effect* dont la valeur peut être "Deny" ou "Permit". Ils peuvent aussi contenir un élément de type *target*, celui-ci hérite des valeurs de l'élément englobant s'il n'est pas présent (c'est-à-dire de l'élément *Policy* ou *Policyset* contenant l'élément *Rule*). Un élément de type *rule* peut aussi contenir un élément de type *condition* correspondant à une fonction booléenne définie sur les sujets, les ressources, les actions et les environnements ou sur les différents attributs présents dans la politique.

```
<Rule
2       RuleId="urn:oasis:names:tc:xacml:1.0:conformance-
            test:IIIA028:rule4"
        Effect="Permit">
    <Description>
        A subject who is at least 100 years older than Bart
        Simpson may perform any action on any
7       resource. NOT APPLICABLE.
    </Description>
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer
        -greater-than-or-equal">
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-
            subtract">
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0
                :function:integer-one-and-only">
12              <SubjectAttributeDesignator
                    AttributeId="urn:oasis:names:tc:xacml:1.0:conformance
                        -test:age"
                    DataType="http://www.w3.org/2001/XMLSchema#integer"/>
```

```
            </Apply>
            <Apply FunctionId="urn:oasis:names:tc:xacml:1.0
                :function:integer-one-and-only">
                <EnvironmentAttributeDesignator
                        AttributeId="urn:oasis:names:tc:xacml:1.0:conformance
                            -test:bart-simpson-age"
                        DataType="http://www.w3.org/2001/XMLSchema#integer"/>
            </Apply>
        </Apply>
        <AttributeValue
        DataType="http://www.w3.org/2001/XMLSchema#integer">100</
            AttributeValue>
    </Condition>
</Rule>
```

Code source 5.5 – Exemple de Rule utilisé en XACML

Algorithmes de combinaison (des règles et des politiques) : Les politiques de CA décrites en XACML correspondent à des listes de permissions et d'interdictions. Plusieurs règles peuvent s'appliquer pour la même requête. Les algorithmes de combinaison permettent de choisir une règle, c'est-à-dire comment tenir compte des différentes règles pertinentes. Si plusieurs règles s'appliquent pour une même requête, la décision sera prise en fonction de l'algorithme de combinaison.

- *Deny-overrides*
  - Si une règle au moins a pour réponse "Deny", le résultat est "Deny".
  - Si toutes les règles ont comme résultat "Permit", alors le résultat est "Permit".
- *Permit-overrides*
  - Si au moins une règle a comme résultat "Permit", le résultat est "Permit".
  - Si au moins une règle a comme résultat "Deny" et toutes les autres règles ont comme résultat "NotApplicable" alors le résultat est "Deny".

  - Si toutes les règles ont comme résultat ”NotApplicable” alors le résultat est ”NotApplicable”.

- *First applicable* : les règles sont évaluées dans l'ordre dans lequel elles sont définies dans la politique.
  - Pour chaque règle, si l'élément *target* correspond et si l'élément *condition* est évalué à vrai, la règle est évaluée. Le résultat peut être ”Permit”, ”Deny” ou ”Indeterminate”.
  - Autrement, l'algorithme passe à la règle suivante. Si aucune règle ne s'applique, le résultat renvoyé est ”NotApplicable”.

- *Only-one-applicable*
  - Pour toutes les politiques définies, si aucune ne s'applique, le résultat est ”NotApplicable”.
  - Si plus d'une politique s'applique, le résultat est ”Indeterminate”.
  - Si une seule règle s'applique, alors le résultat est le résultat de l'évaluation de cette règle.

Obligation **:** Un exemple d'élément *obligation* est donné au code source 5.6. Les éléments de type *obligation* contiennent un identificateur, un élément de type *FulfillOn* indiquant quand l'obligation doit être prise en compte et aussi des éléments permettant d'indiquer les valeurs des différents arguments à prendre en compte. Une obligation correspond à une directive envoyée par le PDP au PEP. Cette directive explique ce qui doit être fait avant et après que l'accès ait été donné. Si le PEP n'est pas capable de se conformer à une de ses directives, l'autorisation doit être annulée. L'identificateur permet au PEP d'identifier la procédure à exécuter. Les éléments de type *attribute assignment* contenus dans un élément de type *obligation* permettent au PEP de connaître avec quelles valeurs il doit exécuter la procédure.

```
<Obligation
        ObligationId="urn:oasis:names:tc:xacml:1.0:conformance-
            test:IIIA014:policy1:obligation-1"
        FulfillOn="Permit">
    <AttributeAssignment
            AttributeId="urn:oasis:names:tc:xacml:1.0:conformance-
                test:IIIA014:policy1:assignment1"
            DataType="http://www.w3.org/2001/XMLSchema#string">assignment1
                </AttributeAssignment>
    <AttributeAssignment
            AttributeId="urn:oasis:names:tc:xacml:1.0:conformance-
                test:IIIA014:policy1:assignment2"
            DataType="http://www.w3.org/2001/XMLSchema#string">assignment2
                </AttributeAssignment>
</Obligation>
```

Code source 5.6 – Example d'Obligation utilisée en XACML

5.3.1.5 Les requêtes

```
<?xml version="1.0" encoding="UTF-8"?>
<Request
        xmlns="urn:oasis:names:tc:xacml:1.0:context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:context
        cs-xacml-schema-context-01.xsd">
    <Subject>
        <Attribute
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-
                    id"
                DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>Julius Hibbert</AttributeValue>
        </Attribute>
    </Subject>
```

```
    <Subject SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-
        category:recipient-subject">
15      <Attribute
            AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-
                id"
            DataType="http://www.w3.org/2001/XMLSchema#string">
          <AttributeValue>Bart Simpson</AttributeValue>
    </Attribute>
20  </Subject>
      <Subject SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-
          category:codebase">
        <Attribute
            AttributeId="urn:oasis:names:tc:xacml:1.0
                :subject:subject-id"
            DataType="http://www.w3.org/2001/XMLSchema#anyURI">
25        <AttributeValue>http://www.medico.com/applications/
            PatientRecordAccess</AttributeValue>
      </Attribute>
    </Subject>
    <Resource>
      <Attribute
30        AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource
            -id"
          DataType="http://www.w3.org/2001/XMLSchema#anyURI">
        <AttributeValue>http://medico.com/record/patient/BartSimpson</
          AttributeValue>
      </Attribute>
    </Resource>
35  <Action>
      <Attribute
          AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
          DataType="http://www.w3.org/2001/XMLSchema#string">
        <AttributeValue>read</AttributeValue>
40    </Attribute>
```

```
    </Action>
</Request>
```

Code source 5.7 – Exemple de requête

Un exemple de requête XACML est donné au code source 5.7, présentant les quatre types d'éléments que peut contenir une requête XACML.

L'élément *subject* représente l'acteur réalisant la requête. Il est défini à l'aide d'un ou plusieurs attributs.

L'élément *resource* définit les données ou les composantes du système auxquel le demandeur de la requête veut avoir accès. Il est défini par un ou plusieurs attributs. Ces attributs ne peuvent toutefois définir qu'un seul objet par requête.

L'élément *action* est unique pour chaque requête et peut être défini par un ou plusieurs attributs. Il décrit ce que le demandeur de la requête planifie de faire avec la ressource demandée.

L'élément *environment* contient tous les attributs nécessaires à la prise de décision pour la requête qui ne sont pas contenus dans les précédents éléments.

Pour chaque attribut utilisé dans les éléments de la requête (*i.e. subject*, *resource*, *action* et *environment*), un identificateur (c'est-à-dire un élément de type *AttributeId*) et un type (c'est-à-dire *DataType*) doivent être précisés. Les éléments de type *AttributeId* sont soit des éléments prédéfinis par le système soit définis par l'utilisateur au sein de la spécification XACML. Les types supportés par XACML sont les types de bases (*i.e. string*, *boolean*, *integer* ...).

5.3.1.6 Les décisions

Un exemple de décision est présenté au code source 5.8. Cette exemple présente les différents éléments que peut contenir une décision selon la norme OASIS.

```
<?xml version="1.0" encoding="UTF-8"?>
<Response
    xmlns="urn:oasis:names:tc:xacml:1.0:context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:context
    cs-xacml-schema-context-01.xsd">
  <Result>
    <Decision>Permit</Decision>
    <Status>
      <StatusCode
       Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
    <Obligations xmlns="urn:oasis:names:tc:xacml:1.0:policy">
      <Obligation
        ObligationId="urn:oasis:names:tc:xacml:1.0:conformance-
          test:IIIA001:obligation-1"
          FulfillOn="Permit">
        <AttributeAssignment
          AttributeId="urn:oasis:names:tc:xacml:1.0:conformance-
            test:IIIA001:assignment1"
          DataType="http://www.w3.org/2001/XMLSchema#string">
            assignment1</AttributeAssignment>
        <AttributeAssignment
          AttributeId="urn:oasis:names:tc:xacml:1.0:conformance-
            test:IIIA001:assignment2"
          DataType="http://www.w3.org/2001/XMLSchema#string">
            assignment2</AttributeAssignment>
      </Obligation>
      <Obligation
        ObligationId="urn:oasis:names:tc:xacml:1.0:conformance-
          test:IIIA001:obligation-2"
          FulfillOn="Permit">
        <AttributeAssignment
```

```
28                  AttributeId="urn:oasis:names:tc:xacml:1.0:conformance-
                        test:IIIA001:assignment1"
                    DataType="http://www.w3.org/2001/XMLSchema#string">
                        assignment1</AttributeAssignment>
                <AttributeAssignment
                    AttributeId="urn:oasis:names:tc:xacml:1.0:conformance-
                        test:IIIA001:assignment2"
                    DataType="http://www.w3.org/2001/XMLSchema#string">
                        assignment2</AttributeAssignment>
33          </Obligation>
        </Obligations>
    </Result>
</Response>
```

Code source 5.8 – Exemple de décision

Les décisions correspondent au résultat pris par le PDP pour une demande d'accès. Les différentes valeurs prises par une décision sont : "Permit", "Deny", "Not applicable" ou "Indeterminate".

Le statut permet de donner plus de précisions sur le résultat. Il est surtout utilisé dans le cas où la réponse correspond à un "Deny". Le statut permet de savoir si l'interdiction est due à la politique de CA, à une erreur de syntaxe dans la politique, à un attribut manquant ou à une erreur d'évaluation de la politique.

Les obligations doivent être exécutées par le PEP avant d'autoriser ou d'interdire l'accès aux données.

Lorsque la décision vaut "Permit", l'accès est autorisé. Dans le cas contraire, si l'accès est refusé la décision vaut "Deny". Si la ressource demandée n'est pas présente dans la politique, la décision vaut "Not Applicable" (aucune règle ne s'applique à la requête). Si la politique de CA ne peut prendre de décision pour la requête, même en utilisant les algorithmes de combinaison,

la décision vaut ”Indeterminate” (grâce au statut, des informations supplémentaires peuvent être récupérées : il manque des valeurs d'attributs, une erreur est survenue ...).

5.3.2 Implémentation

5.3.2.1 L'implémentation de SUN

Cette implémentation [2] a vu son développement commencer en 2003, les dernières événements datent de 2006. Cette implémentation correspond à une API Java. Cette API peut être utilisée pour développer un PEP et/ou un PDP. Une documentation est fournie afin d'expliquer le développement d'un PEP et d'un PDP. Le développement de ces deux modules est à la charge de l'utilisateur. Cette API est faite pour supporter les versions 1.0 et 2.0 de la norme XACML. Rien n'est dit à propos du support de la version 3.0.

5.3.2.2 L'implémentation de Google

Ce projet [3] a commencé en 2008, aucune modification n'est survenue depuis 2009. Cette implémentation ne supporte actuellement que les versions 1.0 et 2.0 de XACML. Les versions à venir de cette implémentation sont sensées supporter la norme 3.0 de XACML. Cette implémentation correspond aussi à une API Java utilisable pour développer sa propre solution. Cependant, un PDP utilisable est déjà fourni.

5.3.3 Les outils

Cette section présente quelques outils pouvant aider à la conception de politiques de CA en XACML.

[2] http ://sunxacml.sourceforge.net/
[3] http ://code.google.com/p/enterprise-java-xacml/

5.3.3.1 L'éditeur XACML UMU

Cette éditeur [4] correspond à un outil d'édition de politiques XACML. Il permet la création de politique de CA compatibles avec le standard XACML. Un outil de vérification est aussi fourni. Cet outil permet de valider la politique de CA créée par rapport aux schémas XML fournis par l'organisation OASIS. Bien que cet outil soit utile, il ne comporte pas d'interface permettant la représentation graphique de la politque de CA ou d'outils de vérification de la consistance de la politique de CA décrite.

5.3.3.2 Un outil de modélisation de politiques de CA de type RBAC

Cet outil se présente sous la forme d'un plugin de la plateforme Eclipse. Le document [79] fournit la documentation et les détails d'implémentation de cet outil. Cet outil permet la modélisation en XACML de politiques de CA respectant le modèle RBAC. Les politiques modélisées à l'aide de cet outil peuvent toutefois contenir des contrainte de SoD résolues de manière statique (SSD). La modélisation se réalise à l'aide d'une interface permettant d'avoir une représentation graphique de la politique. Une fois la politique modélisée, cet outil permet de générer les fichiers correspondant à la représentation XACML de la politique modélisée.

5.4 Quelque extensions XACML

Cette section présente quelques extensions du langage XACML. Ces extensions permettent entre autre de relier le modèle RBAC ou la résolution dynamique de contraintes de SoD au langage XACML. Les différentes illustrations de cette section sont issues de [6].

5.4.1 Un profil du modèle RBAC

Pour spécifier en XACML une politique respectant le standard RBAC, il faut principalement quatre composantes. Ces composantes sont des politiques XACML génériques qu'il faut adapter à la po-

[4] http ://xacml.dif.um.es/

litique modélisée. Des exemples d'utilisation de ces composantes sont décrits dans [6]. Ce profil a pour but de représenter une politique de CA de type RBAC en XACML. Pour fonctionner, ce profil repose sur l'utilisation d'un modèle supplémentaire. Ce module va pour chaque requête reçue ajouter les rôles possibles de l'utilisateur. Ce module, pour prendre sa décision, utilise une politique XACML, décrit dans la section 5.4.1.1. Une fois les rôles de l'utilisateur ajoutés à la requête, la politique de CA est décomposée en deux types de fichiers : Les fichiers correspondant aux rôles et ceux correspondant aux permissions. Les fichiers correspondant aux rôles décrits à la section 5.4.1.2, permettent d'isoler le rôle de l'utilisateur ayant envoyé la requête et d'appeler les permissions adéquates. Ces permissions sont décrites dans des fichiers présentés à la section 5.4.1.3. Il existe un fichier par rôle présent dans la politique de CA. Ce mécanisme permet de pouvoir représenter une hiérarchie de rôles. Finalement, les contraintes de SSD peuvent aussi être représentées avec ce profil, les mécanismes adéquats sont présentés en section 5.4.1.4.

5.4.1.1 Élément de type *Policy* ou *Policyset* utilisé pour l'affectation des rôles

Ces éléments XACML sont utilisés par un module annexe. Ce module permet d'ajouter aux requêtes reçues le ou les rôles que le sujet peut activer. Lors de son fonctionnement ce module ne prend en compte que le champs *sujet* de la requête. Il analyse alors la politique de CA qu'il utilise de manière à ajouter à la requête les attributs nécessaires pour définir le ou les rôles du sujet. Un exemple de politique de CA utilisée par ce module est donné au code source 5.9. Cet élément décrit quel rôle peut être activé par quelle utilisateur du système. Des contraintes (*i.e.* des heures de validité, le nombre maximum de rôles pouvant être activés simultanément, ...) peuvent apparaître dans cet élément. Cet élément doit pouvoir être consulté par les composantes du système qui permettent à l'utilisateur d'utiliser un ou plusieurs rôle durant sa session (ces attributs vont pouvoir alors être utilisés dans les requêtes d'accès envoyés au système). Cet entité est appelé 'activité d'habilitation des rôles' (*Role Enablement Activity*), dans le document [1].

```
<Policy xmlns="urn:oasis:names:tc:xacml:1.0:policy"
        PolicyId="Role:Assignment:Policy"
```

```
        RuleCombiningAlgId="&rule-combine;permit-overrides">
    <Target>
        <Subjects>
            <AnySubject/>
        </Subjects>
        <Resources>
            <AnyResource/>
        </Resources>
        <Actions>
            <AnyAction/>
        </Actions>
    </Target>
    <!-- Employee role requirements rule -->
    <Rule
            RuleId="employee:role:requirements" Effect="Permit">
        <Target>
            <Subjects>
                <Subject>
                    <SubjectMatch MatchId="&function;string-equal">
                        <AttributeValue
                                DataType="&xml;string">
                            Anne
                        </AttributeValue>
                        <SubjectAttributeDesignator
                                AttributeId="&subject;subject-id"
                                DataType="&xml;string"/>
                    </SubjectMatch>
                </Subject>
            </Subjects>
            <Resources>
                <Resource>
                    <ResourceMatch MatchId="&function;string-equal">
                        <AttributeValue DataType="&xml;string">
                            employee
```

```
                </AttributeValue>
                <ResourceAttributeDesignator
39          AttributeId="urn:someapp:attributes:role"
                        DataType="&xml;string"/>
             </ResourceMatch>
          </Resource>
       </Resources>
44     <Actions>
       <Action>
       <ActionMatch MatchId="&function;string-equal">
       <AttributeValue DataType="&xml;string">
       enable
49     </AttributeValue>
       <ActionAttributeDesignator AttributeId="&action;action-id"
       DataType="&xml;string"/>
       </ActionMatch>
       </Action>
54     </Actions>
    </Target>
    <Condition FunctionId="&function;and">
       <Apply FunctionId="&function;time-greater-than-or-equal">
          <Apply
59              FunctionId="&function;time-one-and-only">
             <EnvironmentAttributeDesignator AttributeId="&
                environment;current-time"
                DataType="&xml;time"/>
          </Apply>
          <AttributeValue
64              DataType="&xml;time">
             9h
          </AttributeValue>
       </Apply>
       <Apply FunctionId="&function;time-less-than-or-equal">
69        <Apply
```

```
                    FunctionId="&function;time-one-and-only">
                <EnvironmentAttributeDesignator AttributeId="&
                    environment;current-time"
                    DataType="&xml;time"/>
            </Apply>
            <AttributeValue
                    DataType="&xml;time">
                17h
            </AttributeValue>
        </Apply>
    </Condition>
</Rule>
</Policy>
```

Code source 5.9 – Exemple de Policyset utilisé pour l'assignation des rôles

5.4.1.2 L'élément *Policyset* correspondant aux rôles

Cet élément de type *Policyset* permet pour une requête reçu de déterminer le rôle joué par l'utilisateur ayant envoyé cette requête. Cette reconnaissance s'effectue à l'aide de l'élément *target* et analyse le champ correspondant au rôle dans la requête par le module décrit précédemment. Une fois le rôle déterminé l'élément de type *Policyset* fait un appel aux permissions correspondant à ce rôle à l'aide d'un élément *PolicySetIdReference*. Les permissions données à un rôle sont décrites à l'aide d'un élément de type *Policyset* décrit dans la section 5.4.1.3.

5.4.1.3 Les éléments *Policyset* correspondant aux permissions données à un rôle

Cette élément décrit les permissions données à un rôle. Un élément de type *Policyset* décrit ces permissions. Il peut contenir un ou plusieurs éléments de type *rule*. Chacun de ces élément de type *rule* correspond à une permission élémentaire donnée au rôle correspondant. De manière à modéliser l'héritage de rôles, les éléments contenant les permissions d'un rôle peuvent faire appel aux permissions d'un autre rôle à l'aide d'un élément de type *PolicySetIdReference*.

5.4.1.4 L'élément *Policyset* utilisé pour la SoD

Un exemple est donné à la figure 5.10. Toutefois, on ne peut faire intervenir qu'une résolution statique des problèmes de SoD. Dans l'exemple donné, un utilisateur ne peut plus réaliser d'action s'il possède à la fois les rôles "contractor" et "employee". L'élément de type *Policyset* comporte des liens vers les autres fichiers de la politique car il doit être appelé en premier lors de la résolution d'une requête.

```
<PolicySet xmlns="urn:oasis:names:tc:xacml:1.0:policy"
      PolicySetId="Separation:of:Duty:PolicySet"
      PolicyCombiningAlgId="&policy-combine;deny-overrides">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
      <AnyResource/>
    </Resources>
    <Actions>
      <AnyAction/>
    </Actions>
  </Target>
  <!-- Disallow simultaneous contractor and employee roles -->
  <Policy
       PolicyId="contractor:AND:employee:disallowed"
       RuleCombiningAlgId="&rule-combine;deny-overrides">
    <Target>
      <Subjects>
        <Subject>
        <SubjectMatch MatchId="&function;string-equal">
          <AttributeValue DataType="&xml;string">
            employee
          </AttributeValue>
```

```
                <SubjectAttributeDesignator AttributeId="
                    urn:someapp:attributes:role"
                        DataType="&xml;string"/>
            </SubjectMatch>
            <SubjectMatch MatchId="&function;string-equal">
                <AttributeValue DataType="&xml;string">
                    contractor
                </AttributeValue>
                <SubjectAttributeDesignator AttributeId="
                    urn:someapp:attributes:role"
                        DataType="&xml;string"/>
            </SubjectMatch>
            </Subject>
        </Subjects>
        <Resources>
            <AnyResource/>
        </Resources>
        <Actions>
            <AnyAction/>
        </Actions>
    </Target>
    <Rule RuleId="Deny:target:role:combination" Effect="Deny"/>
</Policy>
<!-- Reference the Role PolicySets that are subject to separation of
     duty -->
<PolicySetIdReference>
    RPS:employee:role
</PolicySetIdReference>
<PolicySetIdReference>
    RPS:contractor:role
</PolicySetIdReference>
<PolicySetIdReference>
    RPS:manager:role
</PolicySetIdReference>
```

```
</PolicySet>
```

Code source 5.10 – Exemple de Policyset utilisé pour les SoD

5.4.2 Un profil XACML pour la SoD

Cette section traite de l'expression de contraintes de SoD dans une politique décrite en XACML. Elle décrit d'abord la résolution statique dans 5.4.2.1 puis la résolution dynamique dans 5.4.3.

5.4.2.1 La SoD statique

La résolution statique des problème de SoD est intégrée au profil RBAC [6]. Un élément de type *Policyset* est utilisé pour stocker les rôles qui ne peuvent être joués par les mêmes personnes. Les outils décrits dans [79] peuvent être uilisés pour créer des politiques contenant des contraintes de SSD.

5.4.3 La SoD dynamique

Les contraintes de DSD peuvent être utilisées en XACML. Dans le document [19], les éléments de type *obligation* sont utilisés pour intégrer des contraintes de DSD dans une politique XACML. En pratique, les éléments de type *obligation* sont utilisés pour mettre à jour la politique de sécurité lorsqu'une action est exécutée. La décision renvoyée par le PDP au PEP permettant à l'utilisateur de réaliser la première action concernée par la SoD comporte dans son élément *obligation* l'appel à une routine. Cette routine devant être exécutée par le PEP permet la mise à jour de la politique de sécurité. Cette routine modifie la politique de manière à ce que l'utilisateur ayant réalisé la première action soit interdit de réaliser la seconde action concernée par la SoD. Cependant une fois la seconde action réalisée, aucune information n'est mentionnée quand à la mise à jour de la politique de sécurité de manière à pouvoir autoriser de nouveau l'utilisateur à pouvoir réaliser la seconde action pour une autre instance du *workflow*.

5.4.4 Le profil OrBAC

Le document [1] présente une extension du profil RBAC de manière à pouvoir prendre en compte tous les contextes nécessaires à l'expression d'une politique OrBAC. Pour modéliser une politique de CA en OrBAC, il faut pouvoir utiliser les concepts de rôle, activité, vue et contexte. De nouvelles composantes appelées REA (*Role Enablement Activity*), AEA (Activity Enablement Activity), VEA (View Enablement Activity) et CEA (Context Enablement Activity) sont introduites, leur fonctionnement est calqué sur celui du REA introduit dans le profil XACML du modèle RBAC [6]. L'orchestration et l'utilisation de ces composantes sont expliquées par le diagramme de séquence de la figure 5.13.

5.5 Un profil XACML pour la méthode EB^{3}SEC

La définition d'un profil XACML pour la méthode EB^{3}SEC permet de comparer les capacités d'expression des deux langages.

5.5.1 Étude de faisabilité du profil

Cette section décrit les différents points à prendre en compte dans l'établissement du profil. De manière à comparer les deux langages, on compare l'expression de différentes contraintes de CA dans les deux langages.

5.5.1.1 Permissions

Cette partie décrit l'expression d'une permission sans contraintes dans les deux langages. L'exemple utilisé est la permission pour le docteur Julius Hilbert d'accéder en lecture au dossier médical du patient Bart Simpson.

En EB^{3}SEC : Pour exprimer l'exemple de permission, il faut d'abord définir le diagramme de classes présenté à la figure 5.14. Le concept d'organisation n'étant pas requis pour cet exemple,

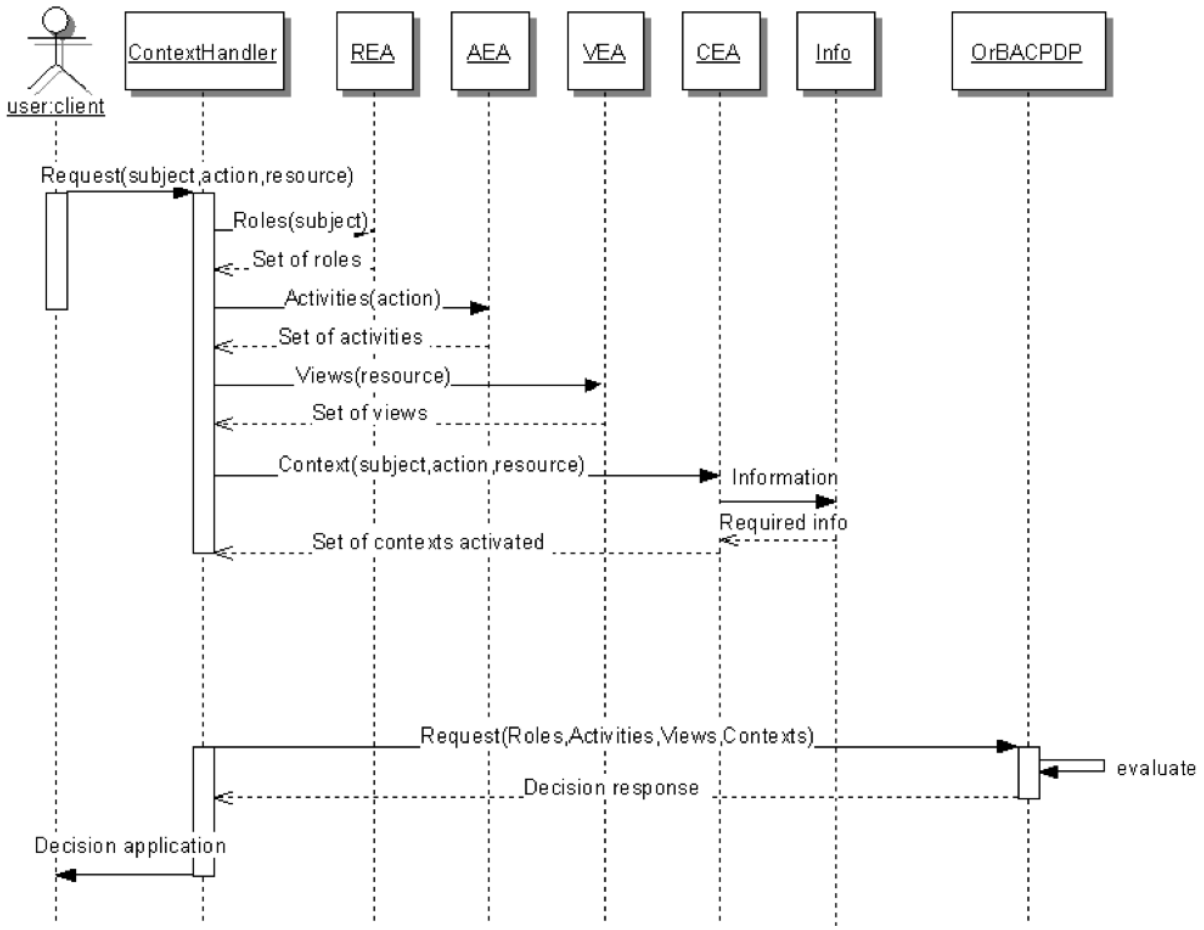

FIGURE 5.13 – Diagramme de séquence utilisé pour l'orchestration des composantes du profil OrBAC

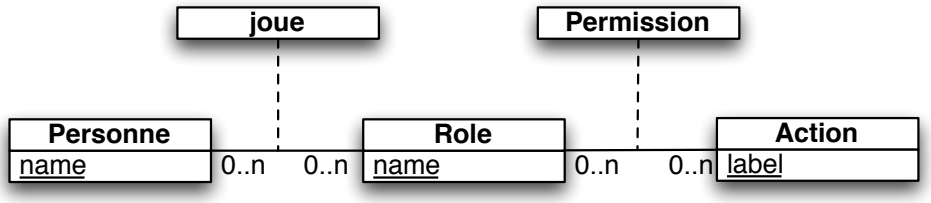

FIGURE 5.14 – Diagramme de classes utilisé en EB^{3}SEC

n'a pas été représenté dans le diagramme de classes. Le prédicat statique défini pour cet exemple est :

$$sp(\langle p, r, evt \rangle) \triangleq \langle p, r \rangle \in joue \wedge \langle r, label(evt) \rangle \in permission$$

La permission s'écrit alors :

$$\langle Julius\ Hibbert, \_, \_, \mathsf{read}(\ Bart\ Simpson\ ) \rangle$$

En XACML : Le code source 5.11 présente la permission modélisée en XACML. Pour modéliser cette règle, un élément de type *Rule* est utilisé.

```
<Rule
    RuleId="urn:oasis:names:tc:xacml:1.0:conformance-test:IIA1:rule"
    Effect="Permit">
  <Description>
    Julius Hibbert can read or write Bart Simpson's medical record.
  </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">Julius
                   Hibbert</AttributeValue>
            <SubjectAttributeDesignator
               AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-
                   id"
               DataType="http://www.w3.org/2001/XMLSchema#string"/>
         </SubjectMatch>
      </Subject>
   </Subjects>
   <Resources>
      <Resource>
         <ResourceMatch
            MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal
                ">
            <AttributeValue
               DataType="http://www.w3.org/2001/XMLSchema#anyURI">http://
                   medico.com/record/patient/BartSimpson</AttributeValue>
            <ResourceAttributeDesignator
               AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource
                   -id"
               DataType="http://www.w3.org/2001/XMLSchema#anyURI"/>
         </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>
```

```
38          <ActionAttributeDesignator
               AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
               DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </ActionMatch>
         </Action>
43    </Actions>
   </Target>
</Rule>
```

Code source 5.11 – Un exemple de permission en XACML

Comparaison : Le diagramme de classes et le prédicat statique sont écrits pour l'ensemble de la spécification. Pour d'autres permissions seuls des n-uplets seront rajoutés. L'expression EB^{3}SEC est beaucoup plus concise que l'expression XACML.

5.5.1.2 Diagramme de classes

En XACML, les seules entités utilisées sont les sujets, les objets, les actions ou l'environnement. Toutefois, tout autre concept peut être utilisé mais sera exprimé comme un attribut d'une entité précédemment citée. Le diagramme de classes utilisé dans une modélisation EB^{3}SEC peut alors être traduit en XACML. Cette traduction est réalisée de différentes manières, selon le contenu du diagramme de classes. Différentes méthodes de traduction sont décrites dans les sections suivantes.

Première version Le diagramme de classes du modèle EB^{3}SEC ne contient que les entités existant en XACML : sujets, objets, actions et environnement. La traduction est obtenue en explicitant en XACML les différentes instances du diagramme de classes.

Deuxième version Si le diagramme de classes contient d'autres entités, il est transformé de manière à ce que les entités qui ne sont pas des concepts utilisés en XACML deviennent des attributs des entités de base en XACML. Une fois cette transformation réalisée, le diagramme peut être transformé en XACML comme précédemment.

Troisième version Le diagramme de classes contient d'autres entités que les concepts de base utilisés en XACML. L'idée est de reprendre le principe utilisé dans [1, 6]. Pour chaque entité non supportée par XACML, il faut créer un élément de type *Enablement Activity*. Ces éléments permettent pour chaque requête reçu de la compléter avec les valeurs des entités supplémentaires du diagramme de classes. Leurs utilisation est comparable au diagramme de séquence décrit dans la figure 5.13.

5.5.1.3 Expression de processus

Le langage XACML permet de décrire une politique de contrôle d'accès de manière statique. Les éléments de l'algèbre de processus EB^{3}SEC n'ont pas d'équivalent en XACML. Cependant l'article [19] décrit une méthode, se basant sur le mécanisme d'obligation de XACML, permettant d'inclure des éléments dynamiques. C'est-à-dire qu'une fois une décision prise, la politique peut être modifiée au travers du champ obligation. Dans ce cas, le champ obligation permet d'exécuter une routine permettant la mise à jour de la politique. Les sections suivantes décrivent comment ce mécanisme peut être utilisé pour traduire les opérateurs EB^{3}SEC.

La séquence Dans cette section nous montrons comment traduire l'opérateur de séquence présent en EB^{3}SEC. Pour des besoins d'illustration nous considérons la séquence suivante :

$$\langle adrian, \_, \_, \_, \textbf{read}\,(\,dossier(alice)\,)\rangle \;\textbf{.}\; \langle boris, \_, \_, \_, \textbf{write}\,(\,dossier(bob)\,)\rangle$$

Cette section se focalise sur la traduction des opérateurs, nous supposons que le diagramme de classes a déjà été traduit à l'aide d'une des méthodes précédemment illustrées dans 5.5.1.2. Pour traduire cette séquence en XACML, nous allons créer un élément de type *Policy* pour chacune des actions sécurisées de l'exemple. Ensuite, nous utilisons le champ obligation pour appeler des routines de mise à jour. Le procédé est expliqué dans la suite.

1. Écrire la politique XACML correspondant à $\langle adrian, \_, \_, \_,$ **read (** $dossier(alice)$ **)**$\rangle$. Cette action sécurisée donne un élément de type *Policy* semblable à celui du code source 5.11. Cependant, le champ obligation appelle la routine `FromAtoB`

2. Écrire la politique XACML correspondant à $\langle boris, \_, \_, \_,$ **write (** $dossier(bob)$ **)**$\rangle$. Comme précédemment le résultat est semblable au code source de la figure 5.11. Cependant, le champ obligation fait appel à la routine `FinB`

La routine `FromAtoB` remplace l'élément *Policy* correspondant à la première action sécurisée de la séquence par l'élément *Policy* correspondant au deuxième élément de la séquence. La routine `FinB` retire de la politique XACML l'élément correspondant à la seconde action sécurisée de la séquence. Au début de l'exécution du système, l'élément *Policy* correspondant à la première action sécurisée de la séquence est présent dans la politique de CA. Lorsque l'événement correspondant à la lecture par *adrian* au dossier d'*alice* aura été exécuté, la routine `FromAtoB` s'exécute. *Adrian* n'a donc plus accès au dossier d'*Alice*, mais *boris* a alors accès au dossier de *bob*. Une fois l'événement correspondant à l'écriture dans le dossier de *bob* par *boris* exécuté, la routine `FinB` s'exécute. *boris* ne peut plus accéder au dossier de *bob*. L'utilisation de ces routine permet la simulation en XACML de l'opérateur de séquence utilisé en EB^{3}SEC.

Entrelacement quantifié Dans cette section, nous nous focalisons sur la traduction de l'opérateur d'entrelacement quantifié. Pour illustrer nos propos, nous considérons l'exemple d'entrelacement quantifié suivant :

$$|||\, x \in ENS \; : \; \langle adrian, \_, \_, \_, \textbf{read} \,(\, dossier(x) \,)\rangle$$

Pour traduire cet exemple en XACML, il faut traduire $\langle adrian, \_, \_, \_,$ **read** $(\, dossier(x) \,)\rangle$, pour chaque élément de ENS. Pour chaque valeur de x, l'action sécurisée correspond à une permission sans contrainte, elle peut être traduite par un élément de type *Policy* comme vu précédemment. Cependant, pour chacun de ces éléments utilisés, le champ obligation est utilisé pour faire appel à

une routine nommée `StopAx` où `x` est remplacé par la valeur de la variable x dans l'élément de type *Policy*. Dans un besoin de compréhension, nous supposons que la valeur de x est $anais$ (avec $anais \in ENS$). L'action sécurisée $\langle adrian, \_, \_, \_,$ **read** ($dossier(anais)$)$\rangle$ est transformée en un élément de type *Policy* faisant appel à la routine `StopAanais` à l'aide du champ obligation. Une fois l'événement correspondant à la lecture du dossier d'*anais* par *adrian*, la routine `StopAanais` de la politique de CA l'élément de type *Policy* permettant à *adrian* de réaliser cette lecture. L'utilisation de ces routines permet la simulation en XACML de l'opérateur d'entrelacement quantifié présent en EB^{3}SEC.

Le choix Dans un soucis d'illustration, nous utilisons l'exemple de choix suivant.

$$\langle adrian, \_, \_, \_,\ \textbf{read}\ (\ dossier(alice)\ )\rangle \mid \langle boris, \_, \_, \_,\ \textbf{write}\ (\ dossier(bob)\ )\rangle$$

Pour traduire cette expression de processus en EB^{3}SEC, il faut traduire chacune des actions sécurisées y participant. Chacune des actions sécurisées la composant fait appel à l'aide du champ obligation à la routine appelée `stopChoix`. Cette routine permet de retirer de la politique de CA les éléments de type *Policy* correspondant à chacun des choix possibles. De cette manière, lorsqu'un événement correspondant à un des choix possibles est reçu, les permissions correspondant à chacun des choix sont supprimées. La routine `stopChoix` permet de simuler en XACML l'utilisation de l'opérateur de choix présent en EB^{3}SEC.

5.5.2 Résultats

Nous donnons ici une étude de faisabilité d'un profil XACML pour la méthode EB^{3}SEC. Les traductions données ne couvrent pas l'ensemble des opérateurs EB^{3}SEC ni même l'ensemble des possibilités d'expression du diagramme de classes. L'étude de l'appel récursif des opérateurs d'algèbre de processus n'est pas abordée. Ces exemples mettent en évidence la difficulté de la réalisation de ce profil.

Une idée plus réaliste est d'exhiber des patrons spécialisés dans le contrôle d'accès en EB^{3}SEC (tels que la permission, l'interdiction, la séparation des devoirs ...), et de réaliser le profil d'un sous-ensemble d'EB^{3}SEC correspondant à ces patrons.

Ce profil permet la mise en place rapide dans un système d'information d'une politique de contrôle d'accès exprimée en EB^{3}SEC ou à l'aide d'un sous-ensemble d'EB^{3}SEC correspondant aux patrons décrits précédemment.

Conclusion

Les méthodes formelles de modélisation ont prouvé leur utilité lors de la réalisation de systèmes critiques. Leur utilisation dans la modélisation de SI n'a été introduite que très récemment, par exemple à l'aide de la méthode EB^3. Le domaine de la sécurité informatique est un domaine vaste recensant beaucoup de spécialités, certaines de ces branches font appel à des techniques utilisées dans le domaine des méthodes formelles : vérification de protocoles, cryptographie ... Le CA, composante de la sécurité informatique, ne s'est que récemment intéressé aux méthodes formelles. Au contraire des travaux actuels qui permettent au CA de profiter des technologies et méthodes utilisées dans le domaine formelle, cette thèse a pour but d'introduire les concepts de CA dans le domaine des méthodes formelles de modélisation de SI.

L'approche proposée présente une étude approfondie des concepts utilisés dans le domaine du CA et leur intégration au sein de la méthode EB^3. Le résultat de cette approche est la création de la méthode EB^{3}SEC, qui permet la modélisation d'une politique de CA. Au sein d'un même modèle peuvent être présentes des contraintes de tous types, aussi bien dynamiques que statiques.

Synthèse

La méthode EB^{3}SEC permet de spécifier des politiques de CA. Elle présente l'avantage de permettre la modélisation dans un même modèle des contraintes de tous types : permissions avec et sans contraintes, interdictions avec et sans contraintes, SoD statique et dynamique et obligations. De manière à aider à la conception d'un modèle EB^{3}SEC, plusieurs méthodologies sont proposées.

- Une première méthode consiste à réaliser une modélisation fonctionnelle du système et d'y inclure les règles de CA ;
- une seconde approche consiste à modéliser chaque règle de CA et ensuite de combiner les modélisations des différentes règles en un seul modèle ;
- la troisième approche crée, pour chaque action du système, une expression de processus tenant compte de toutes les contraintes de CA ciblant cette action ;
- la dernière méthode utilise des patrons. Chaque patron correspond à un type de contraintes de CA. La méthode permet aussi une systématisation de la mise en commun des modèles obtenus.

Une fois la méthode de modélisation choisie et le modèle de la politique de CA obtenu, l'étape de validation et de vérification arrive. Des techniques de vérification sont utilisées pour vérifier les contraintes de CA correspondant à des permissions sans contraintes ou des interdictions sans contraintes. Pour les règles correspondant à d'autres types de contraintes de CA, les techniques de simulation sont utilisées. Un ensemble de propriétés ont été établies. Ces propriétés permettent de vérifier la vivacité du modèle et aussi que si tous les concepts modélisés dans la politique de CA sont bien utiles. Ces propriétés peuvent être vues comme des conditions nécessaires à l'absence de blocage dans le modèle réalisé.

Une fois le modèle validé, il peut être utilisé pour l'implémentation d'un filtre de sécurité. Deux choix sont proposés à l'utilisateur. L'utilisation d'un outil spécialisé dans l'interprétation de modèles EB^{3}SEC peut assurer la prise en charge de la politique de CA. Une autre méthode consiste à traduire le modèle EB^{3}SEC vers un autre langage appelé XACML, langage présentant de nombreuses implémentations reconnues, testées et approuvées.

Perspectives

L'ensemble des types de contraintes de CA représentés par les patrons n'est pas exhaustif. Des patrons peuvent être créés pour chaque type de contraintes de CA existant. La création de ces patrons permet de présenter une approche systématique de création de modèles EB^{3}SEC. Des patrons peuvent aussi être créés pour étendre l'utilisation de la méthode EB^{3}SEC à d'autres domaines, par exemple la gestion du consentement en milieu hospitalier.

La phase de vérification d'un modèle EB^{3}SEC se concentre principalement sur deux types de contraintes et utilise des techniques de simulation pour la validation des autres types de contraintes. L'extension d'utilisation de méthodes de vérification à l'ensemble d'un modèle EB^{3}SEC serait un avantage. Une première idée est de créer un outil utilisant les techniques du domaine de la vérification de modèles et utilisant le langage EB^{3}SEC comme langage de description. Une autre idée est d'étudier la traduction des modèles EB^{3}SEC vers d'autres langages offrant l'opportunité d'utiliser des outils de vérification. Cette seconde option présente cependant un défaut de traçabilité : la méthode de traduction du modèle EB^{3}SEC vers un autre langage doit permettre lorsqu'une erreur est détectée dans le modèle traduit de trouver son origine dans le modèle réalisé en EB^{3}SEC.

La vérification peut aussi être améliorée en prenant en compte les aspects fonctionnels du SI. En effet, on peut envisager que le comportement fonctionnel du SI a été lui aussi modélisé de manière formelle, par exemple à l'aide de la méthode EB^3. Dans ce cas, on peut considérer le modèle global composé de la modélisation de la politique de CA et du comportement fonctionnel, et réaliser un ensemble de vérifications sur ce modèle global. Des problèmes d'interblocage peuvent alors être détectés et résolus.

Nous avons précisé un ensemble de propriétés à vérifier sur le modèle des contraintes de CA. Ces propriétés représentent un ensemble minimal de conditions nécessaires à la vivacité du système. De nouvelles propriétés peuvent être définies, ces propriétés permettraient aux utilisateurs

de la méthode EB^{3}SEC de s'assurer que les modèles réalisés sont corrects. Ces nouvelles propriétés pourront être spécifiques à un domaine donné.

Dans le cadre de l'implémentation, nos travaux se limitent à l'utilisation d'un outil adapté pour la simulation des modèles EB^{3}SEC et à la traduction de modèles EB^{3}SEC vers le langage XACML, de manière à pouvoir utiliser les implémentations du langage XACML. Le développement d'un outil implémentant un filtre de sécurité dans une architecture orientée services et utilisant le langage EB^{3}SEC comme langage d'entrée serait un grand avantage. Des techniques de contrôle de flux pourront être utilisées lors de l'élaboration de ce filtre. En effet, ces techniques pourront assurer le fit que les informations sont transmises au bon utilisateur et que la confidentialité des données est respectée.

Bibliographie

[1] Diala ABI HAIDAR, Nora CUPPENS-BOULAHIA, Frederic CUPPENS et Herve DEBAR. « An extended RBAC profile of XACML ». Dans Proceedings of the 3rd ACM workshop on Secure web services, SWS '06, pages 13–22, New York, NY, USA, 2006. ACM.

[2] Jean-Raymond ABRIAL. The B-Book : Assigning Programs to Meanings. Cambridge University Press, 1996.

[3] Jean-Raymond ABRIAL. Modeling in Event-B : System and Software Engineering. Cambridge University Press, 2009.

[4] Muhammad ALAM, Michael HAFNER et Ruth BREU. « A constraint based role based access control in the SECTET a model-driven approach ». Dans Proceedings of the 2006 International Conference on Privacy, Security and Trust : Bridge the Gap Between PST Technologies and Business Services, PST '06, pages 13 :1–13 :13, New York, NY, USA, 2006. ACM.

[5] Christopher ALM, Michael DROUINEAUD, Ute FALTIN, Karsten SOHR et Ruben WOLF. « A Classification Framework Designed for Advanced Role-based Access Control Models and Mechanisms ». Technical Report, Technologie-Zentrum Informatik Bremen University, 2009.

[6] A ANDERSON. « XACML Profile for Role Based Access Control (RBAC) ». OASIS Standard, 2004.

[7] S. BAJAJ, D. BOX, D. CHAPPELL, F. CURBERA, G. DANIELS, P. HALLAM-BAKER, M. HONDO, C. KALER, H. MARUYAMA, A. NADALIN, D. ORCHARD, H. PRAFULLCHANDRA, C. von RIEGEN, D. ROTH, J. SCHLIMMER, J. SHEWCHUK, A. VEDAMUTHU et U. YALÇINALP. « Web Services Policy Attachment (WSPolicyAttachment) ». Technical Report, mars 2006.

[8] D. BASIN, J. DOSER et T. LODDERSTEDT. « Model driven security : From UML models to access control infrastructures ». ACM Trans. Softw. Eng. Methodol., 15(1) :39–91, 2006.

[9] David BASIN, Ernst-Ruediger OLDEROG et Paul E. SEVINC. « Specifying and analyzing security automata using CSP-OZ ». Dans Proceedings of the 2nd ACM symposium on Information, computer and communications security, ASIACCS '07, pages 70–81, New York, NY, USA, 2007. ACM.

[10] David A. BASIN, Samuel J. BURRI et Günter KARJOTH. « Dynamic Enforcement of Abstract Separation of Duty Constraints ». Dans Michael BACKES et Peng NING, éditeurs, ESORICS, volume 5789 of Lecture Notes in Computer Science, pages 250–267. Springer, 2009.

[11] D E BELL et L J LAPADULA. « Secure Computer Systems : Mathematical Foundations and Model ». The MITRE Corporation Bedford MA Technical Report M74244 May, 1(M74-244) :42, 1973.

[12] Nazim BENAISSA. « La composition des protocoles de sécurité avec la méthode B événementielle ». Thèse de doctorat, Laboratoire Lorrain de Recherche en Informatique et ses Applications, Mai 2010.

[13] Elisa BERTINO, Piero Andrea BONATTI et Elena FERRARI. « TRBAC : A temporal role-based access control model ». ACM Trans. Inf. Syst. Secur., 4 :191–233, August 2001.

[14] Elisa BERTINO, Barbara CATANIA, Elena FERRARI et Paolo PERLASCA. « A logical framework for reasoning about access control models ». Dans Proceedings of the sixth ACM

symposium on Access control models and technologies, SACMAT '01, pages 41–52, New York, NY, USA, 2001. ACM.

[15] Tommaso BOLOGNESI et Ed BRINKSMA. « Introduction to the ISO specification language LOTOS ». Comput. Netw. ISDN Syst., 14(1) :25–59, 1987.

[16] Jery BRYANS. « Reasoning about XACML policies using CSP ». Dans Proceedings of the 2005 workshop on Secure web services, SWS '05, pages 28–35, New York, NY, USA, 2005. ACM.

[17] L. CHOLVY et F. CUPPENS. « Analyzing consistency of security policies ». Dans Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on, pages 103 –112, may 1997.

[18] David D. CLARK et David R. WILSON. « A Comparison of Commercial and Military Computer Security Policies ». Security and Privacy, IEEE Symposium on, 0 :184, 1987.

[19] Jason CRAMPTON et Hemanth KHAMBHAMMETTU. « XACML and role-based access control ». Dans Presentation at DIMACS Workshop on Security of Web Services and e-Commerce, page 174. Springer, 2005.

[20] Frédéric CUPPENS, Nora CUPPENS-BOULAHIA et Céline COMA. « MotOrBAC : un outil d'administration et de simulation de politiques de sécurité ». Dans Security in Network Architectures (SAR) and Security of Information Systems (SSI), First Joint Conference, June 6-9 2006.

[21] A. Abou El Kalam *et al.* « Organization based access control ». Dans 4th Intl. IEEE Workshop Policies for Distributed Systems and Networks, pages 120–130, Como, Italy, 2003. IEEE Press.

[22] Eduardo B. FERNANDEZ. « A methodology for secure software design ». Dans Procs. of the 2004 Int. Conf. on Software Engineering Research and Practice (SERP'04), pages 21–24, 2004.

[23] David F. FERRAIOLO, D. Richard KUHN et Ramaswamy CHANDRAMOULI. Role-Based Access Control. Artech House, Inc., Norwood, MA, USA, 2003.

[24] B. FRAIKIN et M. FRAPPIER. « Efficient Symbolic Execution of Large Quantifications in a Process Algebra ». Dans Jim WOODCOCK et Jin Song DONG, éditeurs, ICFEM 2007, volume 4789 of LNCS, pages 327–344. Springer Berlin/Heidelberg, novembre 2007.

[25] Benoit FRAIKIN et Marc FRAPPIER. « Efficient Symbolic Computation of Process Expressions ». Science of Computer Programming, 74(9) :723–753, jul 2009.

[26] M. FRAPPIER, B. FRAIKIN, F. GERVAIS, R. LALEAU et M. RICHARD. « Synthesizing Information Systems : the APIS Project ». Dans First International Conference on Research Challenges in Information Science, pages 73–84, Ouarzazate, Morocco, 2007.

[27] M. FRAPPIER, F. GERVAIS, R. LALEAU, B. FRAIKIN et R. ST-DENIS. « Extending statecharts with process algebra operators ». Dans Innovations in Systems and Software Engineering, pages 285–292, London, UK, august 2008. Springer London.

[28] M. FRAPPIER et R. ST-DENIS. « EB^3 : an entity-based black-box specification method for information systems ». Software and System Modeling, 2(2) :134–149, 2003.

[29] Marc FRAPPIER, Benoît FRAIKIN, Romain CHOSSART, Raphaël CHANE-YACK-FA et Mohammed OUENZAR. « Comparison of Model Checking Tools for Information Systems ». Dans Jin Song DONG et Huibiao ZHU, éditeurs, ICFEM, volume 6447 of Lecture Notes in Computer Science, pages 581–596. Springer, 2010.

[30] F. GERVAIS, M. FRAPPIER et R. LALEAU. « Generating relational database transactions from EB^3 attribute definitions ». 8(3) :423–445, juillet 2009.

[31] Dimitar GUELEV, Mark RYAN et Pierre SCHOBBENS. Model-Checking Access Control Policies. Dans Kan ZHANG et Yuliang ZHENG, éditeurs, Information Security, volume 3225

of Lecture Notes in Computer Science, pages 219–230. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30144-8_19.

[32] C. GUTIERREZ, E. FERNANDEZ-MEDINA et M. PIATTINI. « PWSSec : Process for Web Services Security ». Dans Proceedings of the IEEE International Conference on Web Services, pages 213–222, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Carlos GUTIÉRREZ, Eduardo FERNÁNDEZ-MEDINA et Mario PIATTINI. « Towards a Process for Web Services Security ». Journal of Research and Practice in Information Technology, 38(1), 2006.

[34] Michael A. HARRISON, Walter L. RUZZO et Jeffrey D. ULLMAN. « Protection in operating systems ». Commun. ACM, 19 :461–471, August 1976.

[35] Thai Son HOANG, David BASIN et Jean-Raymond ABRIAL. « Specifying Access Control in Event-B ». Technical Report 624, Department of Computer Science, ETH Zurich, juin 2009. http://www.inf.ethz.ch/research/disstechreps/techreports.

[36] C. A. R. HOARE. « Communicating sequential processes ». Commun. ACM, 21(8) :666–677, 1978.

[37] IBM et MICROSOFT. « Security in a Web Services World : A Proposed Architecture and Roadmap Version 1.0 ». IBM, Microsoft, April 2002.

[38] International Committee for Information Technology Standards (INCITS), American National Standard for Information Technology (ANSI). « Role-Based Access Control », 359-2004 edition, Februar 2004.

[39] Daniel JACKSON. Software Abstractions. MIT Press, 2006.

[40] Sushil JAJODIA, Pierangela SAMARATI, Maria Luisa SAPINO et V. S. SUBRAHMANIAN. « Flexible support for multiple access control policies ». ACM Trans. Database Syst., 26 :214–260, June 2001.

[41] James B D JOSHI, Elisa BERTINO et Arif GHAFOOR. « Temporal hierarchies and inheritance semantics for GTRBAC ». Dans Proceedings of the seventh ACM symposium on Access control models and technologies, SACMAT '02, pages 74–83, New York, NY, USA, 2002. ACM.

[42] Anas Abou El KALAM. « Modèles et politiques de sécurité pour les domaines de la santé et des affaires sociales ». Thèse de doctorat, Laboratoire d'Analyse et d Architecture des Systèmes du Centre National de la Recherche Scientifique, Décembre 2003.

[43] Anas Abou El KALAM, Salem BENFERHAT, Alexandre MIÈGE, Rania El BAIDA, Frédéric CUPPENS, Claire SAUREL, Philippe BALBIANI, Yves DESWARTE et Gilles TROUESSIN. « Organization based access control ». Dans Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY '03, pages 120–130, Washington, DC, USA, 2003. IEEE Computer Society.

[44] Slim KALLEL, Anis CHARFI, Mira MEZINI, Mohamed JMAIEL et Karl KLOSE. « From Formal Access Control Policies to Runtime Enforcement Aspects ». Dans Proceedings of the 1st International Symposium on Engineering Secure Software and Systems, ESSoS '09, pages 16–31, Berlin, Heidelberg, 2009. Springer-Verlag.

[45] Vladimir KOLOVSKI, James HENDLER et Bijan PARSIA. « Analyzing web access control policies ». Dans WWW '07 : Proceedings of the 16th international conference on World Wide Web, pages 677–686, New York, NY, USA, 2007. ACM.

[46] P. KONOPACKI. « Synthèse automatique de gardes EB3 ». Mémoire de maîtrise, Université de Sherbrooke, 2008.

[47] Pierre KONOPACKI, Marc FRAPPIER et Régine LALEAU. Expressing Access Control Policies with an Event-Based Approach. Dans Camille SALINESI, Oscar PASTOR, Will AALST, John MYLOPOULOS, Norman M. SADEH, Michael J. SHAW et Clemens SZYPERSKI, éditeurs, Advanced Information Systems Engineering Workshops, volume 83 of Lecture Notes in Business Information Processing, pages 607–621. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-22056-2-63.

[48] Yves LEDRU, Akram IDANI, Jeremy MILHAU, Nafees QAMAR, Regine LALEAU, Jean-Luc RICHIER et Mohamed-Amine LABIADH. « Taking into Account Functional Models in the Validation of IS Security Policies ». Dans 1st International Workshop on Information Systems Security Engineering (WISSE) host by CAISE, LNBIP. Springer, 2011. to be published.

[49] Yves LEDRU, Nafees QAMAR, Akram IDANI, Jean-Luc RICHIER et Mohamed-Amine LABIADH. « Validation of security policies by the animation of Z specifications ». Dans Ruth BREU, Jason CRAMPTON et Jorge LOBO, éditeurs, SACMAT, pages 155–164. ACM, 2011.

[50] Ninghui LI et Qihua WANG. « Beyond separation of duty : An algebra for specifying high-level security policies ». J. ACM, 55(3) :1–46, 2008.

[51] Torsten LODDERSTEDT, David A. BASIN et Jürgen DOSER. « SecureUML : A UML-Based Modeling Language for Model-Driven Security ». Dans Proceedings of the 5th International Conference on The Unified Modeling Language, UML '02, pages 426–441, London, UK, 2002. Springer-Verlag.

[52] M. MANKAI et Logrippo L.. « Access Control Policies : Modeling and Validation ». NOTERE, pages 85–91, 2005.

[53] Francis MER. « loi de sécurité financière ». Journal Officiel, (177), january 2003.

[54] Alexandre MIÈGE. « Définition d'un environnement formel d'expression de politiques de sécurité. Modèle Or-BAC et extensions ». Thèse de doctorat, Paristech, ENST, Septembre 2005.

[55] J. MILHAU, B. FRAIKIN et M. FRAPPIER. « Automatic Generation of Error Messages for the Symbolic Execution of EB^3 Process Expressions ». Dans Integrated Formal Methods : 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009, Proceedings, volume 5423 de LNCS, pages 337–351. Springer Berlin/Heidelberg, 2009.

[56] J. MILHAU, A. IDANI, R. LALEAU, M. LABIADH, Y. LEDRU et M. FRAPPIER. « Combining UML, ASTD and B for the formal specification of an access control filter ». Innovations in Systems and Software Engineering, 7 :303–313, 2011. 10.1007/s11334-011-0166-z.

[57] T MOSES. « eXtensible Access Control Markup Langage (XACML) Version 2.0 ». OASIS Standard, 2005.

[58] Anthony NADALIN, Chris KALER, Ronald MONZILLO et Phillip HALLAM-BAKER. « Web Services Security : SOAP Message Security 1.1 (WS-Security 2004) ». OASIS Standard, 2006.

[59] Qun NI, Elisa BERTINO et Jorge LOBO. « An obligation model bridging access control policies and privacy policies ». Dans Proceedings of the 13th ACM symposium on Access control models and technologies, SACMAT '08, pages 133–142, New York, NY, USA, 2008. ACM.

[60] Department of DEFENSE NATIONAL COMPUTER SECURITY CENTER. « Department of Defense Trusted Computer Systems Evaluation Criteria ». DoD, 1985.

[61] Jaehong PARK et Ravi SANDHU. « Towards usage control models : beyond traditional access control ». Dans Proceedings of the seventh ACM symposium on Access control models and technologies, SACMAT '02, pages 57–64, New York, NY, USA, 2002. ACM.

[62] Muhammad-Nafees QAMAR. « Animation of Security Design Models using Z ». Thèse de doctorat, Université de Grenoble, Décembre 2011.

[63] Prathima RAO, Dan LIN, Elisa BERTINO, Ninghui LI et Jorge LOBO. « An algebra for fine-grained integration of XACML policies. ». Dans Barbara CARMINATI et James JOSHI, éditeurs, SACMAT, pages 63–72. ACM, 2009.

[64] Indrakshi RAY, Na LI, Robert FRANCE et Dae-Kyoo KIM. « Using uml to visualize role-based access control constraints ». Dans Proceedings of the ninth ACM symposium on Access control models and technologies, SACMAT '04, pages 115–124, New York, NY, USA, 2004. ACM.

[65] Robert REIX. Système d'information et management des organisations. Vuibert, Paris, France, 1998.

[66] A. W. ROSCOE, J. C. P. WOODCOCK et L. WULF. « A CSP formulation of non-interference ». Dans European Symposium on Research in Computer Security, pages 33–35. Springer-Verlag LNCS 875, 1994.

[67] R. SANDHU. « Transaction Control Expressions For Separation Of Duty ». Dans 4th Aerospace Computer Security Application Conference, pages 282–286, 1988.

[68] R. S. SANDHU, E. J. COYNE, H. L. FEINSTEIN et C. E. YOUMAN. « Role-based access control models ». IEEE Computer, 29(2) :38–47, 1996.

[69] Paul SARBANES et Mike OXLEY. « Sarbanes-Oxley Act ». Public Law, (116) :107–204, 2002.

[70] Andreas SCHAAD et Jonathan D. MOFFETT. « A lightweight approach to specification and analysis of role-based access control extensions ». Dans Proceedings of the seventh ACM symposium on Access control models and technologies, SACMAT '02, pages 13–22, New York, NY, USA, 2002. ACM.

[71] Jeffrey SCHLIMMER. « Web Services Policy Framework (WSPolicy) Version 1.2 ». Microsoft, IBM, VeriSign, Sonic Software, SAP, BEA Systems, March 2006. Editor.

[72] Emin Gün SIRER et Ke WANG. « An access control language for web services ». Dans SACMAT '02 : Proceedings of the seventh ACM symposium on Access control models and technologies, pages 23–30, New York, NY, USA, 2002. ACM.

[73] SOCIÉTÉ-GÉNÉRALE. « Note explicative concernant la la fraude exceptionnelle ». 2008. http ://www.communiques-presse.net/Banque/societe-generale-note-explicative-concernant-fraude-exceptionnel.html.

[74] Karsten SOHR, Michael DROUINEAUD, Gail-Joon AHN et Martin GOGOLLA. « Analyzing and Managing Role-Based Access Control Policies ». IEEE Transactions on Knowledge and Data Engineering, 20 :924–939, 2008.

[75] J. M. SPIVEY. The Z notation : a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[76] Roshan K. THOMAS. « Team-based access control (TMAC) : a primitive for applying role-based access controls in collaborative environments ». Dans Proceedings of the second ACM workshop on Role-based access control, RBAC '97, pages 13–19, New York, NY, USA, 1997. ACM.

[77] G. H. von WRIGHT. « Deontic Logic ». Mind, 60(237) :1–15, 1951.

[78] J. WAINER, P. BARTHELMESS et A. KUMAR. « W-RBAC A Workflow Security Model Incorporating Controlled Overriding Of Constraints ». International Journal of Cooperative Information Systems, 12(4) :455–486, 2003.

[79] Jin XIN. « Applying Model Driven Architecture approach to Model Role Based Access Control System ». Mémoire de maîtrise, University of Ottawa, 2006.

[80] Chunyang YUAN, Yeping HE, Jianbo HE et Zhouyi ZHOU. A Verifiable Formal Specification for RBAC Model with Constraints of Separation of Duty. Dans Helger LIPMAA, Moti YUNG et Dongdai LIN, éditeurs, Information Security and Cryptology, volume 4318 of Lecture Notes in Computer Science, pages 196–210. Springer Berlin / Heidelberg, 2006. 10.1007/11937807_16.

[81] Eric YUAN et Jin TONG. « Attributed Based Access Control (ABAC) for Web Services ». Dans Proceedings of the IEEE International Conference on Web Services, ICWS '05, pages 561–569, Washington, DC, USA, 2005. IEEE Computer Society.

[82] John ZAO, Hoetech WEE, Jonathan CHU et Daniel JACKSON. « RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis ». Dans Proceedings of the 8th ACM symposium on Access control models and technologies, SACMAT '03, 2003.

[83] Nan ZHANG, Mark RYAN et Dimitar P. GUELEV. « Synthesising verified access control systems in XACML ». Dans Proceedings of the 2004 ACM workshop on Formal methods in security engineering, FMSE '04, pages 56–65, New York, NY, USA, 2004. ACM.